Programación de LEGO robots utilizando NQC

(Versión 3.03, Oct 2, 1999)

por Mark Overmars

Departamento de Informática
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Prologo

Los robots Lego MindStorms y CyberMaster son unos nuevos y maravillosos juguetes con los que se puede construir una amplia variedad de robots que pueden ser programados realizando todo tipo de complicadas tareas. Desgraciadamente, el software que viene con los robots es de funciones bastante limitadas, aunque visualmente atractivo. Por eso, sólo puede ser utilizado para tareas sencillas. Para liberar el total potencial de los robots, necesitas un entorno de programación diferente. NQC es un lenguaje de programación, escrito por Dave Baum, diseñado especialmente para Lego robots. Si nunca has escrito un programa, no te preocupes. NQC es realmente fácil de utilizar, y este tutorial te contará todo sobre él. De hecho, programar robots con NQC es mucho más fácil que programar un ordenador normal, así que esta es una oportunidad de convertirse en programador de un modo sencillo.

Para escribir programas de un modo más sencillo, está el RCX Command Center. Esta utilidad te ayuda a escribir tus programas, enviarlos al robot, y arrancar y detener el robot. RCX Command Center trabaja de modo similar a un procesador de textos, pero con algunos extras. Este tutorial utilizará RCX Command Center (versión 3.0 o superior) como entorno de programación. Puedes bajarlo libremente de la dirección web:

http://www.cs.uu.nl/people/markov/lego/

RCX Command Center funciona en PCs con sistema operativo Windows ('95, '98, 2000, NT). Asegúrate que has ejecutado el software que suministra Lego al menos una vez, antes del RCX Command Center. El software Lego instala ciertos componentes que el RCX Command Center necesita. El lenguaje NQC puede ser también utilizado en otras plataformas. Puedes bajarlo de la siguiente dirección de Internet:

http://www.enteract.com/~dbaum/lego/nqc/

La mayoría de este tutorial también es aplicable a otras plataformas (suponiendo que utilizas NQC versión 2.0 o superior), sólo que con el problema de que pierdes algunas herramientas y el código de color.

En este tutorial supongo que tienes el robot MindStorms. La mayoría de los contenidos son también aplicables a los robots CyberMaster aunque algunas de las funcionalidades no están disponibles para estos robots. También los nombres de, por ejemplo, los motores son diferentes, así que debes modificar un poco los ejemplos para que funcionen.

Agradecimientos

Me gustaría darle las gracias a Dave Baum por haber desarrollado NQC. También, muchas gracias a Kevin Saddi por escribir la primera versión de la primera parte de este tutorial.

Índice

Prologo — ***2***
Agradecimientos — 2

Índice — ***3***

I. Escribiendo tu primer programa — ***5***
Construcción de un robot — 5
Arranque del RCX Command Center — 5
Escritura del programa — 6
Ejecutando el programa — 7
Errores en tu programa — 7
Modificación de la velocidad — 8
Resumen — 8

II. Un programa más interesante — ***9***
Giros — 9
Repetición de órdenes — 9
Inclusión de comentarios — 10
Resumen — 11

III. Uso de variables — ***12***
Movimiento en espiral — 12
Números aleatorios — 13
Resumen — 13

IV. Estructuras de control — ***14***
La instrucción if (si condicional) — 14
La instrucción do — 15
Resumen — 15

V. Sensores — ***16***
Esperando al sensor — 16
Actuando sobre el sensor — 16
Sensores de luz — 17
Resumen — 18

VI. Tareas y subrutinas — ***19***
Tareas — 19
Subrutinas — 20
Funciones — 20
Definición de macros — 21
Resumen — 22

VII. Haciendo música — ***23***
Sonidos incluidos — 23
Tocando música — 23
Resumen — 24

VIII. Más sobre motores — ***25***
Parando suavemente — 25
Órdenes avanzadas — 25
Modificando la velocidad del motor — 26
Resumen — 26

IX. Más sobre sensores — ***27***
Modo y tipo de sensor — 27
El sensor de rotación — 28
Colocación de varios sensores en una entrada — 29
Elaboración de un sensor de proximidad — 30

Resumen 31

X. Tareas paralelas 32
Un programa incorrecto 32
Parada y reinicio de tareas 32
Uso de semáforos 33
Resumen 34

XI. Comunicación entre robots 35
Transmisión de órdenes 35
Selección del líder 36
Precauciones 36
Resumen 37

XII. Más ordenes 38
Temporizadores 38
La pantalla 38
Registro de datos 39

XIII. Referencia rápida NQC 40
Instrucciones 40
Condiciones 40
Expresiones 41
Funciones RCX 41
Constantes RCX 43
Palabras clave 43

XIV. Notas finales 44

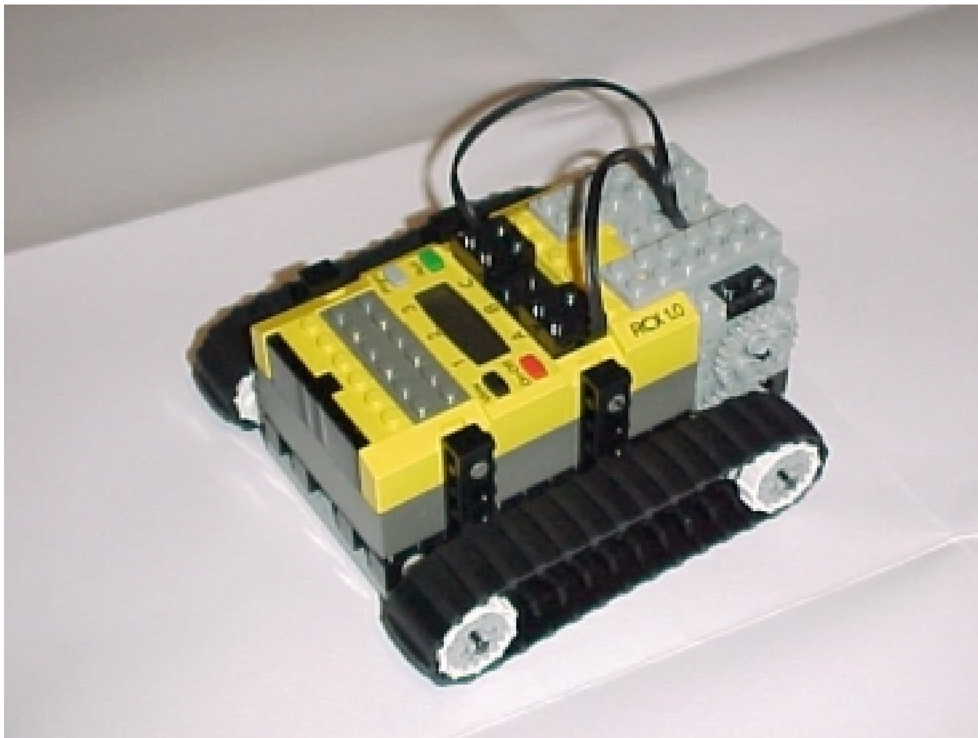
XV. Nota del traductor 45

I. Escribiendo tu primer programa

En este capítulo te voy a enseñar cómo escribir un programa muy sencillo. Vamos a programar un robot que avanzará durante 4 segundos, para retroceder a continuación durante otros 4 segundos. No es muy espectacular, pero te introduce en la idea básica de la programación. Y te mostrará también que fácil es esto. Pero antes de poder escribir un programa necesitamos un robot.

Construcción de un robot

El robot que utilizaremos a lo largo de este tutorial es una sencilla versión del robot top-secret descrito en las páginas 39-46[1] de tu constructopedia. Utilizaremos únicamente el chasis básico. Quita la parte frontal al completo con los dos brazos y los sensores de contacto. Asimismo, conecta los motores de un modo ligeramente diferente al que los cables están conectados al RCX (hacia el exterior). Esto es muy importante para que tu robot circule en la dirección correcta. Tu robot habrá de parecerse al de la figura:



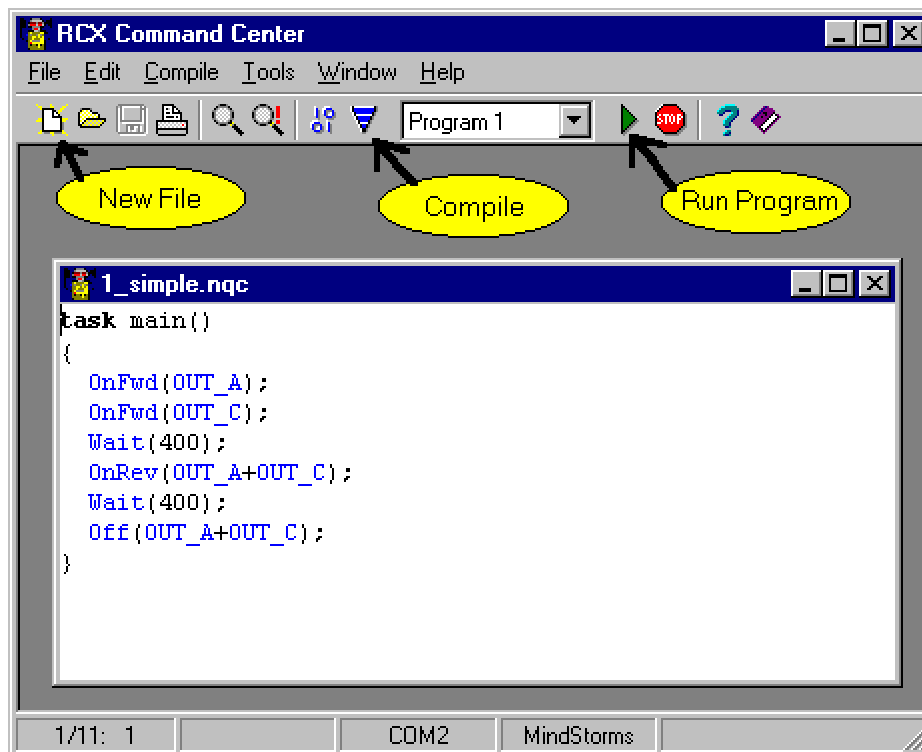
Asegúrate también que el puerto de infrarrojos esté correctamente conectado a tu ordenador, y que esté configurado para largo alcance (puedes chequear el correcto funcionamiento del robot con el software RIS).

Arranque del RCX Command Center

Escribiremos nuestros programas utilizando el RCX Command Center. Inícialo por medio de un doble clic sobre el icono RcxCC (asumo que ya tienes instalado el RCX Command Center, en caso contrario, bájalo del web site[2], descomprímelo y colócalo en el directorio que prefieras). El programa te preguntará dónde localizar el robot. Conecta el robot y pulsa **OK**. El programa encontrará automáticamente el robot (lo más probable). Ahora el interface de usuario aparece tal y como se muestra a continuación (sin la ventana).

---

[1] La constructopedia a la que se refiere repetidamente este manual es la correspondiente a RIS (Robotics Invention System) 1.0 ref. nº 9719 no comercializado en España (Nota del traductor)
[2] http://www.cs.uu.nl/people/markov/lego/



El interface se muestra como un editor de texto estándar, con el habitual menú y los botones para abrir y guardar los archivos, imprimir archivos, editar archivos, etc. Pero también hay unos menús especiales para compilar y transferir los programas al robot y obtener información del robot. Puedes ignorar esto por el momento.

Vamos a escribir un nuevo programa. Presiona el botón **New File** para crear una nueva y vacía ventana.

Escritura del programa

Ahora escribe el siguiente programa:

```
task main()
{
  OnFwd(OUT_A);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Parece un poco complicado al principio, por lo que vamos a analizarlo. Los programas en NQC están compuestos por tareas. Nuestro programa tiene precisamente una tarea, denominada `main` (principal). Todo programa ha de tener una tarea denominada `main`, que es la que será ejecutada por el robot. Aprenderás más sobre tareas en el capítulo VI. Una tarea está compuesta por varias órdenes, también denominadas instrucciones. Hay dos llaves alrededor de las instrucciones, de tal manera que quede claro que todas ellas pertenecen a esta tarea. Cada instrucción finaliza con un punto y coma. De este modo queda claro dónde finaliza una instrucción y dónde comienza la siguiente. En general, una tarea tiene el aspecto siguiente:

```
task main()
{
 declaración1;
 declaración2;
  …
}
```

Nuestro programa tiene seis instrucciones. Veámoslas de una en una:

```
OnFwd(OUT_A);
```

Esta instrucción indica al robot que ha de iniciar la salida A, es decir, que el motor conectado en la salida etiquetada A en el RCX debe moverse hacia adelante. Se moverá a la velocidad máxima, a no ser que previamente se establezca otra velocidad. Veremos más adelante cómo hacerlo.

```
OnFwd(OUT_C);
```

Es la misma instrucción, pero ahora arrancaremos el motor C. Tras estas dos instrucciones, los dos motores estarán en marcha, y el robot avanzará.

```
Wait(400);
```

Ahora es el momento de esperar por un tiempo. Esta instrucción nos pide esperar durante 4 segundos. El argumento, es decir, el número entre paréntesis, determina el número de “tics”. Un tic es una centésima de segundo. Por lo tanto puedes señalar al programa cuanto ha de esperar de un modo muy preciso. De modo que durante 4 segundos el programa no hará nada y el robot continuará avanzando.

```
OnRev(OUT_A+OUT_C);
```

El robot se ha alejado lo suficiente como para que hagamos que se mueva en la dirección opuesta, es decir, hacia atrás. Préstese atención a que nos hemos referido a los dos motores a la vez utilizando como argumento `OUT_A+OUT_C`. Podríamos combinar también las dos primeras de esta manera.

```
Wait(400);
```

Otra vez esperamos 4 segundos.

```
Off(OUT_A+OUT_C);
```

Y para acabar detenemos los dos motores.

Esto es el programa completo. Hace que los motores avancen durante 4 segundos, que retrocedan durante 4 segundos y que finalmente se detengan.

Probablemente te hayas fijado en los colores cuando lo has tecleado. Aparecen automáticamente. Todo lo que aparece en azul es una orden para el robot, o una indicación de un motor u otra cosa que el robot conoce. La palabra **task** aparece en negrilla por que es una palabra importante (reservada) en NQC. Otras palabras importantes también aparecerán en negrilla como bien veremos más adelante. Los colores son útiles para ver que no cometes ningún error mientras tecleas..

Ejecutando el programa

Una vez que tienes escrito el programa es necesario compilarlo (esto es, convertirlo en código que el robot pueda entender y ejecutar) y enviarlo al robot por medio del puerto de infrarrojos (“download” en el programa). Hay un botón que realiza las dos acciones a la vez (véase la figura anterior). Pulsar el botón y, asumiendo que no has cometido ningún error al teclear el programa, será correctamente compilado y transferido al robot (si hay errores en el programa, la aplicación lo notificará; véase a continuación).

Ahora puedes hacer ejecutar el programa. Para acabar, pulsa el botón verde en tu robot, o lo que es más fácil, pulsa el botón de arranque en tu ventana (véase la figura de la página anterior) ¿Ha hecho el robot lo que esperabas? Si no es así, probablemente será por que los cables están erróneamente conectados.

Errores en tu programa

Cuando se teclea un programa existe la posibilidad razonable de cometer algunos errores. El compilador notifica los errores y te informa en la parte inferior de la ventana, tal y como se ve en la siguiente figura:

1_errors.nqc

```
task main()
{
  OnFwd(OUT_D);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Of(OUT_A+OUT_C);
}
```

```
line 3: Error: undefined variable 'OUT_D'
```

Automáticamente selecciona el primer error (nosotros hemos tecleado mal el nombre del motor). Cuando hay más errores, puedes hacer un clic sobre los mensajes de error para acceder a ellos. Adviértase que a menudo errores del principio del programa son causa de más errores en otros lugares. Por ello es mejor corregir sólo los primeros pequeños errores del principio del programa y compilar el programa otra vez. Adviértase también que el código de color ayuda a evitar muchos errores. Por ejemplo, en la última línea hemos tecleado Of en lugar de `Off`. Al ser una orden desconocida, no aparece coloreada en azul.

Hay también errores que no son detectados por el compilador. Si escribimos `OUT_B` este error no será notificado ya que dicho motor existe (aunque no lo hayamos utilizado en el robot). Así que si el robot exhibe un comportamiento inesperado, es más que probable que haya algo equivocado en el programa.

Modificación de la velocidad

Como habrás advertido, el robot se mueve bastante rápido. Por defecto el robot se mueve lo más rápido que puede. Para modificar la velocidad, puedes utilizar la orden `SetPower()`. La potencia es un número entre 0 y 7. 7 es la más rápida, y 0 la más lenta (pero el robot todavía se moverá). A continuación se presenta una nueva versión de nuestro programa en el que el robot se mueve despacio:

```
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Resumen

En este capítulo has escrito tu primer programa en NQC, utilizando RCX Command Center. Ahora sabes cómo escribir un programa, cómo transferirlo al robot y cómo hacer que el robot ejecute el programa. Con el RCX Command Center se pueden hacer más cosas. Para averiguarlo, léase la documentación que lo acompaña. Este tutorial trata principalmente del lenguaje NQC y sólo menciona características del RCX Command Center cuando realmente es necesario.

También has aprendido varios importantes aspectos del lenguaje NQC. El primero es que cada programa tiene una tarea (task) denominada `main` que es siempre ejecutada por el robot. También has aprendido las cuatro órdenes más importante para motores: `OnFwd()`, `OnRev()`, `SetPower()` y `Off()`. Para finalizar, también conoces la instrucción `Wait()`.

II. Un programa más interesante

Nuestro primer programa no era muy espectacular. Vamos a intentar hacer uno más interesante. Vamos a hacerlo en varios pasos, introduciendo varias importantes características de nuestro lenguaje de programación NQC.

Giros

Puedes hacer que el robot gire haciendo parar uno de los dos motores o modificando su dirección de giro. He aquí un ejemplo. Tecléalo, transfiérelo a tu robot y ejecútalo. Avanzará un poco y dará un giro de 90º a la derecha.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  OnRev(OUT_C);
  Wait(85);
  Off(OUT_A+OUT_C);
}
```

Puedes probar con números algo diferentes a 85 en la segunda orden `Wait()` para hacer más preciso el giro de 90º. Esto depende del tipo de superficie sobre el que el robot gira. Algo que puede facilitar esto en el programa es utilizar un nombre para este número. En NQC puedes definir valores constantes tal y como se muestra en el siguiente programa.

```
#define TIEMPO_DE_AVANCE   100
#define TIEMPO_GIRO    85

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(TIEMPO_DE_AVANCE);
  OnRev(OUT_C);
  Wait(TIEMPO_GIRO);
  Off(OUT_A+OUT_C);
}
```

Las dos primeras líneas definen dos constantes. Estas pueden ahora ser usadas a lo largo del programa. Definir constantes es bueno por dos razones: hace el programa más legible, y es más fácil cambiar los valores. Adviértase que el RCX Command Center da colores propios a las instrucciones definidas. Tal y como veremos en el capítulo VI también puedes definir otros elementos diferentes a las constantes.

Repetición de órdenes

Intentemos ahora escribir un programa que haga que el robot circule describiendo una trayectoria cuadrada. Esto quiere decir lo siguiente: circula hacia adelante, gira 90º, circula hacia adelante otra vez, gira 90º, etc. Podríamos repetir el fragmento de código anterior 4 veces, pero esto se puede hacer de un modo más sencillo con la instrucción **`repeat`**.

```
#define TIEMPO_DE_AVANCE   100
#define TIEMPO_GIRO    85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(TIEMPO_DE_AVANCE);
    OnRev(OUT_C);
    Wait(TIEMPO_GIRO);
  }
  Off(OUT_A+OUT_C);
}
```

La cifra que se encuentra a continuación de la instrucción **repeat**, entre paréntesis, indica el número de repeticiones. Las instrucciones que han de ser repetidas, aparecen entre llaves, de la misma manera que la declaración de una tarea. Adviértase también que en el anterior programa hemos dado un sangrado a las instrucciones. Esto no es necesario, pero hace que el programa sea más legible

Como ejemplo final, hagamos que el robot de 10 vueltas describiendo una trayectoria cuadrada. Este es el programa:

```
#define TIEMPO_DE_AVANCE   100
#define TIEMPO_GIRO    85

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(TIEMPO_DE_AVANCE);
      OnRev(OUT_C);
      Wait(TIEMPO_GIRO);
    }
  }
  Off(OUT_A+OUT_C);
}
```

Ahora hay una instrucción de repetición dentro de otra. A esto se le llama instrucción de repetición “anidada”. Puedes anidar tantas instrucciones de repetición como desees. Observa cuidadosamente las llaves y el sangrado utilizado en este programa. La tarea comienza con la primera llave y acaba con la última. La primera repetición comienza en la segunda llave y finaliza en la quinta. La segunda repetición, anidada, comienza en la tercera llave y finaliza en la cuarta. Tal y como puedes ver las llaves siempre van por pares y se sangra la parte comprendida entre ellas.

Inclusión de comentarios

Para que tu programa sea más legible, es conveniente añadir comentarios. Siempre que pongas `//` en una línea, el resto de dicha línea será ignorado y podrá ser utilizado para comentarios. Un comentario largo puede ser colocado entre `/*` y `*/`. Los comentarios aparecen coloreados en verde en el RCX Command Center. El programa completo podría verse de la siguiente manera:

```
/*  10 CUADRADOS

    por Mark Overmars

Este programa hace que un robot recorra 10 veces un cuadrado
*/

#define TIEMPO_DE_AVANCE   100     // Tiempo para avanzar
#define TIEMPO_GIRO    85     // Tiempo para girar 90º

task main()
{
  repeat(10)                // Recorrer 10 cuadrados
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(TIEMPO_DE_AVANCE);
      OnRev(OUT_C);
      Wait(TIEMPO_GIRO);
    }
  }
  Off(OUT_A+OUT_C);         // Ahora se detienen los motores
}
```

Resumen

En este capítulo has aprendido a utilizar la instrucción **`repeat`** y a incluir comentarios. También has visto cómo anidar llaves y el uso del sangrado. Con todo esto, sabes hasta cierto punto cómo hacer que el robot se mueva a lo largo de cualquier tipo de trayectoria. Es un buen ejercicio intentar escribir algunas variaciones de los programas de este capítulo antes de continuar con el siguiente.

III. Uso de variables

El uso de variables constituye un importante aspecto de todo lenguaje de programación. Las variables son posiciones de memoria en las que podemos almacenar un valor. Podemos utilizar este valor en diferentes lugares y también podemos modificarlo. Voy a describir el uso de variables por medio de un ejemplo.

Movimiento en espiral

Supongamos que queremos adaptar el programa anterior para que el robot circule describiendo una espiral. Esto de puede lograr haciendo que el tiempo de espera sea mayor para cada movimiento de avance. Esto es, queremos aumentar el valor de `TIEMPO_DE_AVANCE` cada vez. Pero, ¿cómo podemos hacer esto? `TIEMPO_DE_AVANCE` es una constante, y las constantes no pueden ser modificadas. En su lugar necesitamos una variable. Las variables pueden ser fácilmente definidas en NQC. Puedes tener hasta 32, y darles un nombre a cada una de ellas. Este es el programa espiral:

```
#define TIEMPO_GIRO   85

int tiempo_de_avance;              // definir una variable

task main()
{
  tiempo_de_avance = 20;           // asignar el valor inicial
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(tiempo_de_avance);       // utilizar la variable para la espera
    OnRev(OUT_C);
    Wait(TIEMPO_GIRO);
    tiempo_de_avance += 5;         // incrementar la variable
  }
  Off(OUT_A+OUT_C);
}
```

Las líneas más interesantes están indicadas con comentarios. Primero definimos una variable tecleando la palabra clave **`int`** seguida del nombre que elijamos (generalmente utilizaremos letras minúsculas para los nombres de variables y mayúsculas para constantes, pero no es imprescindible). El nombre ha de empezar por una letra pero puede contener dígitos y el signo de subrayado. Ningún otro símbolo está permitido (esto es aplicable también a constantes, nombres de tareas...). La palabra **`int`** se reserva para números enteros. Sólo números enteros pueden ser almacenados en este tipo de variable. En la segunda línea de interés asignamos el valor 20 a la variable. Desde este momento siempre que utilicemos la variable su valor será 20. A continuación viene el bucle de repetición en el que utilizamos la variable para indicar el tiempo de espera, y al final del bucle se incrementa en valor de la variable en 5 unidades. Así la primera vez el robot avanza durante 20 tics, la segunda durante 25, la tercera durante 30, etc.

Además de sumar valores a una variable, también la podemos multiplicar por otro número utilizando *=, restar utilizando `-=` y dividir utilizando /= (fíjate en que el resultado de una división se redondea al valor entero más cercano). También puedes sumar una variable a otra, y escribir expresiones más complicadas. He aquí algunos ejemplos:

```
int aaa;
int bbb, ccc;

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);   // aaa es ahora igual a 80
}
```

Fíjate que en la segunda línea se definen múltiples variables. También podríamos haber combinado las tres en una sola línea.

Números aleatorios

En los programas anteriores hemos definido exactamente lo que se supone que el robot ha de hacer. Pero se gana en interés cuando el robot puede hacer cosas que desconocemos. Ahora queremos algo de aleatoriedad en los movimientos. En NQC puedes generar números aleatorios. El siguiente programa utiliza esto para circular de un modo aleatorio. Avanza constantemente durante un periodo de tiempo aleatorio y a continuación gira de modo aleatorio.

```
int tiempo_de_avance, tiempo_de_giro;

task main()
{
  while(true)
  {
    tiempo_de_avance = Random(60);
    tiempo_de_giro = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(tiempo_de_avance);
    OnRev(OUT_A);
    Wait(tiempo_de_giro);
  }
}
```

El programa define dos variables y les asigna números aleatorios. `Random(60)` representa un número aleatorio entre 0 y 60 (también puede ser 0 ó 60) Cada vez los números serán diferentes. Sería posible evitar el uso de variables escribiendo `Wait(Random(60))`.

Aquí también has visto un nuevo tipo de bucle. En lugar de utilizar la instrucción de repetición hemos escrito **`while(true)`**. La instrucción **`while`** (mientras) repite las instrucciones que le siguen mientras la condición que se encuentra entre paréntesis se cumpla (sea verdadera). La palabra reservada **`true`** (verdadero) se cumple siempre, por lo que las instrucciones comprendidas entre las llaves se repetirán indefinidamente hasta que queramos. Puedes aprender más sobre la instrucción **`while`** en el capítulo IV.

Resumen

En este capítulo has aprendido sobre el uso de variables. Las variables son muy útiles, pero debido a las restricciones de los robots, son un poco limitadas. Puedes definir solo 32 variables, y solo pueden almacenar números enteros. Pero para muchas tareas de robots esto es suficiente.

También has aprendido cómo crear números aleatorios, y así poder dotar al robot de un comportamiento imprevisible. Para acabar hemos visto cómo se usa la instrucción **`while`** para generar bucles que se repiten sin fin.

IV. Estructuras de control

En los capítulos precedentes hemos visto las instrucciones repeat y while. Estas instrucciones de control hacen que otras instrucciones del programa sean ejecutadas. Son las llamadas “estructuras de control”. En este capítulo veremos otras estructuras de control.

La instrucción if (si condicional)

A veces interesa que una parte en particular de un programa sea sólo ejecutada en ciertas situaciones. En esos casos se utiliza la instrucción if. Veamos un ejemplo. Vamos a modificar otra vez un programa con el que hemos trabajado anteriormente, pero dándole una nueva vuelta. Queremos que el robot circule por una línea recta y a continuación gire a izquierda o derecha. Para hacer esto necesitamos otra vez números aleatorios. Elegiremos un número aleatorio entre 0 y 1, esto es, el 0 o el 1. Si el número es 0, haremos un giro a la derecha, en caso contrario, el giro será a la izquierda. Este es el programa:

```
#define TIEMPO_DE_AVANCE   100
#define TIEMPO_GIRO    85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(TIEMPO_DE_AVANCE);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TIEMPO_DE_GIRO);
  }
}
```

La instrucción **`if`** tiene cierta semejanza con la instrucción **`while`**. Si la condición entre paréntesis se cumple, se ejecutará la parte comprendida entre llaves. En caso contrario, la parte comprendida entre las llaves que se encuentra a continuación de **`else`** será ejecutada. Analicemos un poco más la condición que hemos utilizado: `Random(1) == 0`. esto significa que cuando `Random(1)` sea igual a 0 la condición será verdadera. Probablemente te preguntarás porqué usamos == en lugar de =. La razón es que hay que diferenciarlo de la instrucción que asigna un valor a una variable. Se pueden comparar valores de diferentes maneras. Estas son las más importantes:

`==`	igual a
<	menor que
<=	menor o igual que
>	mayor que
>=	igual o mayor que
!=	diferente de

Se pueden combinar las condiciones utilizando &&, que significa “y”, o `||`, que significa “o”. A continuación se presentan algunos ejemplos de condiciones:

`true`	verdadero siempre
`false`	nunca verdadero
`ttt != 3`	verdadero cuando ttt no es igual a 3
`(ttt >= 5) && (ttt <= 10)`	verdadero cuando ttt está comprendido entre 5 y 10
`(aaa == 10) || (bbb == 10)`	verdadero si aaa o bbb (o los dos) es igual a 10

La instrucción **`if`** tiene dos partes. La parte inmediatamente a continuación de la condición, que se ejecuta cuando la condición es verdadera, y la parte tras **`else`**, que se ejecuta cuando la condición es falsa. La palabra

reservada **else** y la parte tras ella es opcional. Así que se pueden obviar si no hay nada que hacer cuando la condición es falsa.

La instrucción do

He aquí otra estructura de control, la instrucción **do**. Tiene la siguiente forma:

```
do
{
  instrucciones;
}
while (condición);
```

Las instrucciones entre las llaves tras el **do** son ejecutadas mientras la condición sea verdadera. La condición tiene la misma estructura que la que se ha descrito anteriormente para la instrucción **if**. A continuación se presenta un programa ejemplo. El robot circulará de modo aleatorio durante 20 segundos y entonces se detendrá.

```
int tiempo_de_avance, tiempo_de_giro, tiempo_total;

task main()
{
  total_time = 0;
  do
  {
    tiempo_de_avance = Random(100);
    tiempo_de_giro = Random(100);
    OnFwd(OUT_A+OUT_C);
    Wait(tiempo_de_avance);
    OnRev(OUT_C);
    Wait(tiempo_de_giro);
    total_time += tiempo_de_avance; tiempo_total += tiempo_de_giro;
  }
  while (tiempo_total < 2000);
  Off(OUT_A+OUT_C);
}
```

Adviértase que en este ejemplo hemos colocado dos instrucciones en una misma línea. Esto está permitido. Puedes colocar en una misma línea tantas instrucciones como quieras (siempre que se dispongan puntos y comas entre ellas). Pero por cuestiones de legibilidad del programa a menudo no es una buena idea.

Adviértase que la instrucción do se comporta casi del mismo modo que la instrucción **while**. Pero mientras que en la instrucción **while** las condiciones se comprueban antes de ejecutar las instrucciones, en la instrucción **do** esto se hace al final. En una instrucción **while**, las instrucciones que comprende pueden no ser nunca ejecutadas, pero con la instrucción **do** por lo menos se ejecutan una vez.

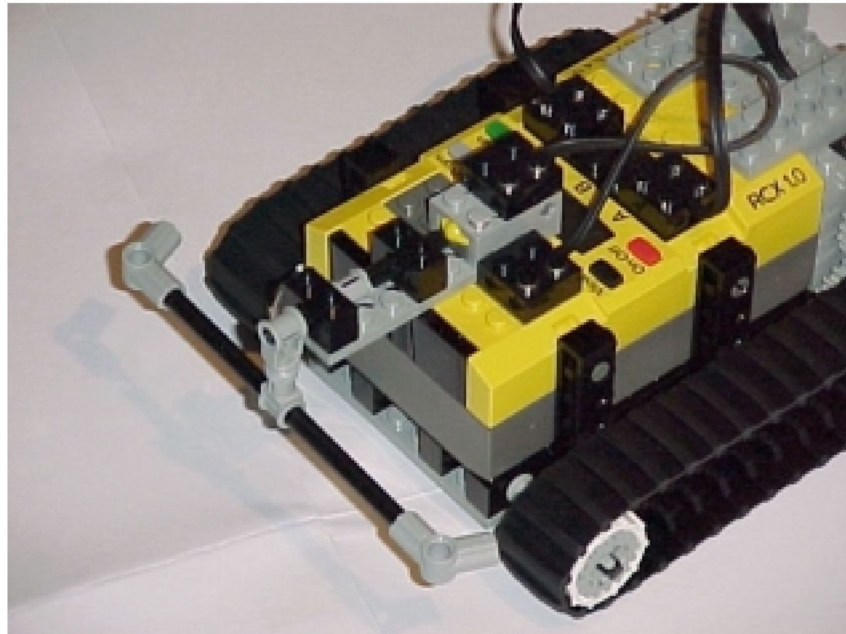
Resumen

En este capítulo hemos visto dos nuevas estructuras de control: la instrucción **if** y la instrucción **do**. Junto a las instrucciones **repeat** y **while** son las instrucciones que controlan el modo en que un programa es ejecutado. Es muy importante entender que hacen. Por ello te conviene tratar de resolver más ejemplos por tu cuenta antes de continuar.

También hemos visto que es posible incluir más de una instrucción en cada línea.

V. Sensores

Uno de los aspectos más interesantes de los LEGO robots es que puedes conectarles sensores y hacer que el robot reaccione a ellos. Para empezar podemos ver como modificar el robot añadiendo un sensor. Para ello, realizar el montaje del sensor como se presenta en la figura 4 de la página 28 de la constructopedia. Puedes hacer alguna pequeña modificación tal y como se ve en el robot de la figura:



Conecta el sensor a la entrada 1 en el RCX.

Esperando al sensor

Comencemos con un simple programa en el que el robot avanza hasta que choca con algo. Veámoslo:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 == 1);
  Off(OUT_A+OUT_C);
}
```

Aquí hay dos líneas importantes. La primera línea del programa comunica al robot que tipo de sensor utilizamos. SENSOR_1 es el número de entrada en la que está conectado el sensor. Las otras dos entradas se denominan SENSOR_2 y SENSOR_3. SENSOR_TOUCH indica que el sensor es un sensor de contacto. Para un sensor de luz deberíamos utilizar SENSOR_LIGHT. Una vez que hemos especificado el tipo de sensor, el programa pone en marcha los dos motores y el robot comienza a avanzar. La siguiente instrucción es una construcción muy útil. Espera hasta que la condición que se encuentra entre paréntesis sea verdadera. Esta condición dice que el valor del sensor SENSOR_1 debe ser 1, que quiere decir que el sensor está presionado. Mientras el sensor no esté presionado, el valor será 0. Es decir, esta instrucción espera hasta que el sensor esté presionado. En ese momento los motores se detendrán y la tarea habrá finalizado.

Actuando sobre el sensor

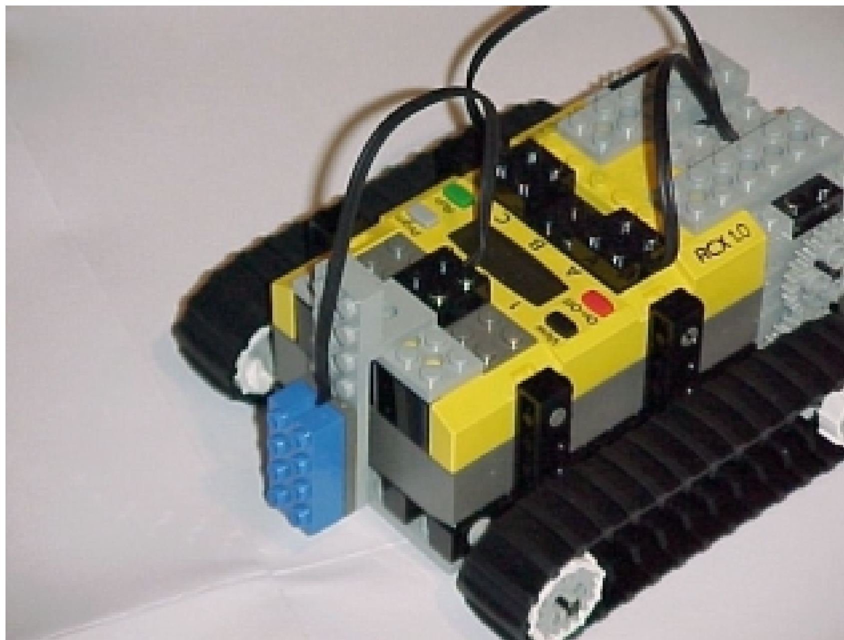
Intentemos hacer ahora un robot que evite obstáculos. Siempre que el robot choque contra un objeto, haremos que retroceda un poco, gire y a continuación continúe. Este es el programa:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Como en el ejemplo anterior primero indicaremos el tipo de sensor. A continuación el robot comenzará a avanzar. Por medio de un bucle **while** sin fin, comprobaremos continuamente si el sensor sufre algún contacto, y si es así retrocederá durante 3 décimas de segundo, girará durante 3 décimas de segundo y, a continuación, continuará avanzando otra vez.

Sensores de luz

Junto a los sensores de contacto, también encuentras un sensor de luz con tu MindStorms System. El sensor de luz mide la cantidad de luz en una dirección particular. El sensor de luz también emite luz. De este modo, es posible apuntar con el sensor de luz en una determinada dirección y distinguir la intensidad de luz reflejada por el objeto en esa dirección. Esto es particularmente útil cuando intentamos que un robot siga una línea en el suelo. Esto es lo que haremos en el ejercicio siguiente. Para comenzar necesitamos añadir un sensor de luz al robot, en la mitad de la parte frontal del robot apuntando hacia abajo. Conéctalo en la entrada 2. Puedes hacer la construcción como sigue:



Necesitamos también el circuito que viene con el kit RIS[3] (el papel con un circuito negro sobre él). La idea ahora es que el robot se asegure que el sensor de luz se encuentra sobre la pista. Siempre que la intensidad de la luz crezca, será que el sensor de luz se encuentra fuera de la pista y tendremos que corregir la dirección. He aquí un programa muy simple para resolver este problema que sólo funciona si el robot se desplaza por el circuito en el sentido de las agujas del reloj.

[3] RIS: Robotics Invention System

```
#define UMBRAL 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > UMBRAL)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= UMBRAL);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

El programa primero indica que el sensor 2 es un sensor de luz. A continuación establece que el robot avance y circule de acuerdo con un bucle indefinido. Siempre que el valor de luminosidad supere 40 (para ello utilizamos una constante fácilmente modificable, ya que depende en gran manera de la luz ambiente) invertiremos el giro de un motor y esperaremos hasta volver a la traza del circuito.

Tal y como observarás cuando ejecutes el programa, el movimiento no es muy uniforme. Prueba a añadir una orden `Wait(10)` antes de la orden **until** para hacer que el robot se mueva mejor. Obsérvese que el programa no funciona si el movimiento es en sentido antihorario. Para permitir el movimiento por una trayectoria arbitraria se requiere un programa mucho más complicado.

Resumen

En este capítulo has visto cómo trabajar con sensores de contacto y sensores de luz. También hemos conocido la orden **until** que es muy útil cuando se utilizan sensores.

Recomiendo que escribas varios programas por tu cuenta en esta fase. Tienes todos los ingredientes para dar ahora a tu robot comportamientos bastante complicados. Por ejemplo, prueba a instalar dos sensores en tu robot, uno en la parte frontal izquierda y el otro en la frontal derecha, y haz que el robot se mueva evitando los obstáculos contra los que choca. También intenta hacer que el robot se mantenga en el interior de un área limitada por una gruesa línea negra en el suelo.

VI. Tareas y subrutinas

Hasta ahora todos nuestros programas estaban formados por una única tarea. Pero los programas en NQC soportan múltiples tareas. También es posible escribir fragmentos de código en las llamadas subrutinas que podrás utilizar en distintos lugares de tu programa. Utilizando tareas y subrutinas tu programa será más fácil de comprender y más compacto. En este capítulo veremos varias de sus posibilidades.

Tareas

Un programa NQC consiste como máximo en 10 tareas. Cada tarea tiene su nombre. Una tarea ha de tener el nombre **main**, y esta será la que se ejecutará. Las otras tareas sólo serán ejecutadas cuando la tarea principal las llame para ser ejecutadas utilizando la orden **start**. A partir de este momento ambas tareas estarán ejecutándose simultáneamente (es decir, la primera tarea sigue ejecutándose). Una tarea en marcha puede detener también otra tarea utilizando el comando **stop**. Esta tarea podrá ser reiniciada más adelante , pero arrancará desde el principio, no desde el lugar en que se detuvo.

Permíteme mostrarte el uso de las tareas. Coloca otra vez el sensor en tu robot. Queremos hacer un programa en el que el robot circule describiendo cuadrados como antes. Pero cuando choque con un obstáculo deberá reaccionar. Es difícil hacer esto en una sola tarea, porque el robot debe hacer dos cosas en el mismo momento: conducir (esto es, conectar y desconectar los motores en los momentos correctos) y mirar los sensores. Así que es mejor utilizar dos tareas para esto, una tarea para circular en cuadrados, y otra que reaccione a los sensores. Este es el programa:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start chequea_sensor;
  start mover_cuadrado;
}

task mover_cuadrado ()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task chequea_sensor ()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop mover_cuadrado;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start mover_cuadrado;
    }
  }
}
```

La tarea **main** sólo define el tipo de sensor e inicia las otras dos tareas. Tras esto la tarea **main** finaliza. La tarea mover_cuadrado mueve el robot siempre en cuadrados. La tarea chequea_sensor comprueba si el sensor de contacto está pulsado. Si es así, realiza las acciones siguientes: lo primero de todo detiene la tarea mover_cuadrado. Esto es muy importante. Ahora chequea_sensor toma el control del movimiento del robot. A continuación el robot retrocede un poco y gira. Entonces puede empezar otra vez mover_cuadrado para permitir al robot circular de nuevo en cuadrados.

Es muy importante recordar que las tareas que has iniciado se ejecutan simultáneamente. Esto puede dar lugar a resultados inesperados. El capítulo X profundiza en estos problemas en detalle y proporciona soluciones.

Subrutinas

A veces necesitas una misma parte del código en múltiples lugares de tu programa. En ese caso puedes escribir esa parte del código como una subrutina y darle un nombre. A partir de entonces puedes ejecutar dicha parte de código simplemente llamándola por su nombre desde el interior de la tarea. NQC (o actualmente el RCX) permite un máximo de 8 subrutinas. Veamos un ejemplo.

```
sub gira()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  gira();
  Wait(200);
  gira();
  Wait(100);
  gira();
  Off(OUT_A+OUT_C);
}
```

En este programa hemos definido una subrutina que hace que el robot gire en torno a sí. La tarea main invoca a la subrutina tres veces. Adviértase que hemos invocado la subrutina escribiendo su nombre seguido de paréntesis. Así que su apariencia es similar a la de muchas de las órdenes que hemos visto. Solo que no lleva parámetros, por lo que no tiene nada entre paréntesis.

Aquí hay que tomar algunas precauciones. Las subrutinas son un poco extrañas. Por ejemplo, las subrutinas no pueden ser invocadas desde otras subrutinas. Las subrutinas pueden ser llamadas desde distintas tareas pero esto no es alentador. Conducen muy fácilmente a problemas, ya que la misma subrutina puede incluso estar ejecutándose doblemente por diferentes tareas. Esto tiende a generar efectos no deseados. También, cuando llamamos una subrutina desde diferentes tareas, debido a limitaciones en el firmware RCX, no se pueden utilizar expresiones complicadas. Así que a no ser que sepas exactamente lo que haces, *no llames una subrutina desde diferentes tareas.*

Funciones

Tal y como se ha indicado, las subrutinas causan ciertos problemas. La parte buena es que sólo se almacenan una vez en el RCX. Esto ahorra memoria y, ya que el RCX no tiene demasiada memoria, es muy útil. Pero cuando las subrutinas son cortas, es mejor utilizar funciones en su lugar. Estas no se almacenan separadamente sino que se copian en cada lugar en que tienen que ser usadas. Esto consume más memoria pero problemas tales como los que se producen al utilizar complicadas expresiones, ya no se presentan. Además, no hay límite en el número de funciones.

La definición e invocación de funciones se hace exactamente de la misma manera que con las subrutinas. Solo que se usa la palabra reservada **void** en lugar de **sub** (la palabra **void** es utilizada porque es la misma que aparece en otros lenguajes, como en C). Así que el ejemplo anterior, utilizando funciones, queda así:

```
void gira()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  gira();
  Wait(200);
  gira();
  Wait(100);
  gira();
  Off(OUT_A+OUT_C);
}
```

Las funciones tienen otra ventaja sobre las subrutinas. Ellas pueden tener argumentos. Los argumentos pueden ser utilizados para transferir un valor a ciertas variables en una función. Por ejemplo, supongamos que en el ejemplo anterior hacemos que el tiempo de giro sea un argumento de la función, como en el siguiente ejemplo:

```
void gira(int tiempo_de_giro)
{
  OnRev(OUT_C); Wait(tiempo_de_giro);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  gira(200);
  Wait(200);
  gira(50);
  Wait(100);
  gira(300);
  Off(OUT_A+OUT_C);
}
```

Adviértase que en el paréntesis tras el nombre de la función especificamos su(s) argumento(s). En este caso indicamos que el argumento es un número entero. (también hay algunas otras opciones) y que el nombre es tiempo_de_giro. Cuando hay más argumentos, hay que separarlos con comas.

Definición de macros

Hay todavía un modo más de dar nombres a pequeños fragmentos de código. Se pueden definir macros en NQC (no confundirlas con las macros del RCX Command Center). Hemos visto anteriormente que podemos definir constantes utilizando #define, dándoles un nombre. Pero en realidad podemos definir cualquier fragmento de código. Ahora veremos el mismo programa pero utilizando macros para girar.

```
#define gira OnRev(OUT_C);Wait(340);OnFwd(OUT_A+OUT_C);

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  gira;
  Wait(200);
  gira;
  Wait(100);
  gira;
  Off(OUT_A+OUT_C);
}
```

Tras la instrucción #define se encuentra la palabra gira y el texto que va tras ella. A partir de ahora siempre que teclees gira, será reemplazada por el texto correspondiente. Adviértase que el texto se encuentra en una sola línea (actualmente hay modos de colocar instrucciones #define en múltiples líneas, pero no es recomendable).

Las instrucciones define son en realidad mucho más potentes. Pueden también tener argumentos. Por ejemplo, podemos poner el tiempo de giro como un argumento en la instrucción. Aquí se ofrece un ejemplo en el que definimos cuatro macros, una para avanzar, otra para retroceder, otra para girar a la izquierda y otra para girar a la derecha. Cada una tiene dos argumentos: la velocidad y el tiempo.

```
#define gira_dcha(s,t)  SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define gira_izq(s,t)   SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define avanza(s,t)    SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define retrocede(s,t)   SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);

task main()
{
  avanza(3,200);
  gira_izq(7,85);
  avanza(7,100);
  retrocede(7,200);
  avanza(7,100);
  gira_dcha(7,85);
  avanza(3,200);
  Off(OUT_A+OUT_C);
}
```

Es muy útil definir las macros. Convierte el código en más compacto y legible. También, puedes adaptar más fácilmente el código, por ejemplo, cuando cambias las conexiones a los motores.

Resumen

En este capítulo has visto el uso de tareas, subrutinas, funciones y macros. Estas tienen diferentes usos. Las tareas generalmente se ejecutan simultáneamente y se ocupan de diferentes cosas que tienen que hacerse a la vez. Las subrutinas son útiles cuando largos e idénticos fragmentos de código son utilizados en distintos lugares de una misma tarea. Las funciones son útiles cuando fragmentos de código han de ser utilizados en diferentes lugares de diferentes tareas, pero utiliza más memoria. Finalmente las macros son muy útiles para pequeños fragmentos de código que han de ser utilizados en diferentes lugares. Ellas pueden tener además parámetros, lo que las convierten en más útiles.

Ahora que has recorrido los capítulos anteriores hasta aquí, tienes el conocimiento que necesitas para que los robots hagan cosas complicadas. El resto de capítulos de este tutorial muestra otras cuestiones que son sólo importantes en ciertas aplicaciones.

VII. Haciendo música

El RCX tiene un altavoz incorporado que puede generar sonidos e incluso tocar composiciones sencillas de música. Esto es en particular útil cuando quieres que el RCX te avise cuando algo suceda. Pero también puede ser divertido que el robot haga música mientras circula.

Sonidos incluidos

Hay seis sonidos incluidos en el RCX, numerados desde 0 a 5. Estos sonidos son los siguientes:

0	Click de tecla
1	Pitido
2	Barrido de frecuencia decreciente
3	Barrido de frecuencia creciente
4	“Buhhh” Sonido de error
5	Barrido de rápido crecimiento

Puedes generar dichos sonidos utilizando la orden `PlaySound()`. He aquí un pequeño programa que toca todos ellos:

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

Debes preguntarte a qué vienen esas órdenes wait. La razón es que la orden que genera el sonido no espera a acabarlo. Inmediatamente ejecuta la siguiente orden. El RCX tiene un pequeño buffer en el que puede almacenar algunos sonidos, pero tras un tiempo el buffer se llena y los sonidos se pierden. Esto no es grave con los sonidos, pero sí es muy importante para la música, como veremos a continuación.

El argumento de `PlaySound()` ha de ser una constante. ¡No puedes utilizar una variable aquí!

Tocando música

Para música más interesante, NQC tiene la orden `PlayTone()`. Tiene dos argumentos. El primero es la frecuencia, y el segundo la duración (en tics de 1/100 de segundo, como en la orden wait). He aquí la tabla con las frecuencias más útiles:

Sonido	1	2	3	4	5	6	7	8
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186
B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

Tal y como hemos señalado anteriormente en el caso de los sonidos, el RCX no espera a acabar el sonido. Así que si utilizas muchas notas seguidas es mejor añadir (un poco más largo) entre ellas ordenes wait. He aquí un ejemplo:

```
task main()
{
  PlayTone(262,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(330,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(262,160); Wait(200);
}
```

Puedes crearlo de un modo muy sencillo utilizando el RCX Piano que ofrece el RCX Command Center.

Si quieres hacer que el RCX toque música mientras circula, es mejor utilizar una tarea para ello. Aquí tienes un ejemplo de un programa muy simplón en el que el RCX avanza y retrocede, generando música continuamente.

```
task musica()
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start musica;
  while(true)
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Resumen

En este capítulo has aprendido a hacer que el RCX genere sonidos y música. También has visto cómo utilizar una tarea por separado para la música.

VIII. Más sobre motores

Hay varias órdenes adicionales para los motores que puedes utilizar para controlarlos más precisamente. En este capítulo vamos a analizarlas.

Parando suavemente

Cuando utilizas la orden `Off()`, el motor se detiene inmediatamente, utilizando el freno. En NQC también es posible detener el motor de un modo más suave, sin utilizar el freno. Para esto se utiliza la orden `Float()`. A veces esto es mejor para la tarea de tu robot. He aquí un ejemplo: primero el robot se para utilizando los frenos, y a continuación sin utilizarlos. Adviértase la diferencia (en realidad la diferencia es muy pequeña para este robot en particular, pero la diferencia puede ser mayor para otros robots).

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);
}
```

Órdenes avanzadas

La orden `OnFwd()` hace dos cosas: conecta el robot y establece la dirección de avance. La orden `OnRev()` también hace dos cosas: conecta el motor y establece la dirección del motor como retroceso. NQC también dispone de ordenes para hacer esto separadamente. Si sólo quieres modificar una de las dos cosas, es más eficiente utilizar órdenes separadas: utiliza menos memoria en el RCX, es más rápido y puede producir movimientos más suaves. Las dos ordenes a utilizar por separado son `SetDirection()`, que establece la dirección (`OUT_FWD`, `OUT_REV` o `OUT_TOGGLE` que invierte la dirección actual), y `SetOutput()`, que establece el modo (`OUT_ON`, `OUT_OFF` o `OUT_FLOAT`). He aquí un sencillo programa que hace que el robot avance, retroceda y avance otra vez.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);
  SetDirection(OUT_A+OUT_C,OUT_FWD);
  SetOutput(OUT_A+OUT_C,OUT_ON);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);
}
```

Adviértase que en el inicio de todo programa los motores tienen establecida la dirección de avance y las velocidades en el nivel 7. así que en el programa anterior las dos primeras líneas no son necesarias.

Hay algunas ordenes más para motores que son atajos para combinaciones de las órdenes anteriores. Esta es la lista completa:

`On('motores')`	Conecta los motores
`Off('motores')`	Desconecta los motores
`Float('motores')`	Desconecta los motores suavemente
`Fwd('motores')`	Establece que los motores avancen (pero no hace que se mueva)
`Rev('motores')`	Establece que los motores retrocedan (pero no hace que se mueva)
`Toggle('motores')`	Invierte la dirección de los motores (avance a retroceso y viceversa)
`OnFwd('motores')`	Establece que los motores avancen y los pone en marcha
`OnRev('motores')`	Establece que los motores retrocedan y los pone en marcha

`OnFor('motores','tics')`	Conecta el motor para un determinado número de tics (tiempo)
`SetOutput('motores','modo')`	Establece el modo de salida (`OUT_ON`, `OUT_OFF` o `OUT_FLOAT`)
`SetDirection('motores','dir')`	Establece la dirección de salida (`OUT_FWD`, `OUT_REV` o `OUT_TOGGLE`)
`SetPower('motores','potencia')`	Establece la potencia de salida (0-9)

Modificando la velocidad del motor

Tal y como probablemente habrás advertido, la modificación de la velocidad de los motores no tiene un gran efecto. La razón es que sobre todo has cambiado el par, no la velocidad. Sólo verás el efecto cuando el motor tenga una pesada carga. E incluso entonces, la diferencia entre 2 y 7 será muy pequeña. Si quieres obtener mejores resultados, el truco es conectar y desconectar el motor en una rápida sucesión. Aquí se ofrece un sencillo programa para hacer esto. Tiene una tarea, denominada motor_en_marcha, que dirige los motores. Constantemente chequea la variable speed para ver cual es la velocidad actual. Un valor positivo es avance, y un valor negativo es retroceso. Fija a los motores la dirección correcta, y espera cierto tiempo, dependiendo de la velocidad, antes de desconectar otra vez los motores. La tarea principal establece simplemente velocidades y esperas.

```
int veloc, __veloc;

task motor_en_marcha ()
{
  while (true)
  {
    __veloc = veloc;
    if (__veloc > 0) {OnFwd(OUT_A+OUT_C);}
    if (__veloc < 0) {OnRev(OUT_A+OUT_C); __veloc = -__veloc;}
    Wait(__veloc);
    Off(OUT_A+OUT_C);
  }
}9

task main()
{
  veloc = 0;
  start motor_en_marcha;
  veloc = 1;   Wait(200);
  veloc = -10; Wait(200);
  veloc = 5;   Wait(200);
  veloc = -2;  Wait(200);
  stop motor_en_marcha;
  Off(OUT_A+OUT_C);
}
```

Este programa puede hacerse mucho más potente, admitiendo rotaciones, e incorporando un tiempo de espera tras la orden `Off()`. Experimenta por tu cuenta.

Resumen

En este capítulo has aprendido las órdenes extra disponibles para motores: `Float()` que detiene el motor suavemente, `SetDirection()` que establece la dirección (`OUT_FWD`, `OUT_REV` o `OUT_TOGGLE` que invierte la dirección activa) y `SetOutput()` que establece el modo (`OUT_ON`, `OUT_OFF` o `OUT_FLOAT`). Has visto la lista completa de órdenes de motores. Has aprendido también un truco para controlar la velocidad del motor de un mejor modo.

IX. Más sobre sensores

En el capítulo V hemos analizado los aspectos básicos del uso de sensores. Pero es mucho más lo que puedes hacer con los sensores. En este capítulo analizaremos las diferencias entre modo de sensor y tipo de sensor, veremos cómo utilizar el sensor de rotación (un tipo de sensor que no suministrado con el RIS pero que puede adquirirse separadamente[4]), y veremos algunos trucos para utilizar más de tres sensores y para hacer un sensor de proximidad.

Modo y tipo de sensor

La orden `SetSensor()` que hemos visto anteriormente hace en realidad dos cosas: establece el tipo de sensor, así como el modo en el que el sensor opera. Estableciendo el modo y tipo de sensor separadamente, puedes controlar el comportamiento del sensor con más precisión, lo cual es útil en ciertas aplicaciones.

El tipo de sensor se establece con la orden `SetSensorType()`. Hay cuatro tipos diferentes: `SENSOR_TYPE_TOUCH`, que es el sensor de contacto, `SENSOR_TYPE_LIGHT`, que es el sensor de luz, `SENSOR_TYPE_TEMPERATURE`, que es el sensor de temperatura (este sensor no forma parte del RIS pero puede ser adquirido separadamente), y `SENSOR_TYPE_ROTATION`, que es el sensor de rotación (tampoco se suministra con el RIS pero puede ser adquirido separadamente). El establecimiento del tipo de sensor es particularmente importante para indicar si el sensor necesita alimentación (tal y como sucede con la luz del sensor de luz). Desconozco si puede resultar útil ajustar un sensor a un ajuste distinto del original.

El modo del sensor se establece con la orden `SetSensorMode()`. Hay ocho modos diferentes. El más importante es `SENSOR_MODE_RAW`. En este modo, el valor que se obtiene cuando se chequea el sensor es un valor entre 0 y 1023. Es el valor sin procesar producido por el sensor. El significado de dicho número depende del tipo de sensor. Por ejemplo, para un sensor de contacto, cuando el sensor no está pulsado el valor se acerca a 1023. Cuando está totalmente pulsado, es cercano a 50. Cuando está parcialmente pulsado el valor está comprendido entre 50 y 1000. Así que si configuras un sensor de contacto en el modo raw puedes realmente detectar si se encuentra parcialmente pulsado. Cuando el sensor es un sensor de luz, el rango de valores es el comprendido entre 300 (mucha luz) y 800 (muy oscuro). Proporciona un valor mucho más exacto que el que obtenemos utilizando la orden `SetSensor()`.

El segundo modo es `SENSOR_MODE_BOOL`. En este modo el valor es 0 ó 1. Cuando el valor sin procesar está por encima de 550 el valor es 0, y en caso contrario es 1. `SENSOR_MODE_BOOL` es el modo por defecto para los sensores de contacto. Los modos `SENSOR_MODE_CELSIUS` y `SENSOR_MODE_FAHRENHEIT` son únicamente útiles con el sensor de temperatura, y proporcionan la temperatura en las unidades deseadas. `SENSOR_MODE_PERCENT` convierte el valor sin procesar en otro comprendido entre 0 y 100. A todo valor sin procesar inferior a 400 le corresponderá el 100%. Si el valor sin procesar es superior el porcentaje desciende paulatinamente hasta el 0. `SENSOR_MODE_PERCENT` es el modo por defecto del sensor de luz. `SENSOR_MODE_ROTATION` parece ser sólo útil con el sensor de rotación (véase más adelante).

Hay otros dos modos de interés: `SENSOR_MODE_EDGE` y `SENSOR_MODE_PULSE`. Ellos cuentan transiciones, es decir, variaciones de un bajo valor a un alto valor de lectura sin procesar o viceversa. Por ejemplo, cuando se pulsa un sensor de contacto, se produce una transición de un alto valor a un bajo valor. Cuando se suelta, se produce una transición en sentido opuesto. Cuando se establece el modo de sensor en `SENSOR_MODE_PULSE`, sólo se contabilizan las transiciones de un valor bajo a uno alto. Así que cada vez que se pulsa y se suelta el sensor de contacto se cuenta uno. Cuando está establecido el modo `SENSOR_MODE_EDGE`, ambas transiciones son contadas. Así que cada vez que se pulsa y se suelta el sensor de contacto se cuenta por dos. Puedes utilizar esto para contar cuantas veces es pulsado un sensor de contacto. O puedes utilizarlo en combinación con un sensor de luz para saber cuantas veces se enciende y se apaga una luz (intensa). Por supuesto que cuando cuentas eventos, has de tener la posibilidad de poner el contador a cero. Para esto se utiliza la orden `ClearSensor()`. Pone a cero el contador para el sensor (o sensores) indicado.

[4]Actualmente puede adquirirse junto a otras piezas y un mando a distancia con el “Ultimate Accesory Set” (Nota del traductor)

Veamos un ejemplo. El programa siguiente utiliza un sensor de contacto para dirigir un robot. Conecta el sensor de contacto a la entrada 1 por medio de un largo cable. Si pulsas rápidamente el sensor dos veces el robot avanzará. Si pulsas una sola vez detendrá el movimiento.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 >0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Adviértase que primero se establece el tipo de sensor y a continuación el modo. Esto es esencial, ya que modificar el tipo tiene consecuencias en el modo.

El sensor de rotación

El sensor de rotación es un tipo de sensor muy útil que desafortunadamente no se suministra con el RIS estándar. Puede ser adquirido separadamente. El sensor de rotación contiene un orificio en el que se puede introducir un eje. El sensor de rotación mide cuanto gira el eje. Una vuelta completa del eje supone 16 pasos (o –16 si la rotación ha sido inversa). El sensor de rotación es muy útil para que el robot haga movimientos controlados con precisión. Puedes hacer que el eje se mueva en la medida exacta que deseas. Si necesitas un control superior a 16 paso por vuelta, siempre puedes utilizar engranajes para conectarlo a un eje que se mueve más rápido, y utilizarlo para contar pasos.

Una aplicación habitual es tener dos sensores de rotación conectados a las dos ruedas del robot que controlas por medio de dos motores. Para un movimiento en línea recta se necesita que las dos ruedas giren a la misma velocidad. Desgraciadamente, los motores generalmente no giran exactamente a la misma velocidad. Utilizando sensores de rotación puedes ver qué rueda gira más rápida. Puedes temporalmente detener ese motor (utilizando `Float()`) hasta que los dos sensores den otra vez el mismo valor. Es lo que hace el siguiente programa. Simplemente hace que el robot circule en línea recta. Para utilizarlo modifica tu robot conectando los dos sensores de rotación a las dos ruedas. Conecta los sensores en las entradas 1 y 3.

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

El programa comienza indicando que los sensores son sensores de rotación, y inicializa sus valores a cero. A continuación comienza un bucle sin fin. En este bucle comprobamos si las lecturas de los dos sensores son iguales. Si es así el robot simplemente continuará hacia adelante. Si una es mayor, el motor correspondiente será detenido hasta que los dos den la misma lectura.

Evidentemente es un programa muy sencillo. Puedes ampliar esto para hacer que el robot recorra una distancia exacta, o para que gire con gran precisión.

Colocación de varios sensores en una entrada

El RCX tiene solo tres entradas, así que solo se pueden conectar tres sensores. Si deseas construir robots más complicados (y has adquirido algunos sensores extras) esto puede no ser suficiente. Afortunadamente, con algunos trucos, puedes conectar dos (o más) sensores en una entrada.

Lo más fácil es conectar dos sensores de contacto en una misma entrada. Si uno de ellos es pulsado (o los dos), el valor de la lectura será 1, en caso contrario 0. No podrás distinguir cuál ha sido, pero a menudo no es necesario. Por ejemplo, cuando le pones a un robot un sensor en la parte delantera y otro en la trasera, sabes cuál ha sido pulsado basándote en la dirección en la que circula. Pero también puedes configurar el modo del sensor como raw (véase en la página 27). Ahora puedes obtener mucha más información. Si tienes suerte, la lectura cuando los sensores están pulsados no serán las mismas para ambos sensores. Si es este el caso, podrás diferenciarlos. Y cuando los dos estén presionados, el valor será mucho menor (alrededor de 30) por lo que podrás detectar esto también.

Puedes también conectar un sensor de contacto y uno de luz en la misma entrada. Establece el tipo como luz (sino el sensor de luz no funcionará). Establece el modo como raw. De este modo, cuando el sensor de contacto está pulsado recibes un valor inferior a 100. Si no está pulsado el valor que se recibe es el correspondiente al sensor de luz que no es nunca inferior a 100. El siguiente programa utiliza dicha idea. El robot ha de ser equipado por un sensor de luz que apunte hacia abajo, y un parachoques en el frente conectado a un sensor de contacto. Conecta los dos en la entrada 1. El robot circulará aleatoriamente en una zona iluminada. Cuando el sensor de luz encuentre una línea oscura (valor raw > 750) retrocederá un poco. Cuando el sensor de contacto toque algo (valor raw inferior a 100) hará lo mismo. Este es el programa:

```
int ttt,tt2;

task mueve_aleat()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start mueve_aleat;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      stop mueve_aleat;
      OnRev(OUT_A+OUT_C);Wait(30);
      start mueve_aleat;
    }
  }
}
```

Creo que el programa está claro. Tiene dos tareas. La tarea `mueve_aleat` hace que el robot se desplace de modo aleatorio. La tarea principal inicializa la tarea `mueve_aleat`, configura el sensor y espera a que algo pase. Si el sensor hace una lectura demasiado baja (contacto) o demasiado alta (fuera del área blanca) se detienen los movimientos aleatorios, retrocede un poco y comienza otra vez sus movimientos aleatorios.

Es también posible conectar dos sensores de luz en la misma entrada. El valor raw está relacionado de algún modo con la combinación de las medidas de luz recogidas por los dos sensores. Pero es algo confuso y parece difícil utilizar. No parece muy útil conectar otros sensores con sensores de temperatura y de rotación.

Elaboración de un sensor de proximidad

Utilizando los sensores de contacto, tu robot puede reaccionar cuando choca contra algo. Pero sería más simpático que el robot reaccionase justo antes de chocar contra algo. Para ello debería saber que se encuentra cerca de algún obstáculo. Desgraciadamente no hay sensores disponibles para ello[5]. Hay un truco que podemos utilizar para esto. El RCX tiene un puerto de infrarrojos con el que puede comunicarse con el ordenador, o con otros robots (veremos más sobre comunicaciones entre robots en el capítulo XI). Resulta que el sensor de luz que viene con el robot es muy sensible a la luz infrarroja. Podemos construir un sensor de proximidad basándonos en ello. La idea es que una tarea envía mensajes en infrarrojos. Otra tarea mide las fluctuaciones de la intensidad de luz que reflejan los objetos. A mayor fluctuación, estaremos más cerca del objeto.

Para poner en práctica esta idea, coloca el sensor de luz encima del Puerto de infrarrojos del robot, apuntando hacia delante. De este modo sólo mide luz infrarroja reflejada. Conéctalo a la entrada 2. Utilizaremos el modo raw para el sensor de luz para apreciar las fluctuaciones lo mejor posible. Aquí hay un sencillo programa que hace que el robot avance hasta que esté cerca de un objeto, y entonces, gire 90º a la derecha.

```
int lastlevel;             // Almacena el nivel previo

task send_signal()
{
  while(true)
    {SendMessage(0); Wait(10);}
}

task check_signal()
{
  while(true)
  {
    lastlevel = SENSOR_2;
    if(SENSOR_2 > lastlevel + 200)
      {OnRev(OUT_C); Wait(85); OnFwd(OUT_A+OUT_C);}
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start send_signal;
  start check_signal;
}
```

La tarea `send_signal` envía 10 señales IR por segundo, utilizando la orden `SendMessage(0)`. La tarea `check_signal` almacena repetidamente el valor del sensor de luz. Además comprueba (un poco después) si se ha convertido en un valor por lo menos 200 unidades superior, indicador de una gran fluctuación. Si es así, el robot gira 90º a la derecha El valor de 200 es bastante arbitrario. Si lo haces más pequeño, el robot gira más lejos de los obstáculos. Si eliges un valor superior, lo hará más cerca de ellos. Pero esto depende del tipo de material y de la cantidad de luz disponible en la habitación. Deberías experimentar o utilizar algunos mecanismos más ingeniosos para determinar el valor correcto.

Una desventaja de esta técnica es que solo trabaja en una dirección. Probablemente todavía necesitarás sensores de contacto en los laterales para evitar las colisiones. Pero la técnica es muy útil para robots que han de circular en laberintos. Otra desventaja es que no puedes comunicarte desde el ordenador con el robot por que interferiría con las órdenes enviadas por el robot (también el mando a distancia del televisor puede no funcionar).

[5] Hay disponible un sensor de proximidad en LegoStuff (Nota del traductor)

Resumen

En este capítulo hemos visto varias cuestiones adicionales sobre sensores. Hemos visto cómo configurar separadamente el tipo y el modo del sensor y cómo podría ser usado para obtener información adicional. Hemos aprendido cómo utilizar el sensor de rotación. Y hemos visto cómo pueden ser conectados múltiples sensores en una sola entrada del RCX. Para acabar, hemos visto un truco para usar la conexión infrarroja del robot en combinación con el sensor de luz para crear un sensor de proximidad. Todos estos trucos son muy útiles cuando se quieren construir robots más complicados. Los sensores siempre jugarán un papel crucial aquí.

X. Tareas paralelas

Tal y como se ha indicado anteriormente, las tareas en NQC son ejecutadas simultáneamente, o en paralelo como se dice comúnmente. Esto es muy útil. Permite observar sensores desde una tarea mientras que otra hace que el robot avance, y otra más que toque música. Pero las tareas en paralelo también pueden crear problemas. Una tarea puede interferir con otra.

Un programa incorrecto

Tomemos el siguiente programa. Aquí una tarea guía el robot para que circule describiendo cuadrados (como hemos hecho anteriormente) y una segunda tarea chequea el sensor de contacto. Cuando el sensor es presionado, retrocede un poco y gira 90

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C);
      Wait(50);
      OnFwd(OUT_A);
      Wait(85);
      OnFwd(OUT_C);
    }
  }
}
```

Seguramente parece un programa perfecto. Pero si lo ejecutas probablemente encontrarás un comportamiento inesperado. Prueba lo siguiente: haz que el robot toque algo mientras gira. Comenzará a retroceder, pero inmediatamente avanzará otra vez chocando con el obstáculo. La razón para esto es que las tareas se interfieren entre sí. Sucede lo siguiente: el robot está girando a la derecha, es decir, la primera tarea se encuentra en la segunda instrucción de espera; ahora el robot hace chocar el sensor; comienza a retroceder, pero en ese mismo instante, la tarea principal está lista y hace avanzar el robot otra vez, hasta el obstáculo. La segunda tarea está durmiente (en espera) en ese momento por lo que no se entera de la colisión. Este evidentemente no es el comportamiento que desearíamos ver. El problema es que, mientras la segunda tarea está durmiente, no nos hemos dado cuenta que la primera tarea estaba todavía en marcha, y que sus acciones interfieren con las acciones de la segunda tarea.

Parada y reinicio de tareas

Un modo de resolver este problema es asegurarse que en todo momento solo una tarea guiará el robot. Esta es la propuesta que hemos visto en el capítulo VI. Permíteme repetir el programa aquí:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start chequea_sensores;
  start mueve_en_cuadrados;
}

task mueve_en_cuadrados ()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task chequea_sensores ()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop mueve_en_cuadrados;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start mueve_en_cuadrados;
    }
  }
}
```

El meollo de la cuestión está en que la tarea `chequea_sensores` solo mueve el robot tras haber detenido la tarea `mueve_en_cuadrados`. Así que esta tarea no podrá interferir con el movimiento de alejamiento del obstáculo. Una vez que el procedimiento de marcha atrás ha acabado, se inicia otra vez la tarea `mueve_en_cuadrados`.

A pesar de que esta es una buena solución para el problema anterior, aquí se presenta un problema. Cuando reiniciamos `mueve_en_cuadrados`, arranca otra vez desde el principio. Esto no es importante en nuestra pequeña tarea, pero a menudo es un comportamiento no deseado. Preferiríamos detener la tarea donde sea, y continuar después en el mismo punto. Desgraciadamente esto no es fácil de hacer.

Uso de semáforos

Una técnica estándar para resolver este programa es utilizar una variable que indica qué tarea está controlando el motor. Las otras tareas no estarán autorizadas para guiar los motores hasta que la primera indique, utilizando el semáforo, que está preparada. Semejante variable es a menudo denominada semáforo. Consideremos que `sem` es un semáforo. Supongamos que el valor 0 indica que ninguna tarea está dirigiendo el motor. Ahora, cuando una tarea quiera hacer algo con los motores se ejecutarán las siguientes órdenes:

```
until (sem == 0);
sem = 1;
// Hacer algo con los motores
sem = 0;
```

Así que primero esperaremos hasta que nadie necesite los motores. Entonces solicitaremos el control dando el valor 1 a `sem`. Ahora podemos controlar los motores. Cuando lo hayamos hecho devolveremos el valor 0 a la variable `sem`. A continuación puedes encontrar el programa anterior, en el que se ha implementado el uso de un semáforo. Cuando el sensor de contacto toca algo, el semáforo se activa y el procedimiento de marcha atrás se ejecuta. Durante la ejecución de este procedimiento la tarea `mueve_en_cuadrados` permanece a la espera. En el momento en la marcha atrás se ha completado, el semáforo vuelve a 0 y `mueve_en_cuadrados` continua.

```
int sem;

task main()
{
  sem = 0;
  start mueve_en_cuadrados;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      sem = 0;
    }
  }
}

task mueve_en_cuadrados ()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_A+OUT_C);
    sem = 0;
    Wait(100);
    until (sem == 0); sem = 1;
    OnRev(OUT_C);
    sem = 0;
    Wait(85);
  }
}
```

Podrías sostener que no es necesario en `mueve_en_cuadrados` dar el valor 1 al semáforo y devolverlo a 0. Sin embargo es muy útil. La razón es que la orden `OnFwd()` consta en realidad de dos órdenes (véase el capítulo IX). Y no deseas que esta secuencia de ordenes sea interrumpida por otra tarea.

Los semáforos son muy útiles y, cuando escribes un programa complicado con tareas paralelas, casi siempre son necesarios (hay todavía una pequeña posibilidad de que puedan fallar, intenta imaginarte porqué).

Resumen

En este capítulo hemos estudiado algunos de los problemas que pueden surgir cuando utilizas distintas tareas. Siempre hay que tener mucho cuidado con los efectos secundarios. Muchos efectos secundarios son debidos a esto. Hemos visto dos modos de resolver tales problemas. La primera solución detiene y reinicia tareas para asegurarse que una sola tarea crítica se ejecuta en cada momento. El segundo enfoque utiliza semáforos para controlar la ejecución de las tareas. Esto garantiza que en cada momento una sola tarea será ejecutada.

XI. Comunicación entre robots

Si tienes más de un RCX este capítulo es para ti. Los robots pueden comunicarse entre sí a través del puerto de infrarrojos. Utilizando esto, puedes hacer que varios robots colaboren (o que peleen entre ellos). También podrás construir un gran robot utilizando dos RCX de tal manera que podrás tener seis motores y seis sensores (o más utilizando los trucos del capítulo IX).

La comunicación entre robots funciona, en general, como sigue. Un robot puede utilizar la orden `SendMessage()` para enviar un valor (0-255) por medio del puerto de infrarrojos. El resto de los robots reciben el mensaje y lo almacenan. El programa de un robot puede buscar el valor del último mensaje recibido utilizando `Message()`. Basándose en este valor el programa puede hacer que el robot ejecute ciertas acciones.

Transmisión de órdenes

A menudo, cuando tienes dos o más robots, uno es el lider. Podemos llamarlo el *patrón*. Los otros robots son los *esclavos*. El robot patrón envían órdenes a los esclavos, y estos las ejecutan. A veces los esclavos pueden devolver información al *patrón*, por ejemplo el valor devuelto por un sensor. Así que tienes que escribir dos programas, uno para el patrón, y otro(s) para el esclavo(s). De ahora en adelante supondremos que tenemos un esclavo. Comenzemos con un sencillo programa. Aquí el esclavo puede ejecutar tres diferentes órdenes: avanzar, retroceder y detenerse. El programa consiste en un sencillo bucle. En este bucle establece el valor del mensaje actual como 0 utilizando la orden `ClearMessage`(). A continuación espera hasta que el mensaje se convierte en otro diferente de 0. Teniendo en cuenta ese valor ejecuta una de las tres órdenes.

```
task main()              // ESCLAVO
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

El patrón tiene un programa más sencillo todavía. Simplemente envía los mensajes correspondientes a las órdenes y espera un poco. En el programa siguiente ordena al esclavo avanzar, tras dos segundos retroceder, y tras otros dos segundos, detenerse.

```
task main()              // PATRÓN
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

Una vez que hayas escrito estos dos programas, necesitas transferirlos a los robots. Cada programa debe ir a uno de los robots. Asegúrate que desconectas el otro durante la transferencia (véanse más adelante las precauciones requeridas). Ahora conecta ambos robots e inicia los programas: primero el del esclavo, y después el del patrón.

Si tienes múltiples esclavos, tienes que transferir los programas a los esclavos por turnos (no simultáneamente, véase más adelante). Ahora todos los esclavos ejecutarán exactamente la misma acción

Para que los robots se comuniquen entre sí, hemos definido lo que se llama un protocolo. Decidimos que 1 significa avanzar, 2 retroceder y 3 detenerse. Es muy importante definir cuidadosamente tales protocolos, especialmente cuando tratamos con muchas comunicaciones. Por ejemplo, cuando hay más esclavos, puedes definir un protocolo en el que se envían dos números (con un pequeño espacio entre ellos): el primer número es el número del esclavo, y el segundo el de la orden actual. El esclavo primero chequea el número y solo ejecuta la acción si es su número (esto requiere que cada esclavo tenga un número propio, lo cual se puede conseguir

haciendo que cada esclavo tenga un programa ligeramente distinto, en el que, por ejemplo, una constante sea diferente).

Selección del líder

Como acabamos de ver, cuando se trata con múltiples robots, cada robot debe tener su propio programa. Sería mucho más fácil si pudiéramos transferir el mismo programa a todos los robots. Pero entonces, la cuestión sería: ¿quién es el patrón? La respuesta es sencilla: dejemos que los robots decidan entre ellos. Dejémosles elegir un líder al que los otros sigan. Pero, ¿cómo podemos hacer esto? La idea es muy sencilla. Dejamos que cada robot espere un periodo de tiempo aleatorio y que a continuación envíe un mensaje. El primero en enviar el mensaje será el líder. Esta estrategia puede fallar si dos robots esperan exactamente el mismo periodo de tiempo aunque es bastante improbable (puedes diseñar estrategias más complicadas que detecten esto e intenten una segunda elección en semejante caso). He aquí el programa que lo hace:

```
task main()
{
  ClearMessage();
  Wait(200);                // Asegurarse que todos están en marcha
  Wait(Random(400));         // espera entre 0 y 4 segundos
  if (Message() > 0)         // comprueba si es el primero
  {
    start esclavo;
  }
  else
  {
    SendMessage(1);          // Yo ahora soy el patrón
    Wait(400);              // asegurarse que el resto lo sabe
    start patron;
  }
}

task patron ()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task esclavo()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Transfiere este programa a todos los robots (uno por uno, no a la vez, véase más adelante). Arrancar los robots simultáneamente y observar lo que pasa. Uno de ellos tomará el mando y el otro(s) seguirá las ordenes. En raras ocasiones ninguno de ellos se convierte en líder. Tal y como hemos indicado anteriormente, esto requiere protocolos más elaborados para resolverlo.

Precauciones

Has de tener mucho cuidado cuando trates con múltiples robots. Hay dos problemas: si dos robots (o un robot y el ordenador) envían información a la vez, esta puede perderse. El segundo problema se produce cuando el ordenador envía un programa a múltiples robots a la vez.

Centremonos en el segundo problema. Cuando transfieres el programa al robot, el robot comunica al ordenador si ha recibido correctamente (las partes de) el programa. El ordenador reacciona enviando nuevas partes o reeenviandolas. Cuando hay dos robots encendidos, los dos comenzarán a comunicar al ordenador si han recibido correctamente el programa. El ordenador no comprenderá esto (¡él no sabe que hay dos robots!). en

consecuencia, las cosas irán mal y el programa se corromperá. El programa no hará lo correcto. *Asegurarse siempre que mientras se transfieren los prgramas sólo hay un robot encendido*.

El otro problema es que el robot puede enviar un mensaje en cualquier momento. Si dos mensajes son enviados aproximadamente en el mismo momento, pueden perderse. Así mismo, un robot no puede enviar y recibir mensajes a la vez. Esto no es un problema cuando un solo robot envía mensajes (sólo hay un patrón) pero en caso contrario puede producirse un serio problema. Por ejemplo, imagínate escribir un programa en el que el esclavo envía un mensaje cuando choca contra algo, de tal manera que el patrón puede entrar en acción. Pero si el patrón envía una orden en el mismo momento, el mensaje puede perderse. Para resolver esto, es importante definir tu protocolo de comunicaciones de tal modo que en el caso en que la comunicación falle esto sea corregido. Por ejemplo, cuando el patrón envía una orden, debería recibir una respuesta del esclavo. Si no recibe la respuesta lo suficientemente rápido, reenviará la orden. Esto sucedería en un fragmento de código semejante al siguiente:

```
do
{
  SendMessage(1);
  ClearMessage();
  Wait(10);
}
while (Message() != 255);
```

Aquí 255 es utilizado para la contestación..

A veces, cuando tratas con múltiples robots, puedes querer que sólo un robot que esté muy cerca reciba la señal. Esto se puede conseguir añadiendo la orden `SetTxPower(TX_POWER_LO)` al programa del patrón. En ese caso la señal IR enviada es muy débil y sólo un robot que se encuentre cerca y de frente podrá “oir” al patrón. Esto es particularmente útil cuando se construye un gran robot con más de 2 RCX .Utiliza `SetTxPower(TX_POWER_HI)` para configurar otra vez el robot en el modo de transmisión de largo alcance.

Resumen

En este capítulo hemos estudiado algunos de los aspectos básicos de la comunicación entre robots. El RCX utiliza órdenes para enviar, borrar y chequear mensajes. Hemos visto que es importante definir cómo funcionan las comunicaciones por medio de un protocolo. Tal protocolo juega un papel crucial en la comunicación entre ordenadores. Hemos visto también que hay varias restricciones en la comunicación entre robots lo que convierte en más importante definir buenos protocolos.

XII. Más ordenes

NQC tiene varias órdenes adicionales. En este capítulo analizaremos tres tipos: el uso de temporizadores, órdenes para el control del display y el uso de la característica de registro de datos del RCX.

Temporizadores

El RCX tiene cuatro temporizadores internos. Estos temporizadores hacen tictac en incrementos de 1/10 de segundo. Los temporizadores están numerados del 0 al 3. Puedes reinicializar el valor de los temporizadores con la orden `ClearTimer()` y establecer el valor actual del temporizador con `Timer()`. Aquí se presenta un ejemplo del uso del temporizador. Este programa hace que el robot circule de un modo aleatorio durante 20 segundos.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

Puedes comparar este programa con uno del capítulo IV que realizaba exactamente la misma tarea. El que utiliza los temporizadores es indudablemente más sencillo.

Los temporizadores son muy útiles para reemplazar la orden `Wait()`. Puedes hacer una espera de una determinada cantidad de tiempo reinicializando el temporizador y esperando hasta que alcance un determinado valor. Pero puedes también reaccionar a otros eventos (por ejemplo, de sensores) mientras esperas. El sencillo programa siguiente es un ejemplo de esto. Permite que el robot circule hasta que pasen 10 segundos, o el sensor de contacto toque algo.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100));
  Off(OUT_A+OUT_C);
}
```

No olvides que los temporizadores trabajan en tics de 1/10 de segundo, mientras, por ejemplo, la orden wait utiliza tics de 1/100 de segundo.

La pantalla

Es posible controlar la pantalla del RCX de dos diferentes maneras. En la primera de ellas, puedes indicar que muestre el reloj del sistema, uno de los sensores, o uno de los motores. Esto es el equivalente de utilizar el botón negro (view) del RCX. Para establecer el tipo de pantalla, se utiliza la orden `SelectDisplay()`. El programa siguiente muestra las siete posibilidades, una detrás de otra.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Entrada 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Entrada 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Entrada 3
  SelectDisplay(DISPLAY_OUT_A);    Wait(100);  // Salida A
  SelectDisplay(DISPLAY_OUT_B);    Wait(100);  // Salida B
  SelectDisplay(DISPLAY_OUT_C);    Wait(100);  // Salida C
  SelectDisplay(DISPLAY_WATCH);    Wait(100);  // Reloj del sistema
}
```

Advierte que no debes utilizar `SelectDisplay(SENSOR_1)`.

El segundo modo por el cual puedes controlar la pantalla es controlando el valor del reloj del sistema. Puedes utilizarlo para mostrar, por ejemplo, la información de diagnóstico. Para ello se utiliza la orden `SetWatch()`. Este es un diminuto programa que lo utiliza:

```
task main()
{
  SetWatch(1,1); Wait(100);
  SetWatch(2,4); Wait(100);
  SetWatch(3,9); Wait(100);
  SetWatch(4,16); Wait(100);
  SetWatch(5,25); Wait(100);
}
```

Adviertase que los argumentos de `SetWatch()` han de ser constantes.

Registro de datos

El RCX puede almacenar valores de variables, lecturas de sensores, y temporizadores, en un espacio de memoria llamado datalog. Los valores contenidos en el datalog no pueden ser utilizados en el interior del RCX, pero pueden ser leidos desde el ordenador. Esto es útil para por ejemplo chequear que ocurre en tu robot. El RCX Command Center tiene una ventana especial en la que puedes ver el contenido actual del datalog.

El uso del registro de datos (datalog) consiste en tres pasos: primero el programa NQC debe definir el tamaño del datalog utilizando la orden `CreateDatalog()`. Esto también borra el contenido actual del datalog. A continuación, los valores pueden ser escritos uno tras otro (si miras en la pantalla del RCX veras una tras otra aparecer las cuatro partes de un disco; cuando el disco se completa, el datalog está completo). Si se alcanza el final del datalog, no pasa nada. Los nuevos valores no serán almacenados. El tercer paso es transferir el datalog al PC. Para esto, seleccionar en el RCX Command Center la orden **Datalog** en el menú **Tools** (Herramientas). A continuación pulsar el botón denominado **Upload Datalog**, y aparecerán todos los valores. Puedes verlos o guardarlos en un archivo para hacer algo con ellos. Hay quien ha utilizado esta característica para hacer, por ejemplo, un escáner con el RCX.

He aquí un pequeño ejemplo de un robot con un sensor de luz. El robot circula por 10 segundos, y cinco veces por segundo el valor del sensor de luz es almacenado en el datalog.

```
task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  CreateDatalog(50);
  repeat (50)
  {
    AddToDatalog(SENSOR_2);
    Wait(20);
  }
  Off(OUT_A+OUT_C);
}
```

XIII. Referencia rápida NQC

A continuación encontrarás la sintaxis de todas las instrucciones, órdenes, constantes, etc. La mayoría de ellas han sido tratadas en los capítulos anteriores, así que sólo se dan escuetas descripciones.

Instrucciones

<table>
<tr><th>Instrucción</th><th>Descripción</th></tr>
<tr><td>while (cond) cuerpo</td><td>Ejecuta el cuerpo cero o más veces mientras la condición es verdadera</td></tr>
<tr><td>do cuerpo while (cond)</td><td>Ejecuta el cuerpo cero o más veces mientras la condición es verdadera</td></tr>
<tr><td>until (cond) cuerpo</td><td>Ejecuta el cuerpo cero o más veces hasta que la condición sea verdadera</td></tr>
<tr><td>break</td><td>Salir del cuerpo de un while/do/until</td></tr>
<tr><td>continue</td><td>Saltarse la siguiente iteración del cuerpo de un while/do/until</td></tr>
<tr><td>repeat (expresión) cuerpo</td><td>Repetir el cuerpo un número determinado de veces</td></tr>
<tr><td>if (cond) inst1
if (cond) inst1 else inst2</td><td>Ejecuta inst1 si la condición es verdadera. Ejecuta inst2 (si existe) si la condición no se cumple.</td></tr>
<tr><td>start nombre_de_tarea</td><td>Inicia la tarea especificada</td></tr>
<tr><td>stop nombre_de_tarea</td><td>Detiene la tarea especificada</td></tr>
<tr><td>function(args)</td><td>Llama una función utilizando los correspondientes argumentos</td></tr>
<tr><td>var = expresión</td><td>Evalúa una expresión y la asigna a una variable</td></tr>
<tr><td>var += expresión</td><td>Evalúa una expresión y la suma a una variable</td></tr>
<tr><td>var -= expresión</td><td>Evalúa una expresión y la resta de una variable</td></tr>
<tr><td>var *= expresión</td><td>Evalúa una expresión y la multiplica por una variable</td></tr>
<tr><td>var /= expresión</td><td>Evalúa una expresión y divide la variable por ella</td></tr>
<tr><td>var |= expresión</td><td>Evalúa una expresión y realiza una comparación OR bit a bit con la variable dejando el resultado en esta última</td></tr>
<tr><td>var &= expresión</td><td>Evalúa una expresión y realiza una comparación AND bit a bit con la variable dejando el resultado en esta última</td></tr>
<tr><td>return</td><td>Devuelve el control de la función al llamador</td></tr>
<tr><td>expresión</td><td>Expresión a evaluar</td></tr>
</table>

Condiciones

Las condiciones son utilizadas en las instrucciones de control para tomar decisiones. En la mayoría de los casos, la condición implica una comparación entre expresiones.

<table>
<tr><th>Condición</th><th>Significado</th></tr>
<tr><td>true</td><td>Siempre verdadero</td></tr>
<tr><td>false</td><td>Siempre falso</td></tr>
<tr><td>expr1 == expr2</td><td>Comprueba si las expresiones son iguales</td></tr>
<tr><td>expr1 != expr2</td><td>Comprueba si las expresiones son diferentes</td></tr>
<tr><td>expr1 < expr2</td><td>Comprueba si una expresiones es inferior a la otra</td></tr>
<tr><td>expr1 <= expr2</td><td>Comprueba si una expresiones es inferior o igual a la otra</td></tr>
<tr><td>expr1 > expr2</td><td>Comprueba si una expresiones es superior a la otra</td></tr>
<tr><td>expr1 >= expr2</td><td>Comprueba si una expresiones es superior o igual a la otra</td></tr>
<tr><td>! condition</td><td>Negación lógica de la condición</td></tr>
<tr><td>cond1 && cond2</td><td>AND lógico entre dos condiciones (verdadero si y sólo si las dos condiciones son verdaderas)</td></tr>
<tr><td>cond1 || cond2</td><td>OR lógico entre dos condiciones (verdadero si y sólo si al menos una de las condiciones es verdadera)</td></tr>
</table>

Expresiones

Hay varios diferentes valores que pueden ser utilizados en las expresiones, incluyendo constantes, variables y valores de sensores. Adviértase que SENSOR_1, SENSOR_2, y SENSOR_3 son macros que se amplían a SensorValue(0), SensorValue(1), y SensorValue(2) respectivamente.

Valor	Descripción
número	Un valor constante (p.ej. "123")
variable	Un nombre de variable (p.ej. "x")
`Timer(`*n*`)`	Valor del temporizador n, donde n está comprendido entre 0 y 3
`Random(`*n*`)`	Número aleatorio entre 0 y n
`SensorValue(`*n*`)`	Valor actual del sensor n, donde n está comprendido entre 0 y 2
`Watch()`	Valor del reloj del sistema
`Message()`	Valor del ultimo mensaje IR recibido

Los valores pueden ser combinados utilizando operadores. Varios operadores sólo pueden ser utilizados para evaluar expresiones constantes, lo cual quiere decir, que sus operandos han de ser o bien constantes, o bien expresiones que incluyan exclusivamente constantes. Los operadores se listan a continuación por orden de prioridad (de mayor a menor).

Operador	Descripción	Asociatividad	Restricción	Ejemplo
abs() **sign**()	Valor absoluto Signo del operando	n/a n/a		abs(x) sign(x)
++ --	Incremento Decremento	izquierda izquierda	Sólo variables Sólo variables	x++ o ++x x-- o --x
- ~	Unary minus Negación a nivel de bit (unary)	derecha derecha	Solo constantes	-x ~123
* / %	Multiplicación División Módulo	izquierda izquierda izquierda	Sólo constantes	x * y x / y 123 % 4
+ -	Adición Substracción	izquierda izquierda		x + y x - y
<<	Desplazamiento de bits a la izquierda	izquierda	Sólo constantes	123 << 4
>>	Desplazamiento de bits a la derecha	izquierda	Sólo constantes	123 >> 4
&	AND a nivel de bit	izquierda		x & y
^	XOR a nivel de bit	izquierda	Sólo constantes	123 ^ 4
|	OR a nivel de bit	izquierda		x | y
&&	AND lógico	izquierda	Sólo constantes	123 && 4
||	OR lógico	izquierda	Sólo constantes	123 || 4

Funciones RCX

La mayoría de las funciones requieren que todos los argumentos sean expresiones constantes (números u operaciones que comprenden otras expresiones constantes). Las excepciones son funciones que utilizan un sensor como argumento, y aquellas que pueden utilizar cualquier expresión. En el caso de los sensores, el argumento debería ser un nombre de sensor: SENSOR_1, SENSOR_2, o SENSOR_3. En algunos casos hay nombres predefinidos (por ejemplo, SENSOR_TOUCH) para ciertas constantes.

Función	Descripción	Ejemplo
`SetSensor(`*sensor, config*`)`	Configura un sensor.	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(`*sensor, modo*`)`	Modo de configuración de sensor	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)`
`SetSensorType(`*sensor, tipo*`)`	Tipo de configuración de sensor	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`

Función	Descripción	Ejemplo
ClearSensor(*sensor*)	Borra el valor de un sensor	ClearSensor(SENSOR_3)
On(*salidas*)	Conecta una o más salidas	On(OUT_A + OUT_B)
Off(*salidas*)	Desconecta una o más salidas	Off(OUT_C)
Float(*salidas*)	Detiene las salidas sin freno.	Float(OUT_B)
Fwd(*salidas*)	Establece las salidas para dirección de avance	Fwd(OUT_A)
Rev(*salidas*)	Establece las salidas para dirección de retroceso	Rev(OUT_B)
Toggle(*salidas*)	Invierte la dirección de las salidas	Toggle(OUT_C)
OnFwd(*salidas*)	Arranca con dirección de avance	OnFwd(OUT_A)
OnRev(*salidas*)	Arranca con dirección de retroceso	OnRev(OUT_B)
OnFor(*salidas, tiempo*)	Arranca por un determinado número de 1/100 de segundo. El tiempo puede ser una expresión.	OnFor(OUT_A, 200)
SetOutput(*salidas, modo*)	Configura el modo de salida	SetOutput(OUT_A, OUT_ON)
SetDirection(*salidas, dir*)	Configura la dirección de salida	SetDirection(OUT_A, OUT_FWD)
SetPower(*salidas,potencia*)	Configura el nivel de potencia de salida (0-7). La potencia puede ser una expresión.	SetPower(OUT_A, 6)
Wait(*time*)	Espera durante el tiempo especificado en 1/100 de segundo. El tiempo puede ser una expresión.	Wait(x)
PlaySound(*sonido*)	Genera el sonido especificado (0-5).	PlaySound(SOUND_CLICK)
PlayTone(*frec, duración*)	Toca un tono de una especificada frecuencia durante el tiempo especificado (en 1/10 de segundo)	PlayTone(440, 5)
ClearTimer(*temporizador*)	Reinicia el temporizador (0-3) al valor 0	ClearTimer(0)
StopAllTasks()	Detiene todas las tareas que se estén ejecutando	StopAllTasks()
SelectDisplay(*modo*)	Selecciona uno de los 7 modos de display: 0: reloj del sistema, 1-3: valor del sensor, 4-6: configuraciones de salidas. Modo puede ser una expresión.	SelectDisplay(1)
SendMessage(*mensaje*)	Envía un mensaje IR (1-255). Mensaje puede ser una expresión.	SendMessage(x)
ClearMessage()	Borra el buffer de mensajes IR	ClearMessage()
CreateDatalog(*tamaño*)	Crea un Nuevo registro de datos del tamaño dado.	CreateDatalog(100)

Función	Descripción	Ejemplo
AddToDatalog(*value*)	Añade un valor al registro de datos. El valor puede ser una expresión.	AddToDatalog(Timer(0))
SetWatch(*horas*, *minutos*)	Establece el valor del reloj del sistema.	SetWatch(1,30)
SetTxPower(*hi_lo*)	Configura la potencia del transmisor de infrarrojos en alta o baja potencia.	SetTxPower(TX_POWER_LO)

Constantes RCX

Muchos de los valores para las funciones del RCX se denominan constantes y ayudan a que el código sea más legible. Siempre que sea posible, es preferible utilizar una constante nombrada a un valor en bruto.

Configuraciones de sensor para SetSensor()	SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE
Modos para SetSensorMode()	SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION
Tipos para SetSensorType()	SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION
Salidas para On(), Off(), etc.	OUT_A, OUT_B, OUT_C
Modos para SetOutput()	OUT_ON, OUT_OFF, OUT_FLOAT
Direcciones para SetDirection()	OUT_FWD, OUT_REV, OUT_TOGGLE
Potencia de salida para SetPower()	OUT_LOW, OUT_HALF, OUT_FULL
Sonidos para PlaySound()	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP
Modos para SelectDisplay()	DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C
Nivel de potencia Tx para SetTxPower()	TX_POWER_LO, TX_POWER_HI

Palabras clave

Palabras clave son algunas palabras reservadas por el compilador de NQC para sí mismo. Es un error utilizar cualquiera de estas como nombre de funciones, tareas o variables. Las palabras reservadas son las siguientes: **__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while.**

XIV. Notas finales

Si has trabajado por tu cuenta mediante este tutorial puedes considerarte un experto en NQC. Si no lo has hecho hasta ahora, este es el momento para comenzar a experimentar por tu cuenta. Siendo creativo en el diseño y la programación, puedes conseguir que los robots hagan cosas sorprendentes.

Este tutorial no cubre todos los aspectos del RCX Command Center. Te recomiendo que leas la documentación en algunos de sus apartados. El NQC está todavía en desarrollo. Futuras versiones pueden incorporar funcionalidades adicionales. Muchos conceptos de programación no han sido tratados en este tutorial. En particular, no hemos tomado en cuenta comportamientos de aprendizaje de robots ni otros aspectos de inteligencia artificial.

Es también posible guiar el robot Lego directamente desde el PC. Esto requiere que escribas un programa en un lenguaje tal y como Visual Basic, Java o Delphi. También es posible que semejante programa trabaje conjuntamente con un programa NCQ en el propio RCX. Este tipo de combinación es muy poderosa. Si estás interesado en este modo de programar el robot, es mejor que empieces por bajar la referencia técnica de spirit del web site Lego MindStorms.

http://www.legomindstorms.com/

La web es una fuente perfecta de información adicional. Otros importantes puntos de partida se encuentran en la página de enlaces de mi propio Web site:

http://www.cs.uu.nl/people/markov/lego/

y LUGNET, el LEGO® Users Group Network (no oficial):

http://www.lugnet.com/

Puedes también encontrar mucha información en los foros lugnet.robotics y lugnet.robotics.rcx.nqc en lugnet.com.

XV. Nota del traductor

Mi nombre es Koldo Olaskoaga (Donostia, Euskadi). He traducido este documento como parte de un proyecto de uso de la robótica y el sistema LEGO MindStorms como herramientas de aprendizaje.

Si haces un uso educativo de este documento, agradecería recibir información al respecto.

Mi dirección electrónica es robotek@donospat.net y mi página Web se encuentra en este momento en http://www.donospat.net/2000/ (mi página personal en http://www.euskalnet.net/kolaskoaga/)