

Edumóvil



**Programando
NXT con NXc**

M.C. Gabriel Gerónimo Castillo
Universidad Tecnológica de la Mixteca
Laboratorio EDUMÓVIL
gcgero@gmail.com



Lego NXT

El ladrillo Inteligente NXT, es el cerebro del robot. Es una pequeña computadora que se programa para crear movimiento.

Contiene una pequeña pantalla de despliegue y un conjunto de botones para comunicarse con los motores y sensores.

La descarga de los programas se pueden realizar vía cable USB o Bluetooth.



Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Sensores

Ultrasonico. Mide la distancia de un objeto u obstáculo.

Tacto. Detecta cuando se presiona el botón de enfrente del sensor.

Luz. Mide la luminosidad del brillo de la luz en la parte frontal del sensor. Con esto puede distinguir entre blanco, negro y tonos de gris.

Color. Puede determinar el color de objetos.

Sonido. Mide el nivel de sonido cercano al robot.



Cada motor de rotación tiene empotrado un sensor para medir la distancia de movimiento del motor.

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Puertos

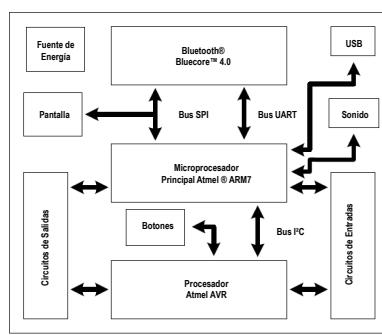


Los puertos de **salida** son **A, B y C** (parte superior) usados para conectar los motores.

Los puertos de **entrada** son el **1, 2, 3 y 4** (para inferior) usados para conectar los sensores.

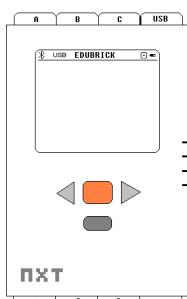
Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Vistazo interno



Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Características Técnicas



Microcontrolador de 32 bits ARM7
 Memoria FLASH de 256 Kbytes
 Memoria RAM de 64 Kbytes
 Microcontrolador de 8 bit AVR
 Memoria FLASH de 4Kbytes
 Memoria RAM de 512 Bytes
 Comunicación Inalámbrica Bluetooth (Bluetooth Class II V2.0)
 Puerta de alta velocidad USB (12 Mbit/s)
 Cuatro puertas de entrada de seis contactos, plataforma digital (cable de 6 hilos)
 Tres puertas de salida de seis contactos, plataforma digital (cable de 6 hilos)
 Pantalla gráfica de cristal líquido de 64 x 100 puntos
 Parlante, calidad de sonido 8KHz
 Fuente de poder, 6 baterías AA

Botones del NXT

Botón Naranja: Se utiliza para encendido, y como Enter.

Botones Gris Claro: Se utilizan para desplazarse en el menú a la izquierda y derecha.

Botón Gris Oscuro: Se utiliza para retroceder en la selección del menú.

Nota:

Para apagar el NXT se presiona el botón gris Oscuro hasta llegar a la opción “Turn off?” y en la pantalla se verán los iconos ✓ ✗, y a continuación se presiona el botón Naranja en la opción deseada.



Botones

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Opciones del Menú

My Files

Software Files. En esta opción se encuentran los programas que se descargan de la computadora, vía USB o Bluetooth.

NXT Files. Programas que ha realizado directamente en el NXT.

Sound Files. Programas de sonidos.

NXT Programs. Al seleccionar este submenú, se muestra primero la pantalla donde se indica el uso de los puertos, y los motores. Al presionar el botón naranja permite programar una secuencia de órdenes a ser ejecutadas por el NXT.

Try Me. Programas precargados para probar motores y sensores, se deben usar los puertos predeterminados (los que se recomiendan en el submenú de NXT Programs).

Opciones del Menú

View. Permite visualizar el estado de los sensores y los motores conectados al NXT.

Setting. Puede ajustar el volumen del sonido del NXT, ajustar el modo Sleep, borrar programas y visualizar la versión del NXT.

Bluetooth. Permite encender o apagar el bluetooth. Si está encendido puede ajustar los distintos parámetros de la comunicación inalámbrica, tales como: contactos, conexiones, visible, on/off, buscar.

Lenguajes de Programación

NXT-G. Programación en modo gráfico (software oficial de LEGO), compatible con Windows y MAC. Basado en LabView.

RobotC. Basado en Ansi C (software oficial de LEGO), compatible con Windows.

NXC. Software Libre similar a C, compatible con Windows, MAC, Linux.

LeJOS. Software Libre basado en Java, compatible con Windows, MAC, Linux.

Lenguajes de Programación

PbLua. Software libre similar a C.

LabView. Software de programación gráfica (propietario), se puede ejecutar en múltiples plataformas.

Brix Command Center (BrixCC)*. Es un programa para Windows de 32-bit conocido como un IDE para robots LEGO MINDSTORMS robots que incluye todas las generaciones de ellos, hasta la actual EV3. BrixCC incluye **Not eXactly C (NXC)**, Next Byte Codes (NBC), NPG que usa el compilador NBC.

* Brix Command Center .<http://bricxcc.sourceforge.net/>

Lenguaje NXC

- **NXC** es el sinónimo de Not eXactly C.
- Es un lenguaje para programar productos LEGO MINDSTORMS NXT.
- El compilador NXC (llamado NBC) traslada el programa fuente a código de bytes del NXT.

Instalando el compilador NBC en Ubuntu

- Descargar el compilador nbc
 - `wget http://downloads.sourceforge.net/bricxcc/nbc-1.2.1.r4.tgz`
- Desempaquetar
 - Crear el directorio donde colocaremos el compilador: `mkdir nbc`
 - desempaquetar el archivo .tgz: `tar xzf nbc-1.2.1.r4.tgz -C nbc`
 - Cambiarse al directorio donde se desempaquete: `cd nbc/NXT`

Instalando el compilador NBC

- Observando la versión
 - Ejecutar el compilador: `./nbc`
Next Byte Codes Compiler versión 1.2
(...)
- Mover al directorio bin para no colocar siempre ./ antes de invocar al nbc
 - `sudo mv nbc /usr/local/bin`

Instalando el compilador NBC

- Probando el compilador

- Abrir un editor para colocar código nxc, en este caso invocar al editor nano: `nano hola.nxc`

```
task main()
{
    TextOut (0,0, "Hola mundo!");
    Wait(1000);
}
```

- Una vez guardo el archivo, procedemos a compilarlo:

`nbc hola.nxc -O=hola.rxe -sm-`

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Comunicación del NXT vía USB

- Crear un grupo llamado: legonxt
`sudo addgroup legonxt`

- Sumar usuarios al grupo

`sudo adduser login legonxt`
 p.e. `adduser gcgero legonxt`

- Crear un archivo: `nano /etc/udev/rules.d/45-legonxt.rules`
 colocar

```
SUBSYSTEM=="usb_device", ACTION=="add", SYSFS
{idVendor}=="0694", SYSFS{idProduct}=="0002",
SYMLINK+="legonxt-%k", GROUP="legonxt",
MODE="0660", RUN+="/etc/udev/legonxt.sh"
```

Comunicación del NXT vía USB

- Crea el archivo **legonxt.sh** en **/etc/udev** usando un editor:
`nano /etc/udev/legonxt.sh`

coloca en el archivo las siguientes líneas:

```
#!/bin/bash
GROUP=legonxt
if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ] then chmod o-rwx "${DEVICE}"
chgrp "${GROUP}" "${DEVICE}"
chmod g+rwx "${DEVICE}"
fi
```

Comunicación del NXT vía USB

- Cambiar modo del archivo
 - `sudo chmod a+x /etc/udev/legonxt.sh`
- Reiniciar el sistema o reinicia dispositivos
 - `reboot` o `service udev restart`
- Ahora, ya podemos compilar y enviar el proceso al brick (ladrillo del lego) escribiendo:
 - `nbc hola.nxc -sm- -d -S=usb`
- Si no puede cargar el programa con la instrucción anterior, se debe revisar el firmware (en el brick opción: **Settings + NXT Version**) y enviar de la siguiente manera (en el ejemplo el firmware es 1.03, por eso se colocó `-v=103`):
 - `nbc -d -v=103 hola.nxc`

Fuente principal: <http://www.jclarsen.dk/index.php/guides/21-programming/52-using-nxc-on-linux>

Compilador nbc

nbc es un programa que compila archivos NXC (Not eXactly C), archivos NBC (NeXT Byte Code) o archivos de imágenes NXT (.ric) usando el lenguaje basado en texto RICScrip.

Características:

- Puede guardar el resultado de la compilación en un archivo o subirlo al NXT para su ejecución.
- Puede compilar diferentes lenguajes (NXT, NBC o RICScrip).
- Las extensiones que reconoce son: .nxc, .nbc, y .rs

<http://manpages.ubuntu.com/manpages/saucy/man1/nbc.1.html>

Compilador nbc

Sintaxis

nbc [opciones] archivo [opciones]

Opciones

- S=<portname>** Especifica el nombre del puerto (COMn o usb), nombre de la fuente, o alias
- d** Descarga el programa
- b** Trata el archivo de entrada como un archivo binario (no le realiza compilación)
- q** modo ligero/sencillo
- n** Evita incluir los archivos de sistema
- D=<sym>[=<value>]** Define macro <sym>
- x** Descompilar programa
- Z[1|2]** Activar las optimizaciones del compilador

Compilador nbc

Opciones

- ER=n** Fija errores máximos antes de abortar (0 == no límite)
- PD=n** Fija la profundidad máxima de recursión del preprocesador (default == 10)
- O=<archivo>** Especifica el archivo de salida
- E=<archivo>** Escribe los errores de compilación al archivo <archivo>
- I=<ruta>** Busca <ruta> para incluir archivos
- nbc=<archivo>** Guarda NXC en código intermedio NBC en <archivo>
- L=<archivo>** Genera un listado de código en <archivo>
- Y=<archivo>** Genera tabla de símbolos en <archivo>

Compilador nbc

Opciones

- w[-|+]** Advertencias apagado o encendido [default es encendido]
- sm[-|+]** Mensaje de estado del compilador apagado o encendido [default es encendido]
- EF** firmware mejorado
- safecall** NXC envolverá todos los llamados de función en Adquirir/Liberar (Acquire/Release)
- api** Descarga el API en stdout
- v=n** Establece la especifica versión del firmware (default == 128, NXT 1.1 == 105)
- help** Muestra las opciones de linea de comando

Programando en NXC

Los programas en NXC consisten de tareas.

Una tarea consiste de un número de comandos, llamadas **Instrucciones**.

Las llaves encierran todas las **Instrucciones**.

```
task main()
{
    Instrucción 1...
    Instrucción 2...
    ...
    ...
}
```

Reglas Léxicas

- **Comentarios.**

Para comentar múltiples líneas, estas se colocan entre

```
/* y */
```

Para comentar una sola línea se realizan por medio

```
// comentario
```

- **Espacios.**

Un espacio puede ser un tabulador, nueva línea o espacio. Es válido colocar

```
x=2;
```

```
x = 2;
```

Reglas Léxicas

- **Constates de cadenas.** Como en C son delimitadas por el carácter de doble comillas

```
TextOut(0, LCD_LINE1 , "Prueba")
```

- **Constates de caracter.** Son delimitadas por comillas simples. El valor del carácter está en valor numérico ASCII

```
char ch = 'a'; // ch == 97
```

- **Constantes numéricas.** Pueden ser escritas en forma decimal o hexadecimal

```
x = 10; // se coloca x a 10
```

```
x = 0x10; // se coloca x a 16 (10 hex)
```

```
f = 10.5; // se coloca f a 10.5
```

NOTA

En el ejemplo anterior se utilizó una constante predefinida

```
#define LCD_LINE1 56
```

Que indica que es la primera línea de la pantalla LCD del NXT.

Los números de líneas para ser usadas por la función del sistema **DrawText** son:

```
#define LCD_LINE8 0
#define LCD_LINE7 8
#define LCD_LINE6 16
#define LCD_LINE5 24
#define LCD_LINE4 32
#define LCD_LINE3 40
#define LCD_LINE2 48
#define LCD_LINE1 56
```

Práctica 1

```
task main ()
{
    // Imprime un mensaje en la pantalla
    TextOut (0,0,"Hola Mundo");
    Wait (1000);
    // Limpia pantalla
    ClearScreen();
    TextOut (0,LCD_LINE6,:);
    Wait(1000);
}
```

Guardar como: p1.nxc

- 1) Compilar:
`nbc p1.nxc -O=p1.rxe -sm-`
- 2) Descargar en el NXT:
`nbc p1.nxc -sm- -d -S=usb`
o
`nbc -d -v=103 p1.nxc`

Práctica 2

```
#define XMAX 99
#define YMAX 63
#define XMID (XMAX)/2
#define YMID (YMAX)/2

task main()
{
    PointOut(1, YMAX-1);
    PointOut(XMAX-1, YMAX-1);
    PointOut(1,1);
    PointOut(XMAX-1,1);
    Wait(2000);
    RectOut(5,5,90,50);
    Wait(2000);
    LineOut(5,95,55);
    Wait(2000);
    LineOut(5,55,95,5);
    Wait(2000);
    CircleOut(XMID,YMID-2,20);
    Wait(1000);
    ClearScreen();
    GraphicOut(30,10,"faceclosed.ric");
    Wait (5000);
    ClearScreen();
    GraphicOut(30,10,"faceopen.ric");
    Wait (2000);
}
```

Identificadores

- Los **identificadores** pueden ser usados para nombrar variables, tareas, funciones, y subrutinas.
- El primer carácter de un identificador puede ser una letra mayúscula, minúscula o un guión bajo. Los caracteres que le siguen pueden ser letras, números o guiones bajos.
- Existe una lista de palabras reservadas por el lenguaje NXC que no pueden ser utilizadas como identificadores, tales como: if, else, do, while, byte, char, float,

Estructura de un programa

- Un programa está compuesto de bloques de código (tareas y funciones) y variables.
- Cada bloque tiene sus propias características pero comparten una estructura común. El máximo número de bloques es 256.
- La tarea principal es llamada **main()**, y es la primera que se ejecuta.
- Una función se ejecuta cada vez que se llama en un bloque.
- Reglas de orden léxico: Cualquier identificador que nombra a una tarea o función debe ser conocido por el compilador antes de ser usado, una vez definida una tarea o función no puede ser redefinida o declarada.

Variables

Todas las variables en NXC están definidas usando uno de los siguientes tipos:

- [bool](#)
- [byte](#)
- [char](#)
- [int](#)
- [short](#)
- [long](#)
- [unsigned](#)
- [float](#)
- [mutex](#)
- [string](#)
- [Structures](#)
- [Arrays](#)

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Variables

Ejemplos:

```
int x;           // declara x
bool y,z;        // declara y y z
long a=1,b;      // declara a y b, inicializado a 1
float f=1.15, g; // declara f y g, inicializado f
int data[10];    // arreglo de 10 ceros en data
bool flags[] = {true, true, false, false};
string msg = "hello world"
```

Arreglos

Un arreglo se declara igual que una variable, pero con corchetes cuadrados [] que siguen al nombre de la variable.

```
int mi_arreglo[]; // declara un arreglo con 0 elementos
```

Se puede declarar de una dimensión o dos o tres o cuatro, que es su máximo en NXC.

```
bool mi_arreglo[][]; //declara un arreglo de 2-dimensiones
```

Arreglos

Un arreglo puede ser inicializado al ser declarado:

```
int x[] = {1, 2, 3, 4}, Y[]={10, 10}; // 2 arreglos
int matrix[][] = {{1, 2, 3}, {4, 5, 6}};
string autos[] = {"honda", "ford", "chevy"};
```

Los elementos de un arreglo son identificados por la posición en la cual se encuentran dentro del arreglo (índice). El primer elemento del arreglo tiene la posición 0, el segundo 1, y así sucesivamente.

```
mi_arreglo[0] = 123; // coloca a elemento uno el valor de 123
mi_arreglo[1] = mi_arreglo[2]; // copia el tercer elemento
en la posición 2 del arreglo
```

Declaraciones

El cuerpo de los bloques de código (tareas o funciones) está compuesta por declaraciones, las cuales son terminadas por punto y coma (",").

Asignaciones

La **Sintaxis** para asignación es:

```
variable Operador Expresión;
```

Ejemplos:

```
x = 2; // coloca x a 2
y = 7; // coloca y a 7
x += y; // x es 9, y sigue siendo 7
```

Operador	Acción
=	Coloca a la variable la expresión
+=	Suma la expresión a la variable
-=	Resta la expresión a la variable
*=	Multiplica la variable por la expresión
/=	Divide la variable por la expresión
%=	Coloca a la variable el resto de la división por la expresión
&=	Coloca el resultado del AND realizado con la expresión
=	Coloca el resultado del OR realizado con la expresión
^=	Coloca el resultado del XOR realizado con la expresión
	Coloca en la variable el valor absoluto de la expresión
+-=	Coloca a la variable el signo (-1, +1, 0) de la expresión
>>=	Realiza un corrimiento a la derecha de la expresión
<<=	Realiza un corrimiento a la izquierda de la expresión

Variables y Arreglos

```

int a;
int b,c;
int valores[];
task main()
{
    a = 10;
    b = 20 * 5;
    c = b;
    c /= a;
    c -= 5;
    a = 10 * (c + 3); // a es 80
    ArrayInit(valores,0,10); //inicializa los 10 elementos a 0
    valores[0] = a;
    valores[1] = b;
    valores[2] = a*b;
    valores[3] = c;
}

```

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Estructuras de control

Sentencia if

La sentencia **if** evalúa una condición. Si la sentencia es verdadera, ejecuta mas sentencias (la consecuencia). El valor de una condición es falsa solo si la evaluación es cero. Para cualquier valor no cero, **if** es verdadero.

Sintaxis:

```
if (condición) consecuencia
```

Ejemplos:

```
if (x==1) y = 2;
if (x==1) { y = 1; z = 2; }
```

Sentencia if-else

La sentencia **if-else** evalúa una condición, si la condición es verdadera se realiza una sentencia (consecuencia), la segunda sentencia (alternativa) que precede a la palabra **else**, se ejecuta si la condición es falsa. La condición se considera falsa solo si la evaluación es cero, si es diferente de cero es verdadero.

Sintaxis:

```
if (condición) consecuencia else alternativa
```

Sentencia if-else

Ejemplo:

```
if (x==1)
    y = 3;
else
    y = 4;
if (x==1) {
    y = 1;
    z = 2;
}
else {
    y = 3;
    z = 5;
}
```

Sentencia while

La sentencia **while** es usada para construir un ciclo condicional. La condición es evaluada, y si es verdadera entonces el cuerpo del ciclo es ejecutada, y después la condición es evaluada nuevamente. Este proceso continua hasta que la condición sea falsa (o se invoca a la sentencia **break** en la ejecución).

Sintaxis:

```
while (condición) cuerpo
```

Ejemplo:

```
while(x < 10)
{
    x = x+1;
    y = y*2;
}
```

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Práctica 3

```
task main()
{
    int cont=0;
    while (cont<10)
    {
        if(ButtonPressed(BTNLEFT, false))
            cont+=1;
        if(ButtonPressed(BTNRIGHT, false))
            cont=0;
        if(ButtonPressed(BTNCENTER, false))
            break;
        ClearScreen();
        NumOut (0,LCD_LINE6, cont);
        Wait(1000);
    }
    ClearScreen();
    TextOut (0,LCD_LINE5, “ADIOS”);
    Wait(1000);
}
```

Guardar como: p3.nxc
1) Compilar:
nbc p3.nxc -o=p3.rxe -sm-
2) Descargar en el NXT:
nbc p3.nxc -sm- -d -S=usb
o
nbc -d -v=103 p3.nxc

Ejercicio

- Tomando como base la practica 2 y 3. Realiza lo siguiente:
 - Al presionar el botón izquierdo muestra un archivo ric
 - Al presionar el botón derecho muestra un archivo ric (diferente al archivo del inciso a)
 - Al presionar el botón del centro muestra ocho círculos en el display que tengan como centro la mitad del display y sus radios sean diferentes
 - Al presionar el botón de exit. Muestre el texto “ADIOS” en el centro de la pantalla.

I. ¿Qué pasa al programar el botón Exit?

Sentencia do-while

Una variante del ciclo while es el ciclo **do-while**.

Sintaxis:

do cuerpo **while** (condición)

La diferencia entre while y do-while es que el **do-while** siempre ejecuta el cuerpo de instrucciones por lo menos una vez, mientras que el while puede no ejecutarlo.

Ejemplo:

```
do
{
    x = x+1;
    y = y*2;
} while (x < 10);
```

Práctica 4

```
task main()
{
    int i;
    for (i=0; i<100; i++)
    {
        OnFwd(OUT_BC, i);
        Wait(100);
    }
    for (; i>0; i--)
    {
        OnRev(OUT_BC, i);
        Wait(100);
    }
    Off(OUT_BC);
```

1. ¿Qué sucede si no colocamos el Wait()?

Práctica 5

```
task main()
{
    while (1)
    {
        if (ButtonPressed(BTNCENTER, false))
            OnFwd(OUT_BC, 75);
        if (ButtonPressed(BTNLEFT, false))
            OnRev(OUT_BC, 75);
        if (ButtonPressed(BTNRIGHT, false))
        {
            Off(OUT_BC);
            break;
        }
    }
}
```

1. En este ejemplo no colocamos Wait(), pero ¿Por qué se queda en movimiento?

Práctica 6

```

int mueve, turno, total;
task main()
{
    total_time = 0;
    do
    {
        mueve = Random(1000);
        turno = Random(1000);
        OnFwd(OUT_AC, 75);
        Wait(mueve);
        OnRev(OUT_C, 75);
        Wait(turno);
        total+= (mueve+turno);
    } while (total< 20000);
    Off(OUT_AC);
}

```

Sentencia for

Este tipo de ciclo permite automáticamente inicializar e incrementar/decrementar una variable que sirve de contador.

Sintaxis:

for(sentencial ; condición ; sentencia2) cuerpo

Un ciclo **for** siempre ejecuta el sentencia1, y repetidamente verifica la condición. Mientras la condición sea verdadera ejecuta el cuerpo y luego ejecuta el sentencia2.

Frecuentemente la sentencia `se` encarga de fijar el valor de la variable que sirve de contador para el ciclo.

Práctica 7

```
task main()
{
    for (int i=0; i<8; i++)
    {
        NumOut(0,LCD_LINE1-i*8, i);
    }
}
```

Nota:

NumOut. Dibuja un número en el display en la posición x,y.

LCD_LINE1. Línea de la pantalla (`#define LCD_LINE1 56`)

Sentencia repeat

El **repeat** ejecuta un ciclo un número específico de veces.

Sintaxis:

repeat (expresión) cuerpo

La expresión determina cuantas veces se va a ejecutar el cuerpo. La expresión seguida de **repeat** es evaluada solo una vez y entonces el cuerpo es repetido el número de veces indicado.

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Práctica 8

```
task main()
{
    int i=0;
    repeat (8)
    {
        NumOut(0, LCD_LINE1-i*8, i++);
    }
}
```

Práctica 9

```
#define TIEMPO 500

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

La primera línea define una constante, llamada:TIEMPO

La repetición es por medio de la declaración **repeat** y entre paréntesis el número de veces (4) que se repiten las instrucciones dentro de las llaves

Sentencia repeat

También se pueden colocar anidaciones de **repeat**

```
repeat(10)
{
    repeat(4)
    {
        .....
        .....
        .....
    }
}
```

Sentencia switch

La sentencia switch ejecuta una de varias secciones de códigos diferentes, dependientes del valor de una expresión. Una o mas etiquetas de casos preceden a cada sección de código.

Cada caso debe ser una constante y única para verificar con la sentencia switch. El **switch** evalúa la expresión y verifica las etiquetas, si coincide la evaluación con alguna de ellas se ejecutan sus sentencias hasta que encuentra un **break** o el fin del switch.

Se puede utilizar una etiqueta llamada **default** para el caso de que no coincida la expresión con alguna de las etiquetas.

Sintaxis:

switch (expresión) cuerpo

Sentencia switch

Ejemplo:

```
switch(x)
{
    case 1:
        // Realizar cuando x es 1
        break;
    case 2:
    case 3:
        // Realizar cuando x es 2 o 3
        break;
    default:
        // Realizar cuando x no es 1, 2, o 3
        break;
}
```

Sentencia break y continue

Con los ciclos (tal como el while) se puede utilizar la sentencia **break** para salir del ciclo de forma inmediata. Además es un componente básico para la sentencia switch.

Sintaxis:

```
break;
```

Dentro de los ciclos se puede usar la sentencia **continue** para saltar al tope de la siguiente iteración del ciclo, saltando el código que se encuentre después de la sentencia **continue**.

Sintaxis:

```
continue;
```

Sentencia break y continue

Ejemplo (break):

```
while (x<100) {
    x = get_new_x();
    if (button_pressed())
        break;
    process(x);
}
```

Ejemplo (continue):

```
while (x<100) {
    ch = get_char();
    if (ch != 's')
        continue;
    process(ch);
}
```

Sentencia goto

La sentencia **goto** indica a un programa que debe saltar a una específica localidad. Las sentencias a ejecutar deben estar después de una etiqueta y deben estar alineados en la columna. La etiqueta sirve para indicar donde saltar. Solo se puede saltar en una tarea o función, no fuera de ella.

Ejemplo:

```
mi_ciclo:
    x++;
    goto mi_ciclo;
```

Sentencia until

Este constructor proporciona una alternativa conveniente para el ciclo while. La definición del **until** es:

```
#define until(c)      while(!(c))
```

En otras palabras, **until** debe realizar el ciclo hasta que la condición sea verdadera.

Es frecuente usar esta conjunción con un cuerpo de sentencias vacías o un cuerpo para realizar otras tareas.

```
until(EVENTOS_OCURRAN); //espera que ocurra algún evento
```

Sentencia start y stop

Se puede iniciar una tarea con la sentencia **start**. Esta sentencia puede ser utilizada por el NBC/NXC firmware estándar o mejorado. La operación resultante es un código nativo en el firmware mejorado pero requiere subrutinas especiales generadas por el compilador para poder trabajar con el firmware estándar.

```
start nombre_tarea;
```

Se puede parar una tarea con la sentencia **stop**. Esta sentencia es solo soportada si se está ejecutando el firmware NBC/NXC mejorado en el NXT.

```
stop nombre_tarea;
```

Expresiones

Las expresiones básicas son las asignaciones de valores, las de mayor complicación son las formadas por valores que usan varios operadores.

Las constantes numéricas en el NXT son representadas por valores enteros o punto flotante. El tipo depende del valor de la constante. Internamente el NXC usa matemática de punto flotante de 32 bit para la evaluación de expresiones constantes. Las constantes numéricas están escritas como decimal o hexadecimal.

Existen dos valores especiales predefinidos: **true** y **false**. El valor **false** es cero (0), mientras que el valor **true** es uno (1). Los mismos valores se mantienen para las relaciones de operadores (p.e. <): cuando la relación es falsa el valor es 0, en otro caso el valor es 1.

Operador	Descripción	Asociatividad	Restricción	Ejemplo
<code>abs()</code>	valor absoluto	n/a		<code>abs(x)</code>
<code>sign()</code>	signo de operando	n/a		<code>sign(x)</code>
<code>++, --</code>	incremento/decremento en posfijo	izquierda	solo variables	<code>x++</code>
<code>++, --</code>	incremento/decremento en prefijo	derecha	solo variables	<code>++x</code>
<code>-</code>	menos unario	derecha		<code>-x</code>
<code>~</code>	negación de bit	derecha		<code>~123</code>
<code>!</code>	negación de lógica	derecha		<code>!x</code>
<code>*, /, %</code>	multiplicación, división, módulo	izquierda		<code>x*y</code>
<code>+, -</code>	suma, resta	izquierda		<code>x+y</code>

Operador	Descripción	Asociatividad	Restricción	Ejemplo
<code><<, >></code>	corrimiento a la izquierda, derecha	izquierda		<code>x<<4</code>
<code><, >, <=, >=</code>	operadores relacionales	izquierda		<code>x < y</code>
	igual a, no igual a	izquierda		<code>x == 1</code>
<code>&</code>	AND para bit	izquierda		<code>x & y</code>
<code>^</code>	XOR para bit	izquierda		<code>x ^ y</code>
<code> </code>	OR para bit	izquierda		<code>x y</code>
<code>&&</code>	AND lógico	izquierda		<code>x && y</code>
<code> </code>	OR lógico	izquierda		<code>x y</code>
<code>?:</code>	condicional ternario	derecha		<code>x == 1 ? y : z</code>

Condiciones

La condición se forma comparando dos expresiones.

Una condición puede ser negada con el operador de negación lógico o combinar dos condiciones con los operadores lógicos AND y OR.

condición	Significado
expr	Verdadero si expr no es igual a 0
expr1 == expr2	Verdadero si expr1 es igual a expr2
expr1 != expr2	Verdadero si expr1 no es igual a expr2
expr1 < expr2	Verdadero si expr1 es menor que expr2
expr1 <= expr2	Verdadero si expr1 es menor que o igual a expr2
expr1 > expr2	Verdadero si expr1 es mayor que expr2
expr1 >= expr2	Verdadero si expr1 es mayor que o igual a expr2
! condicion	Negación lógica de una condición - Verdadero si la condición es falsa
cond1 && cond2	AND lógico de dos condiciones (Verdadero si y solo si ambas condiciones son verdaderas)
cond1 cond2	OR lógico de dos condiciones (Verdadero si y solo si al menos una de las condiciones es verdadera)
true	La palabra true tiene el valor de 1. Representa la condición de siempre verdadero
false	La palabra false tiene el valor de 0. Representa la condición de siempre false

Práctica 10

```
#define TIEMPO 500
task main()
{
    while(true)
    {
        OnFwd(OUT_AC, 75);
        Wait(TIEMPO);
        if (Random() >= 0)
        {
            OnRev(OUT_C, 75);
        }
        else
        {
            OnRev(OUT_A, 75);
        }
        Wait(TIEMPO);
    }
}
```

Funciones

Funciones

- Las funciones son un grupo de instrucciones agrupadas que pueden ser llamadas cuando se necesiten. Estas pueden ser invocadas pasando argumentos y pueden retornar algún valor. Sintaxis
`[safecall] [inline] return_type name(argument_list)`
`{
 // cuerpo de la función
}`
- La lista de argumento puede ser vacía, o puede contener uno o más argumentos. Un argumento está definido como un tipo seguido por un nombre, cuando son múltiples argumentos se separan por comas. Todos los valores representados pueden ser: bool, char, byte, int, short, long, unsigned int, unsigned long, float, string, struct types o array de algún tipo.

Argumentos por valor

- NXC soporta paso de argumentos por **valor**, **valor constante**, **referencia** y **referencia constante**.

Paso por valor

Cuando los argumentos son pasados por valor, el compilador debe asignar una variable temporal para contener el argumento. Por lo tanto los cambios en dicha variable solamente son locales.

```
void foo(int x)
{
    x = 2; // solo afecta a x
}
task main()
{
    int y = 1;      // y es igual a 1
    foo(y); // y sigue siendo igual a 1
}
```

Argumento por valor constante

Este tipo también se pasa por valor. Si la función es una función **inline** entonces los argumentos pueden ser tratados por el compilador como valores constantes verdaderos y pueden ser evaluados en tiempo de compilación.

Si la función no es **inline** entonces el compilador trata los argumentos como si fuera una referencia constante, lo que permite pasar constantes o variables.

Argumento por valor constante

```
void foo(const int x)
{
    PlayTone(x, MS_500);
    x = 1; //Error-no se puede modificar el argumento
    Wait(SEC_1 );
}

task main()
{
    int x = TONE_A4 ;
    foo(TONE_A5); // ok
    foo(4* TONE_A3); //expresión tratada como constante
    foo(x); //x no es una constante pero se puede pasar
}
```

Argumento por referencia

Permite que el llamado de la función modifique el valor y estos estén disponibles después de terminada dicha función. Para adquirir el valor se debe colocar & a la variable que se recibe como argumento.

```
void foo(int &x)
{
    x = 2;
}
task main()
{
    int y = 1;      // y es igual a 1

    foo(y); // y es ahora igual a 2
    foo(2); // Error - Solo se envían variables
}
```

Argumento por referencia constante

También se pasa por **referencia** pero la restricción de que la función que la invoca **no** se le **permite modificar el valor**. Debido a esta restricción el compilador es capaz de que se le puedan pasar no solo variables. Las funciones deben ser invocadas con el número correcto de argumentos.

```
void foo(int bar, const int baz)
{
    // cuerpo de la función...
}
task main()
{
    int x; // Declaración de x
    foo(1, 2); // ok
    foo(x, 2); // ok
    foo(2); // Error-número incorrecto de argumentos
}
```

Detalles de argumentos

safecall

Si la función es marcada como **safecall** entonces el compilador sincroniza la ejecución de esta función a través de múltiples hilos, protegiendo el llamado de la función con los llamados **Acquire** y **Release**. Si un segundo hilo trata de invocar a la función **safecall** mientras otro hilo está en ejecución, el segundo hilo debe esperar hasta que la función retorne del primer hilo.

Detalles de argumentos

safecall

```
safecall void foo(unsigned int frequency)
{
    PlayTone(frequency,SEC_1);
    Wait(SEC_1);
}

task task1()
{
    while(true) {
        foo(TONE_A4);
        Yield();
    }
}

task task2()
{
    while(true) {
        foo(TONE_A5);
        Yield();
    }
}

task main()
{
    Precedes(task1,task2);
}
```

Detalles de argumentos

inline

Las funciones pueden ser marcadas como **inline**. Con esto se indica que se debe crear una copia del código de dicha función, solo hay que tener en consideración que como se copia el código entonces esté en lugar de reducir, aumentar.

El código del ejemplo siguiente muestra como usar **inline** en la función, en este caso la task main contiene 4 llamados a PlayTone y 4 a Wait, además de los 4 llamados a la subrutina foo, dado que el código es expandido.

Detalles de argumentos

inline

```
inline void foo(unsigned int frequency)
{
    PlayTone(frequency, SEC_1);
    Wait(SEC_1);
}

task main()
{
    foo(TONE_A4);
    foo(TONE_B4);
    foo(TONE_C5);
    foo(TONE_D5);
}
```

Detalles de argumentos

void

void permite definir funciones que no retornen datos, estas funciones son referidas como procedimientos o subrutinas. La palabra **sub** es un alias de **void**, ambas son usadas para declarar o definir una función.

El Preprocesador

El preprocesador NCX implementa las siguientes directivas estandares preprocesadas: #include, #define, #ifdef, #ifndef, #endif, #if, #elif, #undef, ##, #line, #error, y #pragma . Además soporta dos directivas no estandares: #download and #import.

MODULOS

Módulo Output

El módulo **Output** abarca todas las salidas de motores. Casi todas las funciones del API NXC toman como argumento una salida o un conjunto de salidas variables o constantes

Macros para especificar el puerto de salida del motor

```
#define OUT_A 0x00 //Salida del puerto A
#define OUT_B 0x01 //Salida del puerto B
#define OUT_C 0x02 //Salida del puerto C
#define OUT_AB 0x03 //Salida del puerto A y puerto B
#define OUT_AC 0x04 //Salida del puerto A y puerto C
#define OUT_BC 0x05 //Salida del puerto B y puerto C
#define OUT_ABC 0x06 //Salida del puerto A, puerto B y puerto C
```

Módulo Output

Macros de modos del puerto de salida del motor

```
#define OUT_MODE_COAST 0x00 //punto muerto del motor
#define OUT_MODE_MOTORON 0x01 //encender motor
#define OUT_MODE BRAKE 0x02 // freno de motor
#define OUT_MODE_REGULATED 0x04 //regulador de motor
#define OUT_MODE_REGMETHOD 0xF0 //registro de método
```

Módulo Output

Funciones para accesar y modificar características del modulo de salida:

void SetMotorPwnFreq(byte): Establece la frecuencia reglamentaria del motor.

void SetMotorRegulationTime(byte): Establece el tiempo reglamentario.

void SetMotorRegulationOptions(byte): Establece opciones reglamentarias.

void OnFwdSyncPID (byte, char, char , byte, byte, byte): Motores de avance sincronizados en los factores PID.

Módulo Output

void OnFwdSyncExPID(byte, char, char, const byte, byte, byte, byte): Motores de avance sincronizados y contadores con factores PID reiniciados.

void OnRevSyncPID(byte, char, char, byte, byte, byte): Motores de reversa sincronizados con los factores PID.

void OnRevSyncExPID(byte, char, char, const byte, byte, byte, byte): Motores de reversa sincronizados y contadores con factores PID reiniciados.

Módulo Output

void OnFwdRegPID(byte, char, byte, byte, byte, byte): Avanzan motores hacia adelante regulados con los factores PID.

void OnFwdRegExPID(byte, char, byte, const byte, byte, byte, byte): Avanzan motores hacia adelante regulados y reiniciando contadores con los factores PID.

void OnRevRegPID(byte, char, byte, byte, byte, byte): Marchan los motores de reversa regulados con los factores PID.

void OnRevRegExPID(byte, char, byte, const byte, byte, byte, byte): Marchan los motores hacia atrás regulados y reiniciando los contadores con los factores PID.

void Off(byte): Apaga motores.

Módulo Output

void **OffEx**(byte, const byte): Apaga motores y reinicia contadores.

void **Coast**(byte): Punto muerto de motores.

void **CoastEx**(byte, const byte): Punto muerto de motores y reinicia contadores.

void **Float** (byte): Motores flotadores (float).

Módulo Output

void **OnFwd**(byte, char): Motores de marcha hacia adelante.

void **OnFwdEx**(byte, char, const byte): Motores de marcha hacia adelante y reinicia contadores.

void **OnRev**(byte, char): Motores de marcha hacia atrás.

void **OnRevEx**(byte, char, const byte): Motores de marcha hacia atrás y reinicia contadores.

void **OnFwdReg**(byte, char, byte): Motores de marcha hacia adelante regulados.

void **OnFwdRegEx**(byte, char, byte, const byte): Motores de marcha hacia delante regulados y contadores reiniciados.

Módulo Output

void **OnRevReg**(byte, char, byte): Ejecuta motores de reversa regulados.

void **OnRevRegEx**(byte, char, byte, const byte): Ejecuta motores de reversa regulados y contadores reiniciados.

void **OnFwdSync**(byte, char, char): Motores de marcha hacia adelante sincronizados.

void **OnFwdSyncEx**(byte outputs, char pwr, char turnpt, const byte reset): Motores de marcha hacia adelante sincronizados y contadores reiniciados.

Módulo Output

void **OnRevSync**(byte, char pwr, char): Ejecuta motores de reversa sincronizados.

void **OnRevSyncEx**(byte, char pwr, char, const byte): Ejecuta motores de reversa sincronizados y contadores reiniciados.

void **RotateMotor**(byte, char, long): Rota el motor.

void **RotateMotorPID**(byte, char, long, byte, byte, byte): Rota el motor con factores PID.

void **RotateMotorEx**(byte, char, long, char, bool, bool): Rota el motor Ex.

Módulo Output

void **RotateMotorExPID**(byte, char, long, char, bool, bool, byte, byte, byte): Rota el motor Ex con factores PID.

void **ResetTachoCount**(byte): Reinicia el contador de tacómetro.

void **ResetBlockTachoCount**(byte): Reinicia el contador relativo al bloque.

void **ResetRotationCount**(byte): Reinicia el contador relativo al programa.

void **ResetAllTachoCounts**(byte): Reinicia todos los contadores del tacómetro.

Módulo Output

void **ResetRotationCount**(byte): Reinicia el contador relativo al programa.

void **ResetAllTachoCounts**(byte): Reinicia todos los contadores de tacómetro.

void **SetOutput**(byte, byte, variant,..., byteN, variant N): Fija campos de salida.

variant **GetOutput**(byte, const byte): Obtiene los valores de campos de salida.

byte **MotorMode**(byte): Obtiene modo de motor.

Módulo Output

char **MotorPower(byte)**: Obtiene el nivel de potencia del motor.

char **MotorActualSpeed(byte)**: Obtiene la velocidad actual del motor.

long **MotorTachoCount(byte)**: Obtiene el contador del tacómetro del motor.

long **MotorTachoLimit(byte)**: Obtiene el límite del tacómetro del motor.

byte **MotorRunState(byte)**: Obtiene el estado de ejecución del motor.

Módulo Output

char **MotorTurnRatio(byte)**: Obtiene la relación de vueltas del motor.

byte **MotorRegulation(byte output)**: Obtiene el modo de regulación del motor.

bool **MotorOverload(byte output)**: Obtiene el estado overload del motor.

byte **MotorRegPValue(byte output)**: Obtiene el valor P del motor P.

byte **MotorRegIValue(byte output)**: Obtiene el valor I del motor.

Módulo Output

byte **MotorRegDValue(byte output)**: Obtiene el valor D del.

long **MotorBlockTachoCount(byte)**: Obtiene el contador relativo al bloque del motor.

long **MotorRotationCount(byte)**: Obtiene el contador relativo del programa del motor.

byte **MotorOutputOptions(byte)**: Obtiene las opciones del motor.

byte **MotorMaxSpeed(byte)**: Obtiene la velocidad máxima del motor.

Módulo Output

byte **MotorMaxAcceleration(byte)**: Obtiene la aceleración máxima del motor.

byte **MotorPwnFreq()**: Obtiene la frecuencia de regulación del motor.

byte **MotorRegulationTime()**: Obtiene el tiempo de regulación del motor.

byte **MotorRegulationOptions()**: Obtiene las opciones de regulación del motor.

Módulo Output

void **PosRegEnable(byte, byte p= PID_3, byte i= PID_1 , byte d= PID_1)**: Habilita la posición de regulación absoluta con los factores PID.

void **PosRegSetAngle(byte, long)**: Cambia el valor actual para el ángulo fijado.

void **PosRegAddAngle(byte, long)**: Suma al valor actual para fijar el ángulo.

void **PosRegSetMax(byte, byte, byte)**: Fija los límites máximos.

Rotación

Rota el motor un específico número de grados.

void **RotateMotor (byte outputs, char pwr, long angle)**

Parámetros:

outputs. Puertos de salida, pueden ser constantes o variables. Si se desea controlar múltiples puertos se usa un arreglo de bytes.

pwr. Potencia que puede ser de 0 a 100. Negativo indica dirección de reversa.

angle. Ángulo límite en grados. Si se coloca negativo es dirección inversa.

Ejemplo:

RotateMotor(OUT_A, 100, 50);

Rotación

```
void RotateMotorEx ( byte outputs, char pwr, long angle, char turnpct, bool sync, bool stop )
```

Ejecutar las salidas especificadas hacia adelante durante el número de grados. Especifique también la sincronización, porcentaje de giro, y las opciones de frenado. Esta función se utiliza principalmente con más de un motor.

Parámetros

outputs: Puertos de salida. Puede ser una constante o una variable. Si utiliza una variable y desea controlar varias salidas en una sola llamada tiene que utilizar una matriz de bytes en lugar de un byte y almacena los valores de puerto de salida de la matriz de bytes antes de pasar en esta función.

Rotación

pwr: Potencia de salida (0..100). Puede ser negativo para invertir la dirección.

angle: Angulo límite en grados. Puede ser negativo para invertir la dirección.

turnpct: Giro de (-100..100). La dirección del vehículo depende de su construcción.

sync: Sincroniza los motores. Debe establecerse en true para producir resultado.

stop: Especifica que los motores deben frenar al final de la rotación.

Ejemplo:

```
RotateMotorEx (OUT_BC, 75, 360, 50, true, true )
```

Moviendo los motores

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

- 1) Indica al robot comunicarse con el motor conectado en su salida A y se mueva hacia adelante a 75% de su máxima velocidad.
- 2) Lo mismo que 1, pero para el motor conectado en C
- 3) Indica que espere 4 segundos
- 4) Se le indica al robot que se mueva en sentido inverso, es decir, hacia atrás. Se combinan los dos motores conectados en A y C.
- 5) Indica que espere 4 segundos
- 6) Apagamos los motores A y C.

Cambiando la velocidad

La velocidad se puede cambiar en el segundo parámetro de los llamados siguientes:

```
OnFwd(OUT_AC, 30); // avanzar
OnRev(OUT_AC, 30); // retroceder
```

La potencia está entre 0 y 100.

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Constantes de puertos de entrada

Las constantes de entrada son usados cuando se llama al API de las funciones de control de los sensores

```
#define IN_1 0x00
#define IN_2 0x01
#define IN_3 0x02
#define IN_4 0x03
```

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Módulo Display

El módulo **display** abarca el soporte para dibujar en el LCD del NXT. El NXT soporta dibujos de puntos, líneas, rectángulos y círculos. Permite dibujar archivos gráficos de iconos, textos y números en la pantalla.

La coordinada origen de la pantalla LCD es (0,0). El API NXC proporciona constantes para ser usadas en las funciones **NumOut** y **TextOut**, las cuales permite colocarse en una línea específica del LCD, las cuales van de 1 a 8, 1 es la línea límite superior y 8 es la línea límite inferior.

Estas constantes (LCD_LINE1, LCD_LINE2, LCD_LINE3, LCD_LINE4, LCD_LINE5, LCD_LINE6, LCD_LINE7, LCD_LINE8) deben ser usadas como la coordenada Y en los llamados **NumOut** y **TextOut**.

Módulo Display

Función **TextOut**

```
char TextOut (int x, int y, string str, unsigned long
options=DRAW_OPT_NORMAL)
```

Dibuja un texto en la coordenada (x,y) de la pantalla. El valor de y debe ser multiplo de 8 (se puede utilizar las constantes de numero de linea: **LCD_LINE**). Si el argumento no se especifica el valor por default es DRAW_OPT_NORMAL. Los valores de este argumento están listados en el grupo de Drawing option constants.

Parámetros:

- x**: Es el valor de la coordenada de inicio de la salida del texto en el LCD
- y**: Es el número de línea para la salida del texto en el LCD
- str**: La cade na a desplegar en el LCD
- options**: Opción de dibujo del texto.

Módulo Display

Constantes de la opción **Font drawing**

Estas constantes solo son usadas cuando se dibujan en el LCD texto o números usando la fuente RIC-base

```
#define DRAW_OPT_NORMAL (0x0000)
#define DRAW_OPT_CLEAR_WHOLE_SCREEN (0x0001)
#define DRAW_OPT_CLEAR_EXCEPT_STAT_SCREEN (0x0002)
#define DRAW_OPT_CLEAR_PIXELS (0x0004)
#define DRAW_OPT_CLEAR (0x0004)
#define DRAW_OPT_INVERT (0x0004)
#define DRAW_OPT_LOGICAL_COPY (0x0000)
#define DRAW_OPT_LOGICAL_AND (0x0008)
#define DRAW_OPT_LOGICAL_OR (0x0010)
#define DRAW_OPT_XOR (0x0018)
#define DRAW_OPT_FILL_SHAPE (0x0020)
#define DRAW_OPT_CLEAR_SCREEN_MODES (0x0003)
#define DRAW_OPT_LOGICAL_OPERATIONS (0x0018)
#define DRAW_OPT_POLYGON_POLYLINE (0x0400)
#define DRAW_OPT_CLEAR_LINE (0x0800)
#define DRAW_OPT_CLEAR_EOL (0x1000)
```

Módulo Display

Función **NumOut**

```
char NumOut (int x, int y, variant value, unsigned long
options=DRAW_OPT_NORMAL)
```

Dibuja el valor numerico en la coordenada (x,y) de la pantalla. El valor de y debe ser multiplo de 8.

Parámetros:

- x**: Es el valor de la coordenada de inicio de la salida del texto en el LCD
- y**: Es el número de línea para la salida del texto en el LCD
- value**: El valor a desplegar en el LCD, cualquier tipo de numero
- options**: Opción de dibujo del texto.

Módulo Display

Función PointOut

```
char PointOut (int x, int y, unsigned long
options=DRAW_OPT_NORMAL)
```

Dibuja un punto en la coordenada (x,y) de la pantalla.

Parámetros:

x: Es el valor de la coordenada en x

y: Es el valor de la coordenada en y

options: Opción de dibujo del punto.

Módulo Display

Función LineOut

```
char LineOut (int x1, int y1, int x2, int y2, unsigned long
options=DRAW_OPT_NORMAL)
```

Dibuja una línea en la pantalla desde la coordenada (x1,y1) a (x2,y2)

Parámetros:

x1, y1: Es el valor de la coordenada inicial

x2, y2: Es el valor de la coordenada final

options: Opción de dibujo.

Módulo Display

Función CircleOut

```
char CircleOut (int x, int y, byte radio, unsigned long
options=DRAW_OPT_NORMAL)
```

Dibuja un circulo en la pantalla con centro en la coordenada (x,y) usando como radio el valor de **radio**.

Parámetros:

x1, y1: Es el valor x del centro del circulo.

x2, y2: Es el valor y del centro del circulo.

radio: El radio del circulo.

options: Opción de dibujo.

Módulo Display

Función **RectOut**

```
char RectOut (int x, int y, int ancho, int alto, unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja un rectángulo en la pantalla en la coordenada (x,y) usada como el tope de la esquina izquierda del rectángulo, con el **ancho** y **alto** especificado.

Parámetros:

x,y: Coordenada (x,y) de la esquina superior izquierda del rectángulo.

ancho: Es el valor del ancho del rectángulo

alto: Es el valor del alto del rectángulo

options: Opción de dibujo.

Módulo Display

Función **PolyOut**

```
char PolyOut (LocationType puntos[], unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja un polígono en la pantalla en la coordenada usando un arreglo de puntos.

Parámetros:

puntos: Es un arreglo de tipo LocationType que contiene los puntos que definen el polígono.

options: Opción de dibujo.

Ejemplo:

```
LocationType puntos[] = {16,16, 8,40, 32,52, 20,36, 52,36, 56,52, 64,32, 44,20, 24,20};  
PolyOut(puntos, DRAW_OPT_LOGICAL_XOR|DRAW_OPT_LOGICAL_XOR|  
DRAW_OPT_FILL_SHAPE);
```

Módulo Display

Función **EllipseOut**

```
char EllipseOut (int x, int y, int radioX, int radioY, unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja una elipse en la pantalla con centro en la coordenada (**x**, **y**) usando los radios **radioX** y **radioY**.

Parámetros:

x,y: El valor de la coordenada del centro, x y y respectivamente

radioX: Radio de en x.

radioY: Radio de en y.

options: Opción de dibujo.

Módulo Display

Función **FontTextOut**

```
char FontTextOut (int x, int y, string archivo, string cadena, unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja un texto en la pantalla en la posición (**x**, **y**) usando una tradicional fuente RIC.

Parámetros:

x,y: El valor de la coordenada donde se inicia el texto, x y respectivamente

archivo: El nombre del archivo con la fuente RIC.

cadena: El texto para desplegarlo en la pantalla LCD.

options: Opción de dibujo.

Módulo Display

Función **FontNumOut**

```
char FontNumOut (int x, int y, string archivo, variant valor, unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja un valor numérico en la pantalla en la posición (**x**, **y**) usando una tradicional fuente RIC.

Parámetros:

x,y: El valor de la coordenada donde se inicia el texto, x y respectivamente

archivo: El nombre del archivo con la fuente RIC.

valor: El valor numérico para ser desplegado en la pantalla LCD.

options: Opción de dibujo.

Módulo Display

Función **GraphicOut**

```
char GraphicOut (int x, int y, string archivo, unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja una imagen gráfica tomada de un archivo en la pantalla en la posición (**x**, **y**).

Parámetros:

x,y: El valor de la coordenada donde se colocará la imagen.

archivo: El nombre del archivo de la imagen gráfica RIC.

options: Opción de dibujo.

Módulo Display

Función **GraphicArrayOut**

```
char GraphicArrayOut (int x, int y, byte datos[], unsigned long options=DRAW_OPT_NORMAL)
```

Dibuja una imagen gráfica a partir de un arreglo de bytes.

Parámetros:

x,y: El valor de la coordenada donde se colocará la imagen.

datos: El arreglo de bytes de la imagen gráfica RIC.

options: Opción de dibujo.

Módulo Display

Funciones: **ResetScreen**, **ClearScreen**, **ClearLine**

```
void ResetScreen ()
```

Reinicia la pantalla LCD.

```
void ClearScreen ()
```

Limpia la pantalla LCD, coloca una pantalla en blanco.

```
char ClearLine (bye linea)
```

Esta función limpia una sola línea en la pantalla.

Parámetro:

linea: Es la línea que quieras borrar, use las constantes de líneas.

Módulo Button

Este módulo está relacionado con los cuatro botones del NXT.

Está formado por un conjunto de constantes asociadas a los botones (offsets, asociadas a los botones, usadas con la función **ButtonState**), funciones para accesar y modificar las características de los botones, y los tipos de datos usados por los botones.

Constantes

```
#define BTN1 0
#define BTN2 1
#define BTN3 2
#define BTN4 3
#define BTNEXTIT BTN1
#define BTNRIGHT BTN2
#define BTNLEFT BTN3
#define BTNCENTER BTN4
#define NO_OF_BTNS 4
```

Módulo Button

Función ButtonPressed

bool ButtonPressed (const byte btn, bool resetCount=false)
 Verifica si se presionó el botón, se puede además reiniciar el contador de presionado.

Parámetros:

btn: El botón a verificar, se puede aplicar las constantes.

resetCount: Bandera para reiniciar o no el contador, default false.

Retorna un valor indicando si se presionó o no el botón.

Ejemplo:

```
while (true)
{
    NumOut(0, LCD_LINE2, ButtonPressed (BTNRIGHT, false));
    NumOut(0, LCD_LINE3, ButtonPressed (BTNLEFT, false));
}
```

Módulo Button

Función ButtonCount

byte ButtonCount (const byte btn, bool resetCount=false)
 Obtiene la cantidad de veces que se presionó el botón, opcionalmente limpiar el contador después de leerlo.

Parámetros:

btn: El botón a verificar, se puede aplicar las constantes.

resetCount: Bandera para reiniciar o no el contador, default false.

Retorna el contador.

Ejemplo:

```
valor= ButtonCount (BTN1, true);
```

Módulo Button

Función ReadButtonEx

bool ReadButtonEx (const byte btn, bool reset, bool &pressed, unsigned int &count)

Lee información de un botón específico, coloca en los parámetros **pressed** y **count** el estado actual de dicho botón, opcionalmente se puede reiniciar (**reset**) el estado de presionado después de la lectura.

Parámetros:

btn: El botón a verificar, se puede aplicar las constantes.

reset: Reiniciar o no el contador de presionado.

pressed: El estado del botón si es presionado

count: El contador que indica las veces que fue presionado el botón

Módulo Button

Función **ButtonPressCount**

byte ButtonPressCount (const byte btn)

Obtiene la cantidad de veces que se presiono el botón.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el contador.

Módulo Button

Función **ButtonLongPressCount**

byte ButtonLongPressCount (const byte btn)

Obtiene la cantidad de veces que se presiono el botón en formato largo.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el contador en formato largo.

Módulo Button

Función **ButtonShortReleaseCount**

byte ButtonShortReleaseCount (const byte btn)

Obtiene la cantidad de veces que se libero el botón en formato corto.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el contador en formato corto.

Módulo Button

Función **ButtonLongReleaseCount**

byte ButtonLongReleaseCount (const byte btn)

Obtiene la cantidad de veces que se libero el botón en formato largo.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el contador en formato largo.

Módulo Button

Función **ButtonReleaseCount**

byte ButtonReleaseCount (const byte btn)

Obtiene la cantidad de veces que se libero el botón.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el contador.

Módulo Button

Función **ButtonState**

byte ButtonState (const byte btn)

Obtiene el estado del botón especificado.

Parámetro:

btn: El botón a verificar, se puede aplicar las constantes.

Retorna el estado.

Módulo Button

Función **SetButtonLongPressCount**

void **SetButtonLongPressCount** (const byte **btn**, const byte **n**)

Fija el contador de presión en formato largo del botón especificado.

Parámetros:

btn: El número de botón.

n: El nuevo valor del contador.

Módulo Button

Función **SetButtonLongReleaseCount**

void **SetButtonLongReleaseCount** (const byte **btn**, const byte **n**)

Fija el contador de liberación de formato largo del botón especificado.

Parámetros:

btn: El número de botón.

n: El nuevo valor del contador.

Módulo Button

Función **SetButtonPressCount**

void **SetButtonPressCount** (const byte **btn**, const byte **n**)

Fija el contador de presión del botón especificado.

Parámetros:

btn: El número de botón.

n: El nuevo valor del contador.

Módulo Button

Función **SetButtonReleaseCount**

void SetButtonReleaseCount (const byte btn, const byte n)

Fija el contador de liberación del botón especificado.

Parámetros:

btn: El número de botón.

n: El nuevo valor del contador.

Módulo Button

Función **SetButtonShortReleaseCount**

void SetButtonShortReleaseCount (const byte btn, const byte n)

Fija el contador de liberación en formato corto del botón especificado.

Parámetros:

btn: El número de botón.

n: El nuevo valor del contador.

Módulo Button

Función **SetButtonState**

void SetButtonState (const byte btn, const byte state)

Fija el estado del botón especificado.

Parámetros:

btn: El botón a fijar, se puede aplicar las constantes.

state: El nuevo estado del botón.

Módulo Play

Este módulo se encarga de proporcionar soporte de reproducción de tonos (archivos de sonido .rso y archivos melody) en el NXT por medio de dos diferentes tipos de archivos.

Cuando un sonido o un archivo es reproducido, la ejecución no espera a que finalice la anterior ejecución de sonido, entonces para reproducir múltiples tonos o secuencia de archivos es necesario esperar a que se termine el primero sonido emitido, esto puede lograrse por medio del uso de la función **Wait** o con el valor del estado del sonido en un ciclo while.

El API NXC define constantes de frecuencia y duración, las cuales pueden ser utilizadas en los llamados **Playtone** o **PlayToneEx**. Las constantes de frecuencia inician con **TONE_A3** (la A de tono en octavo 3) y va a **TONE_B7** (la B de tono en octavo 7). Las constantes de duración inician con **MS_1** (1 milisegundo) y va a **MIN_1** (60000 milisegundos).

Módulo Play

Constantes para ser usada con la función **SoundState()**.

```
#define SOUND_STATE_IDLE 0x00
#define SOUND_STATE_FILE 0x02
#define SOUND_STATE_TONE 0x03
#define SOUND_STATE_STOP 0x04
```

byte SoundState ()

Obtiene el estado del módulo de sonido. Retorna el actual estado del modulo de sonido.

void SetSoundModuleState (byte state)

Fija el estado del módulo de sonido.

Módulo Play

char PlayFile (string filename)

Reproduce un archivo. El archivo puede ser un RSO o un RMD.

Ejemplo:

PlayFile ("starup.rso");

char PlayFileEx (string filename, byte volume, bool loop, unsigned int sr=0)

Reproduce un archivo con opciones extras. El volumen debe ser un número entre 0 (silencio) a 4 (ruidoso). Reproduce el archivo repetidamente si el loop es true.

Ejemplo:

PlayFileEx ("starup.rso", 2, true);

Módulo Play

char PlayTone (unsigned int frequency, unsigned int duration)

Reproduce un tono con una frecuencia (en Hz) y duración especificada (en ms).

Ejemplo:

PlayTone (440, 1000);

Módulo Play

char PlayToneEx (unsigned int frequency, unsigned int duration, byte volume, bool loop)

Reproduce un tono con frecuencia (Hz), duración(ms) y volumen (0..4) específico, si bool es true el sonido es repetitivo.

Ejemplo:

PlayToneEx (400,400,2,false);

byte SoundState ()

Obtiene el estado del modulo de sonido.

Ejemplo:

x=SoundState();

Módulo Play

Los valores que puede retornar están asociados a las siguientes constates

```
#define SOUND_STATE_IDLE 0x00 //Reiniciar (SOUND_UPDATE)
#define SOUND_STATE_FILE 0x02 //Procesando un archivo de
sonido o melodía
#define SOUND_STATE_TONE 0x03 //Procesando un tono
#define SOUND_STATE_STOP 0x04 //Paro del sonido inmediato y
cierre del hardware
```

Módulo Play

byte SoundFlags()

Obtiene las banderas del modulo de sonido.

Ejemplo:

x=SoundFlags();

byte StopSound ()

Detiene un sonido.

Ejemplo:

StopSound();

unsigned int SoundFrequency ()

Obtiene la frecuencia de sonido.

Ejemplo:

x=SoundFrequency();

Módulo Play

unsigned int SoundDuration()

Obtiene la duración del sonido.

Ejemplo:

x=SoundDuration();

unsigned int SoundSampleRate()

Obtiene la frecuencia de muestreo.

Ejemplo:

x=SoundSampleRate();

byte SoundMode()

Obtiene el modo del sonido.

Ejemplo:

x=SoundMode();

Módulo Play

Los valores que puede retornar son:

```
#define SOUND_MODE_ONCE 0x00 // Reproduce solo una vez
#define SOUND_MODE_LOOP 0x01 // Reproduce el archivo
hasta que se escriba SOUND_STATE_STOP en SoundState
#define SOUND_MODE_TONE 0x02 // Reproduce el tono
específico en la frecuencia con una duración de ms
```

byte SoundVolume ()

Obtiene el volumen.

Ejemplo:

x=SoundVolume();

void SetSoundDuration (unsigned int duration)

Fija la duración del sonido.

Ejemplo:

SetSoundDuration(500);

Módulo Play

void SetSoundFlags (byte flags)

Fija las banderas del modulo de sonido. Las Constantes usadas en esta función son:

```
#define SOUND_FLAGS_IDLE 0x00 // Sonido Inactivo
#define SOUND_FLAGS_UPDATE 0x01 // Actualiza los cambios
#define SOUND_FLAGS_RUNNING 0x02 //Procesando un tono o archivo
```

void SetSoundFrequency (unsigned int frequency)

Fija la frecuencia de sonido.

Ejemplo:

```
SetSoundFrequency (400);
```

Módulo Play

void SetSoundMode (byte mode)

Fija el modo del sonido, utilizando las constates de modo.

Ejemplo:

```
SetSoundMode(SOUND_MODE_ONCE);
```

void SetSoundModeState(byte state)

Fija el estado del modulo de sonido.

void SetSoundSampleRate(unsigned int sampleRate)

Fija la frecuencia de muestreo.

Ejemplo:

```
SetSoundSampleRate (4000);
```

void SetSoundVolume(byte volume)

Fija el volumen del sonido.

Ejemplo:

```
SetSoundVolume (3);
```

Módulo Play

void SysSoundPlayFile (SoundPlayFileType &args)

Reproduce un archivo de sonido, que se pasa vía la estructura siguiente.

```
struct SoundPlayFile_Type {
    char Result; // siempre es NO_ERR
    string Filename;
    bool Loop;
    byte SoundLevel;
};
```

Módulo Play

Ejemplo:

```
task main()
{
    SoundPlayFile argu;
    argu.Filename = "hello.rso";
    argu.Loop = false;
    argu.SoundLevel = 3;
    SysSoundPlayFile (argu);
}
```

Módulo Play

void **SysSoundPlayTone** (**SoundPlayToneType** &args)

Reproduce un tono, con las características pasadas vía la estructura siguiente.

```
struct SoundPlayToneType {
    char Result;
    unsigned int Frequency;
    unsigned int Duration;
    bool Loop;
    byte SoundLevel;
};
```

Módulo Play

Ejemplo:

```
task main()
{
    SoundPlayToneType argu;
    argu.Frequency = 400;
    argu.Duration = 2000;
    argu.Loop = false;
    argu.SoundLevel = 3;
    SysSoundPlayTone (argu);
}
```

Módulo Play

```
void SysSoundGetState ( SoundGetType &args)
Obtiene el estado del sonido, vía la siguiente estructura
struct SoundGetType {
    byte State;
    byte Flags;
};
```

Ejemplo:

```
task main ()
{
    SoundGetType argu;
    SysSoundGetState (argu);
    if (argu.State == SOUND_STATE_IDLE)
        { /* realizar algo */ }
}
```

Módulo Play

```
void SysSoundSetState ( SoundSetStateType &args)
Fija el estado del sonido, vía la siguiente estructura
struct SoundSetStateType {
    byte Result;
    byte State;
    byte Flags;
};
```

Ejemplo:

```
task main ()
{
    SoundSetStateType argu;
    argu.State = SOUND_STATE_STOP;
    SysSoundSetState (argu);
}
```

Módulo Play

```
void PlaySound(const int &aCode)
Reproduce un sonido del sistema, usando las siguientes
constantes:
```

aCode	Resultado del Sonido
SOUND_CLICK	Sonido de la tecla click
SOUND_DOUBLE_BEEP	Doble beep
SOUND_DOWN	Barrido hacia abajo
SOUND_UP	Barrido hacia arriba
SOUND_LOW_BEEP	Sonido de error
SOUND_FAST_UP	Barrido rápido

Módulo Play

Ejemplo:

```
task main()
{
    PlaySound (SOUND_UP);
    PlaySound (SOUND_DOWN );
    Wait (1000);
    PlaySound (SOUND_LOW_BEEP);
    Wait (1000);
    PlaySound (SOUND_FAST_UP);
}
```

Módulo Play

void **PlayTones** (Tone tones[])

Reproduce múltiples tonos vía los elementos colocados en el arreglo que serán instancias de la estructura siguiente

```
struct Tone {
    unsigned int Frequency;
    unsigned int Duration;
};
```

Módulo Play

Ejemplo:

```
Tone tonos [] = {
    TONE_C4, MS_50,
    TONE_E4, MS_50,
    TONE_G4, MS_50,
    TONE_C5, MS_50,
    TONE_E5, MS_50,
    TONE_G5, MS_50,
    TONE_C6, MS_50,
};

task main()
{
    PlayTones (tonos);
    Wait (1000);
}
```

Módulo Play

Constantes

```
#define TONE_C4 262 // Cuarto octavo C
#define MS_50 50 // 50 milisegundos
#define TONE_E4 330 // Cuarto octavo de E
#define TONE_G4 392
#define TONE_C5 523
#define TONE_E5 659
#define TONE_G5 784
#define TONE_C6 1047
```

Módulo Input

El módulo input del NXT abarca todos los sensores de entrada excepto los sensores para digital I2C (LowSpeed).

Existen cuatro sensores, los cuales internamente son numerados **0, 1, 2** y **3**. Es un poco confuso dado que externamente se etiquetan como **1, 2, 3** y **4**. Lo que ayuda a mitigar esta confusión, son los nombres de los puertos de los sensores, los cuales se definen como **S1, S2, S3** y **S4**.

Módulo Input

Los nombres de sensores pueden ser usados en cualquier función que requiera el puerto del sensor como argumento, estos nombres de constantes son **IN_1, IN_2, IN_3** e **IN_4**.

Cuando se necesita adquirir los valores de los sensores se utiliza las siguientes constantes: **SENSOR_1, SENSOR_2, SENSOR_3** y **SENSOR_4**. Estos nombres pueden además ser usados siempre que un programa desea leer el valor actual del sensor analógico, por ejemplo:

```
x= SENSOR_1; // lee sensor y almacena en x el valor
```

Módulo Input

Constantes

Las constantes son parte del módulo de entrada del firmware NXT.

Color calibration constants. Constantes para usarse con las funciones de calibración de color.

Color calibration state constants. Constantes para usarse con las funciones de estado de calibración del color.

Color sensor array indices. Constantes para usarse con los arreglos de valores del sensor de color, indexando valores de retorno RGB y blanco.

Color values. Constantes para usarse con ColorValue retornados por el sensor de color en el modo a todo color.

Constants to use with the Input modules Pin function. Constantes para usarse con la funciones del modelo Pin de entrada.

Input field constants. Constantes para usarse con **SetInput()** y **GetInput()**.

Módulo Input

Constantes

Input module IOMAP offsets. Constantes offsets dentro de la estructura IOMAP del módulo de entrada.

Input port constants. Las constantes del puerto de entrada son usadas cuando se invoca a las funciones de control API del sensor NXC.

Input port digital pin constants. Constantes para usarse cuando se controla directamente o se lee el estado del pin digital del puerto.

NBC Input port constants. Las constantes del puerto de entrada son usadas cuando se invoca a las funciones de control API del sensor..

Sensor types and modes. Constantes que son usadas para definir el tipo y modo del sensor.

Macros

```
#define INPUT_CUSTOMINACTIVE 0x00
#define INPUT_CUSTOM9V 0x01
#define INPUT_CUSTOMACTIVE 0x02
#define INPUT_INVALID_DATA 0x01
```

Módulo Input

Constantes

Las constantes de puerto de Input son usadas cuando se invocan a las funciones del API de control del sensor NXC.

Macros

```
#define S1 0 // entrada puerto 1
#define S2 1 // entrada puerto 2
#define S3 2 // entrada puerto 3
#define S4 3 // entrada puerto 4
```

Módulo Input

Funciones

Funciones para accesar y modificar características del módulo input.

`void SetSensorType (const byte &port, byte type)`. Fija el tipo de sensor.

`void SetSensorMode (const byte &port, byte mode)`. Fija el modo del sensor.

`void ClearSensor (const byte &port)`. Limpia el valor del sensor.

`void ResetSensor (const byte &port)`. Reinicia el puerto del sensor.

`void SetSensor (const byte &port, const unsigned int config)`. Fija la configuración del sensor.

`void SetSensorTouch (const byte &port)`. Configura el sensor de tacto.

`void SetSensorLight (const byte &port, bool bActive=true)`. Configura el sensor de luz.

Módulo Input

Funciones

`void SetSensorSound (const byte &port, bool bdBScaling=true)`. Configura el sensor en un puerto específico como un sensor de sonido.

Parámetros:

`port`. El puerto a configurar.

`bdBScaling`. Una bandera booleana que indica si se debe configurar el puerto en dB o escala dBA. El valor predeterminado es true, es decir, dB.

Ejemplo:

`SetSensorSound(S1);`

`void SetSensorLowspeed (const byte &port, bool blsPowered=true)`. Configura el sensor I2C.

Módulo Input

Funciones

`void SetSensorUltrasonic (const byte &port)`. Configura un sensor ultrasonico.

`void SetSensorEMeter (const byte &port)`. Configura un sensor EMeter.

`void SetSensorTemperature (const byte &port)`. Configura un sensor de temperatura.

`void SetSensorColorFull (const byte &port)`. Configura un sensor NXT 2.0 a todo color.

`void SetSensorColorRed (const byte &port)`. Configura un sensor NXT 2.0 de luz roja.

`void SetSensorColorGreen (const byte &port)`. Configura un sensor NXT 2.0 de luz verde.

`void SetSensorColorBlue (const byte &port)`. Configura un sensor NXT 2.0 de luz azul.

`void SetSensorColorNone (const byte &port)`. Configura un sensor NXT 2.0 sin luz.

Módulo Input

Funciones

variant **GetInput** (const byte &port, const byte field). Obtiene un valor del campo de entrada.

void **SetInput** (const byte &port, const int field, variant value). Fija un valor del campo de entrada.

unsigned int **Sensor**(const byte &port). Lee el valor de escala del sensor.

bool **SensorBoolean** (const byte port). Lee el valor booleano del sensor.

byte **SensorDigiPinsDirection** (const byte port). Lee la dirección del sensor pins digital.

byte **SensorDigiPinsOutputLevel** (const byte port). Lee el nivel de salida del sensor pins digital.

byte **SensorDigiPinsStatus** (const byte port). Lee el estado del sensor pins digital.

bool **SensorInvalid** (const byte &port). Lee el sensor de bandera de datos no valido.

Módulo Input

Funciones

byte **SensorMode** (const byte &port). Lee el modo del sensor.

unsigned int **SensorNormalized** (const byte &port). Lee el valor normalizado del sensor.

unsigned int **SensorRaw** (const byte &port). Lee el valor sin tratar del sensor.

unsigned int **SensorScaled** (const byte &port). Lee el valor escala del sensor.

byte **SensorType** (const byte &port). Lee el tipo de sensor.

unsigned int **SensorValue** (const byte &port). Lee el valor escala del sensor.

bool **SensorValueBool** (const byte port). Lee el valor booleano del sensor.

Módulo Input

Funciones

unsigned int **SensorValueRaw** (const byte &port). Lee el valor sin tratar del sensor.

byte **CustomSensorActiveStatus** (byte port). Obtiene el estado activo del sensor personalizado.

byte **CustomSensorPercentFullScale** (byte port). Obtiene la escala completa del porcentaje del sensor personalizado.

unsigned int **CustomSensorZeroOffset** (byte port). Obtiene el offset zero del sensor personalizado.

void **SetCustomSensorActiveStatus** (byte port, byte activeStatus). Fija estados activos.

void **SetCustomSensorPercentFullScale** (byte port, byte pctFullScale). Fija la escala completa de porcentaje.

Módulo Input

Funciones

`void SetCustomSensorZeroOffset (byte port, int zeroOffset)`. Fija el offeset zero personalizado..

`void SetSensorBoolean (byte port, bool value)`. Fija el valor booleano del sensor.

`void SetSensorDigiPinsDirection (byte port, byte direction)`. Fija la dirección de los pins digitales.

`void SetSensorDigiPinsOutputLevel (byte port, byte outputLevel)`. Fija el nivel de salida de los pins digitales.

`void SetSensorDigiPinsStatus (byte port, byte status)`. Fija el estado de los pins digitales.

Módulo Input

Funciones

`void SysColorSensorRead (ColorSensorReadType &args)`. Lee el sensor de color LEGO.

`int ReadSensorColorEx (const byte &port, int &colorval, unsigned int &raw[], unsigned int &norm[], int &scaled[])`. Lee el sensor de color extra LEGO.

`int ReadSensorColorRaw (const byte &port, unsigned int &rawVals[])`. Lee los valores crudos del sensor de color LEGO.

`unsigned int ColorADRaw (byte port, byte color)`. Lee el valor crudo del sensor AD de color LEGO.

`bool ColorBoolean (byte port, byte color)`. Lee el valor booleano del sensor de color LEGO.

Módulo Input

Funciones

`long ColorCalibration (byte port, byte point, byte color)`. Lee el valor de punto de calibración del sensor de color LEGO.

`byte ColorCalibrationState (byte port)`. Lee el estado de calibración del sensor de color LEGO.

`unsigned int ColorCallLimits (byte port, byte point)`. Lee el valor límite del sensor de calibración de color LEGO.

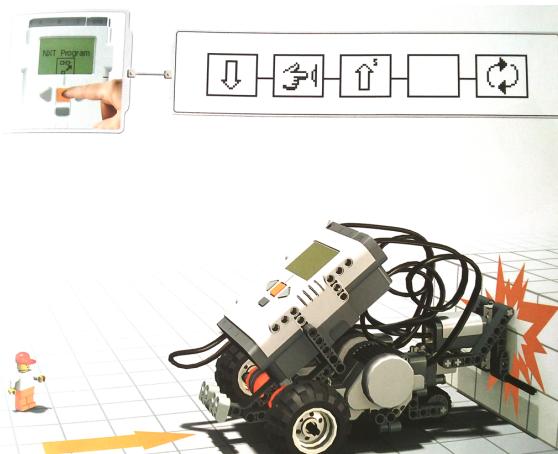
`unsigned int ColorSensorRaw (byte port, byte color)`. Lee el valor crudo del sensor de color LEGO.

`unsigned int ColorSensorValue (byte port, byte color)`. Lee el valor de escala del sensor de color LEGO.

`void SysInputPinFunction (InputPinFunctionType &args)`. Ejecuta la función del modulo pin de entrada.

SENSORES

Sensor de Tacto



- Se instala en la entrada 1 del NXT.

Práctica II



```
task main ()
{
    int cont=0;

    SetSensorTouch(IN_1);
    while (cont < 3)
    {
        OnRev (OUT_AB, 75);
        if (SENSOR_1 == 1)
        {
            OnFwd (OUT_AB, 75);
            Wait (300);
            cont +=1;
        }
    }
    Off(OUT_AB);
}
```

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Práctica 12

```
task main ()
{
int cont=0, ban=0;
SetSensor (IN_1, SENSOR_TOUCH);
SetSensor (IN_2, SENSOR_TOUCH);
while (cont < 6)
{
if (ban==0)
OnFwd (OUT_AB, 75);
else
OnRev (OUT_AB,75);
if (SENSOR_1 == 1)
{
OnRev (OUT_AB, 75);
Wait (300);
cont +=1;
ban=1;
}
if (SENSOR_2==1)
{
OnFwd (OUT_AB, 75);
Wait (300);
cont +=1;
ban=0;
}
}
Off(OUT_AB);
```



Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Sensor Ultrasonico



```
byte SensorUS ( const byte port )
```

Lee el valor del sensor Ultrasonico. Antes de esta función se debe configurar el puerto como Lowspeed. Esta función incluye 15 milisegundos de espera antes de la lectura del sensor.

```
void SetSensorLowspeed ( const byte & port, bool blsPowered = true )
```

Configura un sensor I2C (Ultrasonico).

Laboratorio Edumóvil. M.C. Gabriel Gerónimo C.

Práctica 13

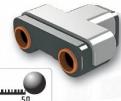


Sensor Ultrasonico

```
#define UMBRAL 70
```

```
task main()
{
SetSensorLowspeed(IN_4)
while (true)
{
OnFwd (OUT_BC,75);
until (SensorUS (IN_4) < UMBRAL);
Wait (1000);
Off(OUT_BC);
OnRev(OUT_C,80);
Wait (800);
}
```

Práctica 14



```
#define UMBRAL 40

task main()
{
    SetSensorLowspeed(IN_3);
    while (true)
    {
        Wait(500);
        ClearScreen();
        NumOut(0,0,SensorUS(IN_3));
        if (SensorUS(IN_3)<UMBRAL)
            Off(OUT_BC);
        else
            OnFwd(OUT_BC,50);
    }
}
```

Utilizando el sensor ultrasonico, muestra en pantalla la distancia a un objeto, si está cerca de dicho objeto (40cm) apaga los motores, sino avanza hasta encontrar un obstáculo.

Sensor de Sonido



```
void SetSensorSound ( const byte & port, bool bdBScaling = true )
```

Configura el sensor de sonido en el puerto especificado.

Parámetros:

port. El puerto a configurar

bdBScaling. Bandera para indicar si el puerto se va a configurar el puerto como sonido en escala dB o dBA. El valor por omisión es true que indica que se configura en dB

```
#define SENSOR_2 Sensor(S2)
```

Constante para leer el valor del sensor en el puerto 2

Práctica 15



Sensor de Sonido

```
#define UMBRAL 75
task main()
{
    SetSensorSound (IN_2);
    do
    {
        until (SENSOR_2 > UMBRAL);
        OnFwd (OUT_BC, 75);
        Wait (1000);
        ResetSensor(IN_2);
        until (SENSOR_2 > UMBRAL);
        OnFwd (OUT_BC, 75);
        Wait(100);
        Off (OUT_BC);
    } while (true);
}
```

Por medio del sensor de sonido se mueve con un sonido mayor de 75 db, y se detiene con un sonido menor de 75db

Sensor de Luz



El Sensor de Luz le permite dar visión al robot distinguiendo entre luz y oscuridad.

El Sensor es monocromático, es decir, puede distinguir entre el blanco y el negro pasando por una gama de grises, y su lectura la entrega en porcentaje.

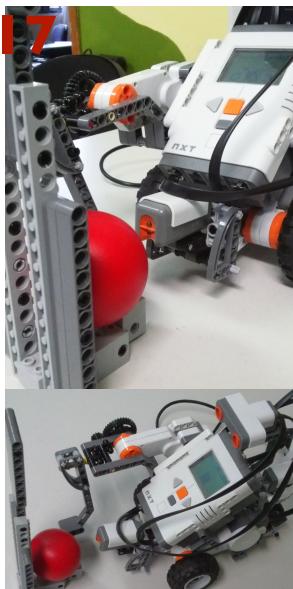
Para indicar en que entrada del NXT se colocará se utilizará la función **SetSensorLight(Puerto de entrada)**, después puede obtener su información por medio del macro **SENSOR_Puertodeentrada**

Práctica 16



```
#define UMBRAL 60
task main()
{
    SetSensorLight (IN_3);
    OnFwd (OUT_BC, 20);
    While (true)
    {
        if (Sensor (IN_3) > UMBRAL)
        {
            OnRev (OUT_C, 30);
            Wait (100);
            until (Sensor (IN_3) <= UMBRAL);
            OnFwd (OUT_BC, 30);
        }
    }
}
```

Práctica 17



```
#define STOP 20
#define ROJO 40
task main()
{
    SetSensorLight(IN_2);
    SetSensorLowspeed(IN_1);
    while(true)
    {
        if (SensorUS(IN_1)<STOP)
        {
            Off(OUT_BC);
            if(SENSOR_2>ROJO)
                RotateMotor(OUT_A,50,90);
            else
                OnFwd(OUT_BC,5);
        }
        else
            OnFwd(OUT_BC,50);
    }
}
```

Concurrencia en el NXT

Tasks

NXT soporta multi-hilos generándolos por medio de una tarea. Por lo que una tarea en NXC corresponde a un hilo en el NXT.

Las tareas son definidas usando la palabra reservada **task** con la sintaxis siguiente:

```
task nombre()
{
    // código de la tarea
}
```

Donde el **nombre** de la tarea puede ser cualquier identificador legal. Un programa debe tener al menos una tarea, llamada **main()**, la cual inicia cuando se ejecuta el programa.

Tasks

Las tareas pueden ser iniciadas o paradas, usando las sentencias **task** y **stop**, respectivamente. Sin embargo, el mecanismo primario para iniciar tareas son las funciones API **Precedes** o **Follows**.

La función API para detener todas las tareas que se están ejecutando es **StopAllTasks**, aunque también puede detenerlas usando la función **Stop**.

Una tarea puede detenerse así misma usando la función **ExitTo**, o simplemente llegando al final de la ejecución de su código.

Tasks

```

task Musica() {
    while (true) {
        PlayTone (TONE_A4, 500);
        Wait(500);
    }
}

task Mover() {
    while (true) {
        OnFwd(OUT_BC, Random(100));
        Wait(Random(SEC_1));
    }
}

task Control() {
    Wait(SEC_10);
    stop music;
    Wait(SEC_5);
    StopAllTasks();
}

```

En el código de ejemplo, la tarea main esquematiza la tarea **Musica**, tarea **Mover** y la tarea **Control** antes de salir y permite que estas tres tareas inicien su ejecución de manera concurrente. La tarea **Control** espera 10 segundos antes de parar la tarea **Musica**, y entonces espera 5 segundos antes de parar todas las tareas y finalizar el programa.

Tasks

void Precedes (task tareal, task tarea2, ..., task tarean)

Declara las precedencias de las tareas. Esquematiza la lista de tareas a ser ejecutadas una vez que la actual tarea a completado su ejecución. Las tareas se ejecutan simultáneamente al menos que otras dependencias les impidan hacerlo. Esta declaración debe ser usada una vez dentro de la tarea - preferentemente al inicio de la definición de la tarea. Cualquier número de tareas puede ser listada en el estado de Precedes.

Parámetros

tarea1. Primer tarea a iniciar a ejecutar después que la tarea actual termine.

tarea2. Segunda tarea a iniciar a ejecutar después que la tarea actual termine.

tareaN. Ultima tarea a iniciar a ejecutar después que la tarea actual termine

Tasks

void StopAllTasks()

Para todas las tareas que se encuentran en ejecución. Esto debe parar el programa completamente, de esta forma el código que se coloca después de esta instrucción es ignorado.

Tasks

```
void StopTask (task t)
```

Para una tarea específica. El parámetro **t** indica la tarea a ser parada.
Debe tenerse en consideración que para usar esta función se requiere tener el firmware NBC/NXC mejorado.

Práctica 18

```
int bandera = 0;

task salir()
{
    while (true)
        if (ButtonPressed(BTN CENTER, false))
            bandera=1;
}

task mover()
{
    while(true) {
        OnFwd(OUT_BC, 75);
        Wait(3000);
        Off(OUT_BC);
        Wait(500);
        OnRev(OUT_BC, 50);
        Wait(2000);
        Off(OUT_BC);
        Wait(500);
    }
}
```

```
task main()
{
    start mover;
    start salir;
    while (true)
    {
        if (bandera) {
            StopAllTasks(); break;
        }
    }
}
```

Realiza dos tareas en forma concurrente (**mover** y **salir**). Al presionar el botón del centro enciende una bandera para detener las tareas.

Práctica 19

```
#define UMBRAL 50
int ultimo=0;
int dentro=0;

sub izq()
{
    OnFwd(OUT_C,50);
    OnRev(OUT_B,25);
}

sub der()
{
    OnFwd(OUT_B,50);
    OnRev(OUT_C,25);
}

sub avanzar()
{
    OnFwd(OUT_BC,50);
}
```

```
task sensores()
{
    SetSensorLight(IN_1);
    SetSensorLight(IN_4);
    while(true)
    {
        if (SENSOR_1 < UMBRAL)
            ultimo=0;
        if (SENSOR_4 < UMBRAL)
            ultimo=1;
        if (SENSOR_1 > UMBRAL &&
            SENSOR_4 > UMBRAL)
            dentro=0;
        else dentro=1;
    }
}
```



Práctica 19

```
task main()
{
start sensores;

while(true)
{
if (dentro==0)
{
if (ultimo==0) izq();
else der();
} else {
avanzar();
}
}
```

En esta práctica se muestra el código de un seguidor de línea usando dos sensores de luz. La rutina sensores, así como la principal se ejecutan en forma concurrente



Referencias



1. NXC: NXC Programmer's guide. <http://bricxcc.sourceforge.net/nbc/nxcdoc/nxcapi/consts.html>
2. BricxCC Command Center .<http://bricxcc.sourceforge.net/>
3. Daniele Benedettelli. Programming LEGO NXT Robots using NXC. 2007.
4. M. Gasperi, P. Hurbain. Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level. ISBN-13: 978-1-4302-2453-2. 2009
5. T. Griffin. The Art of LEGO MINDSTORMS NXT-G Programming. ISBN-10: 1-59327-218-9. 2010
6. JCLarsen. Using NXC on Linux. <http://www.jclarsen.dk/index.php/guides/21-programming/52-using-nxc-on-linux>

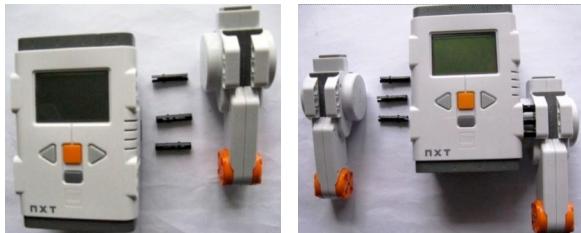


Los siguientes diseños de NXT fueron descargados de sitios de la Internet.

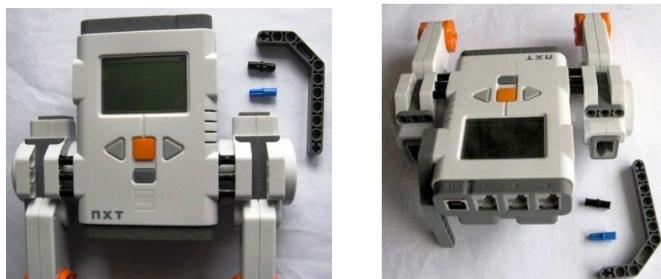
Robot 1



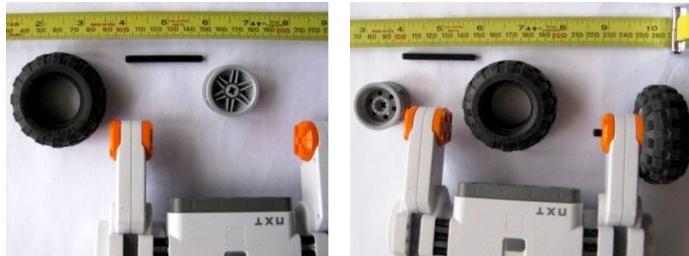
Robot 1 - Pasos



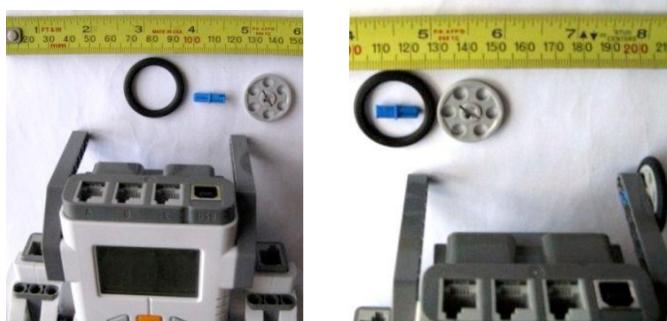
Robot 1 - Pasos



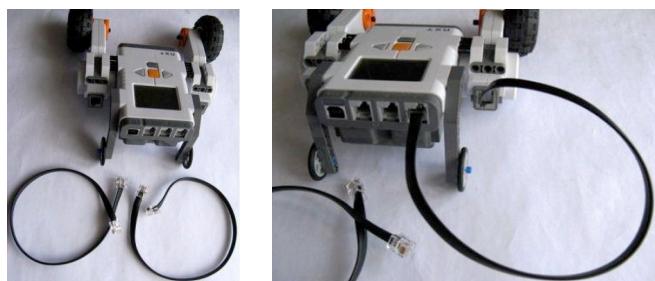
Robot 1 - Pasos



Robot 1 - Pasos



Robot 1 - Pasos



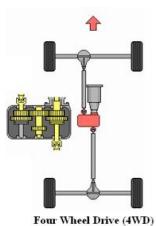
Robot 2



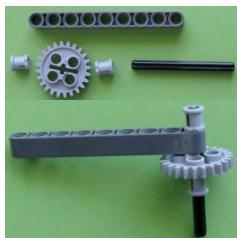
Robot 3



Robot 3 - Pasos



Robot 3 - Pasos



Robot 3 - Pasos



Robot 3 - Pasos



Robot 4



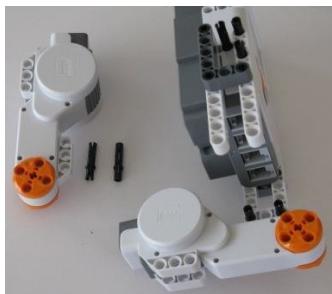
Robot 4 - Pasos



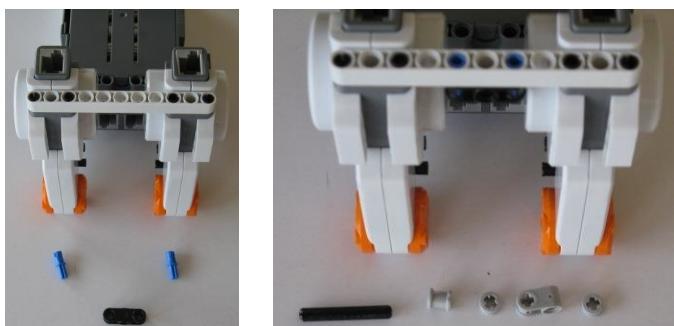
Robot 4 - Pasos



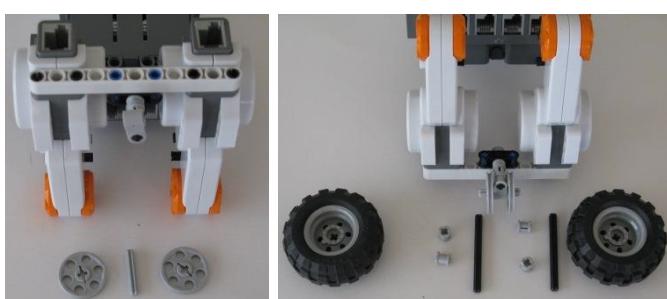
Robot 4 - Pasos



Robot 4 - Pasos



Robot 4 - Pasos



Robot 4 - Pasos



Robot 4 - Pasos



Robot 4 - Pasos

