Quick Reference Brickos guide :

Loops:

```
for(statement, condition, statement){
   //First statement executed on first iteration
   //Body executed while condition true
   //Last Statement executed after each completed
iteration
}
```

```
while(expression){
   //Body executed repeatedly while expression is true
}
```

```
do{
   //Body executed at least once, repeated while
   //expression is true
} while(expression)
```

Threads:

```
variable=start(&myfunc); //Starts a function of name
                         //myfunc as a thread
kill(variable);   //Kills a thread with tid_t of variable
kill_all(PRIO_NORMAL); //Kills all threads currently
                       //running
```

If:

```
if(expression){
   //Body executes if expression is true
}
else{
   //Body executes if expression is false
}
```

Sensors:

```
ds_active(&SENSOR_X);   //Set sensor port to active
ds_passive(&SENSOR_X); //Set sensor port to passive
```

```
SENSOR_X      //Raw sensor value
```

```
TOUCH_X       //Passive sensor value
```

```
LIGHT_X       //Light sensor value
```

```
ROTATION_X  //Rotation saneor value
```

```
ds_rotation_on(SENSOR_X);    //Start rotation counter
ds_rotation_off(SENSOR_X);   //Stop rotation counter
ds_rotation_set(SENSOR_X,Y);//Set rotation counter
                            //on port X to Y
```

Switch:

```
switch(expression){
   case value:
      //Body executes if expression ==value
      break;
   .....
  default:
```

LCD:

```
cls();    //Clear LCD
```

```
cputs("text");    //Write text to LCD
```

```
lcd_int(value);    //Write integer to LCD
```

```
lcd_refresh ( );  //Force a refresh of the screen
```

```
cputc('x','y'); //Write character x to
                //position y (1-5) of LCD
```

Sensors:

```
SENSOR_1
SENSOR_2
SENSOR_3
```

Sleeping:

```
sleep(x);       //Sleep for x seconds
```

```
msleep(x);     //Sleep for x milliseconds
```

Waiting:

```
Wakeup_t  myfunc (wakeup_t data){
          //Your code in here
            return(condition);
            }
…
wait_event(myfunc,0)

//When wait_event is called, program waits
//until Wakeup_t function condition is true
//before proceeding
```

```
get_battery_mv();  //Returns battery voltage in
                   //mV
```

Random numbers:

```
rand(x)   //Returns a random integer  number
          //between 0 and x
```

Main program:

```
Int main() {
//Your code here
return 0;
}
```

Sounds:

```
dsound_system(DSOUND_BEEP);
//Make RCX beep
```

```
static const note_t Music [] = {
{pitch,duration}, {pitch,duration},.....
{PITCH_END,0}
}
//Defines music to be played in an array
```

```
dsound_set_duration(x); //Sets time between
                        //notes in ms
```

```
dsound_finished ( );      //Returns true if
                          //sound has finished
                          //playing
```

```
dsound_playing ( )      //Returns true if sound
                        //has not finished
                        //playing
```

```
        //Default action if none of above cases match
        break;
}
```

Function declaration:

```
variable_type function_name(variable1, variable2...){

//Function body. Takes in variable1, variable2 .... when
called
//Then returns variable variable_type to calling
function(or void for //no return)

return variable_type;
}
```

Mathematical operations:

```
X=1;          //Assign a value
x=y+z;        //Add
x=y/z;        //Divide
x=y%z         //Modulo division (returns remainder
              //from division)
x++;          //Increment variable
x--;          //Decrement variable
x+=y;         //x=x+y shorcut
x*=y;         //x=x*y shortcut
x/=y;         //x=x/y shorcut
```

```
PITCH_A0      //Available pitches. Letter
PITCH_Am0     //denotes note, number denotes
PITCH_H0      //octave.'m' denotes a flat note.
PITCH_C1
PITCH_Cm1
PITCH_D1
PITCH_Dm1
PITCH_E1
PITCH_F1
PITCH_Fm1
PITCH_G1
PITCH_Gm1
```

```
WHOLE     //Available note durations
HALF
QUARTER
EIGTH
```

Defining Functions:

Before any function you have written can be used, it **must be defined at the top of your program**. This ensures that when you refer to it in your code, the computer knows what you are talking about.

We need to tell it the basics of the function in the definition:

Its name
What variables it takes in
What variables it passes out

So for instance we might have:

int adder (**int**,**int**);

as the definition for an adding function that takes in two integers, and returns the sum in of those two in a third integer variable.

If a function does not return a variable, we give it a **void** return type:

void function(**int**,**int**);

Function Structure:

variable0 function_name(variable1, variable2, ...){

//Function code goes in here

}

variable0 : returned to the calling function
variable1, variable2 .... : passed *to* the function from calling function

eg:

int adder(int a, int b){

return (a+b);

}

If we call this function. and pass it two numbers, it returns the sum:

int result;
result=adder(3,4)

result now holds the value 7

We can write our own functions, as above. However, every program must have a main function. This is the function that is executed first when the program is run:

```
int main(){

//Your code here

return 0;
}
```

EXAMPLE 1:

```
#include <conio.h>
#include <lcd.h>

void hello();

void hello(){
        cputs("HELLO");
}

int main(){
        hello();
}
```

Loops:

There are 3 ways of looping in BrickOS

while:

```
while(some condition is met){

//Do this stuff repeatedly

}
```

do-while:

```
do{

//Do this stuff at least once

}while(some condition is met)

for(initialise variable, variable condition, variable change){

//Do this stuff

}
```

eg:

```
for(int i = 0, i<10, i=i+1){
.....
}
```

EXAMPLE 2:

```
#include <conio.h>
#include <lcd.h>

void hello();
void world();

void hello(){
        cputs("HELLO");
}

void world(){
        cputs("WORLD");
}

int main(){

        while(1){
                hello();
                msleep(1000);
                world();
        }

}
```

Mathematical operators:

x=y+z
x=y/z
x=y*z
x=y%z (modulo division – for example, 3%2=1)

Shortcuts:

When we want a variable to act on itself, we can take shortcuts

x=x+1	x++
x=x-1	x--
x=x*y	x*=y
x=x/y	x/=y
x=x+y	x+=y
x=x-y	x-=y

EXAMPLE 3:

```
#include <conio.h>
#include <lcd.h>

int main(){

        int a;
        int b;

        a=4;
        b=6;

        a++;
        b--;
        lcd_int(a+b);
}
```

Logical Operators:

&&	AND
||	OR
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
TRUE	Condition is true
FALSE	Condition is false

!condition Condition is false

eg:
```
while(variable !> 3){
//Do something
}
```

IF:

if(condition is true)}
//Do something
}

else{
//Do something else
}

For the following example, connect two touch sensors to sensor ports 1 and 2 respectively:

EXAMPLE 4:

```
#include <conio.h>
#include <lcd.h>
#include<dsensor.h>

int main(){

        while(1){
                if(TOUCH_1 && TOUCH_2){
                        cputs("OUCH");
                }
        }
}
```

Switch:

This is tidier than using many if statements one after another.

```
switch(variable){

case 1:
//Do something if variable = 1
break;

case 2:
case 3:
case 6:
//Do something else if variable = 2, 3, or 6
break;

default:
//Do something if none of the above true
break;
}
```

Random numbers:

```
#include <random.h>

result=rand(x);
```

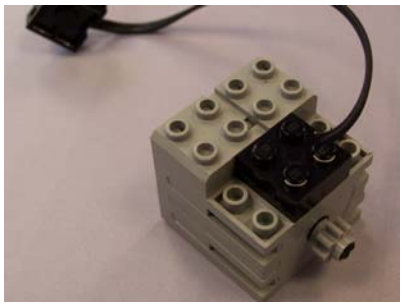
returns a random integer value between 0 and x and stores it in the variable 'result'

EXAMPLE 5:

```
#include <conio.h>
#include <lcd.h>
#include<random.h>

int main(){

        while(1){
                lcd_int(random(5));
                msleep(1000);
                }

}
```

Motor control:

#include<dmotor.h>
#include<motors.h>



motor_fwd(motor_number,motor_speed);
motor_rev(motor_number,motor_speed);

motor number ranges from 1-4

1-3 represent three motor ports, 4 represents all motors.
Speed is a value from 0 to 255

For the following example, connect a motor to port A

EXAMPLE 6:

```
#include<dmotor.h>
#include<motors.h>

int main(){
        motor_fwd(1,100);
        sleep(10000);
        motor_fwd(1,0);
return 0;
}
```

Waiting:

#include <unistd.h>
#include <tm.h>

Basic – good for pausing the program for a set amount of time

```
sleep(x);       //Sleep for x seconds

msleep(x);      //Sleep for x milliseconds
```

Advanced – good for pausing the program until something happens

First, we have the following function:

```
wakeup_t  myfunc (wakeup_t data){
```

```
//Your code in here
return(condition);
}
```

When we call this function, the program waits until 'condition' is true

```
wait_event(myfunc,0)
```

EXAMPLE 7:

```
#include <conio.h>
#include <lcd.h>
#include<dsensor.h>
#include <unistd.h>
#include <tm.h>


wakeup_t touch_wakeup(wakeup_t ignore)
{
 return (TOUCH_3);
}

int main(){
        while(1){
                cputs("HELLO");
                wait_event(touch_wakeup, 0);
                cputs("WORLD");
        }
return 0;
}
```

Multitasking:

#include <threads.h>

We often want to run more than one task at once. This is, unfortunately, impossible with only one processor. However, we can approximate this by switching tasks rapidly, giving each task a slice of the processor time. This is called threading, and each task is a thread.

To make use of threads in BrickOS, we first write a standard function.
Instead of calling it in the normal way, we call it as follows:

```
tid_t varible;                          //This variable lets us keep track of the thread
variable=start(&function_name); //This starts the thread running
...
.....
.........
kill(variable)                          //This stops the thread again
```

EXAMPLE 8:

```
#include <conio.h>
#include <lcd.h>
#include<dsensor.h>
#include <unistd.h>
#include <tm.h>

tid_t task1, task2;

mythread_1(){
        while(1){
                motor_fwd(1,100);
                msleep(1000);
                motor_rev(1,100);
        }
}

mythread_2(){
        while(1){
                cputs("HELLO");
                msleep(1000);
                cputs("WORLD");
        }
}

int main(){
        task1=start(&mythread_1);
        task2=start(&mythread_2);
        msleep(10000);
        while(!shutdown_requested()){
                msleep(100);
        }
        kill(task1);
        kill(task2);
        motor_fwd(1,0);
return 0;
}
```

Making Sounds

#include <dsensor.h>

We can play sounds using our robot. We need to use the following syntax - dont worry too much about the details of what it means for the moment:

static const note_t Music [] = {

{PITCH_C5, HALF}, {PITCH_C5, WHOLE},.....
{PITCH_END, QUARTER}
}

This function is our music. Each set of brackets defines a note and a period

EXAMPLE 9:

```
#include <config.h>
#include <dsound.h>
#include <tm.h>

static const note_t  Music[ ] =
  { {PITCH_C5, 3}, {PITCH_C5, 3}, {PITCH_C5, 2},
    {PITCH_D5, 1}, {PITCH_E5, 3}, {PITCH_E5, 2},
    {PITCH_D5, 1}, {PITCH_E5, 2}, {PITCH_F5, 1},
    {PITCH_G5, 6}, {PITCH_C6, 1}, {PITCH_C6, 1},
    {PITCH_C6, 1}, {PITCH_G5, 1}, {PITCH_G5, 1},
    {PITCH_G5, 1}, {PITCH_E5, 1}, {PITCH_E5, 1},
    {PITCH_E5, 1}, {PITCH_C5, 1}, {PITCH_C5, 1},
    {PITCH_C5, 1}, {PITCH_G5, 2}, {PITCH_F5, 1},
    {PITCH_E5, 2}, {PITCH_D5, 1}, {PITCH_C5, 6},
    { PITCH_END, 0 }
  };



int main() {

  dsound_set_duration(20);
  dsound_play(Music);
  wait_event(dsound_finished, 0);

  return 0;
}
```

Available Pitches:

PITCH_A0
PITCH_Am0
PITCH_H0
PITCH_C1
PITCH_Cm1
PITCH_D1
PITCH_Dm1
PITCH_E1
PITCH_F1
PITCH_Fm1
PITCH_G1
PITCH_Gm1

The final number represents the current octave. The small ‘m’ between note and octave denotes a flat. Note the use of the letter H to represent the note B ☺

Available Durations:

HALF
WHOLE
QUARTER EIGTH

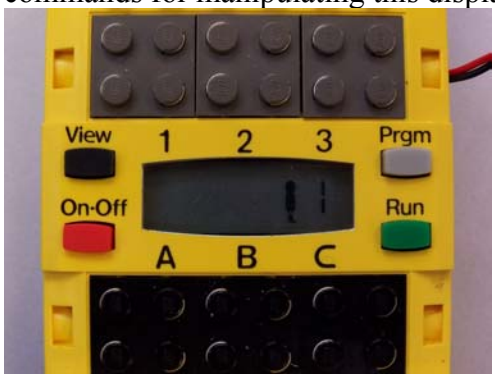
Then, to play our music, we can call it as follows:

```
dsound_set_duration(x);        //set how rapidly the notes are played
dsound_play(Music);
wait_event(dsound_finished, 0);
```

LCD:

```
#include <conio.h>
#include <lcd.h>
```

The RCX unit has an LCD screen we can use to display data. There are various commands for manipulating this display. The main ones are given below:



Writing numbers to the screen:

```
lcd_int(number);
```

Writing characters to the screen:

```
cputs("text_here");
```

Writing to a specific segment of the display:

```
cputs_x("a");          //Writes the character "a" to position x on the display,
                       //where x is between 0 and 5
```

Clearing the screen:

```
cls();
```

EXAMPLE 10:

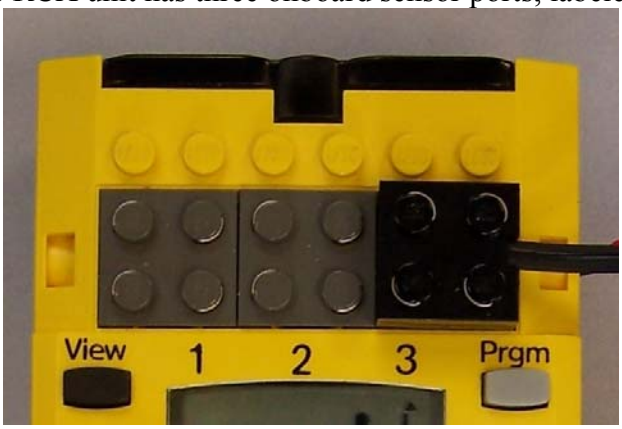
```
#include <conio.h>
#include <lcd.h>
#include <battery.h>


int main(){
        while(1){
                cputs("BATT");
                msleep(1000);
                lcd_int(get_battery_mv());
        }
return 0;
}
```

Sensors:

#include<desensor.h>

Setting up sensors:

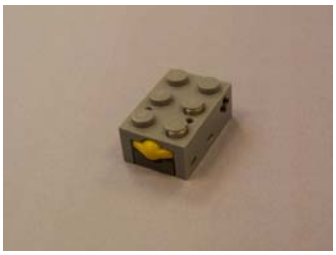
The RCX unit has three onboard sensor ports, labeled 1, 2, and 3.



Each port can be in one of two modes:

- Active..

- Passive....



Components such as Light Sensors require power to run – they need to run in active mode
Components such as switches are passive – they need no power to run

Active sensors work by charging a capacitor for a small time period, then letting the capacitor discharge throught the sensor. The value read by the sensor is the voltage remaining across the capacitor at the end f the measurement period – so the sensors are designed to discharge the capacitor at a rate proportional to the ehat they are sensing.

To set a sensor as active or passive we use the following commands:

```
ds_active(&SENSOR_X);
ds_passive(&SENSOR_X);
```

where X is a number between 1 and 3

When using rotation sensors, there are a few more options we can make use of:

```
ds_rotation_on(&SENSOR_X);     //Start rotation counter
ds_rotation_off(&SENSOR_X);    //Stop rotation counter
ds_rotation_set(&SENSOR_X,Y); //Set rotation counter
                                    //on port X to Y
```

Reading from Sensors:

To read from a sensor, we do the following:

```
int variable;

variable=SENSOR_X          //This stores the raw sensor value into "variable"

variable=TOUCH_X           //Reads either 0 or 1 into "variable. Use for touch
                           //sensors

variable=LIGHT_X           //Reads a light sensor. Value scaled relative to a preset
                           //level for bright light.

variable=ROTATION_X        //Reads a rotation sensor – useful for keeping track of
                            //how far a robot has gone by counting wheel
                            //revolutions.
```

Where X is 1, 2, or 3

For this example, a rotation sensor must be attached to port 1

EXAMPLE 11:

```
#include <conio.h>
#include <lcd.h>
#include <dsensor.h>


int main(){
        cputs("START");
        ds_rotation_on(&SENSOR_1);
        ds_rotation_set(SENSOR_1,0);

        while(ROTATION_1 < 100){
        sleep(100);
        }
        cputs("END");

return 0;
}
```

Multiplexing Sensors:

Although there are only 3 sensor ports, we can connect up to 6 sensors – 3 active and 3 passive.

This is because a passive sensor is always either on or off. However an active sensor, giving an analogue output, will almost never be in the fully on or fully off states.

So if the sensor is reading its maximum value, we know that the passive sensor has been triggered. Otherwise, we know the value read is that of the active sensor.

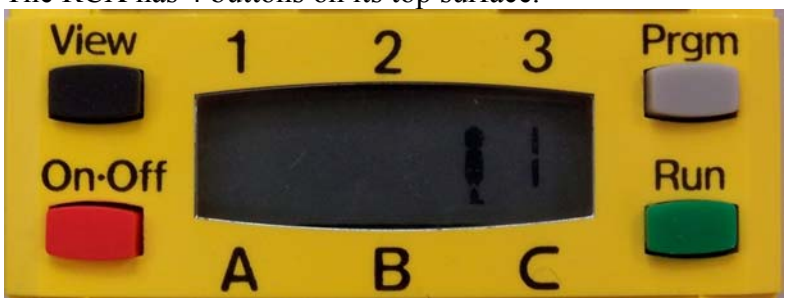
EXAMPLE 12:

```
#include <conio.h>
#include <lcd.h>
#include <dsensor.h>
#include <threads.h>
#include <dsound.h>

tid_t touch_task, light_task;

void touch(){
        while(1){
                if(TOUCH_1){
                    dsound_system(DSOUND_BEEP);
                }
        }
}
void light(){
        while(1){
                lcd_int(LIGHT_1);
        }
}
int main(){
        ds_active(&SENSOR_1);
        touch_task=start(&touch);
        light_task=start(&light);
        while(!shutdown_requested()){}
return 0;
```

Buttons:

#include<dbutton.h>

The RCX has 4 buttons on its top surface.



We can use these to interact with our robot:

PRESSED(dbutton(),buttontype) //This statement is true when buttontype is pressed
Or
RELEASED(dbutton(),buttontype)//This statement is true when buttontype is released

buttontype can be any of:

BUTTON_ONOFF
BUTTON_RUN

BUTTON _VIEW
BUTTON _PRGM
BUTTON _ANY

EXAMPLE 13:

```
#include <unistd.h>
#include <dbutton.h>
#include <conio.h>


int main()
{

  while(1){
if (PRESSED(dbutton(),BUTTON_PROGRAM)){
  cputs("OUCH");
}

if(RELEASED(dbutton(),BUTTON_PROGRAM)){
  cputs("PRESS");
}
  }

  return 0;
}
```

Detecting Shutdown:

Often, we will start a series of threads running in our code. If we then simply turn off the robot, we can corrupt the data and may have to re-download the firmware.

A solution is to have the main program waiting for a shutdown request, which is triggered by pressing the “on/off” button.

When this occurs, the main program can kill al threads, and then quit nicely:

To watch for a shutdown request, we use the **shutdown_requested()** function. This returns true if the power button has been pressed, false otherwise.

EXAMPLE 14:

```
int main(){

id1=start(&thread1);
id2=start(&thread2);

while(!shutdown_requested()){
msleep(100);
}

kill(id1);
kill(id2);

return;

}
```

Note the *msleep(100)* command. This is needed as without any content, the loop would run extremely fast, and would cause problems for out other threads.