
Dr. Barry Fagin
US Air Force Academy
barry.fagin@usafa.af.mil
1.0 Introduction

Ada/Mindstorms 2.0 is an Ada interface to the Lego RCX “brick”, included as part of the Lego Mindstorms™ Robotics Invention System. Programs are written with an Ada compiler, translated into Dave Baum’s Not Quite C language (NQC), then recompiled and downloaded through the Mindstorms IR transmitter to the RCX.

Ada/Mindstorms 2.0 is distributed with the USAFA AdaGIDE Windows Ada compiler. This document assumes that you have obtained and installed AdaGIDE. If you are using another Ada compiler, see Appendix B. We also assume working knowledge of Ada. If you haven’t used Ada before, we recommend learning the basics first before reading further. [http://www.adahome.com/Discover/Introduction.html](http://www.adahome.com/Discover/Introduction.html) is a good place to start. We also assume a general familiarity with the Lego Mindstorms Robotics Invention System and the RCX.

To use Ada/Mindstorms 2.0, you need a PC running Windows and a Windows Ada compiler. To download Ada/Mindstorms programs to an RCX and test them, you need:

  - Dave Baum’s NQC system (2.3a1 or higher) installed on your PC. It’s available from [http://www.enteract.com/~dbaum/nqc/index.html](http://www.enteract.com/~dbaum/nqc/index.html). If you’re using AdaGIDE, the file nqc.exe executable needs to be in c:\gnatpro\bin so that AdaGIDE can find it. Otherwise it can be anywhere convenient.
  - A Lego RCX brick pre-loaded with Mindstorms firmware (firm0328.lgo or later, available from [www.legomindstorms.com](http://www.legomindstorms.com) by following the links to the Software Developers’ Kit).
A Lego I/R transmitter hooked up to your PC’s serial port. These come standard with the Lego Mindstorms Robotics Invention System.

If you have already purchased a Lego Mindstorms Robotics Invention System, you don’t need the CD/ROM and software that come with it. Ada/Mindstorms is a separate programming environment, replacing the Mindstorms graphical programming tools with an Ada interface.

2.0 Ada/Mindstorms 2.0 API

The API is very similar to the NQC API, so you can skip this section and just use Appendix A if you’re already familiar with NQC. Much of the documentation in this section is adapted from the NQC manual.

Basic commands and simple examples follow. The complete API is given in Appendix A.

Most Ada/Mindstorms API commands access either the three sensor ports (inputs) on the RCX or the three output ports. These are indicated using the enumerated types below:

```plaintext
type Sensor_Port is (Input_1, Input_2, Input_3);
type Output_Port is (Output_A, Output_B, Output_C);
```

2.1 Sensor Ports

Sensor ports have a type and a mode: both must be set before the sensor can be used. The type determines how the RCX handles the sensor electrically, while the mode determines how the reading is interpreted. For some sensors, only one mode makes sense, others work with multiple modes.

The possible sensor types are stored in the following Ada enumerated type:

```plaintext
type Sensor_Type is
  (Type_Touch, -- a touch sensor
   Type_Temperature, -- a temperature sensor
   Type_Light, -- a light sensor
   Type_Rotation); -- a rotation sensor
```

This is set with the Set_Sensor_Type command, as in:

```plaintext
Set_Sensor_Type(Sensor => Sensor_1, Mode => Type_Touch);
```

Sensor modes are:

```plaintext
type Sensor_Mode is
  (Mode_Raw, -- number from 0 to 1023
   Mode_Bool, -- 0 or 1
```

This is set with the Set_Sensor_Mode command, as in:

```plaintext
Set_Sensor_Mode(Sensor => Sensor_1, Mode => Mode_Raw);
```
Mode_Edge, -- counts number of boolean transitions
Mode_Pulse, -- counts number of boolean periods
Mode_Percent, -- number from 1 to 100
Mode_Celsius, -- degrees C
Mode_Fahrenheit, -- degrees F
Mode_Rotation); -- counts rotations

This is set with the Set_Sensor_Mode command, as in

Set_Sensor_Mode(Sensor => Sensor_1, Mode => Mode_Edge);

Normally, however, the Set_Sensor_Type and Set_Sensor_Mode commands are not used. For convenience, the most common types and modes are combined into configurations. These are shown below:

```plaintext
type Configuration is (
  Config_Touch, -- type = type_touch, mode = mode_bool
  Config_Pulse, -- type = type_touch, mode = mode_pulse
  Config_Edge, -- type = type_touch, mode = mode_edge
  Config_Light, -- type = type_light, mode = mode_percent
  Config_Rotation, -- type = type_rotation, mode =
    -- mode_rotation
  Config_Celsius, -- type = type_temperature, mode =
    -- mode_celsius
  Config_Fahrenheit); -- type = type_temperature, mode =
    -- mode_fahrenheit
```

This is used in the Config_Sensor command, the most common way to configure a sensor. The previous two commands are normally accomplished with the single command:

Config_Sensor(Sensor => Sensor_1, Configuration => Edge);

In addition to type and mode, some sensors maintain a value when configured. This value is read with the Get_Sensor_Value command, as in

If (Get_Sensor_Value(Sensor => Sensor_1) = 0) then

The value is cleared with the Clear_Sensor command:

Clear_Sensor(Sensor => Sensor_1);

2.2 Output Ports

Output ports are normally where motors are connected. Each port has a mode, a direction, and a power level:

```plaintext
type Output_Mode_Type is (Output_Mode_On, Output_Mode_Off);
type Output_Direction is
```
-- type used to specify the power level of an output.
type Power_Type is range 0 .. 7;

-- power level definitions, supplied as a convenience
Power_Low : constant Power_Type := 1;
Power_Half : constant Power_Type := 4;
Power_High : constant Power_Type := 7;

The mode indicates whether the port is on or off (i.e. whether or not any motors connected to the port will be turning or not). The direction indicates the direction of rotation of the motors. Note that direction is a function of both the API calls and the orientation of the wire connected to the output port. API calls that set the direction of the output port can give different results depending on how the wire is connected. Power levels are from 1 to 7, or you can use the special constants defined above for convenience.

The output mode is set with the Set_Output_Mode command, as in

Set_Output_Mode(Output => Output_1, Mode => Output_Mode_On);

For convenience, a one-parameter command is also available that has the same effect:

Output_On(Output => Output_1);

Similar commands exist for setting direction and power:

Set_Output_Direction(Output => Output_1, Direction =>
Output_Direction_Forward);
Output_On_Forward(Output => Output_1); -- same as above
Output_Power(Power => Power_Half);
Output_Power(Power => 4); -- same as above

2.3 Other Procedures

The Ada/Mindstorms API also includes procedures for RCX sending and receiving messages, playing sounds, and other tasks. The complete API is given in Appendix A.

3.0 Writing Ada/Mindstorms Programs

Ada was designed for a much richer programming environment than the RCX can implement. Accordingly, only a small subset of Ada constructs are supported. Below we will build “test.adb”, a program for a simple robot with a bumper connected to one sensor and motors driven by two output ports. This program will back up and turn if the bumper sensor is depressed, alternating the direction of the turn using a counter variable. As we build the program, we will show what Ada features are supported by Ada/Mindstorms. “test.adb” is part of the Ada/Mindstorms installation.
3.1 Required Packages

Every Ada/Mindstorms program must begin with the lines:

```ada
with Lego;
use Lego;
```

These lines source the package lego.ads, which comes with Ada/Mindstorms and contains the Application Programmer Interface (API) functions that your program will call. **No other packages can be used, including anything from the standard Ada library.**

3.2 Procedures and Program Structure

All Ada/Mindstorms programs must consist of a single main procedure and 0 or more user-defined procedures inside it. Ada/Mindstorms does not support procedure nesting levels greater than 1. User-defined procedures may have 0 or more parameters (see below). The structure of our test program looks like:

```ada
procedure Test is    --main procedure
  ...  
  procedure Go_Foward is    -- some user-defined procedures
     begin
     ... 
     end Go_Foward;

  procedure Go_Back (...) is
    begin
    ... 
    end Go_Back;

  procedure Turn (...) is
    begin
    ... 
    end Turn;

  procedure Initialize_Robot is
    begin
    ... 
    end Initialize_Robot;

begin
  ...    --statements that call your procedures will go here
end Test;
```

3.3 Variables, Constants, Literals, and Parameters

The base type for all program variables must be integer, boolean, or one of the predefined types given in Appendix A. No other predefined data types are supported. Variables can
be either scalar quantities or arrays, defined with the standard Ada array semantics as a user-defined type. No other user-defined types are supported. However, for convenience, lego.ads includes a type called Integer_Array which permits the declaration of array variables without a type statement, like this:

My_Array : Integer_Array(-4..15);

Array elements cannot be used as arguments to procedures. Scalar variables may be initialized when declared, array variables cannot. Multiple variables may declared in a single statement. Global variables are supported, but are of course frowned upon. User-defined procedures may have their own local variables.

All program constants must be integers or one of the supported predefined types. Each must be declared in a separate statement, and initialized with ‘:=’. Both global and local constant definitions are supported.

Only Ada integer literals are supported. They must be either decimal or hex.

“In” and “Out” parameters are supported, as is named parameter association. Similar to variables, all parameters must be of type integer. Arrays and array elements as parameters are not currently supported.

Continuing our example, adding variables, constants, and parameters to the test program gives:

```ada
procedure Test is
  Left : constant Integer := 0;
  Right : constant Integer := 1;

  --user-defined procedure with no parameters
  procedure Go_Forward is
  begin
    ... end Go_Forward;

  --user-defined procedure with a parameter
  procedure Go_Back (Tenhs_Of_A_Second : in Integer ) is
  begin
    ... end Go_Back;

  procedure Turn (Direction : in Integer ) is
  begin
    ... end Turn;

  procedure Initialize_Robot is
  begin
    ... end Initialize_Robot;
```
counter : integer := 1; --variable declaration
begin
...
end Test;

3.4 Operators and Statements

The following Ada operators are supported:

+,-,*,/
and, or, not
:=
=, /=, <, <=, >, >=
abs()
mod()

These operators behave with the standard Ada semantics and precedence, and can be combined to form arbitrarily complex expressions.

The following Ada statements are supported:

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar constant declaration</td>
<td>Max_Val : constant Integer</td>
</tr>
<tr>
<td>Scalar variable declaration</td>
<td>Counter : Integer := 0;</td>
</tr>
<tr>
<td>Array variable declaration</td>
<td>type My_Array_Type is array</td>
</tr>
<tr>
<td></td>
<td>My_Array : My_Array_Type;</td>
</tr>
<tr>
<td>Null statement</td>
<td>null;</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>x := 3*y + 12;</td>
</tr>
<tr>
<td>Exit statement</td>
<td>exit when x=y;</td>
</tr>
<tr>
<td>Ada/Mindstorms API procedure calls</td>
<td>Output_Off(Output =&gt; Output_A);</td>
</tr>
<tr>
<td>User-defined procedure calls</td>
<td>My_Proc(Param1 =&gt; 3);</td>
</tr>
</tbody>
</table>
| Compound statement statements                  | begin
|                                                | 1 or more                                   |
|                                                | end;                                         |
If statement
if (a > 2) then
    Compound statement
end if;

If-then-else statement
if (a > 2) then
    Compound statement
else
    Compound statement
end if;

Cascading if-then-else statement
if (a > 2) then
    Compound statement
elsif (a = 2) then
    Compound statement
(optional other elsif clauses)
else
    Compound statement
end if;

Case statement
case j is
    when 1 =>
        a := 2;
    when 2 =>
        a := 4;
    when 3 | 5 | 12 =>
        a := 6;
    when others =>
        play_sound(up);
end case;

Basic loop
loop
    Compound statement
end loop;

While loop
while (I <= 10) loop
    Compound statement
end loop;

For loop
for I in 1..Limit loop
    Compound statement
End loop;
3.5 Final Test Program

The complete test program, showing user-defined procedure calls, API calls, and statements, is shown below. You can find this file as “test.adb” with the other Ada/Mindstorms files as part of the software installation.

```ada
with Lego;
use Lego;

procedure Test is
    Left : constant Integer := 0;
    Right : constant Integer := 1;

    procedure Go_Forward is
        begin
            Output_On_Forward(Output => Output_A);
            Output_On_Forward(Output => Output_C);
        end Go_Forward;

    procedure Go_Back (Tenths_Of_A_Second : in Integer ) is
        begin
            Output_On_Reverse(Output => Output_A);
            Output_On_Reverse(Output => Output_C);
            Wait(Hundredths_Of_A_Second => Tenths_Of_A_Second * 10);
            Output_Off(Output => Output_A);
            Output_Off(Output => Output_C);
        end Go_Back;

    procedure Turn (Direction : in Integer ) is
        begin
            if Direction = Left then
                Output_Off(Output => Output_A);
                Output_Power(Power  => Power_Low, Output => Output_C);
                Output_On_Forward(Output => Output_C);
                Wait(Hundredths_Of_A_Second => 100);
                Output_Off(Output => Output_C);
            else
                Output_Off(Output => Output_C);
                Output_Power(Output => Output_A,Power  => Power_Low);
                Output_On_Forward(Output => Output_A);
                Wait(Hundredths_Of_A_Second => 100);
                Output_Off(Output => Output_A);
            end if;
        end Turn;

    procedure Initialize_Robot is
        begin
            Config_Sensor(Sensor => Sensor_1, Config => Config_Touch);
            Output_Power(Output => Output_A,Power  => Power_Low);
            Output_Power(Output => Output_C,Power  => Power_Low);
            Output_On_Forward(Output => Output_A);
            Output_On_Forward(Output => Output_C);
        end Initialize_Robot;

    counter : integer := 1;
```
begin
  Initialize_Robot;
  loop
    --touch sensor pressed?
    if Get_Sensor_Value(Sensor => Sensor_1) = 1 then
      Go_Back(Tenths_Of_A_Second => 30);
      if counter = 0 then
        Turn(Direction => Right);
        counter := 1;
      else
        Turn(Direction => Left);
        counter := 0;
      end if;
    end if;
    Go_Forward;
  end loop;
end Test;

3.6 Limitations of Ada/Mindstorms Programs

Because Ada/Mindstorms is built on top of NQC and the RCX, there are many limitations imposed on Ada/Mindstorms programs. Violations of all but one of these constraints are detected by the compiler. The most important limitations are:

a) Ada/Mindstorms supports a maximum of 32 global variables, due to NQC constraints. This shouldn’t be a problem for most programs, since you shouldn’t be using globals anyway!

b) Functions that return values are not supported.

c) Recursion is not supported.

d) Tasking is not supported (if you are not familiar with Ada tasking, don’t worry about this)

Violations of (a) and (b) are detected by the compiler. Violations of (c) are not currently detected, but this is planned for a future release. Violations of (d) are detected by the compiler, and support for tasking is planned for a future release.

4.0 Compiling, Downloading, and Running Ada/Mindstorms Programs With AdaGIDE

To program the RCX with an Ada/Mindstorms program using AdaGIDE, do the following:

Write your program as described in the previous sections. Make sure you ‘with’ and ‘use’ lego.ads at the top of your code.

Select the Target button in AdaGIDE (see below). If you don’t have a Target button, you need to upgrade to the latest version of AdaGIDE.
When the dialog box below comes up, select “Lego Mindstorms” and press OK.

The AdaGIDE target icon will change to a Lego touch sensor:
Now compile the file as you normally would, using the compile button (see below). Successful compilation will be indicated by the message “Completed.” in the window at the bottom of the screen. Errors will be reported there as well.

Once the program has compiled successfully, build it with the build button:

(If you are unable to complete a successful build and instead get the message “Unable to start build Check Path Environment variable Error Code: 2”, make sure that the ada2nqc and nqc executables are in your PATH environment variable.)

After the program builds successfully, make sure your RCX I/R transmitter is plugged into your computer’s serial port and the RCX is turned on. Pressing the run button (shown below) will invoke nqc on your translated code, which will translate it into RCX opcodes and attempt to download it.
When the program has been successfully downloaded, the RCX will play a series of ascending tones and you’ll see this in the AdaGIDE message window (on the bottom of the screen):

Downloading ...
Downloading Program:...........complete
Battery Level = x.x V
Completed successfully.

If your program does not download, an error message will appear in the AdaGIDE window. Possible reasons for download failure include the RCX being off, a disconnected I/R transmitter, bright room lights, failure to install the RCX firmware before downloading, or a malformed output file from Ada/Mindstorms that NQC cannot translate. This last option should never happen, and would indicate a bug in Ada/Mindstorms 2.0. Please report any such occurrences to barry.fagin@usafa.af.mil.

5.0 References


This work is supported by the USAF Institute for Information Technology and Applications.
Appendix A: Ada/Mindstorms API

Predefined Constants and Types

-- RCX sensor ports
type Sensor_Port is (Sensor_1, Sensor_2, Sensor_3);

-- RCX output ports
type Output_Port is (Output_A, Output_B, Output_C);

-- Sensor port configurations
type Configuration is
   Config_Touch, -- type = type_touch, mode = mode_bool
   Config_Pulse, -- type = type_touch, mode = mode_pulse
   Config_Edge, -- type = type_touch, mode = mode_edge
   Config_Light, -- type = type_light, mode = mode_percent
   Config_Rotation, -- type = type_rotation, mode = mode_rotation
   Config_Celsius, -- type = type_temperature, mode = mode_celsius
   Config_Fahrenheit); -- type = type_temperature, mode = mode_fahrenheit

-- Sensor types (normally subsumed by Configuration type above).
type Sensor_Type is
   (Type_Touch,
   Type_Temperature,
   Type_Light,
   Type_Rotation);

-- Sensor modes (normally subsumed by Configuration type above)
type Sensor_Mode is
   (Mode_Raw,
   Mode_Bool,
   Mode_Edge,
   Mode_Pulse,
   Mode_Percent,
   Mode_Celsius,
   Mode_Fahrenheit,
   Mode_Rotation);

-- Output port modes
type Output_Mode is
   (Output_Mode_On,
   Output_Mode_Off,
   Output_Mode_Float);

-- Output port directions
type Output_Direction is
   (Output_Direction_Forward,
   Output_Direction_Reverse,
   Output_Direction_Toggle);

-- Output power levels
type Power_Type is range 0..7;

--power level definitions, supplied as a convenience
Power_Low : constant Power_Type := 1;
Power_Half : constant Power_Type := 4;
Power_High : constant Power_Type := 7;

-- type for API calls where Output_Ports can't be used
type Output_Port_Number is range 0..2;

-- type used to specify a transmitter power setting
type Transmitter_Power_Setting is (Transmitter_Power_Low, Transmitter_Power_High);

-- Predefined sounds
type Sound is (Click, Double_Beep, Down, Up, Low_Beep, Fast_Up);

-- Valid message numbers
type Message is range 1 .. 255;

-- Hour and Minute types for setting the watch
subtype Hour is integer range 0..23;
subtype Minute is integer range 0..59;

-- range of datalog sizes
subtype Datalog_Range is integer range 1..1024;

-- Time duration, guards against negative values
type Duration is new Positive;

-- Frequency, guards against negative values
type Frequency is new Positive;

-- Sensor values
type Sensor_Value is new Integer;

--type for sensor modes
type Sensor_Mode is (Mode_Raw, Mode_Bool, Mode_Edge, Mode_Pulse, Mode_Percent, Mode_Celsius, Mode_Fahrenheit, Mode_Rotation);

--type for timers
type Timer is (Timer_0, Timer_1, Timer_2, Timer_3);

--type for system watch
subtype Watch_Value_In_Minutes is integer range 0..24*60-1;

--for specifying the number of decimal places in "Set_User_Display"
subtype Decimal_Place_Value is integer range 0..4;
## API Routines

Below are the complete set of Ada/Mindstorms routines. Many are not described here, but they are all taken more or less directly from NQC. For more details, the reader is referred to the [NQC web site](http://example.com).

--const means that the indicated argument must be a constant.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>SPECIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_To_Datalog</td>
<td>Adds value to datalog</td>
<td>procedure Add_To_Datalog</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Value : in integer);</td>
</tr>
<tr>
<td>Clear_Message_Buffer</td>
<td>Clears RCX message buffer</td>
<td>procedure Clear_Message_Buffer;</td>
</tr>
<tr>
<td>Clear_Sensor</td>
<td>Sets sensor value to 0</td>
<td>procedure Clear_Sensor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port );</td>
</tr>
<tr>
<td>Clear_Sound_Buffer</td>
<td>Purges all sounds from the sound</td>
<td>procedure Clear_Sound_Buffer;</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td></td>
</tr>
<tr>
<td>Clear_Timer</td>
<td>Clears indicated timer</td>
<td>procedure Clear_Timer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Timer_To_Clear : in Timer );</td>
</tr>
<tr>
<td>Config_Sensor</td>
<td>Configures a sensor</td>
<td>procedure Config_Sensor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port; --const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Config : in Configuration );</td>
</tr>
<tr>
<td>Create_Datalog</td>
<td>Creates datalog of indicated size</td>
<td>procedure Create_Datalog</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Size : in Datalog_Range); --const</td>
</tr>
<tr>
<td>Display_Value</td>
<td>Displays indicated value with #</td>
<td>procedure Display_Value</td>
</tr>
<tr>
<td></td>
<td>decimal places, value must be</td>
<td>(Value : in integer;</td>
</tr>
<tr>
<td></td>
<td>variable in main procedure</td>
<td>Decimal_Places : in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decimal_Place_Value); --const</td>
</tr>
<tr>
<td>Get_Boolean_Sensor_Value</td>
<td>Returns value of sensor as boolean</td>
<td>function Get_Boolean_Sensor_Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port ) --const</td>
</tr>
<tr>
<td>Get_Global_Output_Mode</td>
<td>Returns global mode of indicated</td>
<td>function Get_Global_Output_Mode</td>
</tr>
<tr>
<td></td>
<td>output</td>
<td>(Output : Output_Port_Number) --const</td>
</tr>
<tr>
<td>Get_Message</td>
<td>Returns message in buffer</td>
<td>function Get_Message return Message;</td>
</tr>
<tr>
<td>Get_Output_Mode</td>
<td>Returns mode of indicated output</td>
<td>function Get_Output_Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Output : Output_Port_Number) --const return</td>
</tr>
<tr>
<td>Get_Raw_Sensor_Value</td>
<td>Returns raw value from sensor</td>
<td>function Get_Raw_Sensor_Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port ) --const return Sensor_Value;</td>
</tr>
<tr>
<td>Get_Sensor_Mode</td>
<td>Returns mode of sensor</td>
<td>function Get_Sensor_Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port ) --const return Sensor_Mode;</td>
</tr>
<tr>
<td>Get_Sensor_Type</td>
<td>Returns type of sensor</td>
<td>function Get_Sensor_Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sensor : in Sensor_Port ) --const return Sensor_Type;</td>
</tr>
<tr>
<td>Get_Sensor_Value</td>
<td>Returns calibrated</td>
<td>function Get_Sensor_Value</td>
</tr>
<tr>
<td>NAME</td>
<td>DESCRIPTION</td>
<td>SPECIFICATION</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Get_Timer</td>
<td>Returns current value of indicated timer</td>
<td>function Get_Timer (Timer_To_Read : in Timer) --const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return integer;</td>
</tr>
<tr>
<td>Output_Float</td>
<td>Sets output port mode to float (so that the motor glides to a stop)</td>
<td>procedure Output_Float (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_Forward</td>
<td>Sets output port direction to forward (port must be on)</td>
<td>procedure Output_Forward (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_Off</td>
<td>Turns output port off (makes it stop immediately)</td>
<td>procedure Output_Off (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_On</td>
<td>Turns output port on</td>
<td>procedure Output_On (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_On_For</td>
<td>Turns output port on for a specific length of time.</td>
<td>procedure Output_On_For (Output : in Output_Port; --const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hundredths_Of_A_Second : in Natural );</td>
</tr>
<tr>
<td>Output_On_Forward</td>
<td>Turns output port on, sets direction to forward</td>
<td>procedure Output_On_Forward (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_On_Reverse</td>
<td>Turns output port on, sets direction to reverse</td>
<td>procedure Output_On_Reverse (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_Power</td>
<td>Sets output port power level as indicated</td>
<td>procedure Output_Power (Output : in Output_Port; --const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Power : in Power_Type );</td>
</tr>
<tr>
<td>Output_Reverse</td>
<td>Sets output port direction to reverse (port must be on)</td>
<td>procedure Output_Reverse (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Output_Toggle</td>
<td>Changes output port direction (port must be on)</td>
<td>procedure Output_Toggle (Output : in Output_Port ); --const</td>
</tr>
<tr>
<td>Play_Sound</td>
<td>Plays the indicated sound</td>
<td>procedure Play_Sound (Sound_To_Play : in sound ); --const</td>
</tr>
<tr>
<td>Play_Tone</td>
<td>Plays tone at indicated frequency for indicated duration</td>
<td>procedure Play_Tone (Frequency_In_Hertz : in frequency;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tenths_Of_A_Second : in Natural );</td>
</tr>
<tr>
<td>Random</td>
<td>Returns random # between 0 and arg</td>
<td>function Random (Max : in Integer) --const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return integer;</td>
</tr>
<tr>
<td>Select_Display</td>
<td>Sets what to show on display, can be an expression</td>
<td>procedure Select_Display (What_To_Show : in integer);</td>
</tr>
<tr>
<td>NAME</td>
<td>DESCRIPTION</td>
<td>SPECIFICATION</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Send_Message</td>
<td>Causes RCX to send indicated message</td>
<td><code>procedure Send_Message (Message_To_Send : in Message );</code></td>
</tr>
<tr>
<td>Set_Global_Output_Direction</td>
<td>Sets global direction on an output</td>
<td><code>procedure Set_Global_Output_Direction (Output : in Output_Port; --const Direction : in Output_Direction);</code></td>
</tr>
<tr>
<td>Set_Global_Output_Mode</td>
<td>Sets global mode on an output</td>
<td><code>procedure Set_Global_Output_Mode (Output : in Output_Port; --const Mode : in Output_Mode); --const</code></td>
</tr>
<tr>
<td>Set_Max_Power</td>
<td>Sets maximum power allowed on an output</td>
<td><code>procedure Set_Max_Power (Output : in Output_Port; --const Limit : in Power_Type);</code></td>
</tr>
<tr>
<td>Set_Output_Direction</td>
<td>Sets output port direction</td>
<td><code>procedure Set_Output_Direction (Output : in Output_Port; --const Direction : in Output_Direction); --const</code></td>
</tr>
<tr>
<td>Set_Output_Mode</td>
<td>Sets output port mode</td>
<td><code>procedure Set_Output_Mode (Output : in Output_Port; --const Mode : in Output_Mode);</code></td>
</tr>
<tr>
<td>Set_Sensor_Mode</td>
<td>Sets sensor port mode</td>
<td><code>procedure Set_Sensor_Mode (Sensor : in Sensor_Port; --const Mode : in Sensor_Mode ); --const</code></td>
</tr>
<tr>
<td>Set_Sensor_Type</td>
<td>Sets sensor type</td>
<td><code>procedure Set_Sensor_Type (Sensor : in Sensor_Port; --const Kind : in Sensor_Type ); --const</code></td>
</tr>
<tr>
<td>Set_Transmitter_Power_Level</td>
<td>Sets transmitter power level (high or low)</td>
<td><code>procedure Set_Transmitter_Power_Level (Level : in Transmitter_Power_Setting); --const</code></td>
</tr>
<tr>
<td>Set_Watch</td>
<td>Sets system watch to indicated hours and minutes</td>
<td><code>procedure Set_Watch (Set_Hours_To : in Hour; --const Set_Minutes_To : in Minute); --const</code></td>
</tr>
<tr>
<td>Speaker_Off</td>
<td>Disables speaker</td>
<td><code>procedure Speaker_Off;</code></td>
</tr>
<tr>
<td>Speaker_On</td>
<td>Enables speaker</td>
<td><code>procedure Speaker_On;</code></td>
</tr>
<tr>
<td>Stop_All_Tasks</td>
<td>Stops all tasks running on the RCX</td>
<td><code>Stop_All_Tasks;</code></td>
</tr>
<tr>
<td>Wait</td>
<td>Causes RCX to wait for the indicated amount of time before proceeding to the next command</td>
<td><code>Wait (Hundredths_Of_A_Second : in Natural );</code></td>
</tr>
</tbody>
</table>
Appendix B: Running Ada/Mindstorms 2.0 Without the AdaGIDE Compiler

If you are using a different Ada compiler from AdaGIDE, you will need to run the ada2nqc (the Ada to NQC translator) from a command line on your .adb source file. You will also need Dave Baum’s NQC software installed somewhere on your machine, with the latest RCX firmware if you intend to use all RCX features of Ada/Mindstorms 2.0.

First, write the Ada/Mindstorms program and compile it as you normally would. Make sure you ‘with’ and ‘use’ the lego.ads file that comes with Ada/Mindstorms 2.0. Running your code through your standard Ada compiler first is important, because ada2nqc is designed to detect only errors that would be missed by a standard Ada compiler.

Once the file compiles properly, run your code through the ada2nqc translator at a command line prompt:

> ada2nqc foo.adb

This will produce the file “foo.nqc” in the same directory. This file can then be used as input to NQC, with all the standard options for compiling and downloading. See the NQC programmer’s guide (http://www.enteract.com/~dbaum/nqc/doc/NQC_Guide.html) for details.

NQC should always successfully compile files produced by ada2nqc. Please report any unsuccessful compilations to barry.fagin@usafa.af.mil, along with a sample of the Ada and nqc files.