# Ada/Mindstorms 1.0 User's Guide and Reference Manual

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



## 1.0 Introduction

Ada/Mindstorms 1.0 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 langauge (NQC), then recompiled and downloaded through the Mindstorms IR transmitter to the RCX.

Ada/Mindstorms 1.0 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 A. 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 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 1.0, you need a PC running Windows and a Windows Ada compiler. To download Ada/Mindstorms programs to an RCX and test them, you need:

1) Dave Baum's NQC system installed on your PC. It's available from 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.
2) A Lego RCX brick pre-loaded with Mindstorms firmware.
3) 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 the it. Ada/Mindstorms is a separate programming environment, replacing the Mindstorms graphical programming tools with an Ada interface.

## 2.0 Ada/Mindstorms 1.0 API

The API is very similar to the NQC API, so you can skip this section and just use Lego.ads 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 Lego.ads.

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:

```
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:

```
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:

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

Sensor modes are:

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

```
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:*

```
type Output_Mode_Type is (Output_Mode_On, Output_Mode_Off);

type Output_Direction is
   (Output_Direction_Forward, Output_Direction_Reverse,
    Output_Direction_Toggle);

-- 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 Lego.ads.

## 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 will build 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.

### 3.1 Required Packages

Every Ada/Mindstorms program must begin with the lines:

```
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).

### 3.3 Variables, Constants, Literals, and Parameters

All program variables must be integers. No other predefined data types are supported, and no user-defined types are supported. Variables may be initialized when declared, and 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. 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.

### 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:

| STATEMENT | EXAMPLE |
|---|---|
| Null statement | `null;` |
| Assignment statement | `x := 3*y + 12;` |
| Exit statement | `exit when x=y;` |
| Ada/Mindstorms API procedure calls | `Output_Off(Output => Output_A);` |
| User-defined procedure calls | `My_Proc(Param1 => 3);` |
| Compound statement | `begin`<br>1 or more statements<br>`end;` |
| If statement | `if (a > 2) then`<br>Compound statement |
| If-then-else statement | `if (a > 2) then`<br>Compound statement<br>`else`<br>compound statement |
| Cascading if-then-else statement | `if (a > 2) then`<br>Compound statement<br>`elsif (a = 2) then`<br>compound statement<br>(optional other elsif clauses)<br>`else`<br>compound statement |
| Basic loop | `loop`<br>Compound statement<br>`End loop;` |
| While loop | `while (I <= 10) loop`<br>Compound statement<br>`End loop;` |
| For loop | `for I in 1..Limit loop`<br>Compound statement<br>`End loop;` |

### 3.5 Final Test Program

The complete test program, showing user-defined procedure calls, API calls, and statements, is shown below:

```
with Lego;
use Lego;

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

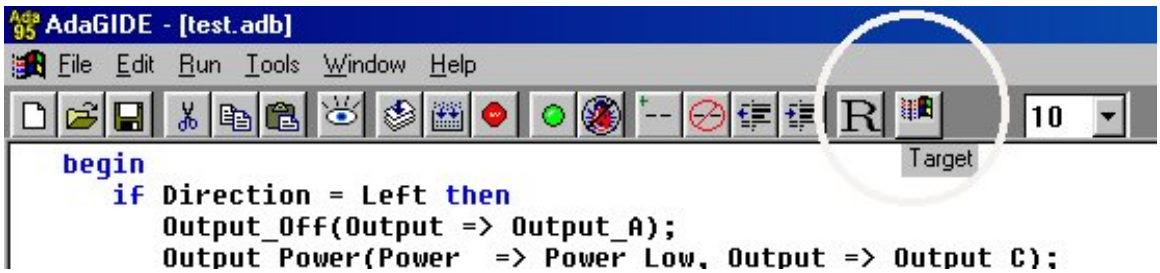
```
   procedure Go_Forward is
   begin
      Output_On_Forward(Output => Output_A);
      Output_On_Forward(Output => Output_C);
   end Go_Forward;

   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 --right
         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
      Set_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;
         Go_Forward;
```

```
      end if;
   end loop;
end Test;
```

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

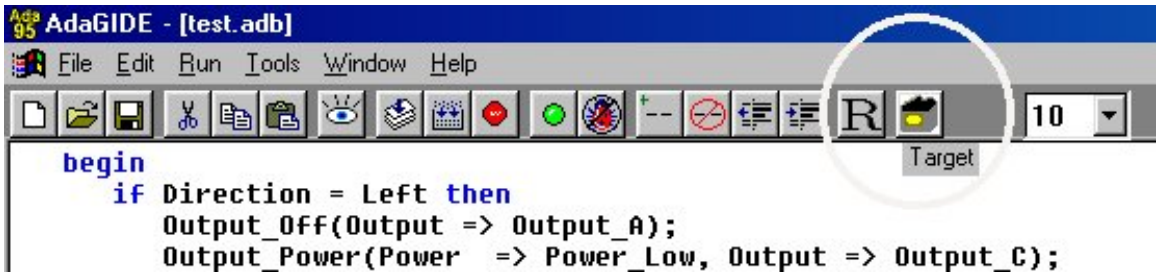
To program the RCX with an Ada/Mindstorms program using AdaGIDE, do the following:

1) Write your program as described in the previous sections. Make sure you 'with' and 'use' lego.ads at the top of your code.
2) 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.




3) When the dialog box below comes up, select "Lego Mindstorms" and press OK.


4) The AdaGIDE target icon will change to a Lego touch sensor:




5) 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.


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


7) 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 download your program into the RCX[1]. When the program has been successfully downloaded, the RCX will play a series of ascending tones.


[1] With the Lego Mindstorms target selected, the run button will invoke nqc on your translated code, which will then translate it into RCX opcodes and attempt to download it.

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 1.0. Please report any such occurrences to barry.fagin@usafa.af.mil.

## Appendix A: Running Ada/Mindstorms 1.0 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 DOS command line on your .adb source file.
You will also need Dave Baum's NQC software installed somewhere on your machine.

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 1.0. 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, open a DOS shell, and run your code through the ada2nqc translator:

DOS> 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.