Infento Robotics

Technical Guide

Programming Guide

Technical guide

Introduction

Infento Robotics is an extension of the general InfentoRides system. It adds actuators and sensors to the system and thus can transform it into a real live size robot. The actuators and sensors connect to the controller via actuator and sensor boards.

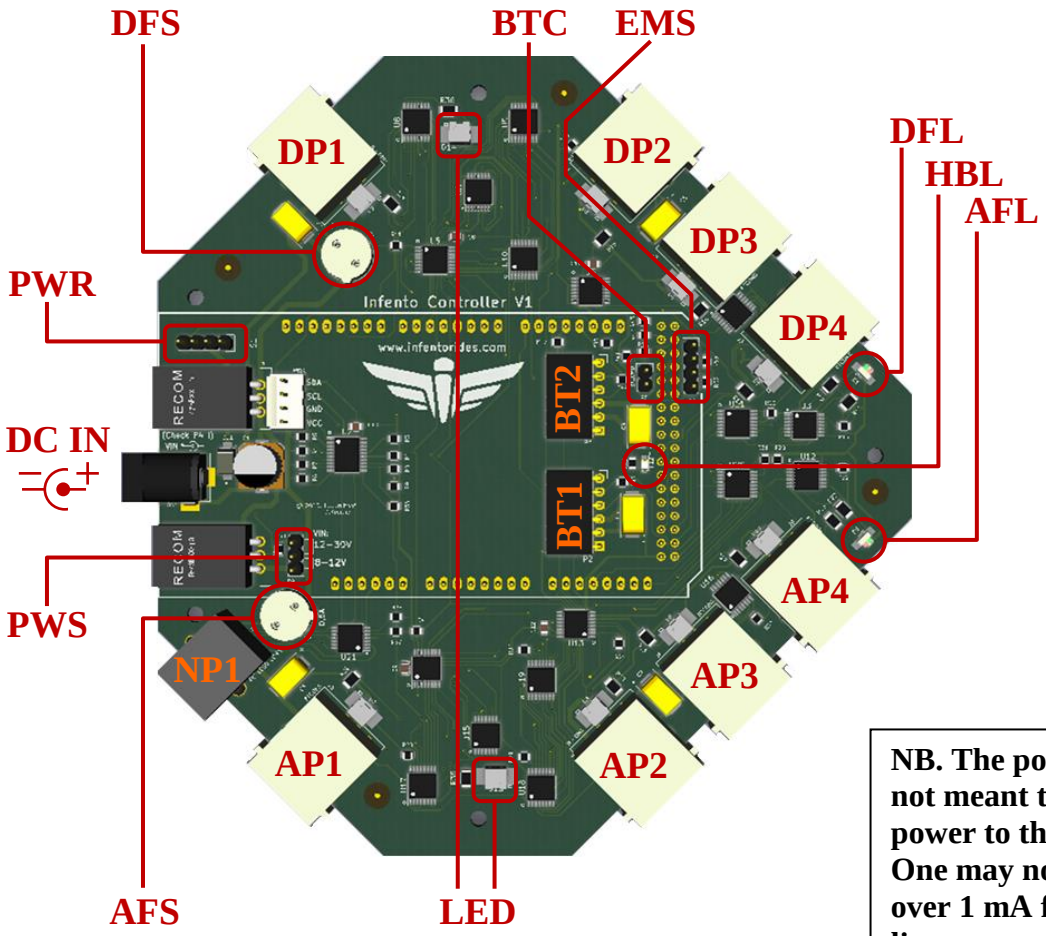
Infento Robotics comes with its own sensors and actuators, but can be extended to other devices. The controller ports exist of standard British Telecom connecters (BTA and BTD) as are used by the Texas Instruments, CMA and Vernier sensors [1]. The Infento controller is designed to accept these sensors as well. In addition there is a Lego Mindstorms connector for support of the Mindstorms I2C and analogue sensors [2]. And one can use home made devices that communicate over I2C, UART or SPI or that handle plain data lines [3].

Although the controller of Infento Robotics does accept foreign devices, it is highly advisable to restrict oneself to the native system since it is designed for easy and safe usage. One is enabled to program a robot with a few instructions only and for example the blocking of motors is detected automatically. In such cases the program is aborted without the need of user intervention. Furthermore one may connect the sensor and actuator boards to whichever controller port without changing a program. The system will find the devices automatically when they are used by the program. Infento Robotics is designed for a flexible, fast and straight forward programming experience.

[1] The auto-id function of the analogue sensors is not supported, but the automatic recognition of i2c sensors is.
[2] The Mindstorms lo-speed motors or ultrasonic sensors are not supported, since the ports cannot supply the needed potential of 9V.
[3] Damage of the controller is not warranted in such cases.

The Infento controller

Board layout



NB. The port lines are not meant to supply power to the devices. One may not consume over 1 mA from the lines.

DC IN	Power supply (see PWS power select)
PWR	Header for power button and power led
PWS	Jumper for DC IN power select (9-12V or 12-30V)
DFS	Fuse (0.5A) for the digital ports
DFL	Digital fuse led (green = ok, red = fuse is blown)
AFS	Fuse (0.5A) for the analogue ports
AFL	Analogue fuse led (green = ok, red = fuse is blown)
NP1	NXT port (see table 1) - Lego Mindstorms connector, without 9V supply
DP1-DP4	Digital ports 1 to 4 (see table 1) - British Telecom Digital connectors
AP1-AP4	Analogue ports 1 to 4 (see table 1) - British Telecom Analogue connectors
LED	Continue leds (red)
BT1-BT2	Bluetooth HC-05 breakout connectors (BT1=uart1, BT2=uart2)
BTC	Header for BT1/BT2 commands (Arduino pins 22/23, output 3,3V)
EMS	Header for emergency button (Arduino pin 25, active low)
HBL	Heart beat led (Arduino pin 40)

All digital and analogue ports (DP1-DP4 and AP1-AP4) have a status led behind them.

General description

The Infento controller exists of an Infento Shield plugged onto an Arduino Mega 2560 R3 (or compatible) board. It can be programmed via the regular Arduino IDE using the Infento library. The connectors of the controller are called ports and its pins are called lines. The pins of the Arduino board are routed to the ports either directly (line 1 and 2) or via a mode switch (line 3 and 4). There are four modes. In data mode line 3 and 4 are handled as regular Arduino pins. In i2c or uart mode these pins aren't available and the Arduino is enabled to communicate via the i2c or uart interfaces respectively. The fourth mode is called reference mode and affects the analogue ports only. In reference mode line 3 is handled as a regular Arduino pin, but line 4 of the analogue port can be used as a voltage divider in combination with one of the other data lines. When a digital port is set to reference mode it acts as if it was set to data mode. For the line layout of the ports and a scheme of the voltage divider see table 1 and figure 1. Each port can be put in a separate mode.

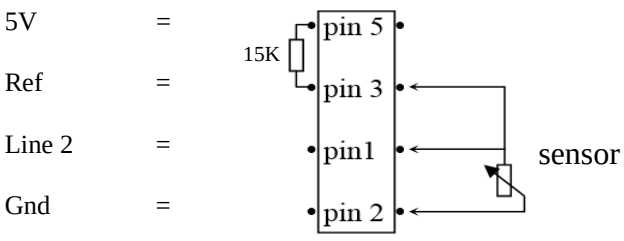
Port description

The port layout of the Infento Controller is described in table 1. All lines of digital port 1, 2 and 3 can handle PWM (pulse width modulation). Digital port 4 can be used as a SPI interface. In the order of the line numbers you have pins for MISO (SO), MOSI (SI), SCK and SS (CS). In addition to the regular usage, line 1 of digital port 1 and 2 will accept a hardware interrupt on the Arduino. For PWM, IRQ and SPI usage a port must be set to data mode.

In reference mode line 4 of an analogue (BTA connector) is neglected. Instead the connector pin is connected to 5V via a reference resistor of 15K. It can be used to create a voltage divider in combination with GND and one of the other lines. See the diagram of figure 2.

Although the NXT port is designed to connect a NXT sensor, it cannot supply 9V at pin 1 like the Lego Mindstorms brick does. This means that the older RCX sensors and the NXT ultrasonic sensor cannot be used. The other NXT sensors however and all new EV3 sensors don't need the 9V supply and can be used therefore.

Figure 1 - BTA connector pins in reference mode



In the figure line 2 is used for the sensor, but line 1 or 3 can be used too.

Table 1 - Connector pins and port lines

DIGITAL		Port mode			BTD connector
Pin	**Vernier Label**	**Data**	**I2C**	**UART**	**Colour in the Vernier cable**
1	DIO0	Line 1	Line 1	Line 1	Yellow
2	DIO1	Line 3	SDA	RX	Black
3	DIO2	Line 4	SCL	TX	Green
4	5V				Brown
5	GND				Orange
6	DIO3	Line 2	Line 2	Line 2	Red

ANALOG		Port mode				BTA connector
Pin	**Vernier Label**	**Data**	**Ref**	**I2C**	**UART**	**Colour in the Vernier cable**
1	SIG2	Line 2	Line 2	Line 2	Line 2	Yellow
2	GND					Black
3	VRES	Line 4	REF	SDA	RX	Green
4	ID [1])	Line 3	REF	SCL	TX	Brown
5	5V					Orange
6	SIG1	Line 1	Line 1	Line 1	Line 1	Red

NXT		Port mode	NXT connector
Pin	**Lego Label**	**I2C**	**Colour in the NXT cable**
1	ANA [2])	Line 1	White
2	GND [3])	Line 2	Black
3	GND		Red
4	PWR		Green
5	DIGI0	SCL	Yellow
6	DIGI1	SDA	Blue

[1]) Vernier resistor-id is not supported. This pin is used as a regular analogue pin SIG3.
[2]) The Infento Controller does not supply the 9V output as the NXT brick does.
[3]) The NXT brick has a data line at the motor output connector here. To be consistent with the other connectors of the Infento Controller, this pin is used as a second data line.

Infento controller with Bluetooth

The Infento controller can be extended by two bluetooth modules HC-05 [4]. When pairing to the modules they are seen as INFENTO-1 and INFENTO-2. You need to enter 1234 as the pairing key. The modules act as general serial interface and communicate via uart1 and uart2. Since the Arduino Mega board accepts hardware interrupts on uart1, two Infento Controllers can communicate over the attached Bluetooth module.

[4] Bluetooth version 3

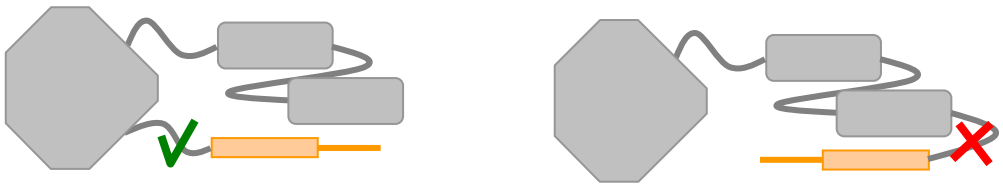
The Infento device boards

General description

Infento devices consist of the devices itself (sensors and actuators) and an i2c interface (boards) that is used to identify the device, to calibrate it, to send or request operational data and to signal the status of the interface. All communication to an Infento device starts by a command for the interface. The command structure itself stays transparent to the user since it is handled by the supporting classes in the Infento programmer library. This is a library for the Arduino IDE - the standard programming environment for the Infento controller. Low level communication to the devices is handled by the *InfentoPort* class. The intermediate class *InfentoDevice* depends on *InfentoPort* for its communication to the devices. This class serves as a base class and all Infento sensor and actuator classes are derived from it. Also foreign and home made devices should have classes that are derived from *InfentoDevice*, but this is the user's responsibility.

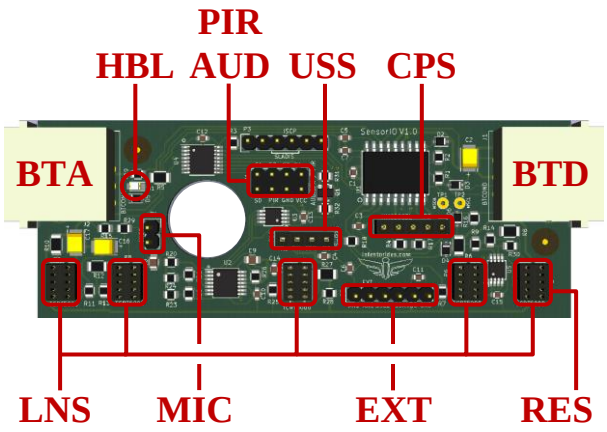
There are three standard Infento device boards: two sensor boards and one actuator board. They are small boards containing the firmware for the interaction with the Infento controller. The sensor board measures 35x95 mm and the actuator board 35x135 mm. Both sensor boards support different sensors. Infento devices are delivered with a factory calibration but the calibration can be adjusted by the user via the *InfentoCalibrate* app. Calibration data is stored in eeprom and is remembered for next sessions therefore.

Infento sensor boards can be chained, but the chain may not contain a foreign device. Foreign devices in a chain disturb the signalling function of the boards and will block the proper working of a program.



The sensor boards

Although Infento Robotics comes with two sensor boards, the hardware of both is identical. The boards differ by their firmware, because technically it is impossible to install all sensors simultaneously. The firmware matches the proposed usage.



BTA	Connector to an analogue port of the Infento controller
BTD	Connector to a digital port of the Infento controller
HBL	Heart beat led
PIR	Header for PIR sensor (external)
AUD	Header for an Adafruit FX sound board (external)
USS	Header for a breakout board of an ultrasonic sensor
CPS	Header for a breakout board of the HMC5883L compass
MIC	Header for a microphone
EXT	Header for external i2c or two pins data line device
LNS	Headers for five TCRT5000 ir-reflection sensors (line follower) These headers can be used for resistance sensors too
RES	This header has an programmable resistor attached

On the outside the two versions of the sensor board can be distinguished by the on board main sensor - either a line follower or a distance sensor (have an underscore in the following lists).

The first sensor board contains the firmware for:

- **Line follower**
- *Compass*
- Melody module
- **Microphone**
- *Luminance sensor*

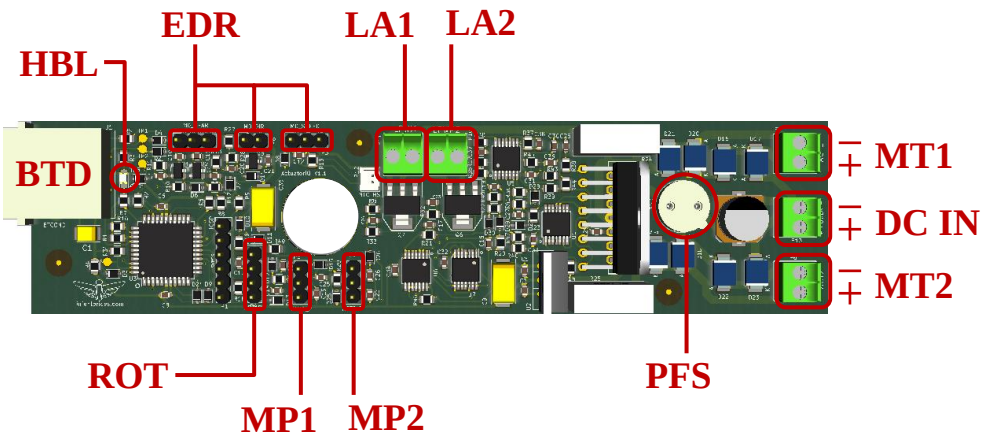
The other sensor board contains the firmware for:

- **Distance sensor**
- *Compass*
- **Microphone**
- Switch
- Melody module
- **PIR sensor**
- *Luminance sensor*
- Resistance
- Light gate

Bold printed sensors are present on the sensor board actually. The other sensors are connected to the board when required. They come with separate housings. Italic printed sensors are plugged into an i2c connector (EXT), the others use a regular three pin header.

Note that the melody module is not a sensor but an actuator. For technical reasons it found its way to the sensor board. It is connected to a six pin header (AUD).

The actuator board



BTD	Connector to a digital port of the Infento controller
HBL	Heart beat led
EDR	Headers for an e-drive motor (gear/direction/speed)
ROT	Header for a rotary sensor
LA1	Header for lamp 1 (max 25 Watt)
LA2	Header for lamp 2 (max 25 Watt)
MT1	Connector for motor 1 (max 1.5A)
MT2	Connector for motor 2 (max 1.5A)
MP1	Header for the position sensor of a linear actuator on MT1
MP2	Header for the position sensor of a linear actuator on MT2
DC IN	Power supply for LA1, LA2, MT1 and MT2
PFS	Fuse (5A) for the power supply

There is one actuator board only. It contains the firmware for:

- Infento e-drive motor
- Standard dc motor
- Rotary sensor
- Linear actuator
- Lamp

None of the actuators reside on board. In order to support a linear actuator, dc motor or lamp the board must powered by an external supply of 12V - 24V. The Infento e-drive motor is powered separately. In the e-drive system of Infento this motor is controlled by a gear handle. The actuator board simply replaces the handle.

Programming guide

Programming via the Arduino IDE

Preparation of the Arduino IDE

Here it is assumed that the Infento controller is programmed via the regular Arduino IDE. The Infento library is tested with version 1.6 of the IDE. One should install the Infento folder containing the library code files as a general Arduino library. It should be copied to the following folder in the *[Arduino]* installation directory:

[Arduino] \ hardware \ arduino \ avr \ libraries

The Infento library code should reside in a folder *Infento* and must contain at least the following files:

- Infento.cpp
- Infento.h
- InfentoCommands.h
- keywords.txt

Once the *Infento* library is installed, the Infento controller can be programmed using the Arduino IDE.

General program layout and behaviour

Be aware that each program should start by including the Infento and Wire header files. Then the required Infento devices can be declared. The hardware will be automatically found on the ports of the controller by calling the *connect(i2c-address)* routine of the corresponding device class. But before this is done, the ports of the controller must have been initiated by a call to *InfentoPort::begin()*. This should be the first call in the *setup* routine of the program. Furthermore the *loop* routine should start with a call to *InfentoPort::emergency()* in order to actuate automatic emergency handling. The emergency handler is responsible for blinking the heart beat led of the controller also.

The inset on the right shows how the above mentioned rules will look like in the Arduino IDE. In the example the *InfentoMelody* device is connected. When the device is found on the controller, the back led of the concerning port turns on. If the melody device was not connected to the controller, the back leds of all unoccupied ports will start blinking and the program sends the name of the missing device to the monitor of the Arduino IDE. After the device is plugged into one of the free ports, the controller should be reset [5]. Then the device is found after all and the program will continue.

```
#include "Wire.h"
#include "Infento.h"

InfentoMelody im;

void setup() {
   InfentoPort::begin();
   im.connect( 8);
   ...
}

void loop() {
   InfentoPort::emergency()
;
   ...
}
```

[5] Without a required device the behaviour of a robot can become unpredictable. Especially when the device would be inserted after all. For safety reasons a deliberate reset is needed to continue.

Port line 1 and 2

In addition to the regular i2c communication to an Infento device, data lines 1 and 2 of the Infento port are used as attention and emergency line respectively. Normally these lines are high. In case of aberrant behaviour of a device (i.e. a motor stops spinning unintentionally) the concerning device board will pull line 2 low. Line 1 is pulled low in order to inform the Infento controller that a requested situation (a sudden sensor value or motor position) is reached. The line is raised again on reading the sensor value or on sending another request to the actuator.

Emergency procedure

The correct response to an emergency call is handled by the *InfentoPort* class and is transparent to the programmer. Two situations can start an emergency procedure: 1) The emergency button of the Infento controller (EMS in the board layout) is activated or 2) one of the devices pulled it's emergency line 2 low. In response the *InfentoPort* class will pull the emergency line of all classes low in order to stop them. The program in the Infento controller is halted. When one of the devices started the emergency procedure, the status of the board is read and reported via the monitor of the Arduino IDE. The led behind the concerning port will start blinking.

Running cycle

In a program Infento device boards are distinguished by their i2c address. In the *setup* routine of the program a device class (derived from *InfentoDevice*) should call its *connect* member supplying the i2c address of the board. Upon calling *connect* the class *InfentoPort* will scan the ports and find the requested device. Now a device can be programmed.

Device classes offer three types of member functions:

- unit settings
- direct communication
- signal driven communication

Most of the time a device can work in different units (e.g. meters, inches, etc.) and the user can use the most convenient unit for his program. Often a device can set more then one unit type.

Direct and signal driven communication mirror each other in the available member functions. For example the *InfentoDistance* class, which supports the combined ultrasonic distance sensor and pir sensor, offers a routine *distance* for direct distance reading and a routine *signalDistance* for signal driven communication. In the latter case one can check if the required distance is reached by the *signal* function or wait until the distance reached by the *waitSignal* function. The code below demonstrates the typical usage the three approaches.

Direct communication

```
#include "Wire.h"
#include "Infento.h"

InfentoMelody im;
InfentoDistance id;

void setup() {
   InfentoPort::begin();
   im.connect(8);
   id.connect(8);
   id.setDistanceUnit(
               UNIT_MTR);
}

void loop() {
   InfentoPort::emergency();
   if ( id.distance() == 2 )
        im.play(1);
}
```

Status signalling

```
#include "Wire.h"
#include "Infento.h"

InfentoMelody im;
InfentoDistance id;

void setup() {
   InfentoPort::begin();
   im.connect(8);
   id.connect(8);
   id.setDistanceUnit(
                UNIT_MTR);
   id.signalDistance(2);
}

void loop() {
   InfentoPort::emergency();
   if ( id.signal() ) {
        im.play(1);
        id.signalDistance(2);
   }
}
```

Wait for a status

```
#include "Wire.h"
#include "Infento.h"

InfentoMelody im;
InfentoDistance id;

void setup() {
   InfentoPort::begin();
   im.connect(8);
   id.connect(8);
   id.setDistanceUnit(
               UNIT_MTR);
}

void loop() {
   id.signalDistance(2);
   id.waitSignal();
   im.play(1);
}
```

Although the direct communication at left gives a shorter code then the middle example, it results in continue data transfer to and from the sensor board. Thereby it slows down the communication and the working of the board. Both other examples depend on signalling via line 1 and there is no data transfer after *signalDistance* has been called. This approach guarantees a quick response of the robot.

Note that in the code most to the right the call to *InfentoPort::emergency* is dropped. There is no need to call it specifically here, since the routine *waitSignal* has this call implemented already.

The class InfentoPort

General layout

All ports of the Infento controller are represented by an instance of the class *InfentoPort*. They are labelled *DP1* through *IDP4* for the digital ports, *AP1* through *AP4* for the analogue ports and *NP1* for the NXT port. Respectively you can use *IFP[0]* through *IFP[3]*, *IP[4]* through *IFP[7]* and *IFP[8]* instead. You may use these labels in your code directly, for instance with the instruction *DP3.setPortMode(IFM_I2C).*

All data lines of the ports are represented by an individual label. The label begins with a port identifier (e.g. DP1 or AP1) and ends with a line identifier (e.g. L1 or L3). Correct labels for data lines look like *DP1L1* or *AP2L3* therefore. You may use these labels in your code directly, for instance with the instruction *digitalWrite(DP3L1, HIGH)*. Notice that some data lines are unavailable in sudden port modes. Line 1 and 2 are available in all port modes, but are used as attention and emergency lines for Infento devices. Line 3 and 4 are available in data mode only, for these lines are used by the i2c or uart interface.

At the beginning of a program, you should include the header files *Infentot.h* and *Wire.h*. Even if you won't use it yourself, *Wire.h* may not be left out, since the Arduino IDE won't link the Wire library needed for the class *InfentoPort* then. Furthermore one should place the instruction *InfentoPort::begin* in the *setup* routine of the program. This is a static member, taking the baud rate for the serial communication to uart devices as a parameter. It isn't possible to have all ports communicating on different baud rates, since they are wired together to one serial interface on the Arduino.

Debugging with the class *InfentoPort*

There is lot going on behind the screens of the class *InfentoPort* and thus there's a lot that can go wrong. A device can malfunction or the interaction with the program behaves unexpected. In order to find out where the problem lies, the class *InfentoPort* can output debug information over USB. Use an USB monitor (e.g. that of the Arduino IDE) to view this output.

You can bring the class *InfentoPort* into debug mode by writing the label *IP_DEBUG_ON* in your code. Debug is turned off again by the label *IP_DEBUG_OFF*. (Notice that there is no semicolon required behind these labels.) The labels can be applied in each part of the code, turning debug on and off as required. This way you can focus on that part of the code, where you expect a problem.

Reference guide

static void begin (int baud = 9600)

This routine starts communication to devices over i2c via *Wire* and over uart3 via *Serial3*. Communication to the PC is also started over uart0 via *Serial* using a fixed baud rate of 9600.The rx/tx pins of uart3 are available on the Infento ports as line 3 and 4. Note that not only all serial devices should use the same baud rate as given here, but also use the same serial protocol. Many modern devices are able to adjust their baud rate on the fly.

The *begin* routine should be called within the Arduino's *setup* before accessing the instances of the *InfentoPort* class.

void setPortMode (int portmode)

Normally the *setPortMode* routine is called during the Arduino's *setup* by the *InfentoDevice*'s *connect* routine and not directly. It must have been called before the actual communication over the port can start. The port is prepared and its connector back light is turned on to indicate that a device should be connected.

The available port modes are:

IPM_I2C - The device communicates over I2C [1]).
IPM_UART - The device communicates over UART3 (Serial3) [1]).
IPM_DATA - The device accesses the io pins of the Arduino board through the port lines directly.
IPM_REF - Like *IPM_DATA*, but now line 4 is connected to 5V via a reference resistor of 15K [2]).

[1]) In addition non Infento devices can access data lines 1 and 2.
[2]) Only available for analogue ports. For digital ports it is the same as IPM_DATA.

static InfentoPort* findDevicePort (int i2c_address)

Normally the *findDevicePort* routine is called during the Arduino's *setup* by the *InfentoDevice*'s *connect* routine and not directly. It scans the Infento controller ports if there is a device with the i2c address plugged in. If so, a pointer to the corresponding instance of *InfentoPort* is returned - else null.

Important note: When there are non i2c devices plugged into the controller, the *connect* routine of their *InfentoDevice* instances must be called before the *connect* routine of the i2c devices are called. If not, the *findDevicePort* routine will hang the program probably. The Arduino *Wire* library will try for contact on non i2c ports endlessly.

static void haltOnIncomplete ()

Normally the *haltOnIncomplete* routine is called during the Arduino's *setup* straight after the *InfentoDevice*'s *connect* routines have been invoked. If one of these connections failed the *haltOnIncomplete* routine will blink the back leds of all empty ports and halt the program.

static bool attention ()

This routine acts on ports that have Infento devices plugged in only. It scans these ports to check if one of the devices asks for attention (line 1 is pulled low). If so, the routine returns true.

static void waitAttention ()

Like the previous routine this one acts on ports that have Infento devices plugged in only. It postpones the continuation of the program until one of these devices asks for attention (line 1 is pulled low). Meanwhile the routine checks for *emergency*.

bool attention (int i2c_address, byte subid)

Normally this *attention* routine is called via the *InfentoDevice*'s *signal* and *waitSignal* routines and not directly. This version of the *attention* routine checks the attention line of the port with the device only and in addition tries if it was the device's subid that asked for attention.

static void emergency ()

The *emergency* routine should be called within the main *loop* of the Arduino, so that it is called as often as possible. It checks on emergency situations and will halt the program if one occurs. Since this is the only routine that should be called on regular basis, it is handling the heart beat led of the controller also. The emergency procedure has been described in the previous chapter already.

int infentoBoard (int i2c_address)

Normally the *infentoBoard* routine is called during the Arduino's *setup* by the *InfentoDevice*'s *connect* routine and not directly. It returns the id of the Infento board with a sudden i2c address. Based on the board id it is known which devices are connected via the board. The id is read automatically if an Infento device is recognized.

VernierParam vernierParameters (int i2c_address)

Normally the *vernierParameters* routine is called during the Arduino's *setup* by the *InfentoDevice*'s *connect* routine and not directly. The routine reads the Vernier sensor parameters by which the calibrated sensor values are calculated from the raw sensor data. The parameters are read automatically if a Vernier sensor is recognized.

static String portLabel (int ix)

A convenience function that returns the port label (e.g. AP1 or DP3) based on the port index. The ports are indexed 0 to 8 in the following order: DP1-DP4, AP1-AP4 and NP1. Returns an empty string if the index is outside the given bounderies.

InfentoPort* port (String label)

A convenience function that returns a pointer to the *InfentoPort* instance based on a port label (e.g. DP1 or AP3). Returns if the label is not recognized.

void setLed (boolean on)

Apart from the port NP1, all port connectors have got a back led that can be turned on or off. It is turned on by a call to *InfentoPort::setPortMode* or *InfentoDevice::connect* automatically to indicate that a device is expected to be connected. A program can override this behaviour by switching the back led via *setLed*.

int pin (int line)

Normally the *pin* routine is called via the *InfentoDevice*'s *line* routine and not directly. It returns the Arduino pin for a given line of the port. The pin number can be used to access the Arduino pin via calls like *pinMode*, *digitalWrite*, *digitalRead* or *analogRead*. Use the labels LINE1, LINE2, LINE3 and LINE4 to specify the line number. For indexed

references, this routine gives a safe way to address the Arduino pin of a port line, but it is a little slower than the direct access to the pin number array *PIN[n]*. In this member 'n' is calculated as '*(port – 1) • 4 + (line – 1)*'. A safe and fast option is the usage of the labels DPnLn ('n' stands for port and line number), but it cannot be used in indexed references. For example DP3L2 represents the Arduino pin connected to line 2 of digital port 3.

void write (byte data[], byte size, byte i2c_address = 0)

Normally the *InfentoPort*'s *write* routine is called via the *InfentoDevice*'s *write* routine and not directly. Use this member to send the specified number of bytes (size) from a buffer (data) to an i2c or uart device. It can be used in IPM_I2C or IPM_UART mode only. For i2c devices the usage of this *InfentoPort* member is compulsory, since it will select the required i2c bus of the Infento controller. When sending to an uart device, the i2c address is not specified.

int read (byte data[], byte size, byte i2c_address = 0)

Normally the *InfentoPort*'s *read* routine is called via the *InfentoDevice*'s *read* routine and not directly. Use this member to receive the specified number of bytes (size) from an i2c or uart device into a buffer (data). It can be used in IPM_I2C or IPM_UART mode only. For i2c devices the usage of this *InfentoPort* member is compulsory, since it will select the required i2c bus of the Infento controller. When receiving from an uart device, the i2c address is not specified. The number of bytes actually read is returned.

The class InfentoDevice

General layout

Class *InfentoDevice* depends on class *InfentoPort* for its communication to the devices. After a device is declared it needs to be connected to a port via *InfentoPort*. Once connected class *InfentoDevice* forms the entry point for the communication.

For i2c devices (like the Infento devices) the connection to an instance of *InfentoPort* is automated in the *connect* routine. When a user connects an i2c device to a different port, the program does not need to be modified. One simply supplies the i2c address of the device to the *connect* routine. From a programmers point of view, there is no need to know which instance of *InfentoPort* is involved. On the other hand, for uart and data pin devices a programmer must specify to which port a device should be connected and supply the corresponding instance of *InfentoPort* to the *connect* routine.

All Infento devices have their own class derived from *InfentoDevice*. In addition to the automated connection, they can take advantage of the attention system of the Infento controller and Infento device boards. When within the corresponding subclasses routines like *signalDistance* or *signalLevel* (starting with 'signal') are used, *InfentoDevice* will catch these signals via the routines *signal* and *waitSignal*.

Classes derived from *InfentoDevice* are not responsible for the correct data transfer to and from the devices. In other words, the *read*, *write*, *getValue* and *setValue* routines are not and should not be overloaded by the subclasses. Instead these routines should be called directly from them. Data transfer is handled in two stages: the actual transfer (*read* and *write*) and the type casting to the required data type (*getValue* and *setValue*). The typical approach is first *read* then *getValue* and first *setValue* then *write*.

Reference guide

bool connect (int i2c_address)

Connects to a device that supports i2c communication. The device is searched for on the ports and if found it is connected automatically to the concerning port.

Important note: When there are non i2c devices plugged into the controller, the *connect* routine of their *InfentoDevice* instances must be called before the *connect* routine of the i2c devices. If not, the program will hang probably. The Arduino *Wire* library will try for contact on non i2c ports endlessly.

bool connect (InfentoPort& ifp, int portmode)

Connects to a device that does not support i2c communication. The connecting process is not automated, so that the desired port and port mode must be supplied.

bool signal ()

Checks the connected port if there is an attention call from the i2c address of the device. This function works with Infento devices only.

void waitSignal ()

Checks the connected port if there is an attention call from the i2c address of the device. If not, the program is postponed until it is. This function works with Infento devices only.

void line (int linenumber, int set, bool pwm = false)

This routine supports direct writing to the pins of the Arduino. The device's port should have been put in data mode. There are several possibilities depending on if it is an analogue or digital port. When parameter *pwm* (pulse width modulation) is set to true, on digital ports the pin is written to by *analogWrite* - otherwise *digitalWrite*. Note that on the Arduino analogue pins cannot be written to by pwm. On the other hand, when not written to by pwm, both the digital and analogue ports will use *digitalWrite* to set the Arduino pin LOW or HIGH. The Arduino does accept digital access to its analogue pins.

int line (int linenumber)

This routine supports direct reading of the Arduino pins. The device's port should have been put in data mode. For devices connected to an analogue port *analogRead* is called, otherwise *digitalRead*.

bool write (int size)

Write transfers the devices data buffer to a device. The routine supports i2c and uart communication. Parameter *size* should correspond to the amount of data that a device expects. If this does not correspond to the amount of data in the data buffer then *write* will return false;

void read (int size)
Read fills the devices data buffer with data from a device. The routine supports i2c and uart communication. Parameter *size* should correspond to the amount of data that a device sends.

void setValue (bool value)
Sets one byte of data in the data buffer to 0 or 1.

void setValue (int value)
Sets two bytes of data in the data buffer to the value of the parameter.

void setValue (long value)
Sets four bytes of data in the data buffer to the value of the parameter.

void setValue (long value)
Sets four bytes of data in the data buffer to the value of the parameter.

void setValue (float value, int decimalpos)
Sets four bytes of data in the data buffer to the value of the parameter. First the float value is multiplied by $10^{decimalpos}$ and then treated as if it was a long value, neglecting the fractional part.

void setValue (String value)
Sets the data buffer to the value of the parameter. The string should be null terminated. Note that the data buffer can contain maximal 10 characters.

void getValue (bool& value)
Type casts the contents of the data buffer to *bool*.

void getValue (int& value)
Type casts the contents of the data buffer to *int*.

void getValue (long& value)
Type casts the contents of the data buffer to *long*.

void getValue (float& value, int decimalpos)
Type casts the contents of the data buffer to *float*. First the buffer is type casted to long and then divided by $10^{decimalpos}$ to get the required *float*.

void getValue (String& value)
Type casts the contents of the data buffer to *String*.

Derived classes

For all Infento devices there is a corresponding derived class from *InfentoDevice*. It is a good habit of programming to derive classes for Vernier or home made devices too. When doing so, you can take advantage of the *line*, *write*, *read*, *setValue* and *getValue* data transfer

framework of the *InfentoDevice* class in the same way as the Infento device classes do. Their routines let you focus on the working of the devices and handle the data transfer framework for you.

The next list matches the Infento devices with the board and the derived classes.

Infento device	Infento board	Derived class
E-drive motor	ActuatorIO	*InfentoMotor*
Linear actuator	ActuatorIO	*InfentoLinAct*
Rotory sensor	ActuatorIO	*InfentoRotary*
Lamp	ActuatorIO	*InfentoLamp*
Melody sound board	Sensor1IO + Sensor2IO	*InfentoMelody*
Line follower	Sensor1IO	*InfentoLineFollow*
Distance sensor	Sensor2IO	*InfentoDistance*
Compass	Sensor1IO + Sensor2IO	*InfentoCompass*
Microphone	Sensor1IO + Sensor2IO	*InfentoMicrophone*
Luminance sensor	Sensor1IO + Sensor2IO	*InfentoLuminance*
Resistance sensor	Sensor2IO	*InfentoResisance*
Switch	Sensor2IO	*InfentoSwitch*
Photo gate	Sensor2IO	*InfentoGate*

All device classes have the same layout of routines. There are routines that:

- set units for the devices
- set the maximum values for actuators
- set the working percentage of the maximum for actuators
- read the current sensor value
- ask to signal a sudden sensor value

Almost all routines that request to signal a sensor value have a parameter *func*, which determines when the sensor board should ask for attention. This can be when the value is reached (*func* = CONST_REACHED) or when the value is abandoned (*func* = CONST_RELEASED). In addition the value can have been reached or abandoned by increasing or decreasing it. Within the various device classes the increasing and decreasing can have different names. When it should be signalled when an increasing changes in decreasing or visa versa, the function is CONST_CHANGED. The functions CONST_REACHED, CONST_RELEASED and CONST_CHANGED are supposed to be known and won't be described with the classes below. The increasing and decreasing function are described.

There can be seven possible values for parameter *func*:

- CONST_REACHED
- CONST_REACHED + increased
- CONST_REACHED + decreased
- CONST_RELEASED
- CONST_RELEASED + increased
- CONST_RELEASED + decreased
- CONST_CHANGED

InfentoMotor

InfentoMotor (InfentoDefinition motornumber)
void setVelocityUnit (InfentoUnit unit)
void setDistanceUnit (InfentoUnit unit)
void setArcUnit (InfentoUnit unit)
void setTimeUnit (InfentoUnit unit)
void setDirection (InfentoConstant direction)
void setVelocity (int velocity)
void setAcceleration (int velocity)
void on (int speed = 100)
void off ()
void run (int time, int speed = 100)
void move (int distance, int speed = 100)
void turn (int arc, int speed = 100)
void attachRotary ()

The Infento actuator board can handle three motors: two general dc motors and one e-drive motor. These motors are connected to MT1, MT2 and EDR on the board (see above). Supply CONST_MOTOR1, CONST_MOTOR2, or CONST_EDRIVE as parameter to the class constructor in order to select the required motor. Additionally the actuator board supports one rotary sensor, so that only one of these motors can use it. A motor gets hold of the rotary by the *attachRotary* routine. The rotary sensor is needed for the routines *move* and *turn* to let them work. Note that instead of a dc motor one may connect a linear activator to MT1 or MT2 also (see the *InfentoLinAct* class).
The following units can be set for a motor.

- velocity — UNIT_MPH, UNIT_KPH, UNIT_MPS, UNIT_RPM
  (miles per hour, km per hour, meter per sec. and rotations per min.)
- distance — UNIT_MTR, UNIT_CM, UNIT_MM, UNIT_INCH
- arc — UNIT_TICK, UNIT_DEG, UNIT_GRAD
  (tick means rotary tick)
- time — UNIT_SEC, UNIT_MIN

The motor direction can be set to CONST_FORWARD and CONST_REVERSE.
The velocity is set in one decimal position. It is the maximum allowed velocity.
The acceleration is set in one decimal position as velocity unit per second.
The acceleration is applied to all velocity adjustments automatically.

The following routines will let the motor work actually.
Parameter *speed* sets the speed of the motor to a certain percentage of the velocity.

- on — turns the motor on
- off — turns the motor off
- run — lets the motor run for a sudden time (one decimal position)
- move — lets the motor move a sudden distance (one decimal position)
- turn — lets the motor turn for a sudden arc (one decimal position)

InfentoLinAct

InfentoLinAct (InfentoDefinition motornumber)
void setPositionUnit (InfentoUnit unit)
void moveTo (int position, int speed = 100)
void moveIn (int distance, int speed = 100)
void moveOut (int distance, int speed = 100)
void off ()
void setHome ()
void goHome ()

The Infento actuator board can handle two linear actuators. They are connected to MT1 and MT2 on the board (see above). Supply CONST_LINACT1 or CONST_LINACT2 as parameter to the class constructor in order to select the required actuator. Both linear actuator have a position sensor available for the position of the shaft.

The following unit can be set for a linear actuator.

- position UNIT_CM, UNIT_INCH
  (applicable for both position and distance)

The position is set in one decimal position.

The following routines will let the linear actuator work actually.
Parameter *speed* sets the speed to a certain percentage of the maximum of the actuator.

- moveTo moves the actuator to a sudden position
- moveIn slides the actuator for a sudden distance in
- moveOut slides the actuator for a sudden distance out

The routines *setHome* and *goHome* facilitate debugging a robot. (It is not possible to slide a linear activator manually.) The first routine makes the actuator board store the current position in its eeprom and the second lets the actuator move to this stored position. Before the programming one should run a small program with *setHome* and during the programming one should let the program start with *goHome* temporary.

InfentoRotary

InfentoRotary ()
void setDistanceUnit (InfentoUnit unit)
void setArcUnit (InfentoUnit unit)
int distance ()
int rotation ()
void signalDistance (int distance, InfentoConstant func = CONST_REACHED)
void signalArc (int arc, InfentoConstant func = CONST_REACHED)

Although the rotary sensor can be used on its own, most of the time it is needed for the e-drive motor. In that case the motor should attach the sensor to itself. When the rotary sensor is attached to a motor, the *move* and *turn* routines of the *InfentoMotor* class are enabled.

The following units can be set for the sensor.

- distance UNIT_MTR, UNIT_CM, UNIT_MM, UNIT_INCH
- arc UNIT_TICK, UNIT_DEG, UNIT_GRAD
  (tick means rotary tick)

The following routines will read the sensor.

- distance returns the moved distance (one decimal position)
- rotation returns the rotated arc (one decimal position)
- signalDistance lets the sensor signal the moved distance (one decimal position)
- signalArc lets the sensor signal the rotated arc (one decimal position)

The following additional functions can be signalled for the distance.

- CONST_APPROACH signalling happened while approaching
- CONST_LEAVING signalling happened while leaving

The following additional functions can be signalled for the arc.

- CONST_CLOCKWISE signalling happened while turning clockwise
- CONST_COUNTERCW signalling happened while turning anticlockwise

InfentoLamp

```
InfentoLamp (InfentoDefinition lampnumber)
void setLuminosity ( int luminosity)
void on ( int brightness = 100)
void blink (int time_a = 1000, int brightness_a = 100,
            int time_b = 1000, int brightness_b = 0)
void slide ( int time_a = 1000, int brightness_a = 100,
            int time_b = 1000, int brightness_b = 0)
void off ( bool immediate = false)
```

The Infento actuator board can handle two lamps. They are connected to LA1 and LA2 on the board (see above). Supply CONST_LAMP1 or CONST_LAMP2 as parameter to the class constructor in order to select the required actuator.

The luminosity is set in lumen (one decimal position). Use -1 for full brightness.

The following routines will let the lamp work actually.
The *brightness* parameters set the brightness to a certain percentage of the maximum of the luminosity.

- on switches the lamp on
- blink repeatedly the lamp switches between *brightness_a* for *time_a* and *brightness_b* for *time_b*
- slides repeatedly the lamp slides in *time_a* to *brightness_a* and in *time_b* to *brightness_b* and visa versa
- off switches the lamp off by completion the current cycle (except when *immediate* is set to true)

InfentoMelody

InfentoMelody()
void play(int tracknumber, int volume = 100)
void off()

InfentoMelody can play five different tracks from the Adafruit FX Sound board. The tracks are stored on the sound board directly via a mini USB cable.

The following routine will let the lamp work actually.
Parameter *volume* sets the volume to a certain percentage of the maximum of the sound board.

- play starts playing a track number
- off stops playing

InfentoLineFollow

InfentoLineFollow()
void setLineColour(InfentoConstant colour)
InfentoConstant position()
void signalPosition(InfentoConstant mask = (InfentoConstant) 0xFB,
InfentoConstant func = CONST_REACHED)

The line follower of Infento consists of five infrared leds and reflection sensors in a row, called far left, left, centre, right and far right. Measured from the centre, the inner and outer sensors are placed at a distance of 30 and 42 mm. You could use a 50 mm tape on the ground as the line to be followed, since the distance between the inner sensors is 60 mm. The sensor does not measure colours.

You may set the sensor to recognize dark tape on a light floor (CONST_DARK or visa versa (CONST_LIGHT).

The following routines will read the sensor.

- position returns the position of the sensor above the followed line.
- signalPosition lets the sensor signal if a sudden position is detected.

No additional functions can be signalled for the position.

The sensor position forms a compound of the values CONST_FARLEFT, CONST_LEFT, CONST_CENTRE, CONST_RIGHT and CONST_FARRIGHT. These values reflect the lowest five bits in a byte, respectively: 0x01, 0x02, 0x04, 0x08 and 0x10. Use a bitwise operators to find out which sensors have detected the line ('and' - &) or to set the sensors that should signal the line ('or' - |). In the last case, it is enough that one of the selected sensors detects the line in order to trigger an attention so that the function *signal* will return true.

Example:

```
InfentoLineFollow ilf;
ilf.signalPosition( CONST_FARLEFT | CONST_FARRIGHT);
ilf.waitSignal();
if ( ilf.position() & CONST_FARLEFT )
       // turn right
if ( ilf.position() & CONST_FARRIGHT )
       // turn left
```

InfentoDistance

InfentoDistance()
void setDistanceUnit(InfentoUnit unit)
bool objectAhead()
int distance()
void signalObjectAhead(InfentoConstant func = CONST_REACHED)
void signalDistance(int distance, InfentoConstant func = CONST_REACHED)

The Infento distances sensor exists of both an ultrasonic sensor and a motion (pir) sensor. Ultrasonic sensors can fail to detect soft materials like cloth so that a person in front of the robot would not be noticed. In such cases the motion sensor will detect the person - either because the person is moving in front of the robot or the robot is moving towards the person.

The following units can be set for the sensor.

- distance UNIT_MTR, UNIT_CM, UNIT_MM, UNIT_INCH

The following routines will read the sensor.

- distance returns the distance of an object (one decimal position)
- objectsAhead returns true if a moving object is detected
- signalDistance lets the sensor signal a sudden distance of an object (one decimal position)
- signalObjectAhead lets the sensor signal if a moving object is detected

The following additional functions can be signalled for the distance.

- CONST_APPROACH signalling happened while approaching
- CONST_LEAVING signalling happened while leaving

No additional functions can be signalled for object ahead.

InfentoCompass

InfentoCompass()
void setPrecision(int precision)
int azimuth()
InfentoDefinition orientation()
void signalAzimuth(int azimuth, InfentoConstant func = CONST_REACHED)
void signalOrientation(InfentoDefinition orientation,
InfentoConstant func = CONST_REACHED)

The Infento compass is of a type that is found in most smartphones. Although you can read the values in tenth of degrees, its accuracy will hardly ever become better then five degrees. Moreover, only very expensive compasses will give steady output, but smartphone compass types always fluctuate over time. Furthermore the reliability and maximum precision of a compass depends on its surroundings. For example inside a building it is far less then outside. And of course the robot itself will affect the results of the compass. In fact one should calibrate a compass always in position, but that is hardly doable with a live size robot. Thus it is the best way to have the compass assembled at the outside of a robot and away from its motors.

Depending on the robot construction and the place of usage, you should set a higher or lower precision of the heading.

The following routines will read the sensor.
Note that *azimuth* use a heading in degrees, while *orientation* uses labels like CONST_N, CONST_NE, CONST_E, CONST_SE, etc.

- azimuth returns the distance to an object (one decimal position)
- orientation returns true if a moving object is detected
- signalAzimuth lets the sensor signal a sudden distance to an object (one decimal position)
- signalOrientation lets the sensor signal if a moving object is detected

The following additional functions can be signalled for the azimuth and orientation.

- CONST_CLOCKWISE signalling happened while turning clockwise
- CONST_COUNTERCW signalling happened while turning anticlockwise

InfentoMicrophone

InfentoMicrophone()
void setLevelUnit(InfentoUnit unit)
int level()
void signalLevel(int level, InfentoConstant func = CONST_REACHED)

Both sensor boards have a microphone on board. When placed at opposite sides of a robot you may experiment with orientation based on sound levels.

The following units can be set for a microphone.

- level unit UNIT_PERC, UNIT_DB

Note that the unit decibel will generate rough indications of the perceived sound level only. The main difference between a percentage and decibel reading is that the first gives a linear and the latter a logarithmic interpretation of the sound levels.

The following routines will read the sensor.

- level returns the sound level (one decimal position)
- signalLevel lets the sensor signal a sudden sound level (one decimal position)

The following additional functions can be signalled for the sound level.

- CONST_INCREASE signalling while increasing the level
- CONST_DECREASE signalling happened while decreasing the level

InfentoLuminance

InfentoLuminance()
void setLevelUnit(InfentoUnit unit)
int level()
void signalLevel(int level)

The Infento luminance sensor consists of a lux meter. In combination with a Infento lamp you might find out that a change of the lumen of a lamp does not correspond to the change of the received lux by the luminance sensor. Lumen and lux are technically different quantities.

The following units can be set for a luminance sensor.

- level unit UNIT_PERC, UNIT_LUX

The following routines will read the sensor.

- level returns the brightness (one decimal position)
- signalLevel lets the sensor signal a sudden brightness (one decimal position)

The following additional functions can be signalled for the luminance level.

- CONST_INCREASE signalling happened while increasing the level
- CONST_DECREASE signalling happened while decreasing the level

InfentoResistance

InfentoResistance()
int level()
void signalLevel(int level, InfentoConstant func = CONST_REACHED)

This is not a special purpose sensor. Everything that outputs a changing resistance level can be used. Therefore the class works in percentage levels only.

The following routines will read the sensor.

- level returns the resistance level (one decimal position)
- signalLevel lets the sensor signal a sudden level (one decimal position)

The following additional functions can be signalled for the resistance level.

- CONST_INCREASE signalling happened while increasing the level
- CONST_DECREASE signalling happened while decreasing the level

InfentoSwitch

InfentoSwitch()
bool on()
bool off()
void signalOn()
void signalOff()

A switch can detect only two levels: on or off.

The following routines will read the sensor.

- on returns true if the switch is on
- off returns true if the switch is off
- signalOn lets the sensor signal if the switch is turned on
- signalOff lets the sensor signal if the switch is turned off

InfentoGate

InfentoGate()
bool open()
bool closed()
void signalOpened()
void signalClosed()

A gate can detect only two levels: open or closed. Most of the time the sensor will be used as a photo gate in order to detect if an object passes by.

The following routines will read the sensor.

- open returns true if the gate is open
- closed returns true if the gate is closed
- signalOpened lets the sensor signal when the gate is opened
- signalClosed lets the sensor signal when the gate is turned closed

Example program

The next code demonstrates how a robot can be programmed to follow a line on the floor. The robot starts moving straight ahead but will soon detect that the line is a little more to the right or left of the line follower. Then the robot will steer to left or right depending on how far it deviated from its course. The robot will postpone its journey when an object is discovered on the track.

Robot

```
#include "Infento.h";
#include "Wire.h"

InfentoMotor mleft( CONST_MOTOR1);
InfentoMotor mright( CONST_MOTOR2);
InfentoDistance dist;
InfentoLineFollow fllw;

void setup() {
  InfentoPort::begin();
  mleft.connect( 8);
  mright.connect( 8);
  dist.connect( 9);
  fllw.connect( 10);
  fllw.setLineColour( CONST_DARK);
  mright.on( 50);
  mleft.on( 50);
}

void loop() {
  fllw.signalPosition();
  dist.signalObjectAhead();

  InfentoPort::waitAttention();

  if ( dist.signal() ) {
    mright.off();
    mleft.off();
    dist.signalObjectAhead( CONST_RELEASED);
    dist.waitSignal();
    mright.on( 50);
    mleft.on( 50);
  }

  if ( fllw.signal() ) {
    InfentoConstant pos = fllw.position();
    if ( pos & CONST_FARLEFT ) { mright.on( 100); mleft.on( 50); }
    else
    if ( pos & CONST_LEFT ) { mright.on( 75); mleft.on( 50); }
    else
    if ( pos & CONST_FARRIGHT ) { mright.on( 50); mleft.on( 100); }
    else
    if ( pos & CONST_RIGHT ) { mright.on( 50); mleft.on( 75); }
  }
}
```