Tutorial

ROBOTC Driver Suite

Xander Soldaat

I'd Rather Be Building Robots

11-Apr-12

Contents

1 Introduction

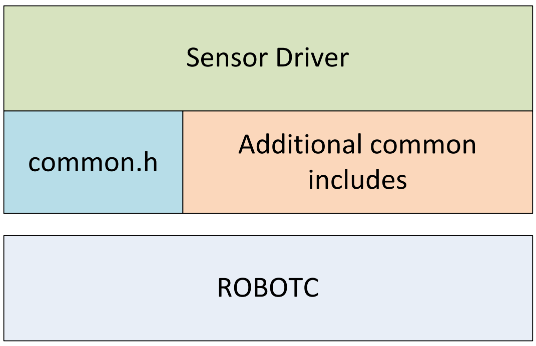
1.1 What is the ROBOTC Driver Suite?

ROBOTC comes with plenty of built in drivers for a wide variety of sensors. However, they don't expose all of the power that some of these sensors have. They will often allow you to read the sensor's primary value and nothing more. For example, the HiTechnic Compass can only be read, not calibrated through the standard drivers, you can't read the raw RGB values of a colour sensor. Other sensors simply aren't represented at all. There is a good reason for that, ROBOTC is a tool for the masses, which means that it caters to it. Some of the more exotic sensors on the market cater to a much smaller audience.

That's where the Driver Suite comes in. It started with me wanting support for the HiTechnic IRLink back in 2008 and now has support for 60+ sensors from many manufacturers. The aim of the Suite is to make the drivers as easy and straight-forward to use as the ones that are built-in to ROBOTC. The Suite will also expose most, if not all, of the functionality the sensor offers.

1.2 Design architecture

The majority of drivers in the Suite use the same architecture. After writing the first couple of drivers, it became clear that a lot of functionality pertaining to communicating with the sensors was duplicated. To avoid this kind of thing, the Suite uses a layered approach:



The sensor driver contains the logic and commands that the sensor requires to function. This is the part that you use in your own program.

The common.h include file contains the I2C communication code and a collection of utility functions. These functions are generally not used by your program, although you are free to do so.

The "Additional common includes" contain functions that are common to a group of sensors, for example, the Motor MUXes of Mindsensors and Holit Data Systems share a lot of features, so it makes sense to de-duplicate them.

The ROBOTC layer is not part of the suite.

1.3 Types of sensors

There are three main types of sensors:

- Analogue
- Digital using I2C
- Digital using RS485

1.3.1 Analogue

The analogue sensors are the simplest in their use. The NXT reads the voltage the sensor applies to the sensor port and converts it to a 10 bit digital value between 0 and 1023. 0 means 0 volts and 1023 is between 4.3V and 4.8V, depending on the load on the NXT.

1.3.2 Digital I2C

Digital sensors that use I2C are by far the most common of the digital sensors. They use an industry-standard communications protocol called I2C, which stands for Inter-Integrated Circuit. I2C uses a master/slave configuration where the NXT is the master and the sensor acts as the slave. Each slave has an address between 0 and 127, with the first bit used to signal whether the master wants to either write to (0) or read from (1) the slave.

Communicating with such sensors is a lot more complicated than simply reading a voltage on a pin. Although LEGO has defined a standard for communicating with these sensors, not all manufacturers adhere to this, for any number of reasons.

1.3.3 Digital using RS485

There are now several sensors on the market that use the NXT's high speed port, S4, which is able to communicate at up to almost 1Mb/s using RS485. Compared to the max speed of 10KB/s or 30Kb/s for I2C sensors, that's a massive speed increase.

1.4 Prerequisites

This tutorial assumes you are using at least ROBOTC 3.08. If you are not using this, or a later version, please download it from http://www.robotc.net/download/nxt/

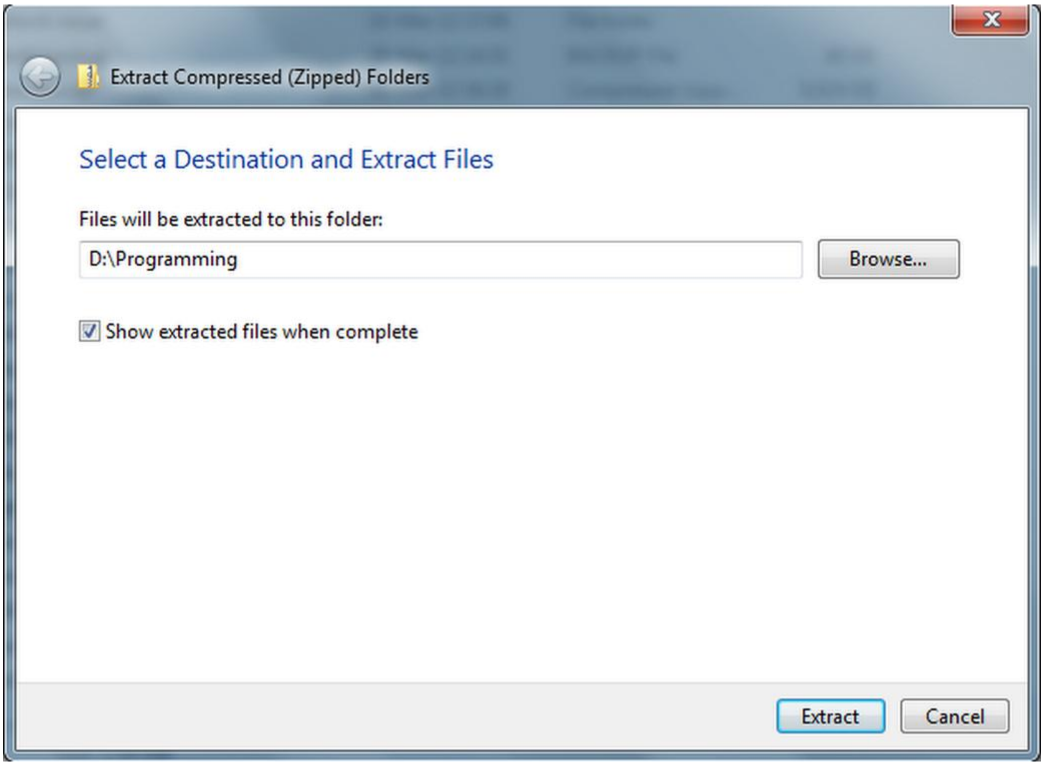
Please note that the Driver Suite has been written for the Mindstorms NXT and Tetrix platform and will not work with VEX Cortex or PIC.

If you have not already purchased ROBOTC, please consider doing so through my Associate link, doing so will help support me to continue writing sensor drivers and tutorials like this one! http://secure.softwarekey.com/solo/products/info.asp?A=91555. Thank you!

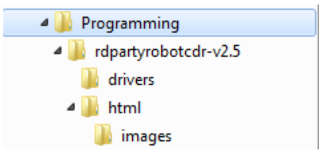
2 Setting up the drivers

2.1 Extracting the files

Download the latest version of the suite from the Project Page. Extract them to a folder of your choice. This example uses D:\Programming, but anything is fine. Preferably, extract it into a folder where you normally store your ROBOTC programs.

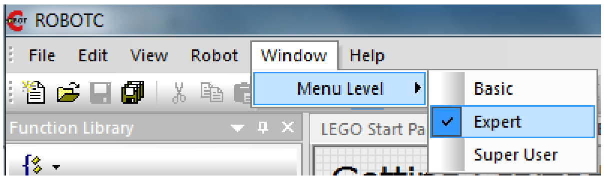


This should create the following folder structure under C:\Programming

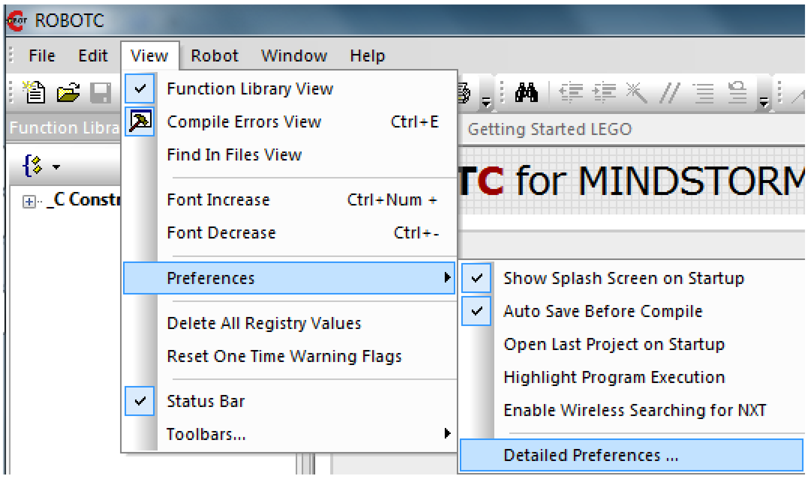


2.2 Setting up ROBOTC

Navigate to the Menu Level option in the Window menu and select “Expert”. This will give access to additional options in the preferences, which are needed to use the Driver Suite.



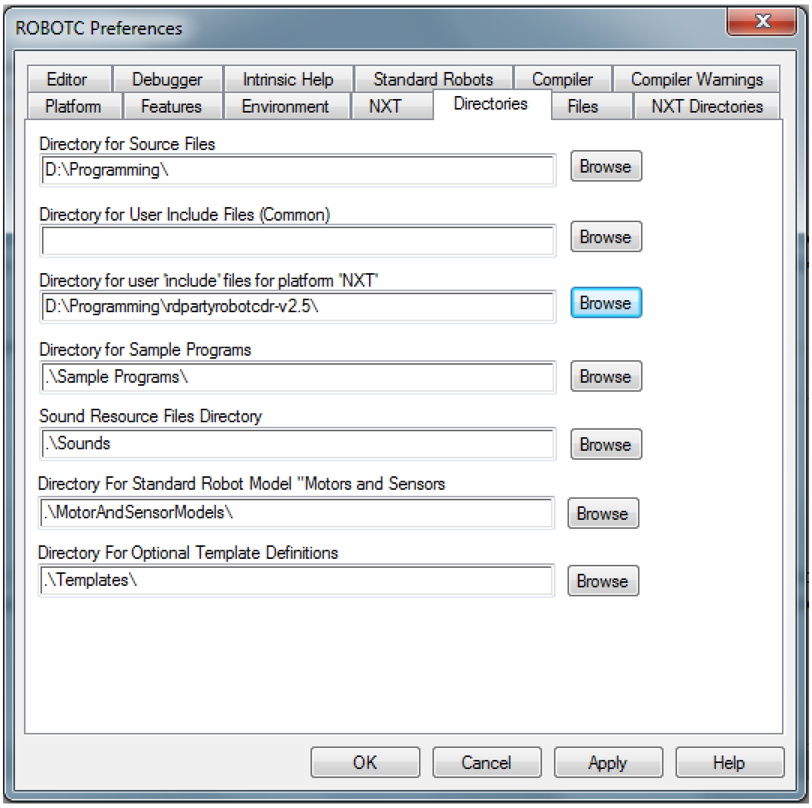
Open the Preferences UI by navigating to the View menu and selecting “Detailed Preferences”



Select the “Directories” tab in the Preferences UI.

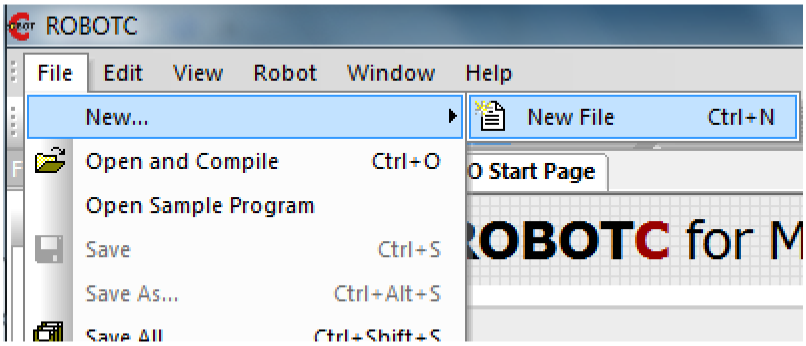
Click on the Browse button for the Source Files directory path and select whatever path you’ve been using to store your ROBOTC programs. This tutorial assumes you are using D:\Programming

Click on the Browse button next to the Include files for Platform NXT and select D:\Programming\rdpartyrobotcdr-2.5. The final result should resemble the window below:

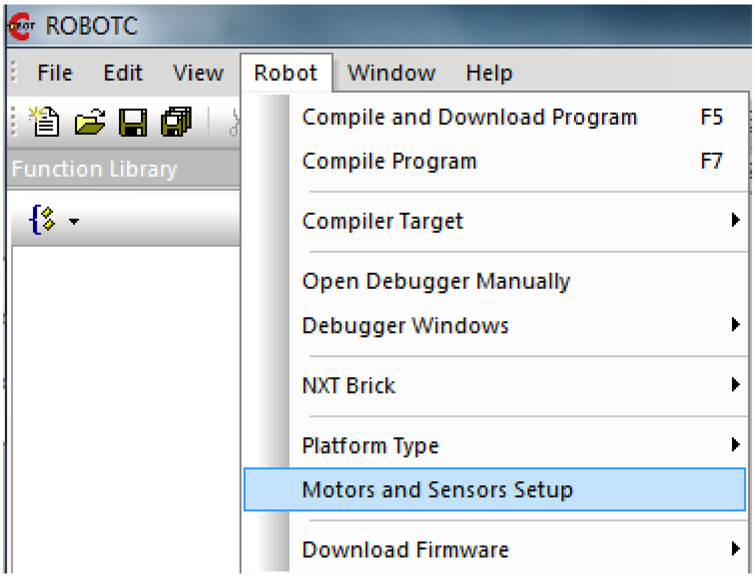


To get access to the custom I2C sensors in your Motor and Sensors setup UI, do the following:

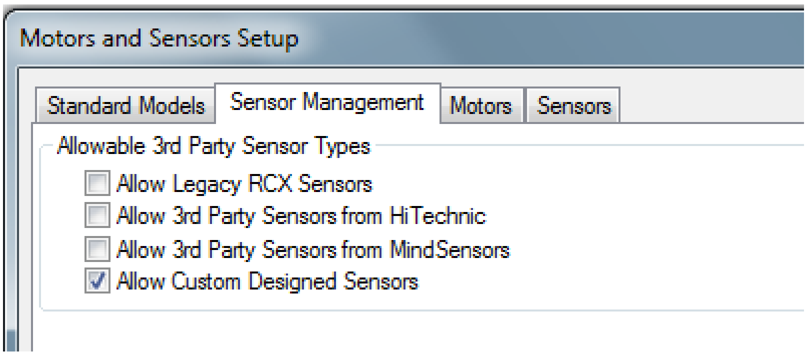
Open a new file:



Now open the Motor and Sensor Setup from the Robot menu



Navigate to the Sensor Management tab and ensure the “Allow Custom Designed Sensors” is ticked and click OK.



ROBOTC is now ready to go!

3 Getting started

Now that you have a basic understanding of how the Suite has been designed and your ROBOTC environment has been setup, it is time to get our hands dirty. I have created a number of examples of how to create your own program from scratch. I realise that not all people who read this will have those sensors but lucky for you, the Suite comes with at least one example program for each sensor driver, often more. You will have to draw your own parallels between those programs and the examples provided in this tutorial.

When using the Driver Suite, it is imperative that you don't use the built-in device specific sensor drivers that are part of ROBOTC. They will cause a conflict and will give you very strange sensor values or nothing at all. You must always check the example of the individual sensor driver to ensure you are using the right sensor type.

Another important thing to note is that you should always use the functions from the Driver Suite and **not** ROBOTC's own `SensorValue[]` variable. It will not contain any meaningful information and may interfere with the Driver Suite.

3.1 Example 1: The fields are alive with the sound of…beeping

The Dexter Industries dCompass is a 3D compass sensor; it can detect magnetic field strengths on all three axes. Normally you would use two of these axes (X and Y) to calculate your current heading. However, using a little math, you can easily use it to detect total magnetic field strength. This is very useful if you want to use it to find cables in the wall or other metal objects.

To calculate the total field strength you can use the following formula:

$$strength = \sqrt{fieldX^2 + fieldY^2 + fieldZ^2}$$

In ROBOTC code that would look like:

```
int fieldX = 0;
int fieldY = 0;
int fieldZ = 0;
int strength = 0;

strength = sqrt(pow(fieldX, 2) + pow(fieldY, 2) + pow(fieldZ, 2));
```

If we look at the documentation for the Dexter Industries dCompass, we can see there is a function called `DIMCreadAxes()`, which allows us to read all three fields in one go:

```
bool DIMCreadAxes ( tSensors link,
                    int &    _x,
                    int &    _y,
                    int &    _z
                  )
```

Read all three axes of the Compass

Parameters:

link	the port number
_x	data for x axis in degrees per second
_y	data for y axis in degrees per second
_z	data for z axis in degrees per second

Returns:
true if no error occured, false if it did

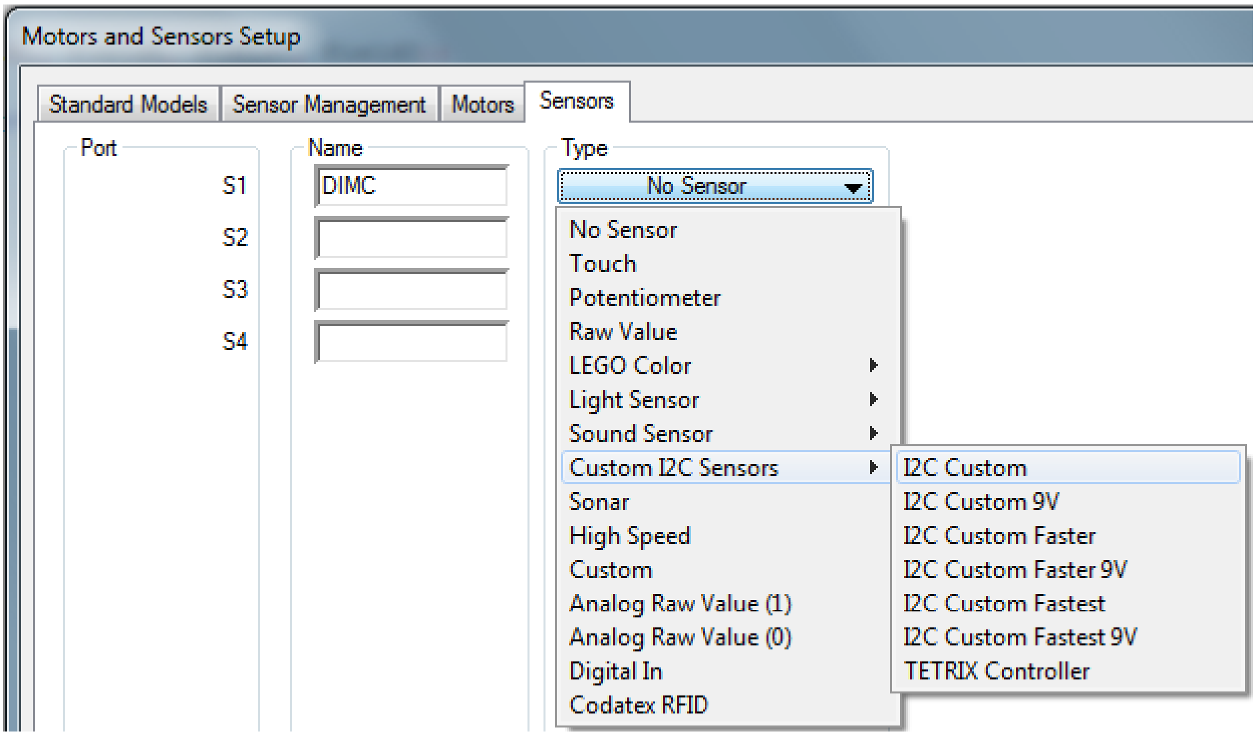
Examples:
DIMC-test1.c.

Definition at line 182 of file DIMC-driver.h.

Assuming the dCompass is connected to Sensor Port 1 (S1), the code for using this would looks like this:

```
int fieldX = 0;
int fieldY = 0;
int fieldZ = 0;
int strength = 0;

DIMCreadAxes(S1, fieldX, fieldY, fieldZ);

strength = sqrt(pow(fieldX, 2) + pow(fieldY, 2) + pow(fieldZ, 2));
```

Before we can really use the sensor, it needs to be configured. The dCompass is a digital sensor that uses I2C to communicate with the NXT. Open the Motors and Sensors Setup tool and navigate to the Sensors tab. From the drop down menu, select “I2C Custom” and give the sensor a name like “DIMC” or “COMPASS”



This will generate a set of “pragma” statements at the top of the program:

```
#pragma config(Sensor, S1,     DIMC,           sensorI2CCustom)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
```

These statements are used by the compiler to ensure the sensor is configured correctly.

The dCompass needs to be initialised before we can start reading from it; the correct gain, sample rate and mode need to be selected. This is done with `DIMCinit()`:

bool DIMCinit (tSensors link)

Configure the Compass

Parameters:
link the port number

Returns:
true if no error occured, false if it did

Definition at line **143** of file **DIMC-driver.h**.

In order to allow the compiler to find the driver, `DIMC-driver.h` in this case, we'll have to add an `#include` statement below the pragma. Putting it all together, the program will end up looking like this:

```
#pragma config(Sensor, S1,     DIMC,           sensorI2CCustom)
//*!!Code automatically generated by 'ROBOTC' configuration wizard

#include "drivers/DIMC-driver.h"

task main ()
{
  int fieldX = 0;
  int fieldY = 0;
  int fieldZ = 0;
  int strength = 0;

  // Initialise the sensor
  DIMCinit(DIMC);

  // Loop forever, reading the sensor and calulating total
  // field strength
  while (true)
  {
    // read the individual axes
    DIMCreadAxes(DIMC, fieldX, fieldY, fieldZ);

    // calculate the field strength
    strength = sqrt(pow(fieldX, 2) + pow(fieldY, 2) + pow(fieldZ, 2));

    // display on the screen
    nxtDisplayCenteredBigTextLine(3, "%d", strength);
    wait1Msec(50);
  }
}
```

Our magnetic field detector is now ready to go! If you attach the sensor to the end of a beam like in the picture below, you can wave it around like a magic wand.

You can watch a video of it on YouTube: http://www.youtube.com/watch?v=icrtNMrK3qY

The source code for this program is available in the Driver Suite , just open `DIMC-test3.c` in ROBOTC.

3.2 Example 2: Heading somewhere?

The HiTechnic Gyroscopic Sensor is an analogue sensor that returns the current rate of rotation (angular velocity) in degrees per second.

Most of the time this sensor is used in things like balancing robots but you can also use it as a compass of sorts.

How would that work? Seeing as the sensor measures rate of turn in degrees per second, we can simply integrate to find the number of degrees the sensor has turned. In other words, if we keep track of how fast we've been turning at constant intervals, we can keep track of our current position. A more mathematical approach (thanks, Laurens Valk) can be found below:

If the angular velocity at time t is given as $\omega(t)$,

$$\omega(t) = \frac{d\varphi(t)}{dt} \quad [deg/s]$$

Then the robot's heading $\varphi(t)$ $[deg]$ at time t $[s]$ is given as follows

$$\varphi(t) = \int_0^t \omega(t)dt \qquad [deg]$$

Here, $t = 0$ $[s]$ is defined as the start of the program, so t describes the time elapsed since the beginning of the program.

Actual measurement is not continuous. The angular velocity is measured once at each interval Δt $[s]$. For example, if an interval is 20ms, then $\Delta t = 0.02$. $\varphi(t)$ can therefore only be found as the sum of small angle changes:

$$\varphi(t) = \int_0^t \omega(t)dt \ \approx \sum_{i=0}^{t/\Delta t} \omega(i \cdot \Delta t)\,\Delta t$$

This is evaluated with a loop, from i = 0 to $t/\Delta t$ (the amount of measurements). $\omega(i \cdot \Delta t)$ is the sensor measurement at interval i, so at time $i \cdot \Delta t$. Multiplying this measurement by Δt gives the small angle change during that interval. Adding all these small angle changes gives the total change in angle, which is your robot's current heading.

In ROBOTC, we're going to create a loop that always takes 20ms to complete. This makes it much easier to determine Δt, which will then simply be 20ms, or 0.02s. The resulting code looks like this:

```
while (true)
{
  // Wait until 20ms has passed
  while (time1[T1] < 20)
    wait1Msec(1);

  // Reset the timer
  time1[T1]=0;

  // Calculate the new heading by adding the amount of degrees
  // we've turned in the last 20ms
  // If our current rate of rotation is 100 degrees/second,
  // then we will have turned 100 * (20/1000) = 2 degrees since
  // the last time we measured.
  heading += rotSpeed * 0.02;
}
```

It is sometimes surprising how little code the math boils down to. This is not always the case, though. Keep in mind that if you make Δt smaller, you may get more accurate results. 20ms was chosen to allow the program to do other things as well, which we'll add later.

The angular velocity can be read with the HTGYROreadRot() function:

float HTGYROreadRot (tSensors link)

Read the value of the gyro

Parameters:
link the HTGYRO port number

Returns:
the value of the gyro

Examples:
HTGYRO-SMUX-test1.c, and **HTGYRO-test1.c**.

Definition at line **70** of file **HTGYRO-driver.h**.

It is important to note that the Gyro is a sensor that needs to be calibrated before you use it. If you were to read the sensor's raw value, its value would be around 620 at rest. Negative angular velocity would take away from that value, positve velocity would add to it. This isn't very useful and "around" isn't the level of accuracy we need if we want to calculate the heading. This is where HTGYROstartCal() comes in:

float HTGYROstartCal (tSensors link)

Calibrate the gyro by calculating the average offset of 5 raw readings.

Parameters:
link the HTGYRO port number

Returns:
the new offset value for the gyro

Examples:
HTGYRO-SMUX-test1.c, and **HTGYRO-test1.c**.

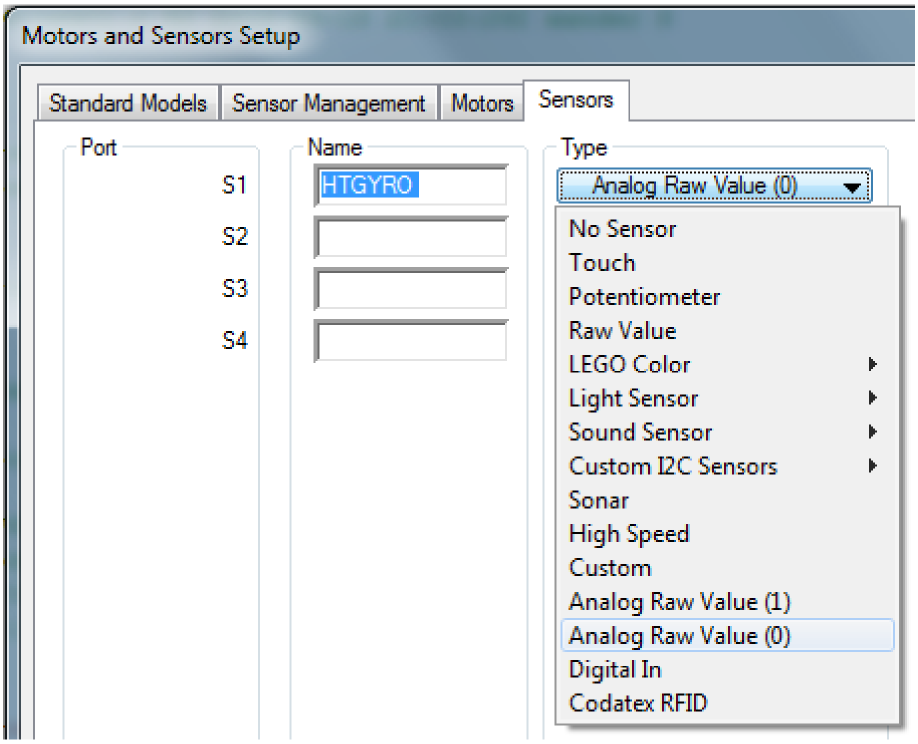
Definition at line **98** of file **HTGYRO-driver.h**.

Once we have this offset value which is based on a number of measurements taken while the sensor was at rest, it is subtracted from all measurements returned by HTGYROreadRot(). That means that the angular velocity values are now -300 to +300 deg/sec, rather than 320 deg/s to 920 deg/s.

Assuming the Gyro is hooked up to Sensor Port 1 (S1), our code now looks like this:

```
// Calibrate the gyro, make sure you hold the sensor still
HTGYROstartCal(S1);

// Reset the timer.
time1[T1] = 0;

while (true)
{
  // Wait until 20ms has passed
  while (time1[T1] < 20)
    wait1Msec(1);

  // Reset the timer
  time1[T1]=0;

  // Read the current rotation speed
  rotSpeed = HTGYROreadRot(S1);

  // Calculate the new heading by adding the amount of degrees
  // we've turned in the last 20ms
  // If our current rate of rotation is 100 degrees/second,
  // then we will have turned 100 * (20/1000) = 2 degrees since
  // the last time we measured.
  heading += rotSpeed * 0.02;
}
```

ROBOTC will need to be configured to use this sensor correctly, and to do that we have to use the Motors and Sensor Setup tool. Configure the sensor to be “Analogue Raw Value (0)” and give it a meaningful name like HTGYRO:



That should generate the following pragma statements at the top of your file:

```
#pragma config(Sensor, S1,     HTGYRO,          sensorAnalogInactive)
//*!!Code automatically generated by 'ROBOTC' configuration wizard          !!*//
```

Next step is to add the #include statement to ensure the compiler can find the driver, delcare the variables we need and display our current heading on the screen. Below you can see the final result:

```
#pragma config(Sensor, S1,     HTGYRO,              sensorAnalogInactive)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "drivers/HTGYRO-driver.h"

task main ()
{
  float rotSpeed = 0;
  float heading = 0;

  // Calibrate the gyro, make sure you hold the sensor still
  HTGYROstartCal(HTGYRO);

  // Reset the timer.
  time1[T1] = 0;

  while (true)
  {
    // Wait until 20ms has passed
    while (time1[T1] < 20)
      wait1Msec(1);

    // Reset the timer
    time1[T1]=0;

    // Read the current rotation speed
    rotSpeed = HTGYROreadRot(HTGYRO);

    // Calculate the new heading by adding the amount of degrees
    // we've turned in the last 20ms
    // If our current rate of rotation is 100 degrees/second,
    // then we will have turned 100 * (20/1000) = 2 degrees since
    // the last time we measured.
    heading += rotSpeed * 0.02;

    // Display our current heading on the screen
    nxtDisplayCenteredBigTextLine(3, "%2.0f", heading);
  }
}
```

Just mount the sensor to the side of your NXT brick and rotate it around so you can see the heading change. You can watch a video of it in action here: http://www.youtube.com/watch?v=2bAcR3hktKs

This program is part of the Suite and can be found by opening HTGYRO-test2.c in ROBOTC.

3.3 Example 3: Tilting Tones

The Mindsensors ACCEL-Nx is a digital acceleration and tilt sensor that can measure X, Y and Z axes values. This sensor is great if you want to know how quickly you are accelerating, for collision detection or, using the tilt data, check your robot is upside down or not.

We're not going to do anything sensible with this sensor, though, that would be too easy. This example is going to a noisy one, like the first one, only more annoying; we're going to use the 3D tilt data to control sounds generated by the NXT.

There's no maths involved in this example, we're going to generate two sounds based on the X and Z axes and control the time between the sounds using the Y tilt data.

The tilt data generated by the sensor seems to go between -20 and 20, so if we want to use the whole range, we'll have to add 20 to the value. Just to be sure, we'll force the data to always be 0 or more. This is the ROBOTC code to turn the tilt data into two frequencies and a wait time:

```
// Tilt values seem to go from about -20 to +20.
// Adding 20 to them makes them go from 0 to 40.

// Make sure the tones are at least 0Hz
tone1 = max2(0, (_x_tilt + 20) * 20);
tone2 = max2(0, (_z_tilt + 20) * 25);

// Make sure the wait time is at least 10ms
waitTime = max2(10, (_y_tilt + 20));
```

Multiplying the titlt data by 20 and 25 makes the frequencies a little more interesting. The max2 function takes the biggest number in the arguments given. That means that if the tilt data does go below 0, `max2()` will simply return 0. This function is actually part of the Driver Suite's common.h file:

```
#define max2 (  a,
                b
             )   (a > b ? a : b)
```

This function returns the bigger of the two numbers

Definition at line 103 of file **common.h**.

There is also a `min2()`, `min3()` and a `max3()`, which return the smallest of two, smallest of three and largest of three numbers, respectively.

The tilt data can be read from the sensor using `MSACreadTilt()`:

```
bool MSACreadTilt ( tSensors  link,
                    int &     x_tilt,
                    int &     y_tilt,
                    int &     z_tilt
                  )
```

Read tilt data from the sensor

Parameters:

link	the sensor port number
x_tilt	X tilt data
y_tilt	Y tilt data
z_tilt	Z tilt data

Returns:
true if no error occured, false if it did

Examples:
MSAC-test1.c.

Definition at line **76** of file **MSAC-driver.h.**

It is important to understand that the ACCEL-nNx is capable of operating on a number of scales and needs to be configured properly before you can start reading from it. The scale we're going to use is ±10G. The scale is set using `MSACsetRange()`:

```
bool MSACsetRange ( tSensors  link,
                    int       range
                  )
```

Set sensitivity range of sensor.

Parameters:

link	the sensor port number
range	1 = 2.5G, 2 = 3.3G, 3 = 6.7G, 4 = 10G

Returns:
true if no error occured, false if it did

Examples:
MSAC-test1.c.

Definition at line **150** of file **MSAC-driver.h.**

Instead of the numbers 1 – 4, you can also use these aliases (defines):

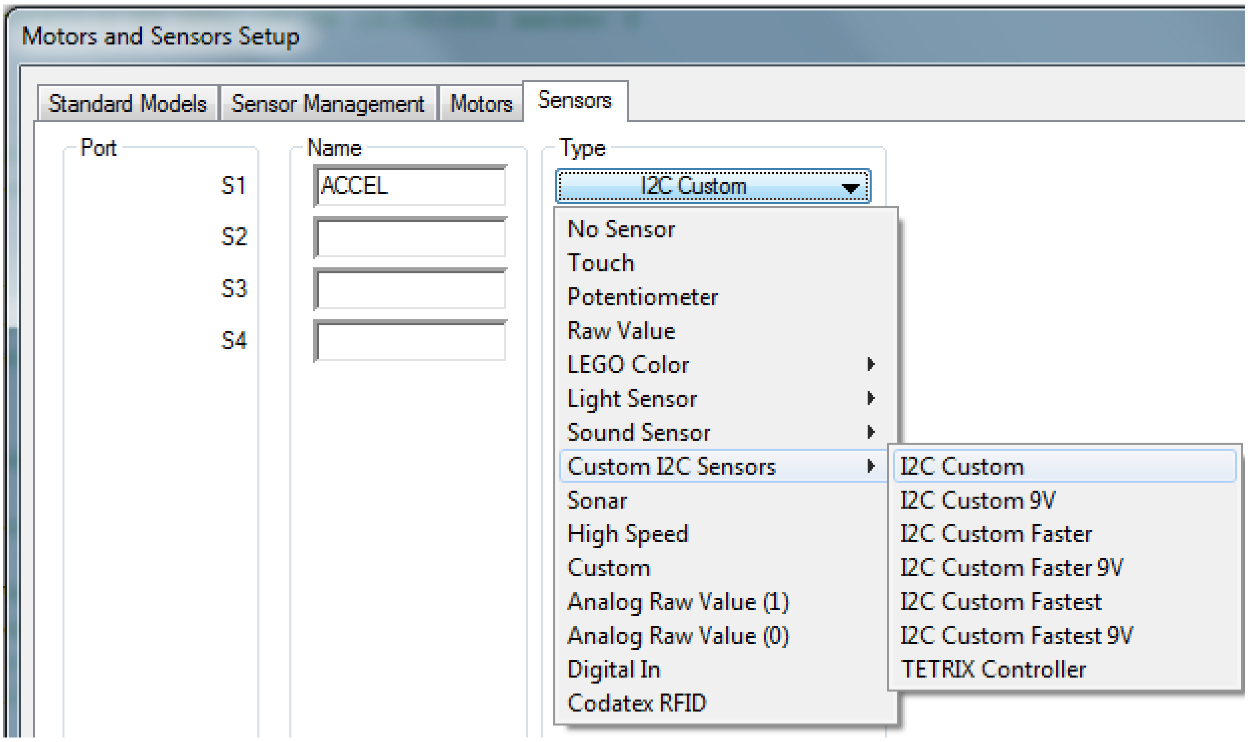
```
#define MSAC_RANGE_2_5  1
#define MSAC_RANGE_3_3  2
#define MSAC_RANGE_6_7  3
#define MSAC_RANGE_10   4
```

Using these will make your code a lot more readable.

Putting it all together, we've got the following ROBOTC code:

```
// There are four ranges the ACCL-Nx can measure in
// up to 2.5G - MSAC_RANGE_2_5
// up to 3.3G - MSAC_RANGE_3_3
// up to 6.7G - MSAC_RANGE_6_7
// up to 10G  - MSAC_RANGE_10
MSACsetRange(S1, MSAC_RANGE_10);

while (true) {
  // Read the tilt data from the sensor
  MSACreadTilt(S1, _x_tilt, _y_tilt, _z_tilt);

  // Tilt values seem to go from about -20 to +20.
  // Adding 20 to them makes them go from 0 to 40.

  // Make sure the tones are at least 0Hz
  tone1 = max2(0, (_x_tilt + 20) * 20);
  tone2 = max2(0, (_z_tilt + 20) * 25);

  // Make sure the wait time is at least 10ms
  waitTime = max2(10, (_y_tilt + 20));

  PlayImmediateTone(tone1, 5);
  wait1Msec(waitTime);
  PlayImmediateTone(tone2, 1);
}
```

This is a digital sensor, so we'll open the Motors and Sensors Setup tool and navigate to the Sensors tab. From the drop down menu, select "I2C Custom" and give the sensor a name like "ACCEL".



Putting it all together, we end up with the code below:

```
#pragma config(Sensor, S1,     MSAC,                sensorI2CCustom)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "drivers/MSAC-driver.h"

task main () {
  int _x_tilt = 0;
  int _y_tilt = 0;
  int _z_tilt = 0;

  int tone1;
  int tone2;
  int waitTime;

  // There are four ranges the ACCL-Nx can measure in
  // up to 2.5G - MSAC_RANGE_2_5
  // up to 3.3G - MSAC_RANGE_3_3
  // up to 6.7G - MSAC_RANGE_6_7
  // up to 10G  - MSAC_RANGE_10
  MSACsetRange(MSAC, MSAC_RANGE_10);

  while (true) {
    // Read the tilt data from the sensor
    if (!MSACreadTilt(MSAC, _x_tilt, _y_tilt, _z_tilt)) {
      nxtDisplayTextLine(4, "ERROR!!");
      wait1Msec(2000);
      StopAllTasks();
    }

    // Tilt values seem to go from about -20 to +20.
    // Adding 20 to them makes them go from 0 to 40.

    // Make sure the tones are at least 0Hz
    tone1 = max2(0, (_x_tilt + 20) * 20);
    tone2 = max2(0, (_z_tilt + 20) * 25);

    // Make sure the wait time is at least 10ms
    waitTime = max2(10, (_y_tilt + 20));

    PlayImmediateTone(tone1, 5);
    wait1Msec(waitTime);
    PlayImmediateTone(tone2, 1);
  }
}
```

So now all you need to do is hook up your accelerometer to the side of your brick, connect it to Sensor Port 1 and run it. You can find the complete program in the Driver Suite, just open MSAC-test2.c in ROBOTC.

If you're curious to find out what mine sounded like, be sure to check out the video: http://www.youtube.com/watch?v=PjGu2g-oVZk

4 Advanced Topics

4.1 Turning your sensor all the way to eleven

ROBOTC has a nifty feature that allows you to communicate with your your digital I2C sensors at a much higher clock speed. The standard clock speed is around 10 KHz, which allows you to squeeze about 1200 bytes/s. That excludes overhead of addresses, start and stop conditions. It comes down to about 165 I2C transactions if you're only reading 1 byte from your sensor, much less if you're looking to read multiple bytes.

You can speed things up dramatically if you use ROBOTC's fast I2C clock speed, which runs at 30KHz, 3 times the normal speed. Below are the results of some read tests that were performed a little while ago:

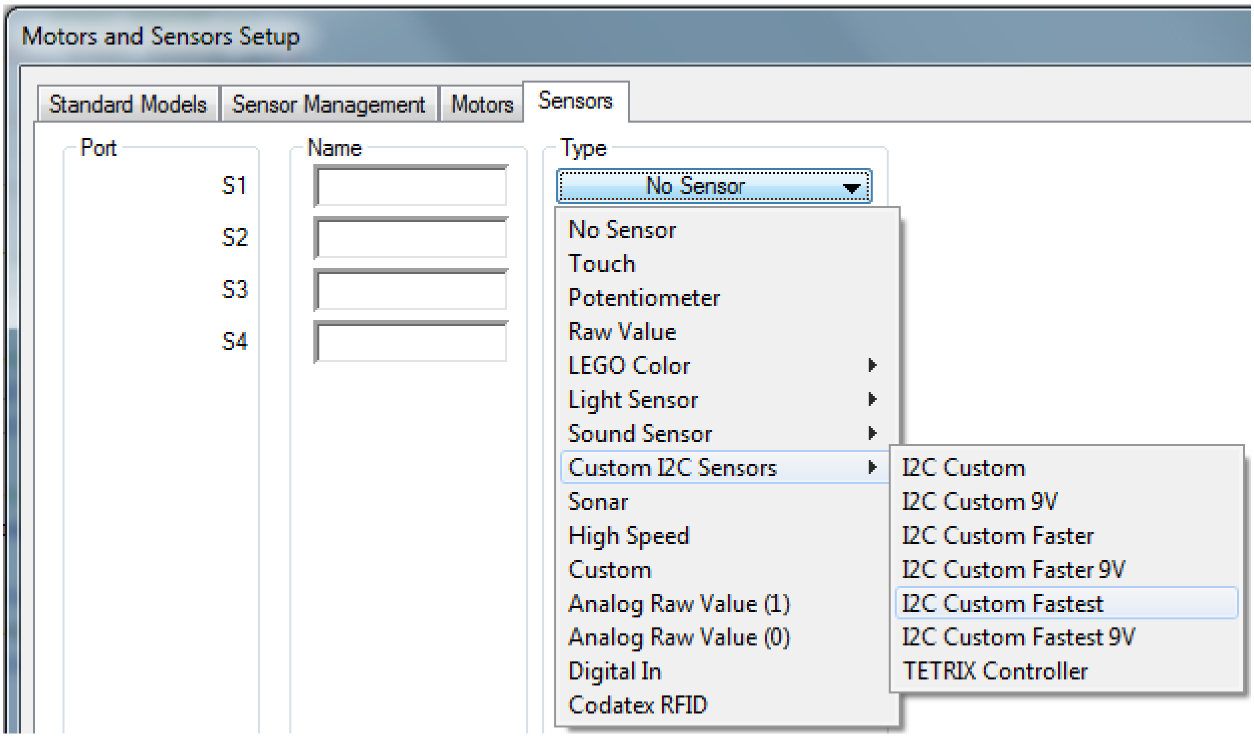
Transactions per second		
Bytes	Standard Clock	Fast Clock
1	167	501
2	143	500
3	125	500
4	125	334
5	111	333
6	100	333
7	91	251
8	83	250
9	77	250
10	71	220
11	67	200
12	63	200
13	59	187
14	56	167
15	53	167
16	50	166

Unfortunately, not all sensors like to communicate at this speed. HiTechnic and LEGO sensors tend to behave better at the lower speed but Dexter Industries, Mindsensors and several others that have been tried, seem to have no problems with the higher clock speeds.

Why you would even consider cranking up the speed? A good example would be when using multiple sensors that require you to read 6 bytes at once, like the Mindsensors ACCEL-nx. Your sensor reading loop can now go around much quicker, which can be a very good thing if speed is of the essence. Who needs a soccer bot that plays in slow-mo?

4.2 Configuring ROBOTC to use faster I2C

To configure ROBOTC to use the high speed I2C clock, simply open the Motors and Sensors Setup tool and navigate to the Custom I2C Sensors sensor type. Depending on whether you sensor requires a 9V supply or not, you can pick either "I2C Custom Fastest" or "I2C Custom Fastest 9V". Don't bother with the "Faster" ones, they're pretty useless.



Don't worry about breaking your sensor by trying out the higher speeds. The worst that can happen is that you cause the sensor to hang, which is easily remedied by unplugging the cable and plugging it back in. Then just reconfigure the sensor to use the normal clock speed, which is "I2C Custom" or "I2C Custom 9V".

The pragmas for the various types:

```
#pragma config(Sensor, S1,      NORMAL,         sensorI2CCustom)
#pragma config(Sensor, S2,      FASTEST,        sensorI2CCustomFastSkipStates)
#pragma config(Sensor, S3,      NORMAL_9V,      sensorI2CCustom9V)
#pragma config(Sensor, S4,      FASTEST_9V,     sensorI2CCustomFastSkipStates9V)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
```

You don't need to change anything in your code, other than the pragmas to make use of the faster clock speed. Do keep in mind, though, that a transaction between the NXT and sensor now takes roughly a third of the time it did before. If you have specific timing in your code, you may need to compensate for that.

5 Sensors and Driver Names

5.1 Dexter Industries

Name	Description
DFLEX-driver.h	ROBOTC Dexter Industries dFlex Sensor driver
DGPS-driver.h	Dexter Industries GPS Sensor driver
DIMC-driver.h	Dexter Industries IMU Sensor driver
DIMU-driver.h	Dexter Industries IMU Sensor driver
DPRESS-driver.h	ROBOTC dPressure Sensor Driver
DTMP-driver.h	ROBOTC DI Temp Probe Driver
TIR-driver.h	Dexter Industries Thermal Infrared Sensor driver

5.2 HiTechnic

Name	Description
HTAC-driver.h	Acceleration Sensor driver
HTANG-driver.h	Angle Sensor driver
HTBM-driver.h	Barometric Sensor driver
HTCS-driver.h	Color Sensor driver
HTCS2-driver.h	Color Sensor V2 driver
HTEOPD-driver.h	EOPD Sensor driver
HTGYRO-driver.h	Gyroscopic Sensor driver
HTIRL-driver.h	IR Link Sensor driver
HTIRR-driver.h	IR Receiver Sensor driver
HTIRS-driver.h	IR Seeker driver
HTIRS2-driver.h	IR Seeker V2 driver
HTMAG-driver.h	Magnetic Field Sensor driver
HTMC-driver.h	Magnetic Compass Sensor Driver
HTPB-driver.h	Prototype Board driver
HTRCX-driver.h	IR Link RCX Comms Driver
HTSMUX-driver.h	Commonly used SMUX functions used by drivers
HTSPB-driver.h	SuperPro Prototype Board driver
HTTMUX-driver.h	Touch Sensor Multiplexer Sensor driver

5.3 LEGO

Name	Description
LEGOEM-driver.h	RobotC Energy Meter Driver
LEGOLS-driver.h	Light Sensor driver
LEGOSND-driver.h	SMUX driver for the Lego Sound sensor
LEGOTMP-driver.h	New Temperature Sensor Driver
LEGOTS-driver.h	Touch Sensor driver
LEGOUS-driver.h	SMUX driver for the Lego US sensor

5.4 Mindsensors

Name	Description
MSAC-driver.h	ACCEL-nx driver
MSDIST-driver.h	DIST-Nx driver
MSHID-driver.h	HID Sensor driver
MSLL-driver.h	Line Tracking Sensor
MSMMUX-driver.h	Motor MUX driver
MSMTRMX-driver.h	RCX Motor MUX Driver
MSNP-driver.h	Numeric Keypad Sensor driver
MSPFM-driver.h	PFMate Sensor driver
MSPM-driver.h	Power Meter Sensor
MSPPS-driver.h	PPS-v3 driver
MSRXMUX-driver.h	MSRXMUX RCX Sensor MUX Sensor driver
MSSUMO-driver.h	Sumo Eyes Sensor driver
MSTMUX-driver.h	Touch Multiplexer Sensor driver
MSTP-driver.h	TouchPanel
NXTCAM-driver.h	NXTCam driver
NXTServo-driver.h	NXTServo Sensor Driver

5.5 Others

Name	Description
common.h	Commonly used functions used by drivers
CTRFID-driver.h	Codatex RFID driver
EEPROM-driver.h	EEPROM Driver
FLAC-driver.h	Firgelli Linear Actuator driver
HDMMUX-driver.h	Holit Data Systems Motor MUX driver
MAX127-driver.h	MAXIM MAX127 ADC driver
MCP23008-driver.h	Microchip MCP23008 driver
MICC-driver.h	MicroInfinity CruizCore XG1300L driver
PCF8574-driver.h	Philips PCF8574 IO MUX driver
STATS-driver.h	Statistics functions
TMR-driver.h	Additional _timers