NXT I2C Device Driver Design

The NXT supports two kinds of sensor interfaces.

1. "Dumb" analog sensors provides a voltage level that the NXT converts to a digital value. For example, the NXT sound and the NXT light sensors. There is only a single value associated with an analog sensor.
2. "Smart" digital sensors that communicate with the NXT via the industry standard I2C protocol. The sensor can have multiple values – for example, a three-axis gyroscope has three different sensor values. The NXT Ultrasonic sensor uses the I2C protocol. The sensors are call "smart" because they typically contained an embedded microcontroller (CPU) chip.

This document describes how to write device drivers for the I2C sensors.

Typical Digital Sensors

I2C is a multi-byte master-slave messaging protocol. The NXT is always the master and the sensor is the slave. The master device always initiates the messaging. It sends two types of messages.

Sensor Type	Description
`Ultrasonic`	Measures the distance to an object by bouncing an ultrasonic pulse off the object and measuring the elapsed time for the echo to be received. Basic mode returns the closest object found. An expanded mode returns the distance to up to five objects.
`Gyroscope`	Typically contains one to three axis of measurement for a small gyroscope. Each axis may have one or two bytes of precision.
`Accelerometer`	Typically contains one to three axis of measurement for a small accelerometer. Each axis may have one or two bytes of precision.
`Gyroscope + Accelerometer`	Combined gyroscope and accelerometer.
`Motor Multiplexer`	Provides control for additional motors. Typically four motors are supported.
`RCX IR Communication`	Provides interface to an infrared communications link. Used to talk to the RCX and selected other LEGO IR devices.
`Video Camera`	Provides access to smart video camera sensors. This is a smart camera where, rather than returning a video image, it returns the location in the camera image of a shape or color. I2C messages are sent to the camera to tell it the shape to look for.

Hitechnic (www.hitechnic.com) and Mindsensors (www.mindsensors.com) are two suppliers of 3rd party sensors for the NXT.

I2C Protocol Overview

I2C is a multi-byte master-slave messaging protocol. The NXT is always the master and the sensor is the slave. The master device always initiates the messaging. It sends two types of messages.

- A "write" message is used to send one or more bytes of data to the sensor.

- A “read” message is used to read one or more bytes from the sensor.

Both message types begin with a header consisting of the sensor address and a “register index” within the sensor. For most NXT devices, the register address is 2. The register index is the internal location (address) within the sensor that you want to read/write.

Each slave I2C device is assigned an address because the I2C protocol supports multiple slaves connected to a single master. This architecture is possible with some 3rd party NXT devices but is beyond the scope of this document.

The NXT I2C implementation operates at 9600 bits / second. Or about one byte of data transferred per millisecond. Read messages are N + 3 bytes in length where ‘N’ is the number of bytes to read and ‘3’ is two bytes of protocol overhead. Write messages are N + 2 bytes in length. To read the data from a 3-axis accelerometer takes 9 bytes – 3 bytes of overhead and two bytes of data for each axis. The time to perform I2C I/O can become significant!

Fortunately, ROBOTC has a “fast” transmission mode which operates about five times faster than the 9600 baud found in the standard NXT-G firmware. At the time of writing, all 3rd party sensors have been found compatible with this faster mode and its use is strongly encouraged. Only the LEGO developed Ultrasonic sensor needs to operate at the slower 9600 baud. Third party sensors generally contain a microcontroller that has integrated hardware support for I2C whereas the Ultrasonic sensor uses a firmware implementation for the I2C protocol.

NOTE: the I2C standard specifies that I2C device should support clock rates up to 400 KHz. This is a little faster than the clock used for NXT I2C sensors!

ROBOTC I2C Low-Level Interface

ROBOTC provides several built-in functions for easily sending read or write messages to an I2C sensor.

Function	Description
`sendI2CMsg(nPort, sendMsg, nReplySize);`	Send an I2C message on the specified sensor port.
`nI2CBytesReady[]`	This array contains the number of bytes available from a I2C read on the specified sensor port.
`readI2CReply(nPort, replyBytes, nBytesToRead);`	Retrieve the reply bytes from an I2C message.
`nI2CStatus[]`	Currents status of the selected sensor I2C link.
`nI2CRetries`	This variable allows changing the number of message retries. The default action tries to send every I2C message three times before giving up and reporting an error. Unfortunately, this many retries can easily mask any faults that can exist.
`SensorType[]`	This array is used to configure a sensor for I2C operation. It also indicates whether ‘standard’ or ‘fast’ transmission should be used with this sensor.

There are several well-commented sample programs in the ROBOTC distribution that illustrate the use of these functions.

"Inline" Device Driver Design

The simplest approach to implementing a "device driver" for an I2C sensor is to use "inline" code that reads the value of the sensor whenever you want to retrieve the current value. Generally you call a function to perform the I2C read. The function needs to do the following steps:

1. Wait for any previous I2C sensor activity to complete. I2C messages must be performed one at a time!
2. Initiate an I2C message to read the sensor value.
3. Wait for the I2C read to complete.
4. Extract the sensor value from the reply received from the sensor.

Several of the NXT programming environments – NXT-G, NXC – use this method for handling the Ultrasonic sensor. A significant drawback of this method is that it takes about four milliseconds to read the current sensor value; this is time when you program execution is suspended waiting on the I2C transaction. This can be a "long" time for a real time system!

An alternative approach is to continuously read the sensor value as a background activity. When your program wants the value of the sensor it simply uses the value that was last read. This preferred implementation is discussed later in this document.

The above four step description presented a simplified version of the activities that needed to be performed. The additional complexity that needs to be dealt with includes:

1. The application should initialize the `SensorType` to an I2C type.
2. There may be some one-time initialization messages that need to be sent to the sensor. For the Ultrasonic sensor this includes:
    - Reset the sensor.
    - Configure the measurement scale to be 'cm' or 'inch'.
3. Some devices require a setup delay while they perform initialization. You should not send new messages to the device during this time
4. Recover from error conditions from the I2C messaging to the sensor. The cable may not be connected or the sensor may use an external power source that is not available.
5. Some sensors require a "guard" time between I2C transactions. The sensor may give inaccurate results if values are polled too fast. On the Ultrasonic sensor, 25 milliseconds should elapse between read requests.
6. Some sensors only update their results on a periodic basis. For example, a GPS sensor may update values only once per second. It doesn't make sense to poll the sensor on a faster basis than its update rate!
7. For devices like a four-port motor multiplexer, an application program may typically update the values of all four motors on a frequent periodic basis. You usually won't want to send a single I2C message every time a motor setting is updated; at four milliseconds per message, 12 milliseconds may elapse between the time the first and last motors are update. This is

enough time that a pair of motors may not travel straight! Instead, you'd usually want to update the four motors using a single I2C message.

Depending on the device's characteristics and "quirkiness", the implementation of a driver for a digital sensor can become complex if it needs to deal with all of the considerations described above. Fortunately, you only need to develop the driver once and then this can become common code used in many applications. Most developers of 3rd party sensors also develop the device driver code that they provide to purchasers of their sensors.

"Background Task" Device Driver Design

As mentioned above, generally the preferred implementation of an I2C sensor is a "background" activity that continuously polls the sensor. When the user's program – i.e. the "foreground" activity – is running it simply reads or writes variables that manage the sensor and the background task performs the appropriate actions.

ROBOTC is a multi-tasking solution. This means that you can have several "`tasks`" (or "threads") concurrently executing. The ROBOTC firmware will share the CPU time among all executable tasks. The primary, or '`main`' task, is the user application program. It's very easy to implement a program with multiple tasks.

1. The application program is '`task main`'.
2. Use the '`task`' keyword to implement a second task for the device driver. For example '`task sensorDriver`' or '`task SensorDriver()`'. The "`()`" are optional. A task has the same syntax as a function or subroutine definitino with the addition of the "`task`" keyword.
3. Only execution of the "main" task is started when a program is run. Add a '`StartTask(sensorDriver);`' statement at the beginning of your "main" task to cause the device driver task to begin execution.
4. There's no need to add an explicit '`StopTask(sensorDriver)`' when your "main" task is finished. The ROBOTC firmware will automatically abort all secondary tasks when the '`main`' task terminates.

A multi-tasking solution shares the available CPU time among all executable tasks. If you had two tasks, and they both had the same execution priority, then they would each get 50% of the available time. However, there is really no need for the background task to get 50% of the CPU time as it spends most of its time waiting! It's either waiting for I2C transaction to complete or waiting for delay between 'polling' cycles. ROBOTC tasks that are explicitly waiting (i.e. via a 'wait1Msec(<time to wait>)' function call) do not consume CPU cycles. A "bad" design to wait for I2C transaction to complete is:

```
while (nI2CStatus[nDDPortIndex] != NO_ERR)
{}
```

A "good" design to wait for I2C transaction uses the explicit wait statement

```
while (nI2CStatus[nDDPortIndex] != NO_ERR)
{
  Wait1Msec(2); // Relinquish CPU so other tasks can execute
}
```

NOTE: ROBOTC on the NXT will typically execute 200 or more lines of code in a single millisecond. The I2C status is only updated by the firmware once per millisecond, so as soon as you detect that I2C is not ready, you might as well go ahead and wait.

You want to write your device driver as common code that can be reused in many application programs. You could use 'cut and paste' to copy the code into each application program; but this is not a good technique. You want a solution where there is only one copy of the driver and, if it is updated, all application programs can utilize the latest version. The ROBOTC '`#include`' preprocessor directive provides this implement. The '`#include`' statement contains the name of a file that should be included in your application program. You implement this as follows:

1. Write the device driver task as a separate file. Assume the name is 'sensorDriver.c' and that it is located in the same file directory as your application program.
2. Add the statement '`#include "sensorDriver.c"`' to your main application program. When your program is compiled, the source code from the 'include' file will be added at this point.
3. Once you have the driver fully tested, you may want to add it to the directory 'C:\Program Files\Robotics Academy\Includes\'. When searching for an include file, the compiler will first look in the same folder as the file being compiled; it will then check the directory mentioned here which is used as a repository for common include files. [NOTE: Advanced users can redirect the location of the common files using the 'Preferences' options on the ROBOTC menu].

Sample Device Driver Design Files – Ultrasonic Sensor

The Ultrasonic sensor is such a popular device that the driver for it is built into the ROBOTC firmware. The built-in driver performs just like the 'background' task approach discussed above. The program in the ROBOTC samples folder, "NXT Ultrasonic User Device Driver.c" provides a similar implementation as a standalone ROBOTC program. The program is heavily commented and provides a good introduction for the beginner on how to write a simple I2C device driver.

There's a subtle programming "trick" in the above implementation. For the built-in sensor drivers – i.e. analog sensors and the Ultrasonic sensor – the array variable 'SensorValue' contains the value of a sensor. This is a read-write variable. So the driver task writes to this variable so that an application program can refer to the sensor's value using this familiar, and common, variable.

Another subtlety to notice is that the driver only works on port S4. It assumes that this port is always connected to a Ultrasonic sensor.

Sample Device Driver Design Files – Compass Sensor

A compass sensor is a popular third party I2C sensor. There's only a single value from this sensor and the hardware resolution of the sensor is around a single degree. So the 3rd party designers of commercial sensors implemented a compatibility mode where the sensor behaves just like a ultrasonic sensor. You simply treat the compass like a ultrasonic sensor! The ultrasonic sensor only returns a one-byte value (i.e. 0 to 255) and two approaches have been taken by manufacturers to represent the 0 to 359 degree scale.

- One returns a value in the range 0 to 179. You multiply by two to convert to a degree.
- One returns a value in the range 0 to 255. You convert to degrees with the formula <value> * 360 / 256.

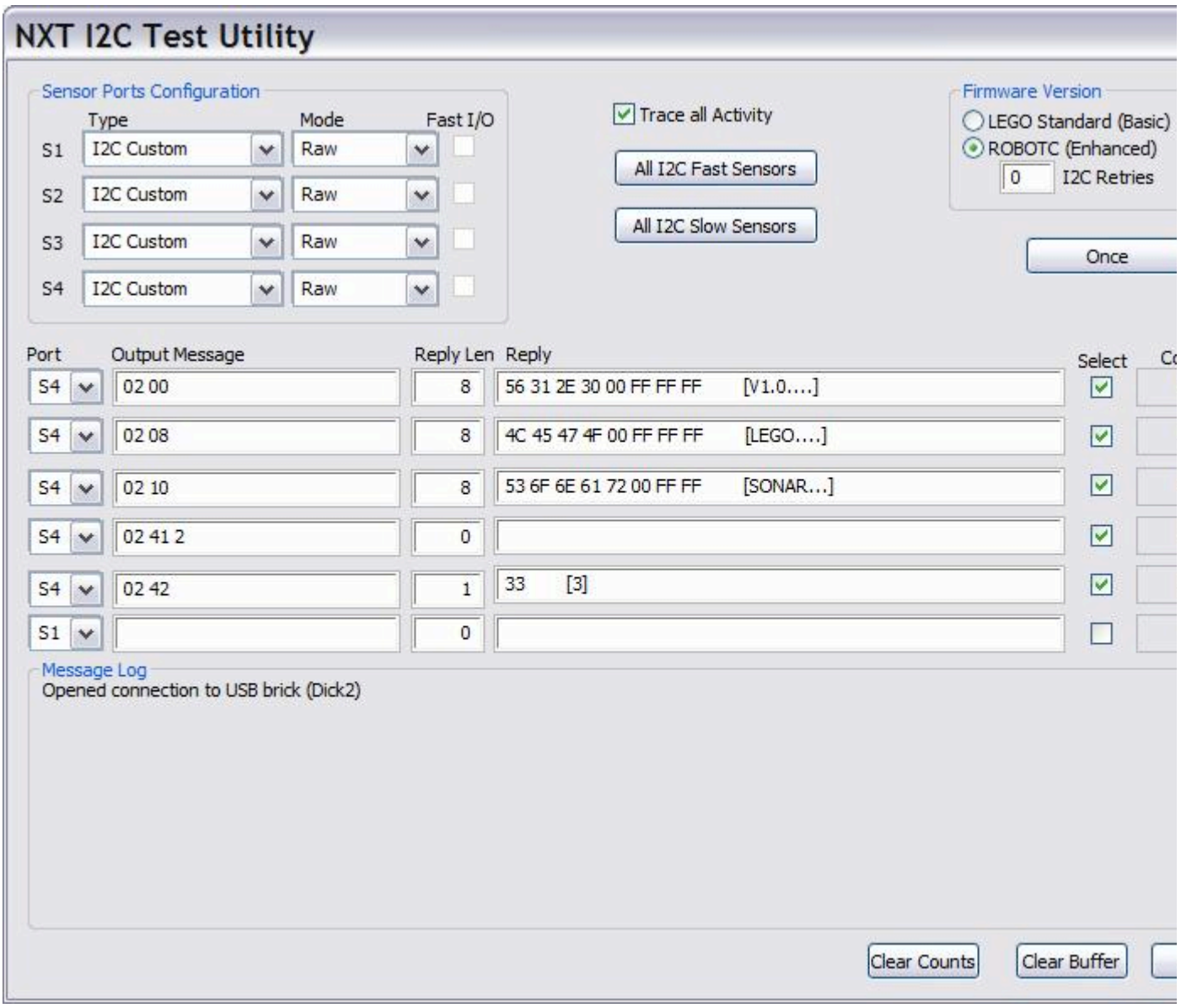
Both compasses offer an enhanced mode where you can read the full degree range with a two-byte I2C read. The files “Compass Sensor.c” and “Compass Sensor Driver.c” in the ROBOTC sample programs illustrate this. This sample is more sophisticated than the previous ultrasonic sensor example. It:

- Works on one or more of the four sensor ports. The driver has a new outer loop that checks each of the four sensors to see whether they are configured as a compass.
- Does a two-byte read to obtain the sensor value.

ROBOTC I2C Test Utility

Before beginning to write the device driver for a sensor you should confirm that the sensor hardware is functioning as expected. This is especially true if the hardware is being designed coincident with the driver. ROBOTC has a unique, and excellent, built-in utility for testing I2C sensors. It is described in the file NXT I2C Test Utility.htm.

A screenshot of the utility is shown in the following picture. With this utility, you can define up to six I2C transactions that can be performed on a single or continuous basis.


NXT I2C Test Utility
Sensor Ports Configuration
Type
Mode
Fast I/O
S1 I2C Custom Raw
S2 I2C Custom Raw
S3 I2C Custom Raw
S4 I2C Custom Raw
Trace all Activity
All I2C Fast Sensors
All I2C Slow Sensors
Firmware Version
LEGO Standard (Basic)
ROBOTC (Enhanced)
0 I2C Retries
Once
Port
Output Message
Reply Len
Reply
Select
S4 02 00 8 56 31 2E 30 00 FF FF FF [V1.0....]
S4 02 08 8 4C 45 47 4F 00 FF FF FF [LEGO....]
S4 02 10 8 53 6F 6E 61 72 00 FF FF [SONAR...]
S4 02 41 2 0
S4 02 42 1 33 [3]
S1 0
Message Log
Opened connection to USB brick (Dick2)
Clear Counts
Clear Buffer