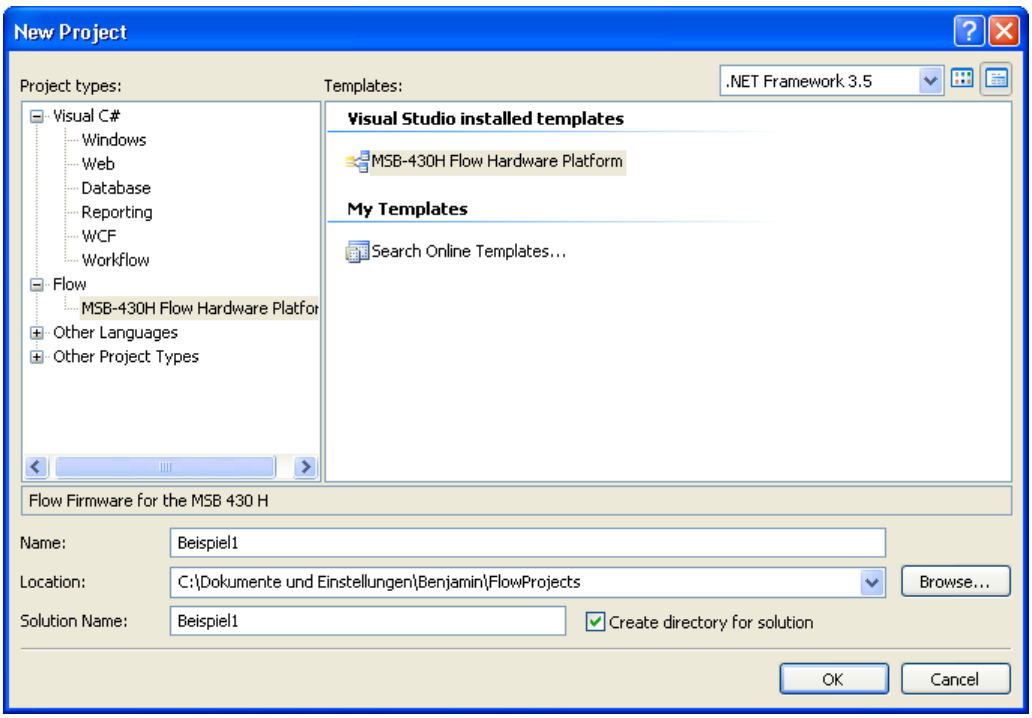
Flow Anwenderhandbuch

Im Folgenden soll beschrieben werden, wie *Flow* innerhalb von Visual Studio eingesetzt wird, um Sensorknotenanwendungen zu modellieren. Primär soll auf die Bedienung und Elemente der Benutzeroberfläche eingegangen werden und nicht auf die Modellierung in den verschiedenen Sprachen. Dazu sei auf Kapitel 5 der Diplomarbeit[1], in deren Rahmen *Flow* entstanden ist, verwiesen.

Um die Beispiele der nächsten Seiten ausführen zu können, wird ein lauffähiges System zur Sensorknotenentwicklung mit Visual Studio 2008 und *Flow* benötigt. Des Weiteren sollten *(für die Funkkommunikation)* mehrere Sensorknoten vom Typ MSB-430H und das Erweiterungsboard MSB-430S zur Verfügung stehen.

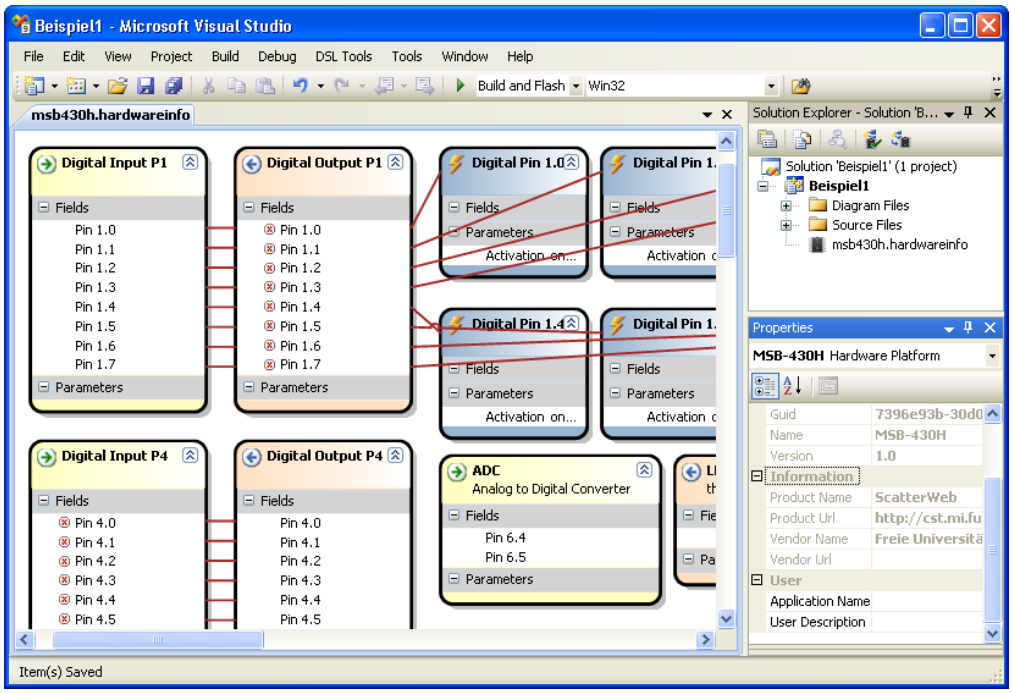
Beispielanwendung I

Nachdem die MSB-430H-Plattform auf dem Entwicklungsrechner installiert wurde, steht in Visual Studio ein neuer Projekttyp bereit, der über den New Project Dialog erzeugt werden kann. Für das erste Beispielprogramm wird ein solches Projekt mit dem Namen „Beispiel1“ angelegt.

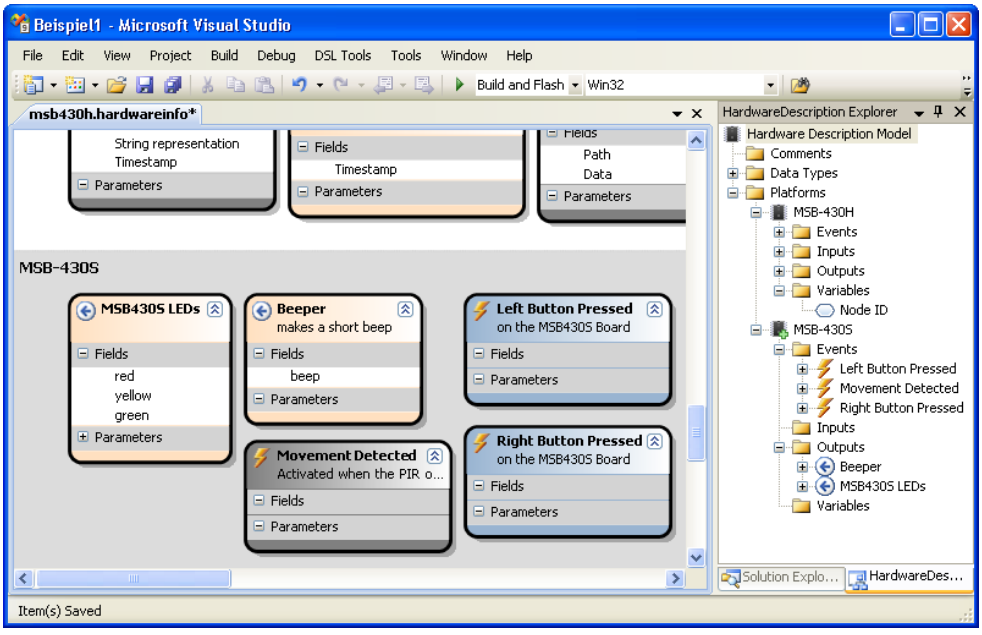


[1] Benjamin Schröter: *Software Factories for Embedded Systems* (2008)

Das neue Projekt enthält bereits die Hardwaredescription der Sensorknotenhardware.



Da für diese Beispielanwendung die Erweiterung MSB-430S verwendet werden soll, muss der entsprechende Treiber geladen werden. Dazu bietet dieses Modell im Kontextmenü[2] den Befehl „Import Driver...“ an. Nachdem der Befehl angeklickt wurde, muss die Datei `MSB-430S Driver.flowdrv` ausgewählt werden. Das Modell wird dadurch um die Elemente des Treibers erweitert. Da dabei auch Kanten angelegt werden, die das Modell unübersichtlich machen können, empfiehlt es sich, ebenfalls über das Kontextmenü mit dem Befehl „Show Connectors“, die Kanten auszublenden.

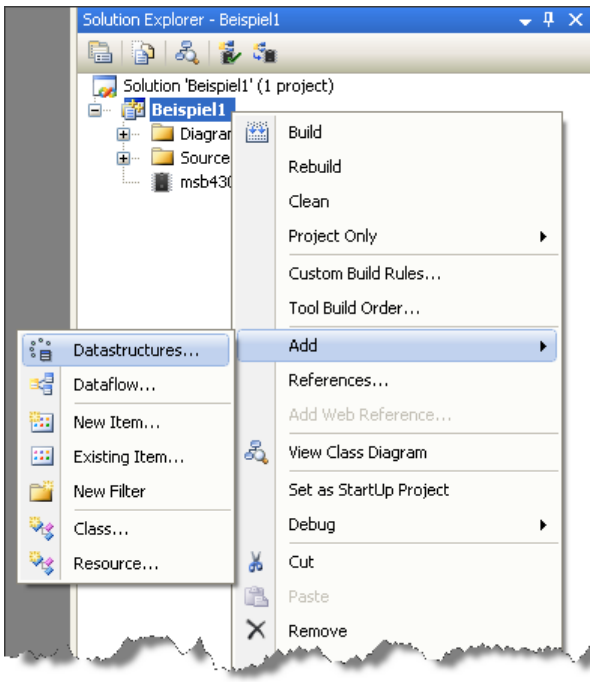


Alle Elemente der Hardwaredescription sind auch im „Hardwaredescription Explorer“ auf der rechten Seite zu sehen, der ebenfalls über das Kontextmenü des Modells geöffnet werden kann.

Die Sensorknotenanwendung soll beim Drücken eines Tasters die grüne LED umschalten. Wird der andere Taster gedrückt und die grüne LED leuchtet, soll der Beeper kurz aktiviert werden. Um zu signalisieren, dass der Sensorknoten aktiv ist, soll währenddessen eine weitere LED ständig blinken. Da für diese Anforderungen viele Elemente der Hardwaredescription nicht benötigt werden, werden sie bereits in diesem Modell über das Visual Studio Properties Window deaktiviert *(enabled = false)*. Dies sind:

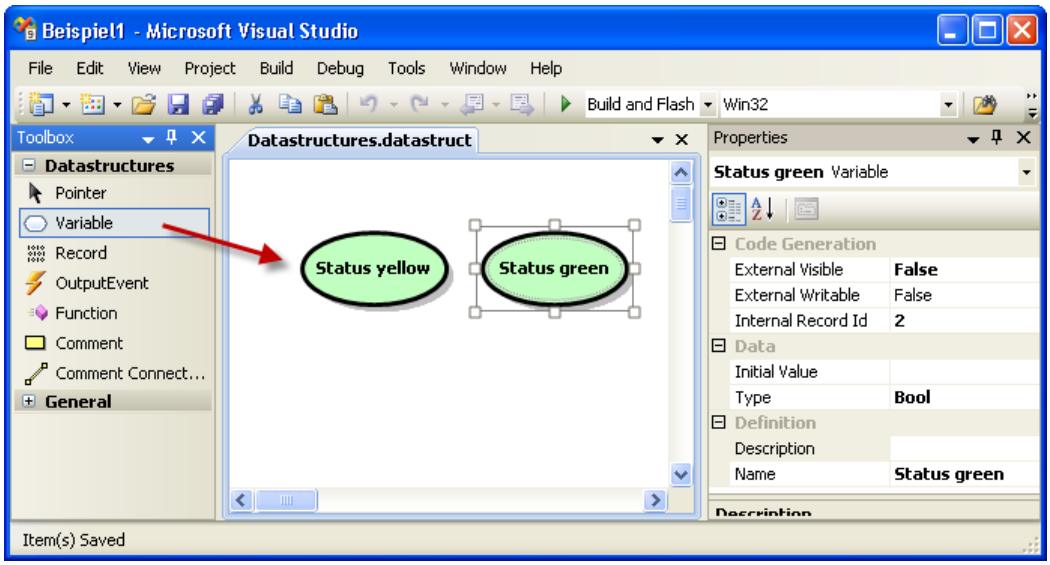
- alle digitalen I/O-Ports sowie die Events
- der analoge ADC-Port
- die Elemente der seriellen Schnittstelle und die Real Time Clock
- die „Write File“-Output-Ports
- der Bewegungsmelder des MSB-430S

Da für die beiden LEDs im weiteren Verlauf der aktuelle Zustand benötigt wird, muss dieser in Variablen gespeichert werden. Dazu wird im Projekt ein neues Datastructure-Modell erstellt.

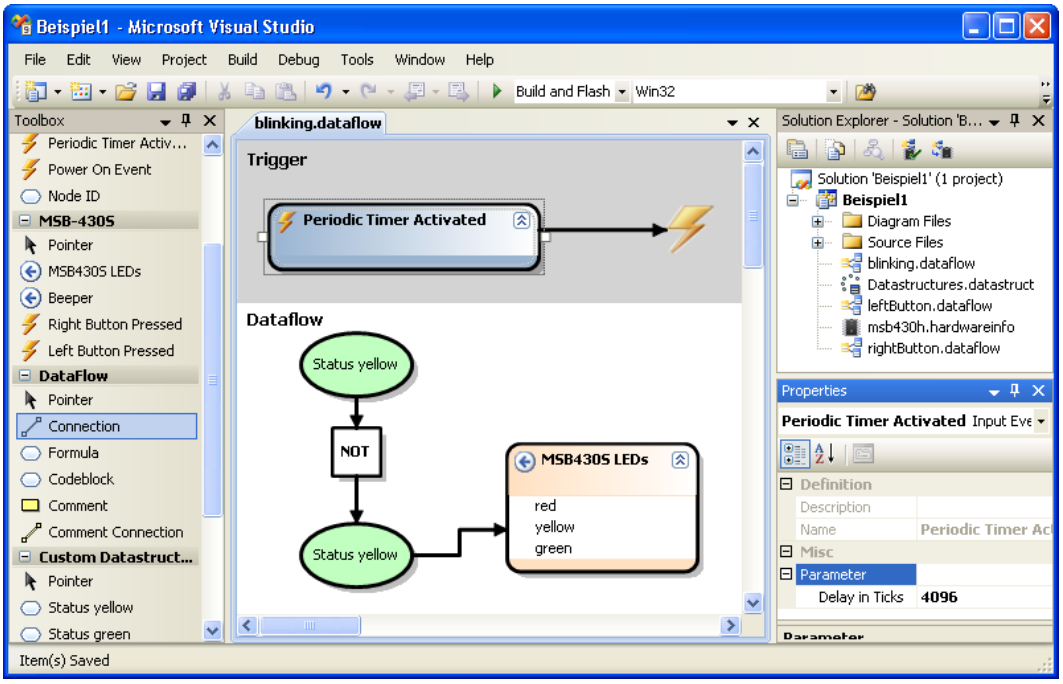


[2] Das Kontextmenü kann durch Rechtsklick auf der Zeichenfläche *(dem Hintergrund)* des Modells aufgerufen werden.

Diesem Modell können aus der Toolbox auf der linken Seite Elemente per drag-and-drop hinzugefügt werden. Für die Beispielanwendung werden zwei Variablen benötigt. Mittels der Eigenschaften im Properties Window können diese Variablen konfiguriert werden. Ihnen wird der angezeigte Name gegeben und der Datentyp „Bool“ ausgewählt.



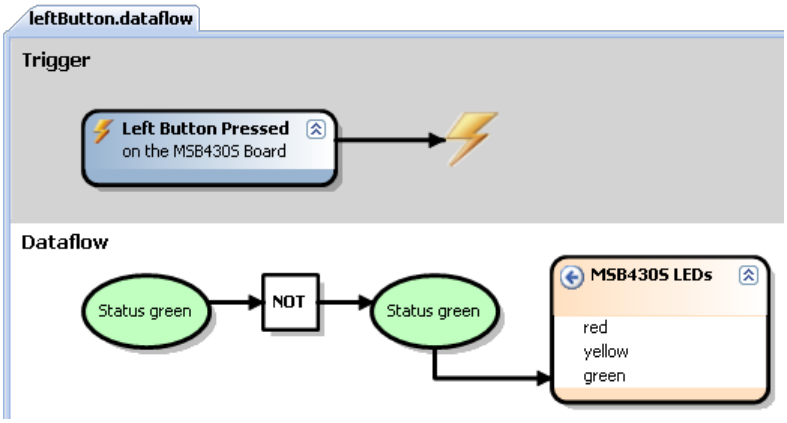
Nachdem nun die Hardware und die Datastructures modelliert und konfiguriert sind, kann begonnen werden, das Verhalten der Anwendung zu definieren. Dazu wird *(so wie bei den Datastructures)* ein neues Dataflow-Modell mit dem Namen „blinking“ angelegt. Dieser erste Dataflow soll dafür sorgen, dass die gelbe LED blinkt. Dazu werden die Elemente aus der Toolbox wie folgt auf dem Modell positioniert und mittels des „Connection“-Tools verbunden:



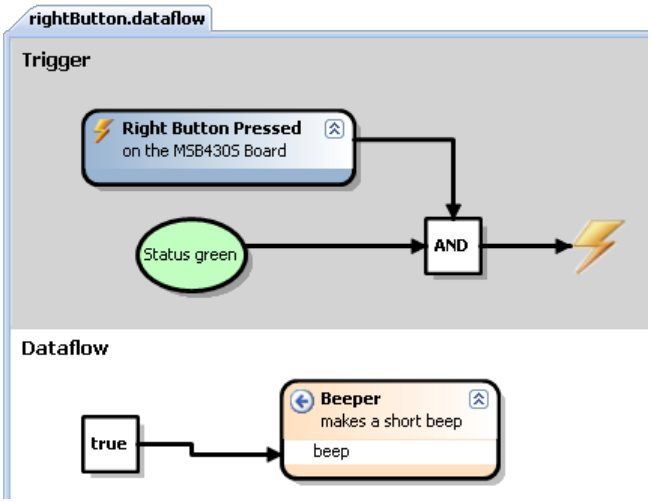
Das „Periodic Timer Activated“-Event im Trigger-Bereich gibt immer, wenn es ausgelöst wird, einen „Impuls“ an den Master-Trigger, so dass der Dataflow ausgeführt wird. Das Event besitzt eine Eigenschaft „Delay in Ticks“, die über das Properties Window auf 4096, was einer halben Sekunde entspricht, eingestellt wird.

Im Dataflow-Bereich wird zuerst die Variable „Status yellow“ gelesen, negiert und dann wieder geschrieben. Zum Schluss wird der Wert der Variable über die gelbe LED des „MSB430S LEDs“-Output-Ports angezeigt, wodurch sie in einem Rhythmus von 0,5 Sekunden blinkt.

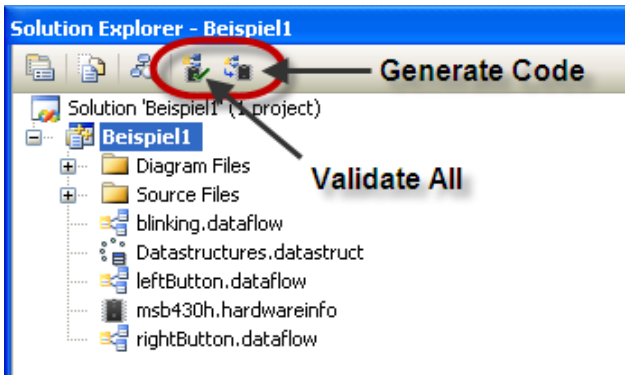
Der zweite Dataflow mit dem Namen „leftButton“ funktioniert analog, wird allerdings von einem anderen Event ausgelöst.



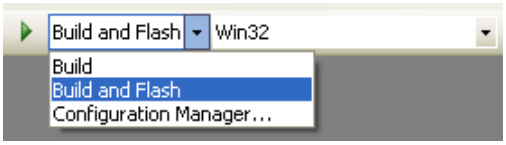
Der dritte und letzte Dataflow „rightButton“ reagiert nun auf den anderen Taster, aber führt den Dataflow nur dann aus, wenn auch die Variable „Status green“ gesetzt ist. Der Dataflow-Bereich selbst ist trivial, da hier lediglich der Beeper aktiviert werden muss.



Nachdem die Anwendung vollständig modelliert ist, kann zunächst geprüft werden, ob die Modelle korrekt gebildet sind und sich fehlerfreier Code generieren lässt. Dies kann durch den linken Button[3] im Solution Explorer durchgeführt werden. Wenn Fehler gefunden werden, werden diese in der Error List von Visual Studio angezeigt. Ist dahingegen die Validierung fehlerfrei verlaufen, so kann mit dem rechten Button der Quellcode für die Sensorknoten generiert werden[4].



Zu guter Letzt kann der Code kompiliert und auf die Sensorknoten geflasht werden. Dazu stehen zwei Konfigurationen bereit, die entweder nur den Code kompilieren *(„Build“)* oder ihn im Anschluss über ein USB-JTAG-Interface auf den Sensorknoten aufbringen *(„Build and Flash“)*. Diese Konfigurationen können über die Symbolleiste von Visual Studio ausgewählt werden. Während des Kompilierens[5] wird die Ausgabe des Compilers im Visual Studio Output Window angezeigt. Sollten Fehler auftreten, so sind diese im Output Window und der Error List zu sehen.

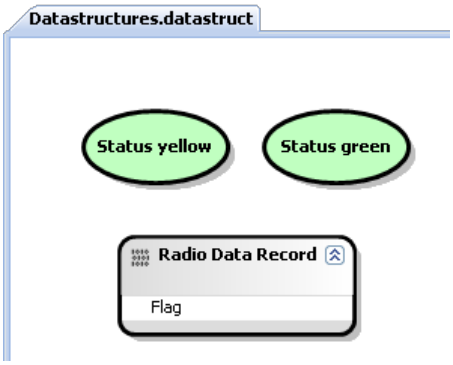


[3] Die *Flow*-Buttons im Solution Explorer sind nur sichtbar, wenn ein *Flow*-Projekt ausgewählt ist.
[4] Jedesmal, wenn die Modelle verändert wurden, muss der Quellcode erneut generiert werden, da ansonsten veralteter Code auf die Sensorknoten programmiert würde.
[5] Das Kompilieren kann entweder über das Menü oder über die Tastenkombination `Shift+Strg+B` gestartet werden.

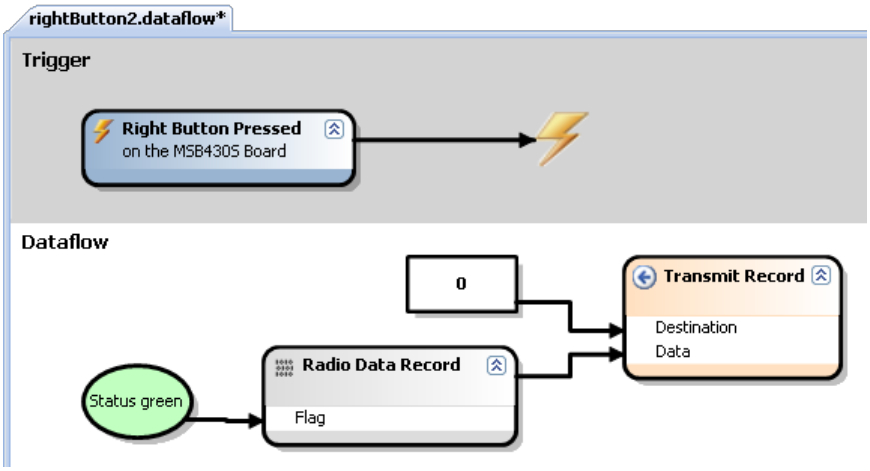
Beispielanwendung II

Das zweite Beispiel erweitert die Anwendung von oben um Funkkommunikation. Immer wenn der rechte Taster gedrückt wird, soll per Broadcast eine Nachricht verschickt werden. Diese Nachricht enthält als Daten den Zustand des grünen LEDs des Senders. Wenn ein anderer Sensorknoten eine solche Nachricht empfängt, prüft dieser die übertragenen Daten und falls das Flag gesetzt ist, gibt auch er einen Pfeifton aus.

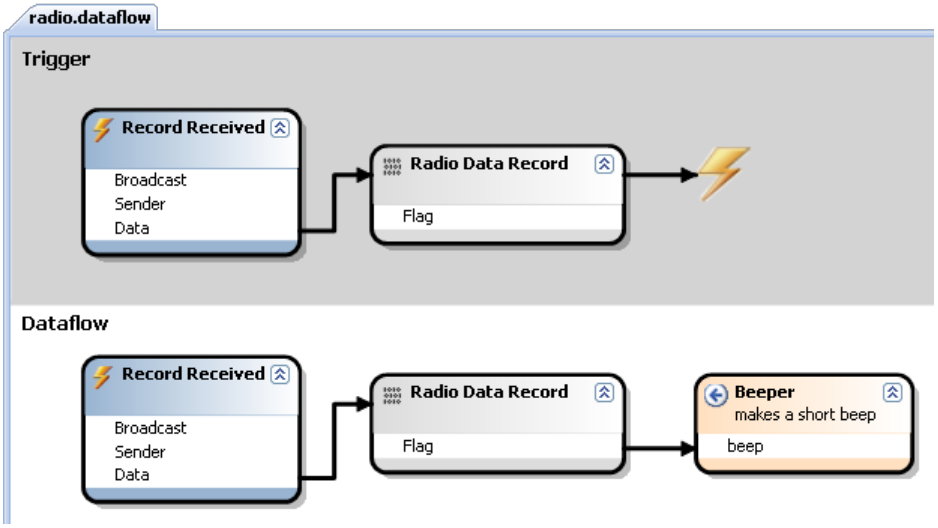
Für die Funkübertragung wird ein Record mit nur einem Feld benötigt. Dies kann im bereits vorhandenen Datastructure-Modell durch drag-and-drop aus der Toolbar hinzugefügt werden. Anschließend lässt sich dem Record über sein Kontextmenü ein Feld hinzufügen. Im Properties Window kann der Name und der Datentyp *(„Bool“)* des Feldes eingestellt werden.



Ein neuer Dataflow „rightButton2“ soll, wenn der Taster betätigt wird, ein Record erstellen und versenden. Ein zweiter Dataflow ist notwendig, da der Dataflow „right-Button“ aus dem Beispiel oben nur dann ausgeführt wird, wenn die Variable „Status green“ gesetzt ist, aber nun bei jedem Tastendruck ein Paket verschickt werden soll. Im Dataflow wird die Variable „Status green“ in ein Record von Typ „Radio Data Record“ verpackt und mittels des „Transmit Record“-Output-Ports an die Broadcastadresse 0 verschickt. Um die Adresse angeben zu können wird ein Formel-Shape verwendet, in dem die Zahl eingetragen ist.



Auf diese Funkpakete soll nun in einem weiteren Dataflow reagiert werden. Wenn ein Funkpaket empfangen wird prüft der Tigger-Bereich dieses Modells, ob das Event ein Record vom Typ „Radio Data Record“ geliefert hat. Wenn dem so ist, wird ein „Impuls“ zum Master-Trigger gegeben und der Dataflow ausgeführt. Im Dataflow wird ein Record mit den Daten aus dem Event gefüllt und das Flag an den Beeper weitergegeben. Der Beeper wird nur dann einen Pfeifton ausgeben, wenn das Flag im Record `TRUE` ist.



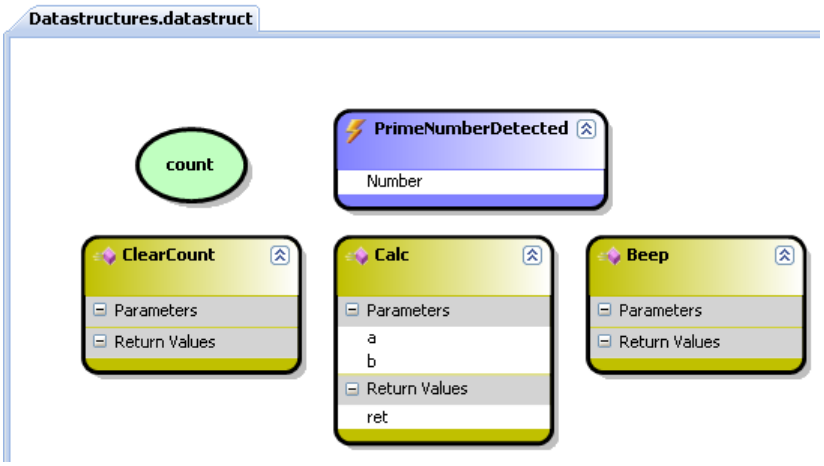
Wie oben beschrieben, kann nun auch aus dieser Anwendung Code generiert und auf mehrere Sensorknoten verteilt werden. Da in diesem Beispiel alle Knoten gleichberechtigt sind und die Kommunikation mittels Broadcasts durchgeführt wird, sollten die Knoten problemlos zusammenarbeiten.

Beispielanwendung III

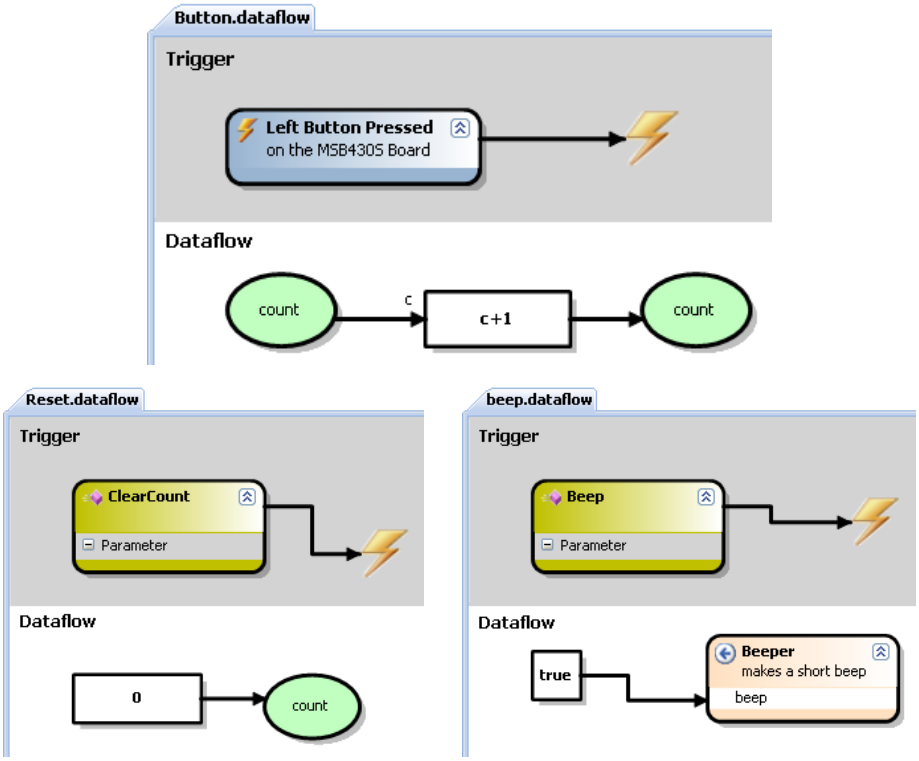
Das folgende Beispiel zeigt, wie *Flow*-Sensorknoten unter Verwendung des ScatterWeb .NET SDKs mit einem PC zusammenarbeiten können. Dazu wird eine Sensorknotenanwendung mit *Flow* modelliert, aber auch der Code der PC-Anwendung vorgestellt.

Die Sensorknoten sollen einen Zustand in Form einer numerischen Variable besitzen, der sich durch Ereignisse *(Tastendrücke)* verändert. Der PC soll auf diese Variable lesend zugreifen und sie mittels eines Funktionsaufrufs auf Null setzen können. Um weitere Möglichkeiten der remote Funktionsaufrufe zu demonstrieren, soll eine Berechnung mit zwei Argumenten auf den Sensorknoten durchgeführt werden, so dass der PC die Berechnung anstoßen kann und das Ergebnis geliefert bekommt. Außerdem soll der PC den Beeper des Sensorknotens aktivieren können. Normalerweise werden Sensornetzwerke eingesetzt, weil die Sensorknoten eine gewisse Intelligenz besitzen und so nicht permanent von einer Basisstation ferngesteuert werden müssen. Um auch dieses Einsatzgebiet zu demonstrieren, soll der Sensorknoten immer dann ein Output Event an den PC auslösen, wenn die numerische Variable ihren Wert ändert und eine Primzahl erreicht.

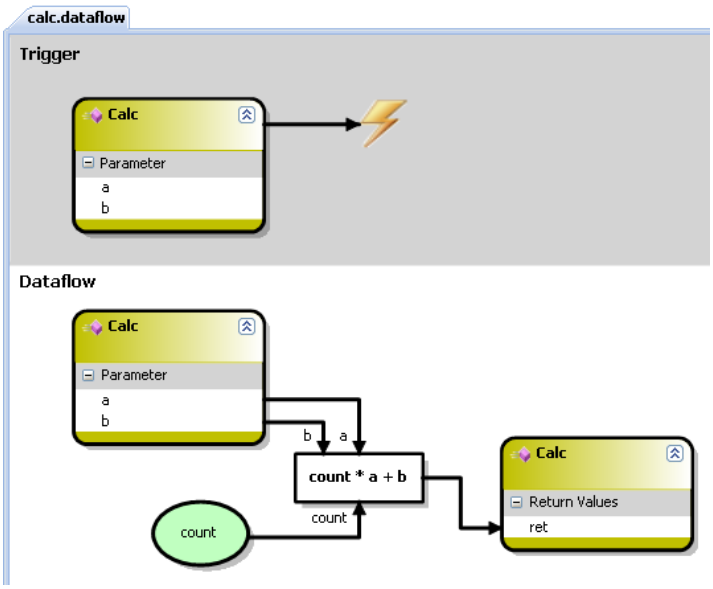
Wie bei den vorhergehenden Beispielanwendungen wird ein neues Sensorknotenprojekt angelegt. Auch bei diesem Beispiel wird der Treiber für die Erweiterung MSB-430S in die Hardwaredescription geladen und ein Datastructure-Modell angelegt. In diesem Modell ist die numerische Variable *count*, die drei beschriebenen Funktionen, die von einem PC aufgerufen werden können, sowie das Output Event zu sehen.



Die ersten drei Dataflow-Modelle sind den der anderen Beispiele sehr ähnlich. Wird der Taster betätigt, so wird die Variable um eins erhöht und beim Funktionsaufruf `ClearCount()` vom PC wird sie auf Null gesetzt. Der Funktionsaufruf `Beep()` aktiviert den Beeper.

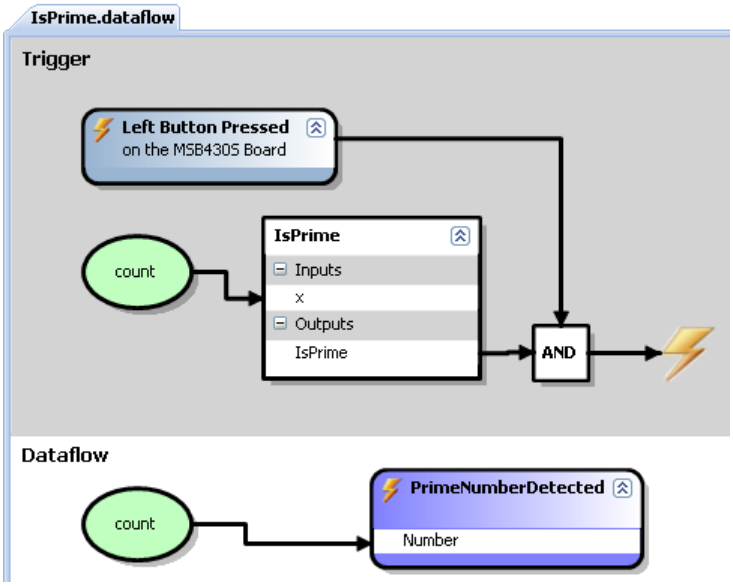


Der Funktionsaufruf `Calc()` vom PC verwendet die mit dem Funktionsaufruf gesendeten Argumente *a* und *b* sowie den Wert der Variable *count*, um mittels eines Formel-Shapes das Ergebnis zu berechnen. Dieses wird als Rückgabewert der Funktion an den PC geschickt.



Der letzte Dataflow wird immer dann ausgeführt, wenn der Taster gedrückt wird und die Variable *count* durch die Veränderung im oben beschriebenen Dataflow nun eine Primzahl darstellt. Da diese Prüfung nicht ohne weiteres in einem Dataflow darzustellen ist, wird hier ein Codeblock im Trigger-Bereich verwendet. Ein Codeblock kann aus der Toolbox dem Diagramm hinzugefügt und dort mit Ein- und Ausgängen bestückt werden. Durch Doppelklick auf den Codeblock wird ein C-Code-Editor geöffnet, in dem der Code aus Abbildung 1 eingetragen werden muss. Dieser Code prüft, ob der Parameter `x` eine Primzahl ist und setzt den Parameter `IsPrime` als Rückgabe entsprechend.

Wenn der Codeblock den Dataflow mittels des Master-Triggers aktiviert, wird das Output Event `PrimeNumberDetected` ausgelöst, auf das ein PC reagieren kann.



Diese wenigen Dataflows implementieren das öffentliche Interface des Sensorknotens, der so auf Anfragen eines PCs reagieren bzw. Events an den PC schicken kann. Die Anwendung kann, wie bereits beschrieben, kompiliert und auf Sensorknoten programmiert werden.

```
/* This file contains user code for the
   Codeblock 'IsPrime'
   from the Model 'IsPrime' */

// the following line must match exactly the definition in the model
// please do not change this line:
void IsPrime_IsPrime(uint32_t x, bool* IsPrime)
{
    if (x < 2) {
        *IsPrime = false;
        return;
    }
    else if (x == 2) {
        *IsPrime = true;
        return;
    }
    else if (x % 2 == 0) {
        *IsPrime = false;
        return;
    }
    else {
        int i=3;
        for (; i*i <= x; i+=2) {
            if (x%i == 0) {
                *IsPrime = false;
                return;
            }
        }
        *IsPrime = true;
        return;
    }
}
```

Abbildung 1: Quellcode des Codeblocks „IsPrime“

PC-Anwendung

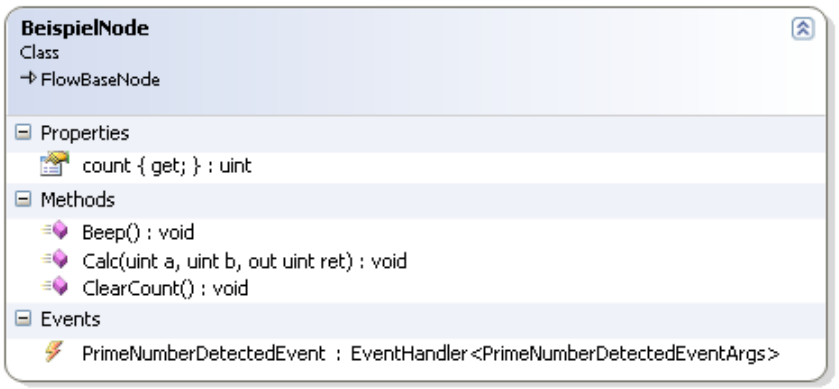
Im nächten Schritt soll eine Windows-Anwendung entstehen, die mit diesen Sensorknoten kommuniziert. Zunächst wird in der Solution ein neues Projekt *(C# Windows Forms Application)* angelegt. Um in diesem Projekt das ScatterWeb .NET SDK zu verwenden, müssen dem Projekt drei Referenzen auf die folgenden Assemblies hinzugefügt werden. Diese Assemblies befinden sich in einem Unterverzeichnis der *Flow*-Installation:

- `ScatterWeb.API.dll`
- `ScatterWeb.SDK.dll`
- `ScatterWeb.SDK.Nodes.Flow.FlowBaseNode.dll`

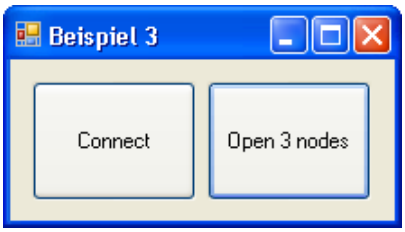
Für den oben beschriebenen Sensorknoten soll ein Proxy generiert werden. Dieser Proxy wird ebenfalls dem Projekt hinzugefügt indem eine Textdatei mit der Endung `.tt` angelegt und mit dem folgenden Code gefüllt wird:

```
<#@ ProjectFile processor="VsProjectFileDirectiveProcessor"
        FileName="..\Beispiel3\Beispiel3.vcproj" #>
<#@ include file="proxy.tt" #>
```

Unter Umständen muss der relative Pfad zum Sensorknotenprojekt angepasst werden. Es ist durchaus möglich, einer Anwendung auf diese Weise mehrere Proxies hinzuzufügen, die den Code für verschiedene Sensorknotenprojekte erzeugen. Durch die Dateiendung `.tt` wird beim Speichern der Datei automatisch der Proxycode erzeugt, der im Solution Explorer unterhalb dieser Datei als C#-Code gefunden werden kann. Die folgende Abbildung zeigt das öffentliche Interface des generierten Proxies, in dem die oben modellierten Elemente wiedergefunden werden können.



Bevor nun mit dieser Klasse gearbeitet werden kann, ist ein wenig Infrastrukturcode notwendig, der mit einem lokal am PC angeschlossenen Sensorknoten, der ein Gateway zum gesamten Sensornetzwerk darstellt, kommunizieren kann. Beim Starten der Anwendung wird das `MainForm` geöffnet, das lediglich zwei Buttons enthält.

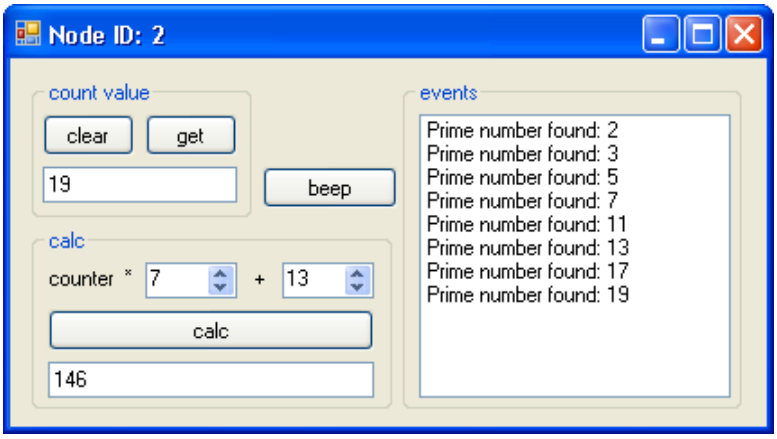


Wenn der Button „Connect“ betätigt wird *(siehe Abbildung 2, Zeile 17)*, wird der Port zur seriellen Kommunikation ermittelt, ein `NetworkManager` instanziert und nach Sensorknoten gesucht. Wenn Knoten gefunden wurden, wird der zweite Button „Open x nodes“ freigegeben. Mit diesem Button *(Zeile 41)* wird für jeden Knoten im Netzwerk ein `NodeForm` erzeugt und angezeigt. Der in Abbildung 2 gezeigte Code kann in ähnlicher Art und Weise auch in anderen Anwendungen verwendet werden, auch wenn hier auf Fehlerbehandlung verzichtet wurde, um die relevanten Stellen hervorzuheben.

```
using System;
using System.Linq;
using System.Windows.Forms;
using ScatterWeb.SDK;
using ScatterWeb.SDK.Nodes;
using ScatterWeb.API.Messaging.Channels;
using Flow.SensorNodes;

namespace WindowsFormsApplication1 {
 public partial class MainForm : Form {
   NetworkManager netman;

   public MainForm() {
     InitializeComponent();
   }

   private void buttonConnect_Click(object sender, EventArgs e) {
     // Use the HardwareFinder to get the COM-Port for communication
     ScatterWeb.API.Platform.HardwareFinder hwf = new HardwareFinder();
     string port = hwf.FindConntectedHardware(System.IO.Ports.SerialPort
         .GetPortNames());
     if (String.IsNullOrEmpty(port)) return;

     // create a Channel and a NetworlManager
     Channel channel = new SerialPortChannel(sport);
     netman = new ScatterWeb.SDK.NetworkManager(channel);

     // announce the Nodetype
     netman.RegisterNodeType<BeispielNode>();

     // Open the network
     netman.Open();

     // enable the second button if any nodes available
     int c = netman.Network.OfType<BeispielNode>().Count();
     if (c > 0) {
       this.buttonOpen.Text = string.Format("Open {0} nodes", c);
       this.buttonOpen.Visible = true;
     }
   }

   private void buttonOpen_Click(object sender, EventArgs e) {
     // get all BeispielNode nodes form the NetworkManager
     foreach (BeispielNode n in netman.Network.OfType<BeispielNode>()) {
       // and open a NodeForm for every node
       NodeForm f = new NodeForm(n);
       f.Show();
     }
   }

   private void MainForm_FormClosing(object s, FormClosingEventArgs e) {
     // close the NetworkManager when the application is closing
     if ( netman != null && netman.IsOpen )
       netman.Close();
   }
 }
}
```

Abbildung 2: Quellcode von `MainForm`

Der `NetworkManager` erzeugt für jeden Sensorknoten abhängig von der installierten Anwendung eine spezialisierte Klasse. Für alle Sensorknoten der oben modellierten Sensorknotenanwendung wird eine Instanz der von Proxy generierten `BeispielNode`-Klasse erzeugt. Diese wird in der Schleife des zweiten Buttons an den Konstruktor des `NodeForms` übergeben *(Zeile 45)*, das in der folgenden Abbildung dargestellt ist:



Auf die Elemente des Forms soll nicht detailliert eingegangen werden, da sie lediglich Funktionen des Sensorknotens wiederspiegeln. Der relevante Code dieses Forms ist in Abbildung 3 gezeigt und besteht meist nur aus wenigen Zeilen, um auf die Benutzer- und Sensorknotenereignisse zu reagieren.

Die Buttons rufen entweder Funktionen in der Sensorknoten-Klasse auf *(`Beep()` in Zeile 33, `CleartCount()` in Zeile 38 und `Calc()` in Zeile 52)* oder fragen die Property *count* ab *(Zeile 43)*. Alle Aufrufe bewirken, dass der entsprechende Dataflow des Sensorknotens ausgeführt wird. Wenn für eine Funktion Rückgabewerte definiert sind blockiert sie solange, bis der Knoten eine Antwort liefert oder ein Timeout auftritt. Bei den Funktionen ohne Rückgabewerte *(`Beep()` und `CleartCount()`)* wird die Ausführung fortgesetzt, sobald das Gateway das Record entgegengenommen hat.

Für das Output Event wird eine EventHandler-Methode registriert, was im Beispiel im Konstruktor des Forms in Zeile 19 geschieht. Innerhalb dieser Methode muss lediglich beachtet werden, dass das Form nicht direkt manipuliert werden darf, da dieser Aufruf aus einem anderen Thread heraus geschieht. Daher ist eine Konstruktion mittels `this.Invoke()` wie in Zeile 25 gezeigt notwendig.

Über diese Anwendung kann nun transparent auf das Sensornetzwerk zugegriffen werden. Es ist irrelevant, ob der Knoten mit dem man kommuniziert direkt an der Schnittstelle des PCs angeschlossen oder über Funk erreichbar ist. Lediglich die Reaktionszeit variiert deutlich.

```
using System;
using System.Windows.Forms;
using Flow.SensorNodes;

namespace WindowsFormsApplication1 {
 public partial class NodeForm : Form {
   BeispielNode node;

   public NodeForm() {
     InitializeComponent();
   }

   public NodeForm(BeispielNode node) : this() {
     // store the node in a global variable
     this.node = node;
     // display the NodeId in the title of the form
     this.Text = "Node ID: " + node.NodeID;
     // and register for the event
     this.node.PrimeNumberDetectedEvent += new EventHandler<BeispielNode
         .PrimeNumberDetectedEventArgs>(node_PrimeNumberDetectedEvent);
   }

   // the Event from the Sensornode
   void node_PrimeNumberDetectedEvent(object sender, BeispielNode.
       PrimeNumberDetectedEventArgs e) {
       // Invoke is required for the event
       this.Invoke((Action)delegate() {
       // show the data of the event in the listbox
       this.listBox1.Items.Add("Prime number found: " + e.Number);
     });
   }

   private void buttonBeep_Click(object sender, EventArgs e) {
     // let the node beep when the beep button is pressed
     node.Beep();
   }

   private void buttonClear_Click(object sender, EventArgs e) {
     // call the ClearCount function on the node
     node.ClearCount();
   }

   private void buttonGet_Click(object sender, EventArgs e) {
     // query the variable from the node
     uint c = node.count;
     this.textBoxCount.Text = c.ToString();
   }

   private void buttonCalc_Click(object sender, EventArgs e) {
     // call the Calc function of the sensor node with two parameters
     uint x;
     node.Calc((uint)this.numA.Value, (uint)this.numB.Value, out x);
     // and show the return value in the textbox
     this.textBoxCalcResult.Text = x.ToString();
   }
 }
}
```

Abbildung 3: Quellcode von `NodeForm`