KTH Electrical Engineering

Communication between PC and motes

in TinyOS

AITOR HERNÁNDEZ

Stockholm July 25, 2011

TRITA-EE 2011:XXX
Version: 0.1

Contents

Chapter 1

Introduction

In Wireless Sensor Network (WSN) the nodes are usually designed to be completely independent and autonomous. In some environments it is useful to have them connected to a machine with more computational capabilities, like a computer. This connection could give more computational power for computing a better controller, having one node acting as a manager communicating with the computer or simply, one node analysing the status of the network to log data or analysing traffic in the network.

This document presents the tools and platforms that we have been using at the Automatic Control department. This tools are used in some experiments, for instance [2, 3]. In these cases the tools are modified, but here we provide tutorials and templates to learn how to use them.

The code is available on the following URL [1]: `http://code.google.com/p/kth-wsn/source/browse/trunk/kth-wsn/apps.tutorials/CommPC`.

[1]Check [1] to checkout the code

CHAPTER 2

Platforms and Tools

In this chapter we present the tools that we have been using. We start with the Serial-forwarder and then we show different ways to communicate between the Serial-forwarder and Matlab or LabView.

2.1 Serial-forwarder

There are two different implementations of these applications, but for simplicity and efficiency we have chosen the C-version. It is very powerful because is implemented under C, and uses a very simple code. Moreover, it could be modified to fulfil our special requirements.

Table 2.1 shows all the applications included in the `sf` suite.

In case you want to use the Serial-Forwarder suite in Windows, skip the next Section and download the following file:

`http://kth-wsn.googlecode.com/svn/trunk/kth-wsn/extra-tools/CygwinBin/CygwinBin.tar.gz`

2.1.1 Compilation

The compilation method is only described for a Debian based Linux distribution, but it should be similar for other distributions.

Before starting the compilation process, be sure that you have the TinyOS properly installed and the environmental variables are set, otherwise it will not compile. The applications need some of the headers files in the TinyOS `tos/` folder.

The instructions could be found on the README file as well.

Application	**Description**
`sf`	This application forwards the messages that it receives from the serial port to the TCP/IP server that it creates. To read the packets, we only need to open a TCP/IP connection with the server created and we will read the packets. In case of packets errors, the `sf` discard the packet, which means that the Serial-Forwarder assure that the data is fully valid.
`sflisten`	It creates a TCP/IP connection with the `sf` server and shows the results in the standard output (screen)
`sfsend`	It creates a TCP/IP connection with the `sf` server and send the packet passed as an argument
`seriallisten`	Instead of creating a server, and then using a client to read the packets, this application forward the packet from the serial port automatically to the standard output (screen)
`serialsend`	Instead of creating a server, and then using a client to read the packets, this application send the message passes as an argument

Table 2.1: Description of the different application of the `sf` suite

1. Install the automake package and all its dependences

   `$ sudo apt-get install automake`

2. Go to the folder where your TinyOS source code is. Then go to the path `support/sdk/c/sf`.

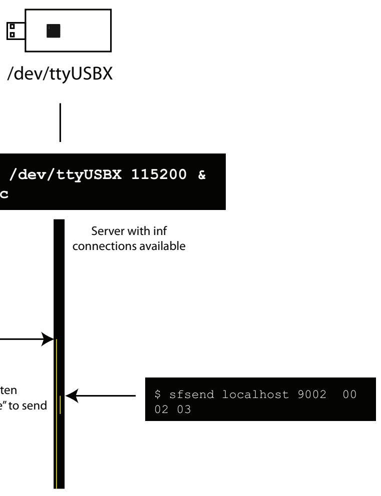
3. `$./bootstrap`

4. `$./configure`

5. `$make`

6. At this point, I recommend to copy the binary files to a `bin/` folder like `~/bin` or `/usr/bin`. Be sure that the folder is on the PATH environmental variable (`$ printenv PATH`)

7. Go to your home folder (`$ cd`) and check if you could run the `$ sf` from there. If not, please check the previous step.


/dev/ttyUSBX
Server with inf connections available
Although the sflisten is running, it is "possible" to send

```
$ sf 9002 /dev/ttyUSBX 115200 &
Note: Sync
```

```
$ sflisten localhost 9002
```

```
$ sfsend localhost 9002  00
02 03
```

Figure 2.1: Example of using the `sf` server

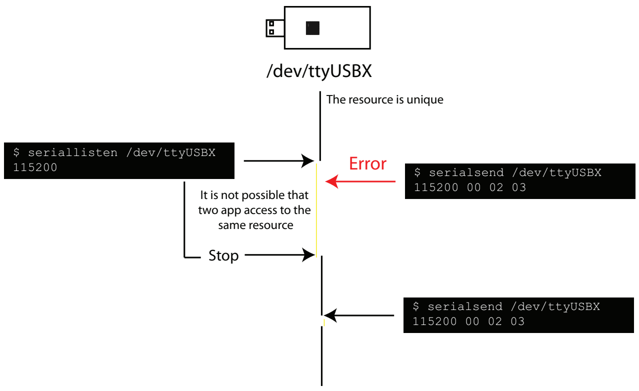
2.1.2 Usages

After the compilation and set-up of the applications, it is time to know how to use them. We show the usage functions for each application and an example for each of them.

Application	**Usage**	**Example**
`sf`	`sf <port> <device> <rate>`	`sf 9002 /dev/ttyUSB0 115200`
`sflisten`	`sflisten <host> <port>`	`sflisten localhost 9002`
`sfsend`	`sfsend <host> <port> <bytes>`	`sfsend localhost 9002 0 2 3 4`
`seriallisten`	`seriallisten <device> <rate>`	`seriallisten /dev/ttyUSB0 115200`
`serialsend`	`serialsend <device> <rate> <bytes>`	`serialsend /dev/ttyUSB0 115200 0 2 3 4`

To make it simply, below there are some figure with different ways to do the same by using some of utilities form the Serial-Forwarder package. Both examples show a way to listen and send packets from/to the mote. In the example, we also show the difference between using one group or the other.

Figure 2.1 shows an example where one mote is selected to run the `sf`. After that, the



```
$ seriallisten /dev/ttyUSBX
115200
```

```
$ serialsend /dev/ttyUSBX
115200 00 02 03
```

```
$ serialsend /dev/ttyUSBX
115200 00 02 03
```

Figure 2.2: Example of using the `serial*` tools

`sflsiten` is called to listen the serial packets received. With the `sflsiten` running and receiving messages with a low packet arrival probability, we try to send a packet using the `sfsend`. If at the moment we execute the `sfsend` the serial port is free we send a packet to the mote. In this scenario we could have multiple connections using the `sflisten` without any problem. For example we want to log the raw data that we receive from the motes, but we want to run a LabView Vi at the same time.

Figure 2.2 shows an example where we want to read/write from/to the mote without using the `sf` server. In that case, every time we want to open a new listener (`seriallisten`), we open a connection with the serial resource. For this reason if we try to send at the same time we are receiving, we will get an error. It is not possible to use more that one listener neither.

2.1.3 Examples

In the previous Section, we have learned have the command should be used. In this Section, we purpose different exercises to play with the different tools.

2.1.3.1 Exercises

`printf` vs packets In TinyOS, it is important to distinguish between `printf` messages and other kind of packets.

In the folder `CommPC/MotesFiles/TestWriteReadSerialComm` is stored the application which sends periodically a custom message. Check the header file for more details of the message structure. In the folder `CommPC/MotesFiles/TestPrintf` is stored the application which sends periodically `printf` messages to the PC.

By using these application and the Serial-Forwarder suite. Answer the next questions:

- *What is the destination and source address for both messages?*
- *How many bytes are in the `TestSerialCommMsg` message?*

- *What does the `TestSerialCommMsg` message contains?*
- *What are the types of both messages? Why could it be useful?*
- *Below we have an output of the `sflisten` for the `TestWriteReadSerialComm` application. Which part is the payload? What is its meaning?*

  00 ff ff 00 00 04 00 06 00 00 00 41
- *Below we have an output of the `sflisten` for the `TestPrintf` application. Which part is the payload? What is its meaning?*

  00 ff ff 00 00 1c 00 64 48 65 72 65 20 69 73 20 61 20 75 69 6e 74 38 3a 20 31 31 0a 48 65 72 65 20 69 73 20

`sflisten` vs `seriallisten` The difference between `sflisten` and `seriallisten` is explained in Figures 2.1 and 2.2. But try to answer yourself.

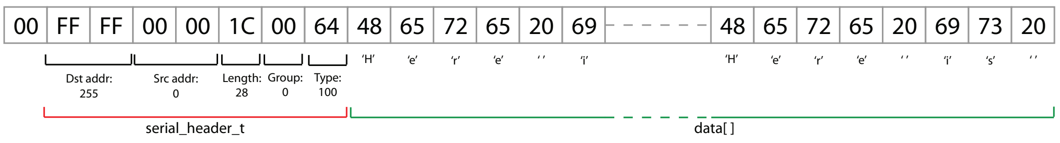
Modification of the `seriallisten` or `sflisten` If you did the exercise *`printf` vs packets* you understand which are the difference between these packets. In this exercise, I would purpose to create a *PrintfClient*, but a C version of it.

To simply the compile method, I recommend to create a backup of the `sflisten`, and modify the `sflisten`.

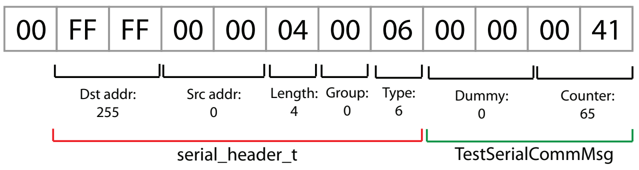
The solution needs to have less than five more lines comparing to the current version.

2.1.3.2 Solutions

`printf` vs packets We usually we do not distinguish between a `printf` and a normal packet transmitted through Universal Serial Bus (USB). The `printf` is just an special packet transmitted through USB, the packet contains a certain number of characters on his payload.



(a) Structure of a `printf` packet received in the Serial-Forwarder



(b) Structure of a custom packet received in the Serial-Forwarder

Figure 2.3: Structure of a serial packet transmitted over the USB port from the mote to the PC. Examples for the `printf` and a custom packet

Figure 2.3 shows the structure of two different packets. Figure 2.3a shows the structure for a `printf`. The `printf` sends packets of 28 bytes with the characters that it has in the buffer. Every time we call a `printf`, we are adding characters to the buffer, for this reason the same packet could contains parts of two different `printf` commands, as we have in the figure.

Figure 2.3b show an example where we are transmitting a customize message, the `TestSerialCommMsg`. The message contains two fields, two `nx_uint16_t` numbers. It is important that when we create this kind of structures the types are external (`nx_`).

`printf` is useful for debugging purpose, even though it waste a lot of computational time and resource. If your purpose is to send data to the PC to log it, create you own message.

`sflisten` vs `seriallisten` The difference between `sflisten` and `seriallisten` is explained in Figures 2.1 and 2.2.

- To run the `sflisten` we need to connect first to the `sf` server.
- The `seriallisten` connects directly to the motes.
- It is possible to open multiple `sflisten` and open a connections to the `sf` server from other programs, like Matlab or LabView
- `seriallisten` could only be opened once and there is no possible to use it with any other programs

Modification of the `sflisten` or `seriallisten` As we have seen in the first exercise, the `printf` sends the message, and every byte in the payload is one character. So, to convert the `sflisten` into a `sfprintf` we only need to change the `printf` and skip the header bytes.

Below we have the solution.

```
#include <stdio.h>
#include <stdlib.h>

#include "sfsource.h"

#define HEADER_OFFSET 8
 int main(int argc, char **argv)
 {
   int fd;
    unsigned char * currentField;
   unsigned char i, j;
   if (argc != 3)
     {
       fprintf(stderr, "Usage: %s <host> <port> - dump packets
       from a serial forwarder\n", argv[0]);
       exit(2);
     }
   fd = open_sf_source(argv[1], atoi(argv[2]));
   if (fd < 0)
     {
       fprintf(stderr, "Couldn't open serial forwarder at %s:%s\n",
               argv[1], argv[2]);
       exit(1);
     }


   for (;;) {
     int len, i;
     const unsigned char *packet = read_sf_packet(fd, &len);

     if (!packet) exit(0);

     for (j=HEADER_OFFSET; j < len ; j++)
         printf("%c", packet[j]);


     fflush(stdout);

     free((void *) packet);
     }
 }
```

2.2 LabView

In this Section, we present the LabView templates to communicate between the mote and the PC. The SubVi provides an easy way to use the different modules and simplify our LabView programs. Table 2.2 shows a description of all the dependences implemented for this purpose.

In all the templates, the `sf` server needs to run before the LabView.

At this point, you need to have the source code in your computer: http://code.google.com/p/kth-wsn/source/browse/trunk/kth-wsn/apps.tutorials/CommPC.

Following we show different templates and their motes program to make them running. To sum up what we see there, we have Table 2.3

	SubVi name	Icon	Description
Serial-Forwarder	`OpenSF`	OpenSF	Open a connection with the `sf` server
	`ReadSF`	ReadSF 1.1	Read (listen) packets from the `sf`, from the mote. We receive a burst of bytes to be parsed
	`WriteSF`	WriteSF	Write (send) packets from the `sf`, . We receive a burst of bytes to be parsed
	`CloseSF`	CloseSF	Close the connection with the `sf` server.
LabView to Mote	`BuildPacket`	BuildPck	Build the structure required to be received by the motes
	`NumberToArrayUint8`	Num2Arr	Convert a number which is represented with more than 1 byte, in an array to be properly understandable in the motes
	`ToFloatMote`	ToFloat 123	Convert a float value in LabView to a float to be understandable in the motes
Mote to LabView	`ToFloat`	ToFloat 123	Convert the value received from the mote to a float
	`ToUint32`	ToUint 123	Convert the value received from the mote to a uint32
	`ToUint16`	ToUint 123	Convert the value received from the mote to a uint16
	`ToUint8`	ToUint 123	Convert the value received from the mote to a uint8

Table 2.2: Dependences to use the `sf` in LabView

LabView template	**Motes applications**	**Description**
`TestReadFromMote`	`TestWriteReadSerialComm/`	Listen or read the packet from the mote
`TestWriteFromMote`	`TestWriteReadSerialComm/`	Send or write packets to the mote
`TestMoteComm`	`TestSerialComm/`	Bidirectional communication using one mote
`TestHIL`	`TestHIL/`	The idea is to have a real scenario by using two motes, one sensor and one actuator transmitting packet between each other using IEEE 802.15.4
`TestControllerPC`	`TestWriteReadSerialComm`	Template of how to implement a “Controller in the PC” scenario

Table 2.3: Summary of the templates and the motes applications related

2.2.1 Mote to PC template

2.2.1.1 Mote: TestWriteReadSerialComm application

This application sends periodically a packet to the serial port and increments the counter each iteration. To check the correct behaviour is needed to check the results on the PC. You should get a ramp (y=x). If the writing is enabled on the PC, the Leds blink every time the motes receives a message and show the (received value in binary) % 8.

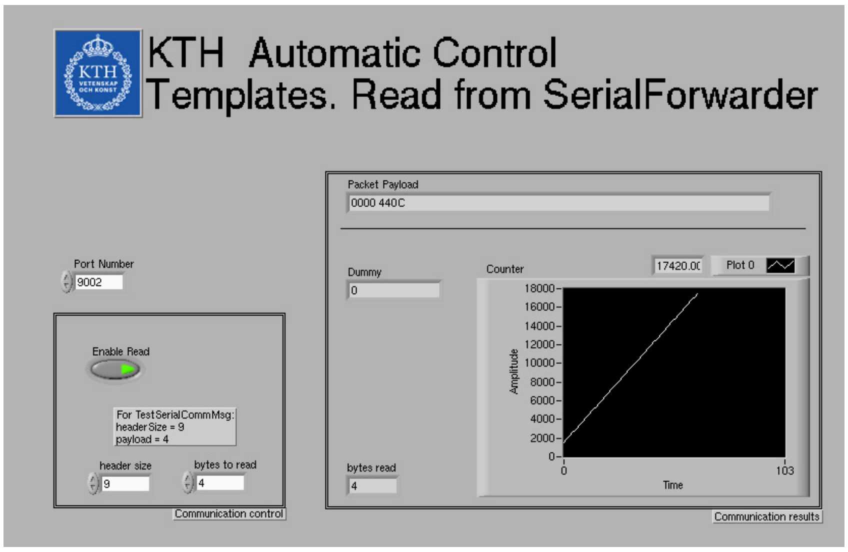
Before starting the LabView file, we need to create the TCP/IP server in the port number where the mote is connected. For example, assuming that we have the mote with the TestWriteReadSerialComm application already installed in the port */dev/ttyUSB0*, we need to call:

```
sf 9002 /dev/ttyUSB0 115200 &
```

2.2.1.2 PC: TestReadfromMote.vi

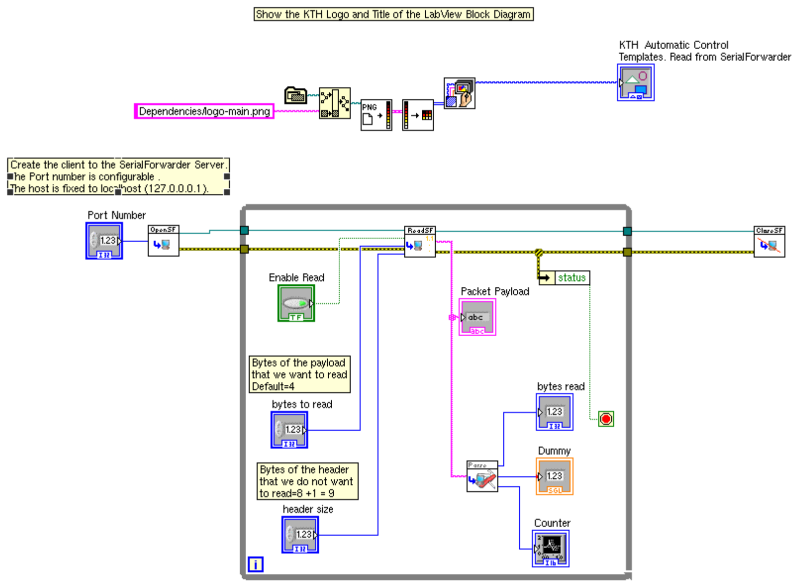
Figure 2.4 shows the front panel of the Vi that reads the values from the mote. In the *Communication control panel* we need to configure the number of header in the byte and the payload that we have configured in order to read the packets properly. In the *Communication results panel* we can see the *dummy* value that we have received and the counter in the chart.

Figure 2.5 shows the block diagram of the loop that we use for reading. Starting from the left of the diagram, we found the port number of the TCP/IP port that we need to configure in order to open the TCP/IP connection in the port where we have the `sf`. The



For TestSerialCommMsg:
headerSize = 9
payload = 4

Figure 2.4: Front panel TestReadFromMote



Create the client to the SerialForwarder Server.
The Port number is configurable .
The host is fixed to localhost (127.0.0.0.1).

Bytes of the payload
that we want to read
Default=4

Bytes of the header
that we do not want
to read=8 +1 = 9

Figure 2.5: Block diagram of TestReadFromMote

next block is the *OpenSF*, that opens the connection to the port and gives the status and the connection ID to the next blocks. Inside the for loop we found the controllers which selects the bytes for the header and the payload, by default in this example are 9 and 4 respectively. It is really important to be sure that these values are set properly, for this we could check the *Packet payload* indicator in the front panel.

With the bytes for the header and payload configured, the *ReadSF* SubVi, try to read the packet from the TCP/IP server. This is an important SubVi, because it could block the loop. Depending on the application, we need to wait until we receive a packet, in some it is enough just to have an offset. To configure this, we could modify the timeout field. By default is set to Infinity and block all the Vi if it does not receive packets. So, take special attention to this SubVi.

After receiving a packet, we use a *ParsePacket* SubVi which will convert the received string to the values that we need.

2.2.2 PC to Mote template

This example uses the same program for the mote as the previous example, `TestWriteReadSerialComm` 2.2.1.1

Before starting the LabView file, we need to create the TCP/IP server in the port number where the mote is connected. For example, assuming that we have the mote with the TestWriteReadSerialComm application already installed in the port */dev/ttyUSB0*, we need to call:

```
sf 9002 /dev/ttyUSB0 115200 &
```

2.2.2.1 PC: TestWriteFromMote.vi

Figure 2.6 shows the front panel of the Vi that writes (sends) information to the mote. In the front panel, the *Communication control panel* allows the modification of the TCP/IP port where the `sf` is connected. Moreover, in the *Communication results panel* we could change the *dummy* field of the packet that we send.

Figure 2.7 shows the for loop where we implement our sending functions. In this example we are sending packets to the motes every 250ms. On each iteration we send the previous value +1, so the results that we have in the chart shall be ramp.

2.2.3 Bidirectional communication template

We have created three templates where we implement different scenarios for bidirectional communication. On the first case (TestCommunication) we have one mote connected to the computer receiving and sending information from/to the mote/PC, in that case the reception is mandatory so we block the Vi if we do not receive any message. The second template (TestHIL), involves two motes, one of them acts a a receiver and the other as a sender, the reception is also mandatory. In the last example (**??**), with only one mote, we have the freedom to configure if it is mandatory to receive packet or not, and the transmission is optional by clicking on the button.

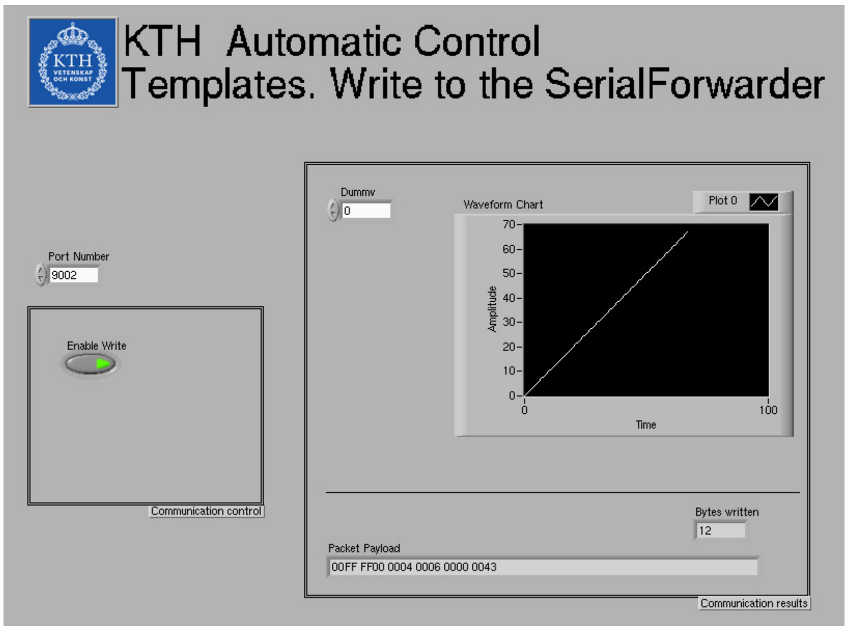


Figure 2.6: Front panel TestWriteToMote

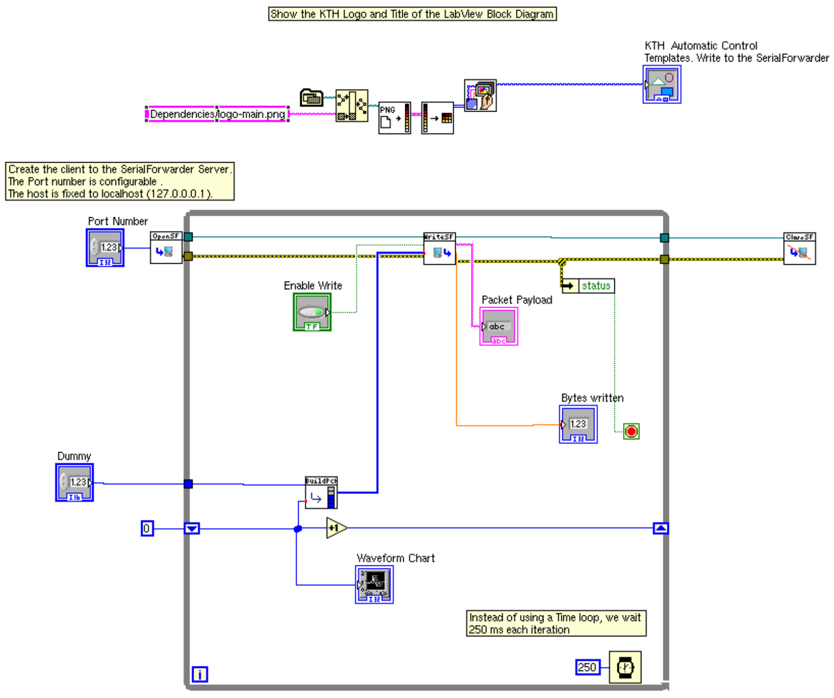
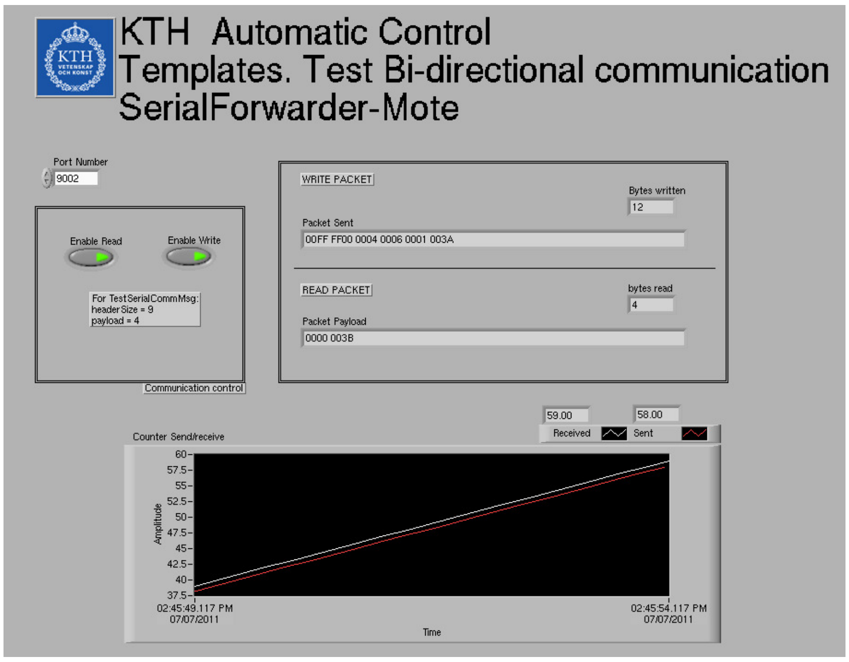

Show the KTH Logo and Title of the LabView Block Diagram
KTH Automatic Control
Templates. Write to the SerialForwarder
Dependencies/logo-main.png
Create the client to the SerialForwarder Server.
The Port number is configurable .
The host is fixed to localhost (127.0.0.1).
Port Number
OpenSF
WriteSF
CloseSF
Enable Write
status
Packet Payload
Bytes written
Dummy
BuildPck
Waveform Chart
Instead of using a Time loop, we wait
250 ms each iteration
250

Figure 2.7: Block diagram of TestWriteToMote



KTH Automatic Control
Templates. Test Bi-directional communication
SerialForwarder-Mote

Port Number: 9002

Enable Read
Enable Write

For TestSerialCommMsg:
headerSize = 9
payload = 4

Communication control

WRITE PACKET
Bytes written: 12
Packet Sent: 00FF FF00 0004 0006 0001 003A

READ PACKET
bytes read: 4
Packet Payload: 0000 003B

Received: 59.00
Sent: 58.00

Figure 2.8: Front panel TestMoteComm

2.2.3.1 TestCommunication

Mote: TestSerialComm application This program involves reading and sending from the serial port. The program is event-driven by the reception of a packet through the serial port.

When we receive a packet into the serial port, an event is generated, `UartReceive.receive`, In this function if the packet length is equal to the expected, we blink the LEDs according to the counter value that we have received. Then we increase the counter by adding 1 unit, and we send the packet back to the PC.

Before starting the LabView file, we need to create the TCP/IP server in the port number where the mote is connected. For example, assuming that we have the mote with the `TestSerialComm` application already installed in the port */dev/ttyUSB0*, we need to call:

```
sf 9002 /dev/ttyUSB0 115200 &
```

PC: TestMoteComm.vi Figure 2.8 shows the front panel of the Vi that reads/writes in the serial port periodically. We can see the raw data of the packet sent to the mote, and the payload data of the packet received from he mote. At the bottom, we have a chart that shows the difference between the received and sent value. As we see in Figure 2.9, in the LabView we do not modify the counter value, we just forward the information.

Figure 2.9 shows the block diagram of the `TestMoteComm` vi. In this example we have added another variation compared to the previous one. In this case the timeout is not set

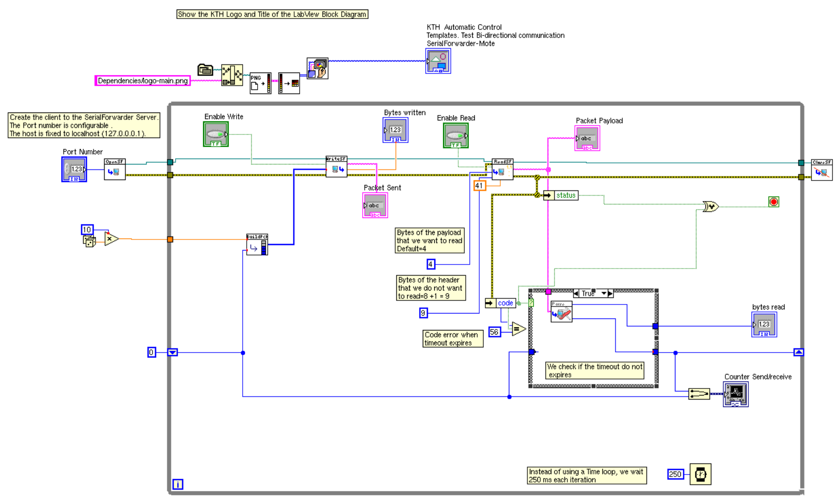


Figure 2.9: Block diagram of TestMoteComm

to Infinity, so the `ReadSF` does not block the model. By doing this, we need to take into account that the SubVi will not always give us data, so we need to control that. This is what we do in the If case structure. We compare if we receive an error, meaning that the timeout has expired.

2.2.3.2 TestHIL

For this example, we have tried to simulate a Hardware In Loop, where we have a sensor and actuator scenario. But we can see them as receiver (actuator, read from PC), and the sender (sensor, write to the PC).

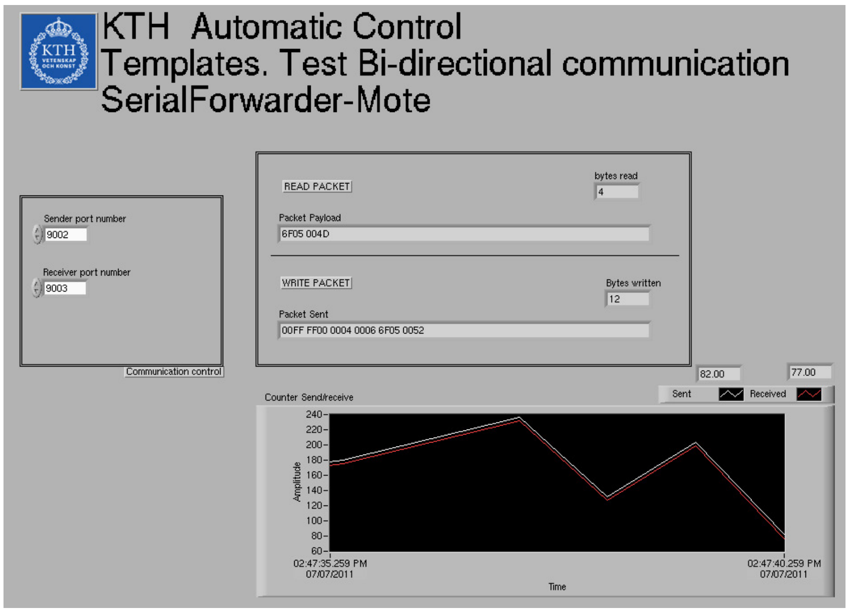
Mote: TestHIL application In this case, we have two different roles, one for the sensor and another for the actuator.

The sensor is sending random values periodically to the PC, the PC compute something with the value (increment + 5, for instance) and sends the value to the actuator mote. When the actuator mote receives a packet, it shows the value in the LEDS and transmit the value to the sensor node through radio using the IEEE 802.15.4 standard protocol.

The communication between actuator and sensor has been included in this example, in order to have the most extensive example possible. In most of our applications it is used the IEEE 802.15.4 to communicate between different motes.

Before starting the LabView file, we need to create the TCP/IP server in the port number where the mote are connected. For example, assuming that we have the mote with the *TestHIL/actuator* application already installed in the port */dev/ttyUSB0*, we need to call:

```
sf 9002 /dev/ttyUSB0 115200 &
```


KTH Automatic Control
Templates. Test Bi-directional communication
SerialForwarder-Mote

Sender port number 9002
Receiver port number 9003
Communication control

READ PACKET
bytes read 4
Packet Payload 6F05 004D

WRITE PACKET
Bytes written 12
Packet Sent 00FF FF00 0004 0006 6F05 0052

82.00
77.00

Figure 2.10: Front panel TestHIL

And if the mote the mote with the *TestHIL/sensor* is on the port */dev/ttyUSB1*, we need to call:

```
sf 9003 /dev/ttyUSB1 115200 &
```

PC: TestHIL.vi Figure 2.10 shows the front panel of the `TestHIL` template. In the *Communication control panel* we can configure the ports for the sender and receiver TCP/IP servers. The rest of the front panel is the same as the previous ones.

Figure 2.11 shows the block diagram of the `TestHIL` template. In this case, we need to use a *Sequential frame*, because we need to have the two motes connected. With this frame, we ensure that until we do not receive the packet from the sensor, we do not send any actuator value to the actuator mote.

2.2.3.3 TestControllerPC

This application simulates a scenario where the controller is implemented in the computer, which has more computational capabilities than the mote itself. This scenario a unique mote connected to the PC, which could be considered as a bridge between the process and the PC.

This example uses the same program for the mote as the previous example, `TestWriteReadSerialComm` 2.2.1.1

Before starting the LabView file, we need to create the TCP/IP server in the port number where the mote is connected. For example, assuming that we have the mote with

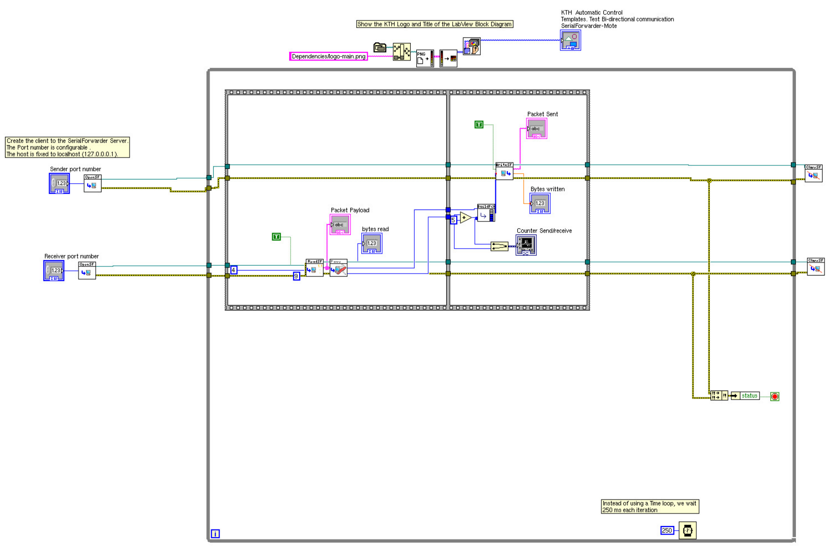

Show the KTH Logo and Title of the LabView Block Diagram
KTH. Automatic Control
Templates. Test Bi-directional communication
SerialForwarder-Mote
Dependencies/logo-main.png
Create the client to the SerialForwarder Server
The Port number is configurable
The host is fixed to localhost (127.0.0.1)
Sender port number
Receiver port number
Packet Sent
Bytes written
Packet Payload
bytes read
Counter Send/receive
status
Instead of using a Time loop, we wait 250 ms each iteration
250

Figure 2.11: Block diagram of TestHIL

the `TestWriteReadSerialComm` application already installed in the port */dev/ttyUSB0*, we need to call:

```
sf 9002 /dev/ttyUSB0 115200 &
```

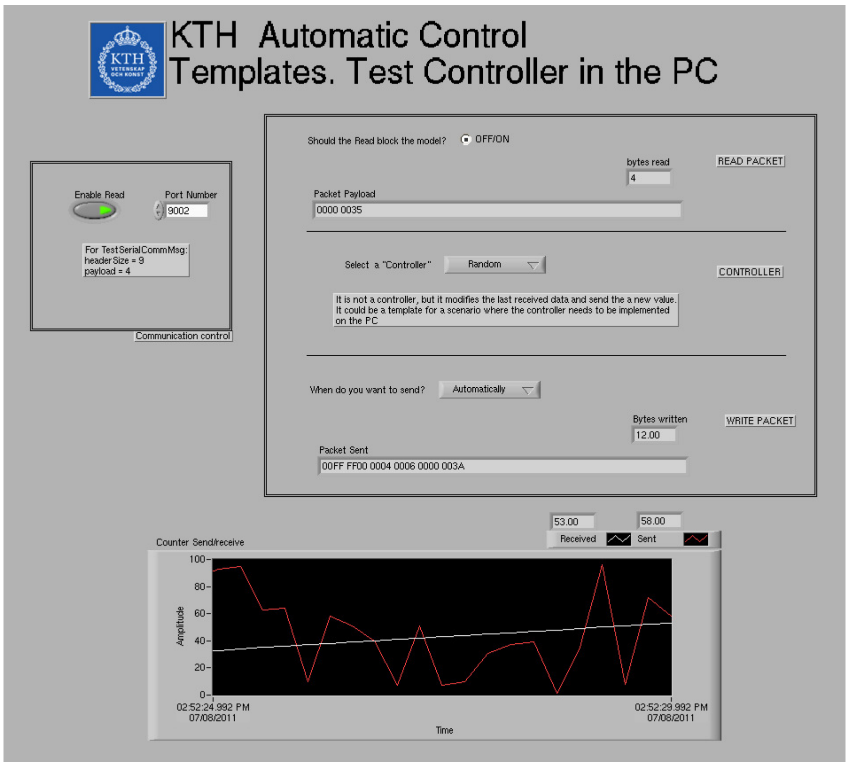
PC: TestControllerPC.vi In this template, we want to show another set of options which could give the desired configuration for different cases.

Figure 2.12 shows the front panel for the `TestControllerPC` template. The front panel is similar to the previous examples. However, the Vi shows different options that are interested to explain.

The example includes different options for the reading and the writing part. For the reading is possible to modify the timeout. The timeout will allow to choose if the reception of a packet is mandatory or not for the normal running of the application. In this example, we can switch between Infinity and 250 ms by pressing on the On/Off button. We also have a panel called *Controller* which allows the user to switch between different transfer functions.

Figure 2.13 shows the block diagram of the *TestControllerPC*. All the functionality is inside the for loop. It could be divided in three blocks, starting from the left to right: reading, controller and writing.

After all the previous templates, we are quite familiar with the block and tools involve. For the reading, we have modified the timeout, which could be modified with a switch. The if case in the middle, allow the user to have a place where the controller needs to be place. And the third part, the writing, optionally we can modify if we want to send packets automatically after each reception, or we want to do it manually by pressing a button in the front panel.



KTH Automatic Control
Templates. Test Controller in the PC

Enable Read
Port Number: 9002

For TestSerialCommMsg:
header Size = 9
payload = 4

Communication control

Should the Read block the model? OFF/ON

bytes read: 4
READ PACKET
Packet Payload: 0000 0035

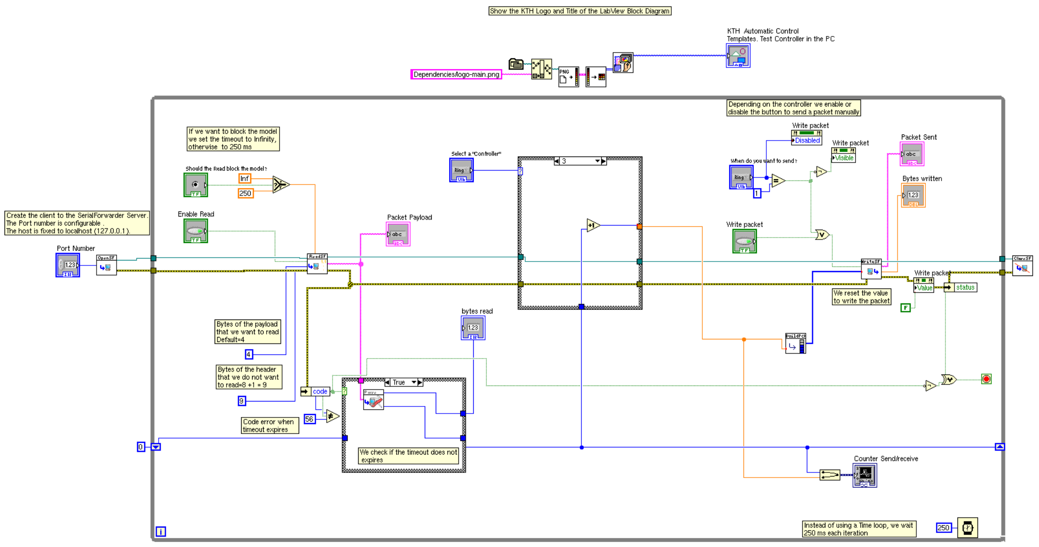
Select a "Controller": Random
CONTROLLER

It is not a controller, but it modifies the last received data and send the a new value.
It could be a template for a scenario where the controller needs to be implemented on the PC

When do you want to send? Automatically

Bytes written: 12.00
WRITE PACKET
Packet Sent: 00FF FF00 0004 0006 0000 003A

Figure 2.12: Front panel TestControllerPC



Show the KTH Logo and Title of the LabView Block Diagram

Create the client to the SerialForwarder Server.
The Port number is configurable .
The host is fixed to localhost (127.0.0.1).

If we want to block the model we set the timeout to Infinity, otherwise to 250 ms

Bytes of the payload that we want to read Default=4

Bytes of the header that we do not want to read=8 +1 = 9

Code error when timeout expires

We check if the timeout does not expires

Depending on the controller we enable or disable the button to send a packet manually

We reset the value to write the packet

Instead of using a Time loop, we wait 250 ms each iteration

Figure 2.13: Block diagram of TestControllerPC

2.2.4 Exercices

Change packet structure In Section 2.3.1.3 we explain the messages are set in order to transmit and received them easily. What we propose is to modify the message structure with more fields and different types. Follow the step below:

1. Add a `float` field to the `TestSerialCommMsg` in the `CommPc/MotesFiles/app_profile.h`. Remember that the variable needs to be external, so the new fields should be `nx_float`.
2. Add a vector field of `uint16` elements to the `TestSerialCommMsg`
3. Delete the `*.java` and `*.class`
4. Modify the program to write on the new fields.
5. Compile the application. The `*.java` and `*.class` shall be created.
6. Modify a LabView model, for example the `TestReadFromMote`, to read the new fields. If the modifications are done in a SubVi, I recommend to create a separate SubVi.
7. In the `functions/readFromBaseStation.m`, we can read the new values.

If we do the same exercise in Matlab, we can easily realize that the modification of the `m-file` is much easier than LabView.

2.3 Matlab

Apart from the LabView file, Matlab is a good tool to enable the communication between the motes and the PC. LabView provides the simplicity, but Matlab allows a more complex controllers and it is easier for people with control background. An extensive comparison is shown in Section 2.5.

At this point, you need to have the source code in your computer: `http://code.google.com/p/kth-wsn/source/browse/trunk/kth-wsn/apps.tutorials/CommPC`.

2.3.1 Setup

To be able to use Matlab and communicate with the motes, first we need to configure and go through some steps. These steps are based in a Unix environment. There is no restriction of using Windows. Everything shall work on the same way.

2.3.1.1 Matlab setup

Classpath The `jmi.jar` library is needed for the compilation and used of the libraries in Matlab. To install it, you should modify your `~/.bashrc` in order to add the classpath for `jmi.jar`. Add the following line:

```
export CLASSPATH=$CLASSPATH:/opt/matlab/java/jar/jmi.jar
```

Modify the folder where you have Matlab installed. And restart the terminal after finishing.

Java libraries For this configuration, we need to compile extra libraries that are not included in TinyOS 2.x. The files are extracted from the TinyOS 1. Two different methods to have it are provided: compile them, or the use of the compiled `tinyos.jar`.

1. *Compiled tinyos.jar*: For simplicity, I would recommend to copy and paste the `*.jar` , and if it does not work, try to second option. Copy and paste the tinyos.jar place in the folder `Libraries/` into the `support/sdk/java/` folder.
2. *Compile:* Copy and merge the folder `Libraries/support/` in the `support/` folder of the TinyOS used for compiling. The folder only contains the `*.java` files. To recompile the `tinyos.jar`:
   - `$ cd support/sdk/java/`
   - `$ make tinyos.jar`

Application example In order to use and application example, you need to set the Matlab path manually, and add the application folder. The application is placed in the `MatlabFiles/` folder. In Matlab:

1. File -> Set path ...
2. Add Folder...
3. Select `MatlabFiles/`
4. Save

With this, every time you open Matlab, the `startup.m` script will be launch. Before restarting Matlab, modify the absolute paths that are in the `startup.m` script, with your own information.

2.3.1.2 Motes setup

For this template, we include all the different controllers and options within the same application. But we need to be sure that the application is the suitable for the selected options.

If we use this application with the Graphical User Interface (GUI) we have the name of the related application between parenthesis.

2.3.1.3 Message Java libraries

When we compile any of the applications, we see that a `.java` and `.class` is generated. The `.class` files will be used by the application to parse and send the packets. With this functionality we can easily read and write values in the message, as a structure.

At this point, I would like to show the steps in order to generate these messages:

1. Define the struct of your message:

```
typedef nx_struct TestSerialCommMsg {
nx_uint16_t dummy;
nx_uint16_t counter;
} TestSerialCommMsg;
```

2. Set your message ID. Create a field in the header file called, `AM_structureNAME`, in our case:

   ```
   AM_TESTSERIALCOMMMSG = 6
   ```

3. Compile the application with the following extra option in the `Makefile`.

   ```
   BUILD_EXTRA_DEPS += TestSerialCommMsg.class
   TestSerialCommMsg.java:
   mig java -target=telosb -java-classname=TestSerialCommMsg ../app_profile.h
   TestSerialCommMsg -o $
   TestSerialCommMsg.class:  TestSerialCommMsg.java
   javac TestSerialCommMsg.java
   ```

2.3.2 Application example

After setting up the environment, we can focus on the explanation of the different functions and folders that the application includes. Table 2.4 explains all the files and functions.

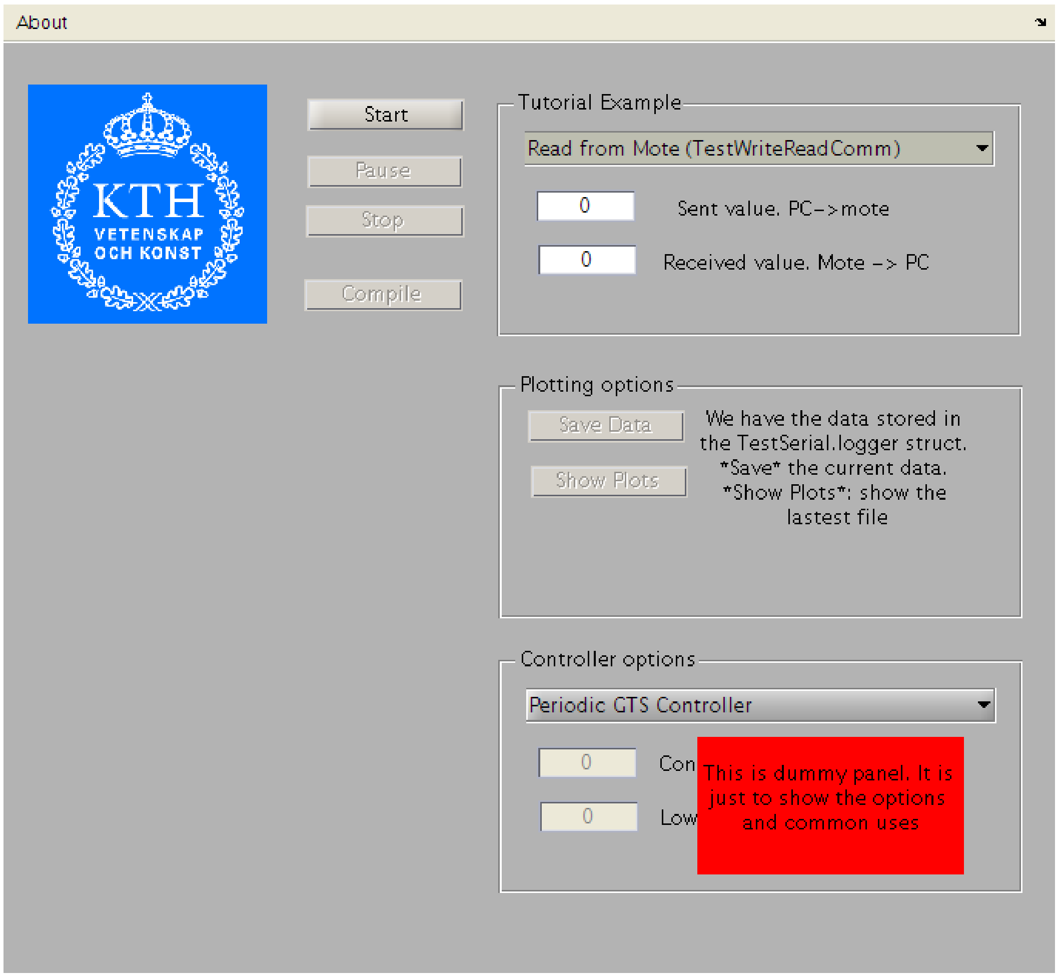
Folder	**File**	**Description**
`functions/`	`plotGlobal.m`	It is an example for plotting the data. The information is stored in the structure `TestSerial.logger`
	`readFromBaseStation.m`	This function is called when we receive a new message from the mote. In this example, we read the a *counter* value, increase it by 5 units and we send it back to the mote if the writing is enabled. Moreover store the received information in a structure `TestSerial.logger`, for plotting data.
	`sendPacket.m`	Send the new message to the `sf` that we are connected.
	`timerFired.m`	This is the callback for the timer created in the `testSerialComm.m` if the selected mode needs it.
	`testSerialComm.m`	Main application which uses all the other functions. It is placed as a function because we consider the main application the GUI
`functions/comm`	*	Libraries to communicate with the `sf`. They use the Java libraries.
`functions/gui`	`testSerialComm_gui.fig` `testSerialComm_gui.m`	This file allow the developer to modify the GUI easily. It contains the intelligence of the GUI, with all the callback and functions, that we need to run the experiments.

Table 2.4: Brief explanation of the functions involved in the application example

2.3.3 Graphical User Interface

If everything is properly set, by pressing `> testSerialComm_gui` the GUI will be launched.

Figure 2.14 show the GUI for the `TestCommunication` example. The GUI includes different panels: *Tutorial example*, *Plotting options* and *Controller options*. Mention, that not all the buttons and panel are available in this example. It gives an idea what could be done easily. Apart from these options, a figure could be added, to show the results on real-time.



0 Sent value. PC->mote
0 Received value. Mote -> PC

We have the data stored in the TestSerial.logger struct. *Save* the current data. *Show Plots*: show the lastest file

This is dummy panel. It is just to show the options and common uses

Figure 2.14: GUI for the `TestCommunication` example

2.3.4 Exercises

At this point, we would like to provide some exercises and examples to get familiar with the program and the GUI. The complexity of the exercises increases, so I will recommend to go step by step, and skip the exercises if you really know how to do it, and you did it before. Some of the exercises requires several hours do be completely implemented.

Change packet structure In Section 2.3.1.3 we explain the messages are set in order to transmit and received them easily. Now, we would like to check if the process is clear. What we propose is to modify the message structure with more fields and different types. Follow the step below:

1. Add a `float` field to the `TestSerialCommMsg` in the `CommPc/MotesFiles/app_profile.h`. Remember that the variable needs to be external, so the new fields should be `nx_float`.

2. Add a vector field of `uint16` elements to the `TestSerialCommMsg`
3. Delete the `*.java` and `*.class`
4. Modify the program to write on the new fields.
5. Compile the application. The `*.java` and `*.class` shall be created.
6. Restart Matlab, to reload the Java libraries.
7. In the `functions/readFromBaseStation.m`, we can read the new values.

Add a new tutorial example: TestController In the LabView case, we have seen an example where we implement a example which simulates a controller, `TestControllerPC.vi`. In this exercise we propose to generate the option to use a controller, and be able to change it with the *Controller options*. Below, we have some tips:

- To add/remove some buttons or options in the visualization of the GUI, check the `gui/testSerialComm_gui.fig`
- To modify the behaviour of the GUI, check the `gui/testSerialComm_gui.m`
- To modify the functionality of the program, check the following files `functions/testSerialComm.m`, `functions/readFromBaseStation.m` and/or `functions/timerFired.m`

Enable *Compile* button In this exercise, we expect you to be able to enable the *Compile* button, and compile the correct application depending on the tutorial example that is selected. Some guidance is provided below:

- Create a script (Shell, Python, Perl, PHP....) to compile an application for a given application name as an argument. For example:

  `moteinstallByFolder 0 TestWriteReadSerialComm`, could be the instruction to compile and install the `TestWriteReadSerialComm` application into to mote connected to the */dev/ttyUSB0*
- By default, disable the *Start* button until the compilation has finished
- Every time the tutorial example changes, disable the *Start* button.
- Enable the *Controller options* panel, and add the options that we have in LabView.

Plotting in the GUI Currently, in the GUI we only show the received and sent values in a text box. We suggest to modify the GUI and create a figure to update automatically a plot with the received and/or sent values.

No tips are given the last exercise.

2.4 Matlab/Simulink

There is also the possibility to use Simulink for reading packet from the `sf`. We do not considered this option important comparing to Matlab or LabView. Therefore, we just refer it to the placed where we have an example. You could find it for example on the

http://code.google.com/p/kth-wsn/source/browse/trunk/kth-wsn/apps.water-tank/CentralizedControllerGeneral/MatlabSimulnk/

or check the following Wiki page:

http://www2.ee.kth.se/web_page/netcon/mediawiki/index.php/Wireless_Process_Control

2.5 Comparison

We have presented different ways to communicate between the PC and the motes. Depending on the case and the purpose is better to use one or another. What we want to explain here, is, under our experience, in which cases is better and which are the good and bad points for each.

LabView

- ✓ Programming with schematics
- × Not common programming language at university
- ✓ Easy for graphical user interfaces
- ✓ Possibility to export data to a files to plot it nicely in Matlab.
- × Control library not included by default
- ✓ No special configuration for TinyOS.
- × Difficult to parse packets.
- × Difficult to automatize processes

Matlab

- ✓ Programming with Matlab language
- ✓ People are used to program in Matlab
- × Not easy to create complex graphical user interfaces
- ✓ Plots for papers are done in Matlab.
- ✓ Configuration for TinyOS is needed, Java libraries
- ✓ With the previous configuration, we can read packets as a structure.
- ✓ Combining Matlab/Bash/PHP script we can automatize everything.

Below, there are some questions that could help you to decide with program you should use for your application.

Do you prefer programming with code or schematics? It is really important to choose the program which you feel more comfortable. If you like writing code, perhaps you will find LabView hard and complicated. On the other hand, having the schematic helps to see the functionality and it is more intuitive.

Have you ever used LabView? Actually, it is not really necessary that you have worked before with LabView, it is quite intuitive and easy. If it is your first time, please start using our templates.

Are you going to receive different packets types? In Matlab is much easier to parse the packets, because with the proper configuration, we receive them as structures (for example `meessage.get_counter()`). On the other hand, in LabView we need to decode the packet manually. We have SubVi to parse the information in LabView.

Are you familiar with any script language? If you are not, this should not influence in your answer and skip the next question.

Are you in the developing step? If you need to compile, debug, change the controller quite often, it is really interesting to think about generating an script to compile them automatically with new controllers. For example: if we realize that the controller is wrong and it is implemented on the mote we need to:

- Stop Matlab/LabView
- Kill Serial-Forwarder server
- Program again, the motes or motes that we have in the correct order
- Wait until the programming has finished
- Start the Serial-Forwarder server again
- Start Matlab/LabView

What if, we would have:

- Stop Matlab/LabView
- Run the script to stop Serial-Forwarder, program the motes and start Serial-Forwarder-
- Start Matlab/LabView

The process has been simplified and we could same time. We have not tried to call external programs from LabView, but maybe it is also possible. In Maltab is possible by using:

```
> unix('myScript')
```

or

```
> system('myScript')
```

Do you need a complex GUI and the user needs to interact with it? Definitely, in LabView this is much more easier. It is drag, place and wire it. In Matlab, it is not complicated to add the figures and button, it is also drag and place. But for the functionality, you need to write it yourself. in the `.m` file.

Table 2.5 shows a list of applications where we use LabView or Matlab, and why we decided to use that one.

	Experiment	**Reason**
Matlab	Water Tanks - Self-Triggered	In this experiment, we have decided to use Matlab with the GUI for three reasons: 1. The scheduler was already implemented in Matlab 2. The experiment is set for 8 water tanks, and having automatics process for compiling, modeling, running and logging is simpler 3. We wanted to show the results in Matlab
	Water Tanks - Dynamic scheduler	Here we have the same case than before 1. The controller and scheduler are implemented in Matlab and they require and specific toolbox The experiment is set for 8 water tanks, and having automatics process for compiling, modelling, running and logging is simpler 2. We wanted to show the results in Matlab
LabView	Pendulum - Performance analysis	In this case, we do not control the pendulum with the mote attached to the computer. The mote is used for analysing the communication and performance of the controller. For this case we have both implementation but we have selected the LabView one due to: 1. LabView is better to plot and show results in real-time
	Wireless Crane	In this case, the LabView model was already provided in LabView.
	Home Smart Grid	This experiment requires a complex GUI and has a simple controller that could be implemented easily in LabView.

Table 2.5: List of experiments where we use LabView or Matlab

References

[1] Aitor Hernandez. Getting started with tinyos at the automatic control lab. Technical report, Royal Institute of Technology (KTH), July 2011.

[2] Aitor Hernandez, Faisal Altaf, and Jose Araujo. Wireless crane. event triggered control. Technical report, Royal Institute of Technology (KTH), July 2011.

[3] Aitor Hernandez, Joao Faria, and Jose Araujo. Wireless water tanks. diferent scenarios and controllers. Technical report, Royal Institute of Technology (KTH), July 2011.