KTH Electrical Engineering

Communication between PC and motes

in TinyOS

AITOR HERNÁNDEZ, DANIEL PÉREZ HUERTAS

Stockholm May 25, 2012

TRITA-EE 2011:XXX
Version: 0.1

Contents

Chapter 1

Introduction

In Wireless Sensor Network (WSN), 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 provides a better controllability by means of having one node acting as a manager communicating with the computer or simply, one node analyzing the status of the network,log data or analyze traffic in the network.

This document presents the tools and platforms that we are being used at KTH Automatic Control department.

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

Chapter 2

Platforms and Tools

In this chapter the tools that are being used in the control lab are presented, starting with the Serial-forwarder, and continuing with description of different ways to communicate between the Serial-forwarder and Matlab or LabView.

2.1 Serial-forwarder

For implementation of serial-forwarder's applications, the C-version will be used as it is simple,efficient, and powerful because of being implemented under C. Moreover, it is capable of being modified to fulfill 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 (Compilation) and install the latest version of Cygwin on the `C:\` drive using the following link:

`http://cygwin.com/install.html`

Cygwin is a collection of tools which provide a Linux console look and feel environment for Windows. Once Cygwin has been installed, download the following file:

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

and unzip all the executable files in the folder `C:\cygwin\bin\`.

2.1.1 Compilation

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

Application	Description
`motelist`	This command presents the list of motes that are connected to the computer via the USB ports.
`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, it is only required to open a TCP/IP connection with the created server and packets will be read easily. In case of packets error, `sf` discards the packet, which means that serial-forwarder assures 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 sends the packet passed as an argument
`seriallisten`	Instead of creating a server, and then using a client to read the packets, this application forwards 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 sends the message passes as an argument

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

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. A proper instruction for installing TinyOS on Ubuntu is available in the document "GettingStartedWithTinyOS" at `http://code.google.com/p/kth-wsn/downloads/list`

The applications need some of the headers files in the TinyOS `tos/` folder. The instructions could be found on the README file as well.

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, it is recommend to copy the executable files made in the `sf` folder, with the names (sf, sfsend, sflisten, serialsend, seriallisten, prettylisten) to a `bin/` folder like ∼`/bin` or `/usr/bin` and make sure that the folder is on the PATH environmental variable (`$ printenv PATH`). You may do this by copying the files into a folder already on the path or add your own folder to the path as described below.
   - Open the bashrc file entering the command `$ gedit .bashrc`
   - Copy the following line to the end of the file, save changes and close it `export PATH=$PATH:/path-of-the-bin-directory`
   - Now, close the previous terminal windows. Open a new terminal, go to your home folder (`$ cd`) and check if you could run the `$ sf` from there. If not, verify the previous steps.

2.1.2 Usages

After the compilation and set-up of the applications needed to communicate with the motes, it is time to know how to use each of them.

2.1.2.1 Input parameters

The input parameters of the applications are described in the following.

- `<port>`

  Communication between PC and motes is based on the TCP/IP protocol. When a TCP/IP connection is opened a suitable port that is not being used by any other device should be chosen. Normally we will choose the port `9002` if it is free, and move to the `9003`, `9004`, and so, if more ports are needed. A different port will be necessary for each mote, connected to the computer.

- `<device>`

  To communicate with the motes that are connected to a computer via USB it is necessary to know the device name that the computer gives to the motes. The device name does not depend on the mote or the physical USB port where they are connected, therefore it becomes necessary to use the command `motelist`. Figure 2.1 presents the result of the command `motelist` in Linux

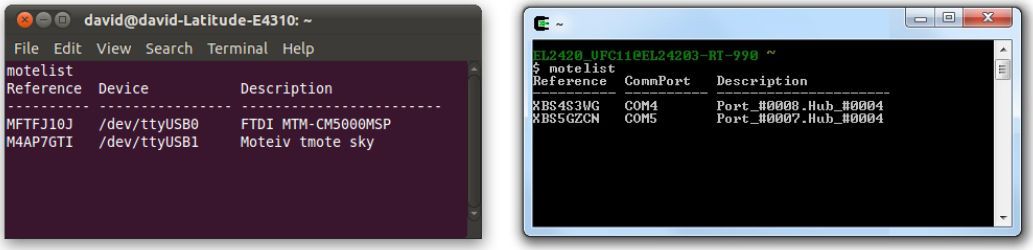


Figure 2.1: `motelist` command in the Linux console (left) and Cygwin console (right) when two motes are connected to the computer.

and Cygwin console when there are two motes connected to the computer. As can be seen, the device names of the connected motes are `/dev/ttyUSB0` and `/dev/ttyUSB1` in the Linux case and `COM4` and `COM5` in the Windows case. In general the devices are always called `/dev/ttyUSBX` in Linux and `COMX` in Windows, where `X` is an integer number.

- `<rate>` The rate refers to the baudrate of the mote connected to the computer, which can be found in the manual or datasheet of the device. For instance in the TelosB or tmote the baudrate is `115200`.

- `<host>` The host to which the motes are connected is the local host, for which the word `localhost` is used.

Using the introduced input parameters, the functions for each application and some examples are presented in the following.

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`

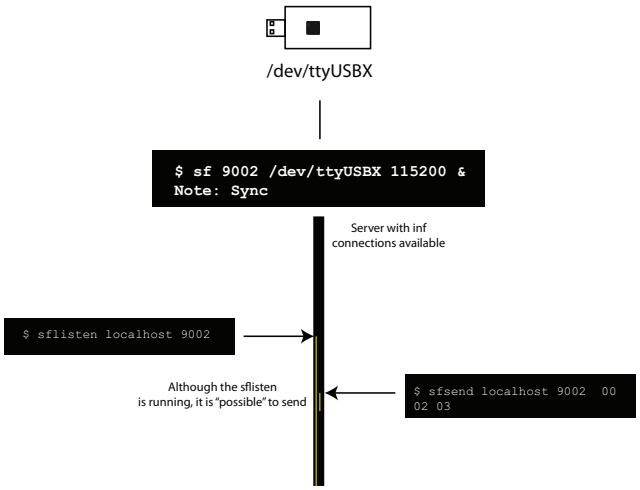


Figure 2.2: Example of using the `sf` server

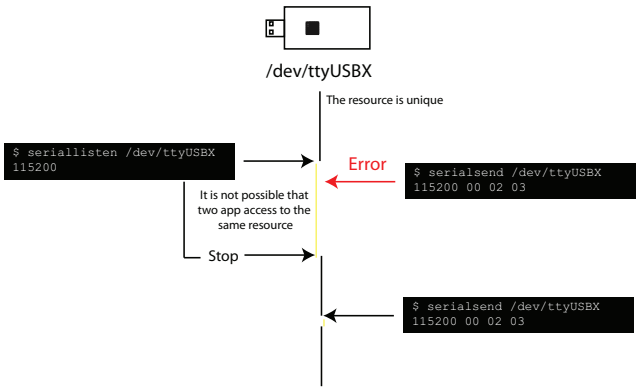


Figure 2.3: Example of using the `serial*` tools

The above figures depict how one can use different ways to do the same task by means of the Serial-Forwarder package utilities. Both examples show a way to listen and send packets from/to a mote.

In Figure 2.2 one mote is selected to run the `sf`. Then `sflsiten` is called to listen to the serial packets received. With the `sflsiten` running and receiving messages, we try to send a packet using `sfsend`. If at the moment we execute `sfsend`, the serial port is free, the packet will be sent to the mote. In this scenario multiple connections could be established using the `sflisten` without any problem, like logging the raw data being received from the motes, and running LabView at the same time.

Figure 2.3 shows an example where the target is to read/write from/to the mote without using the `sf` server. To open a new listener (`seriallisten`), a connection with the serial resource should be opened. So, any try to send and receive at the same time results in error. Thus, there is no possibility to use more than one listener in this case.

2.1.3 Exercises

In the previous Section,different commands and their usage were discussed. In this section, a number of exercises are proposed to play with different tools.

2.1.3.1 Questions

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

You can find the applications which send a custom message periodically in the folder `CommPC/MotesFiles/TestWriteReadSerialComm`, and those which send `printf` messages in the folder `CommPC/MotesFiles/TestPrintf`. Check the header files (with extension `.h`) for more details of the message structure.

Using the above applications and the Serial-Forwarder suite, answer the following 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 2.2 and 2.3. But try to answer yourself.

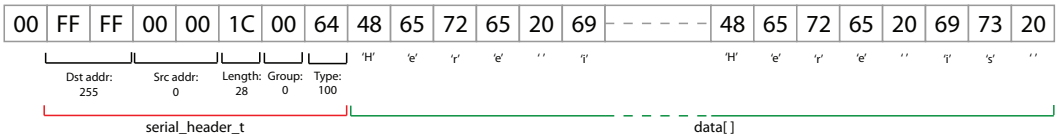
⇒ **Modification of the `seriallisten` or `sflisten`** Doing the exercise *`printf` packets vs. other packets* can help you understand the differences between these packets. In this exercise, the purpose is to create a C version of *PrintfClient*.

To simplify the compilation method, it is recommended to create a backup of the `sflisten`, and modify the `sflisten` application directly.

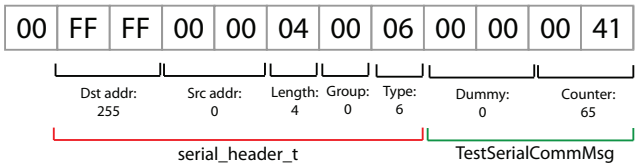
The solution needs to have less than five more lines compared to the current version.

2.1.3.2 Solutions

⇒ **`printf` packets vs. other packets** We usually do not distinguish between a `printf` and a normal packet transmitted through Universal Serial Bus (USB). A `printf` packet is just an special packet transmitted through USB, and contains a certain number of characters on its payload.



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



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

Figure 2.4: 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 2.4 shows the structure of two different packets. In Figure 2.4a the structure of a `printf` is depicted. `printf` sends packets of 28 bytes with the characters that it has in the buffer. With every call of `printf`, characters are added to the buffer. Thus,the same packet can contain parts of two different `printf` commands, as shown in the figure.The buffer contents are displayed by execution of `printfflush` command.

Figure 2.4b shows an example of transmitting a customized message, the `TestSerialCommMsg`. The message contains two fields, two `nx_uint16_t` (Linux platform data type equivalent to integer) numbers. It is important that when we create this kind of structures the types are (`nx_`), the external structures defined by TinyOS to ensure cross-platform compatibility. For more details in this regard refer to `http://www.tinyos.net/tinyos-2.1.0/doc/html/tep111.html`.

In spite of using a lot of computational time and resources, `printf` is useful for debugging purposes, as it enables one to view the value of the variables at different places in the code.

⇒ **`sflisten` vs. `seriallisten`** The difference between `sflisten` and `seriallisten` is explained in Figures 2.2 and 2.3.

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

⇒ **Modification of the `sflisten` or `seriallisten`** As seen in the first exercise, `printf` sends messages, in which every byte of the payload is one character. So, to convert `sflisten` into `sfprintf`, it is only required 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, an easy and direct way to establish communication between a mote and a PC by means of LabView is presented. This method makes it possible to read the packets sent from the mote to the PC in LabView and also to send packets from LabView to the mote.

The first step is to open an `sf` server using the `sf` command in Linux or Cygwin console as explained before:

```
sf <port> <device> <rate>
```

Normally the `<port>` is `9002` while `9003`, `9004`, and so are used if more motes are connected to the PC. It is necessary to open as many `sf` servers in different consoles as the motes connected to the PC and keep the console windows open. Once a `sf` server is opened, the four following steps should be done in LabView for establishing communication with the motes. The steps are:

- Opening a TCP/IP connection between LabView and the `sf` server (Only at the beginning of the program).
- Reading a packet in LabView from the mote.
- Writing a packet from LabView to the mote.
- Closing the TCP/IP connection between LabView and the `sf` server (Only at the end of the program).

In order to simplify the implementation of the above steps, it is recommended to use *SubVi*s.

2.2.1 Establishment of communication using *SubVi*s

This subsection discusses the basics of using *SubVi*s for establishment of communication with the motes.

2.2.1.1 Opening a TCP/IP connection

The basic schematic of this *SubVi* is shown in the Figure 2.5. First a TCP/IP connection should be opened with `localhost` while the port to which the mote is connected is normally chosen to be `9002` when starting the `sf`. After opening the TCP/IP connection one should connect the LabView with the `sf` by means of a "handshake", which consists of writing two bytes with the decimal values, `85 32` into the TCP/IP connection and then reading another 2 bytes. If the connection with the `sf` was successful, a message will be shown in the Linux or Cygwin console indicating that the `sf` server has a new client.

The output of the *SubVi* is `connection ID`, which will be used in the other *SubVi*s to communicate with the connected mote.

2.2.1.2 Reading a message

This *SubVi*'s schematic is presented in the Figure 2.6. To read a message one needs to start by reading the first byte, which shows the length of the message (in bytes); by, knowing the length of the payload we can also separate the header from the payload.

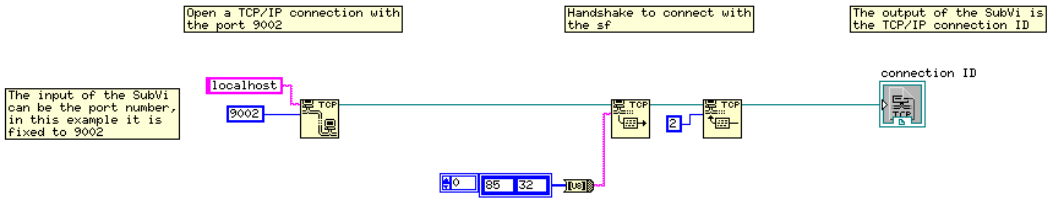


Figure 2.5: SubVi to open a TCP/IP connection between LabView and the `sf`.

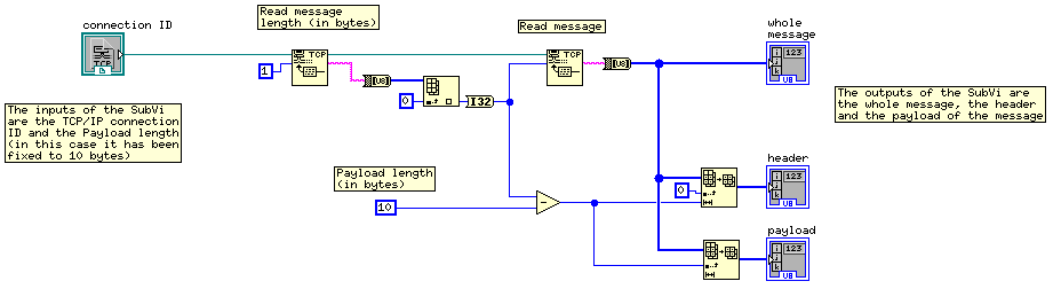


Figure 2.6: SubVi to read from the mote in LabView.

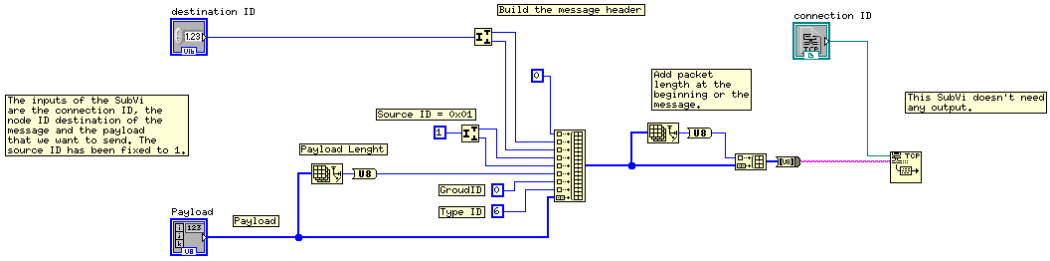


Figure 2.7: SubVi to send a message from the LabView to a mote.

The output of this *SubVi* may be the whole message or the payload; while in both cases it will be a one-dimensional array of `UINT8` from which the desired part of the message can be extracted depending on the purpose of the application.

2.2.1.3 Writing a message

In this case, the inputs of the *SubVi* are the connection ID and the data to be sent (payload), which is stored in a one-dimensional array of `UINT8`. Other optional inputs may be the destination ID, the source ID, etc.

Before sending the payload to the mote the header of the message should be built and added to the beginning of the payload as shown in the Figure 2.7. The final message (array of `UINT8`) has to include the following fields:

- `UINT8` Length of the whole message.
- `UINT8` 0x00. (To indicate the beginning of the message).

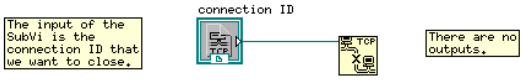


Figure 2.8: SubVi to close the TCP/IP connection between LabView and `sf`.

- `UINT16` Destination ID (node ID of the destination). It has to be separated into two bytes as shown in Figure 2.7. To broadcast the message use `0xFFFF`.
- `UINT16` Source ID (ID of the node connected to the computer). Also it should be split into two bytes to be put it into the `UINT8` array.
- `UINT8` Payload length.
- `UINT8` Ground ID (Usually 0x00).
- `UINT8` Type ID (Usually 0x06).
- `x*UINT8` Payload.

In TinyOS-2.x, the length of payload is limited to 28 bytes.

2.2.1.4 Closing a connection

A TCP/IP connection with `sf` can be closed by means of a simple close connection block, which is depicted in Figure 2.8.

2.2.2 Main Read/Write program example

This subsection presents an example of a main program in LabView that uses the previously explained *SubVi*s; *OpenSF*, *ReadSF*, *WriteSF* and *CloseSF* blocks. As it is shown in Figure 2.9, firstly one has to open a connection with `sf`; then there is a ”while” loop that reads from the motes and writes to the motes, and finally the connection is closed when the loop is stopped.

One of the most common problems in a LabView program occurs when one tries to read and write to the motes at the same time. To prevent this, one can add a "Flat sequence” (see Figure) or connect the blocks one after another.

If there is more than one mote connected to the computer, then it will be necessary to include another four *SubVi*s to open the connection with the other port (presumably `9003`), read, write and close the connection. Furthermore, if all of the motes are using the same radio channel a delay must be included between the packets being sent.

2.2.3 Control example using a MATLAB script

MATLAB is one of the most common programs used in control engineering. So, it is interesting to interconnect the control programs that are implemented in MATLAB with the main LabView program to control real processes in real time (hardware in the loop). Figure 2.10 shows a simple way to do so.

Any MATLAB function or code that one wants to implement for control algorithms, can be written in a MATLAB script. And to use a previously created m-file its path should

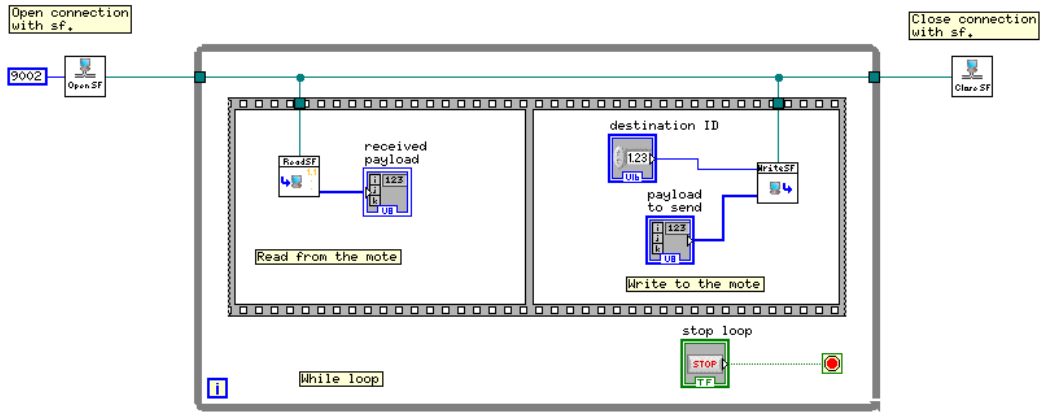


Figure 2.9: Example of a main program in LabView that uses the previous SubVis to interact with the connected motes.

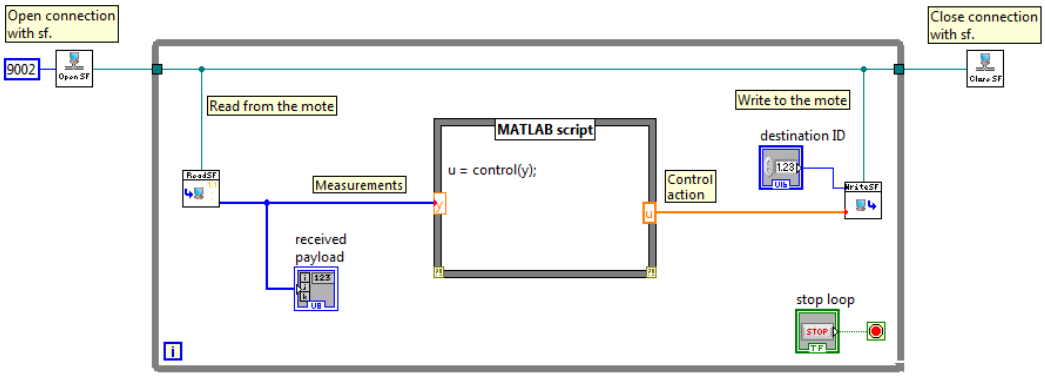


Figure 2.10: Example of a control program in LabView that uses the previous SubVi's and a MATLAB script to calculate the control actions.

be added to the directory path of MATLAB. This can be done by going to the MATLAB command window which is automatically opened when the MATLAB script is put in the LabView program, and using the function `addpath`.

For instance, if the `control.m` file is in the directory `c:/programs` we have to write

```
addpath c:/programs
```