Contents

Chapter 1

Introduction

This is a c++ implementation of the SCIP 2.0 protocol for Hokuyo products. The current version implements only those commands supported by the UTM-30LX model.

Chapter 2

Setup

1. Make sure the driver is installed.
2. Connect the laser to a 12v DC power source. Typical current consumption goes from 0.8A to 1A. **WARNING:** Inverting the polarity might cause severe damage to the device, consult the user guide before powering the laser.
3. Connect the laser's USB cable to your computer. The device will report itself as a virtual serial port (COMx in Windows, /dev/ttyACMx in linux).

2.1 Building with UDG_Hokuyo

This guide uses Microsoft Visual Studio Express for Desktop 2012 and Windows 8.1 or g++ 4.7 and Ubuntu 13.10. Other compiler/OS/Distribution combnations might work as well but haven't been tested.

2.1.1 Windows

1. Copy the UDG_Hokuyo release folder to a suitable location.
2. Open your Visual Studio solution.
3. As an optional step, you might want to add UDG_Hokuyo include (<Release_Folder >\include) and library (<Release_Folder>\include\bin\<Architecture>) directories to your default search folders. To do so, for the include folder go to Project\<Name>Properties\Configuration Properties\C/C++\General and enter the appropriate path in Additional Include Directories. Similarly, for the binary folder edit the Additional library directories property in Project\<Name>Properties\Configuration Properties\Linker\General.
4. Your source files should use #include <UDG_Hokuyo.h>or #include "UDG_Hokuyo.h"[1]
5. Make sure you're linking with UDG_Hokuyo.lib. To do so add UDG_Hokuyo.lib

[1]Remember that, when using quotes, If the header file is not on your current directory the full path should be included i.e. #include "c:\foo\bar\UDG_Hokuyo.h".

to Additional Dependencies in Project\<Name>Properties\Configuration Properties\Linker\Input or add #pragma comment(lib,“UDG_Hokuyo.lib”) to your code. [2]

2.1.2 Linux (Ubuntu)

1. Copy the UDG_Hokuyo release folder to a suitable location.
2. As an optional step you might want to place the header and library files into your default search directories. Typically you’ll need to copy <Release_Folder>/include/UDG_Hokuy
to /usr/include/ and <Release_Folder>/bin/<Architecture>/libUDG_Hokuyo.a to /usr/lib.[3]
3. Your source files should use #include <UDG_Hokuyo.h>or #include “UDG_Hokuyo.h”[4]
4. To build your application use -std=c++11 [5] and link with -lUDG_Hokuyo.a [6]

[2]Use the whole path if you skipped step 3.

[3]These default libraries might vary depending on your compiler settings.

[4]Remember that, when using quotes, If the header file is not on your current directory the full path should be included or you can also use the -I option (see g++ help).

[5]-std=c++0x might work as well if you’re using an older g++ version

[6]Or simply UDG_Hokuyo.a if you skipped step 2. If the file is not on your current folder you can use its whole path or the -L option (see g++ help).

Chapter 3

The UDG_Hokuyo API

3.1 LaserData class

Public members

int firstStep

The first of the 1080 steps of the motor containing laser data. This is also the first position of the buffer (*int data[]* that contains useful data. To obtain the actual angle, multiply be 0.25.

int lastStep

The last of the 1080 steps of the motor containing laser data. This is also the last position of the buffer (*int data[]* that contains useful data. To obtain the actual angle, multiply be 0.25.

HokuyoEncoding encoding

Defines the type of encoding to be used for measurements. Posible values are:

HokuyoEncoding.TWO_CHARACTER: Uses 12 bits to represent measurements, that is, from 0 to 4095mm. Further distances are capped to the max value.

HokuyoEncoding.THREE_CHARACTER: Uses 18 bits to represent measurements, that is, from 0 to 262143mm (although 30000 is the max range for the Hokuyo UTM-30LX).

LaserData()

Default constructor for the class.

bool SetData(unsigned char* buffer)

Decodes the raw information from a laser reading contained in *buffer* and stores it in *LaserData::data.*
unsigned char* buffer: A byte buffer containing the raw data from a laser reading.

Return value: true if the operation succeeds, false if it doesn't.

int GetReadingsCount()

Gets the number of steps that where actually read in the last update.

Return value: returns *LaserData::lastStep - LaseData::firstStep +1*

3.1.1 Private members

bool CharToNumber(unsigned char* string,int nDigits,int* result)

Converts an ascii string representing a decimal number stored at *string* with *nDigits* digits and stores it in *result.*

3.2 The Hokuyo class

3.2.1 Public members

HANDLE serialPortDescriptor

The device descriptor associated to the device,that is, a COM port in Windows or /dev/ttyACMx in Linux.

Hokuyo()

The default constructor for the class, initializes *Hokuyo::serialPortDescriptor* to NULL.

Hokuyo(HANDLE portDescriptor)

Constructor for the class, initializes *Hokuyo::serialPortDescriptor portDescriptor.*

Hokuyo()

Destructor method for the class.

bool GetData(HokuyoEncoding encoding, unsigned short startingStep, unsigned short endStep, unsigned short clusterCount,unsigned short scanInterval, unsigned short nScans, unsigned char* response)

This operation is described as "MDMS-Command" in the documentation for the SCIP 2.0. protocol. Gets distance measurements in the selected range.
HokuyoEncoding encoding: Indicates the encoding format of the data and thus, its max range (see 3.1).
unsigned short startingStep: The first measurement step to read with th laser (between 0 and 1080 for the UTM-30LX model).
unsigned short endStep: The last measurement step to read with th laser (between 0 and 1080 for the UTM-30LX model).
unsigned short clusterCount: The Hokuyo allows the user to use clustering, that is, a group of 0 to 99 measurement steps that are interpreted as one. The minor distance amongst them is the one returned. For more info, check the SCIP 2.0 protocol documentation.
unsigned short scanInterval: The number of scans to be skipped. See the SCIP 2.0 documentation for more info.
unsigned short nScans: The number of scans to perform. **unsigned char* response:** A byte buffer to store the response.

Return Value: true if it succeeds, false if it doesn't.

bool SwitchLaserOn(unsigned char* response)

This section is referred to as 'BM-Command' in the SCIP 2.0 documentation. Turns the laser on.
unsigned char* buffer : a byte buffer containng the raw response from the instruction. this is useful to diagnose and correct errors.

Return Value: true if it succeeds, false if it doesn't.

bool SwitchLaserOff(unsigned char* response)

This section is referred to as 'QT-Command' in the SCIP 2.0 documentation. Turns the laser off.
unsigned char* buffer : a byte buffer containng the raw response from the instruction. this is useful to diagnose and correct errors.

Return Value: true if it succeeds, false if it doesn't.

bool Reset(unsigned char* response);

This command, also known as 'RS-Command' will reset all the settings that were changed after sensor was switched on. This turns Laser off, sets motor speed and bit rate back to default as well as reset sensors internal timer.
unsigned char* buffer : a byte buffer containng the raw response from the instruction. this is useful to diagnose and correct errors.

Return Value: true if it succeeds, false if it doesn't.

bool GetVersionDetails(unsigned char* response)

Refered to as 'VV-Command', Transmits version details such as, serial number, firmware version etc on receiving this command.
unsigned char* buffer : a byte buffer containng the response from the instruction as a null terminated string.

Return Value: true if it succeeds, false if it doesn't.

bool GetSpecs(unsigned char* response

Refered to as 'PP-Command', makes the sensor transmit its specifications.
unsigned char* buffer : a byte buffer containng the response from the instruction as a null terminated string.

Return Value: true if it succeeds, false if it doesn't.

bool GetRunningState(unsigned char* response)

Also known as 'II-Command' makes the laser transmit its running state.
unsigned char* buffer : a byte buffer containng the response from the instruction as a null terminated string.

Return Value: true if it succeeds, false if it doesn't.

3.2.2 Private members

int readDelayMs

The time in milisecons to wait after a read.

bool WriteToSerialPort(unsigned char* buffer, int bytesToWrite)

Generic write funtion associated with Hokuyo::serialPortDescriptor.

int ReadFromSerialPort(unsigned char* buffer)

Generic read funtion associated with Hokuyo::serialPortDescriptor.

void NumberToChar(unsigned short number, int nCharacters,unsigned char string[])

Converts an unsigned short into its ascii representation as a sring with *nCharacters* characters.

Chapter 4

Code Samples

The code samples work for the Microsoft c++ compiler (cl, using Visual Studio Express 2012 for desktop) and g++ 3.8. Other compilers might work as well but haven't been tested yet. Windows examples assume an unicode build.

4.1 Opening a serial port descriptor

For both Linux and Windows implementations, you need to provide a file descriptor associated with a serial port. The following samples demonstrate how to code a function to do so.
textbfNOTE: The name of the serial port associated with your board varies from system to system. You'll notice that two virtual communication ports with contiguous numbering will register themselves when connecting the board. You need to use the one with the higher number. Typically this will be COMx for Windows and ttyACMx for linux.

4.1.1 Windows

```
HANDLE OpenSerialPort(LPWSTR name)
{
        HANDLE portDescriptor;
        DCB deviceControlBlock;
        portDescriptor = CreateFile(name,
                                GENERIC_READ|
                                    GENERIC_WRITE,
                                0,0,OPEN_EXISTING,0,0);
        if(portDescriptor == INVALID_HANDLE_VALUE)
        {
                printf("Failed opening %ls\n", name);
                return INVALID_HANDLE_VALUE;
        }

        FillMemory(&deviceControlBlock,sizeof(DCB),0);
        deviceControlBlock.DCBlength = sizeof(DCB);
        if(BuildCommDCB(L"57600,n,8,1",&
           deviceControlBlock))
        {
                printf("Successfully configured DCB!\n");

        }
        else
        {
                printf("Something went wrong while
                   building the DCB\n");
                return INVALID_HANDLE_VALUE;
        }

        if (!SetCommState(portDescriptor,&
           deviceControlBlock))
        {
                printf("Error configuring port\n");
                return INVALID_HANDLE_VALUE;
        }
        else
        {
                printf("Serial port ready\n");
        }

        return portDescriptor;
}
```

4.1.2 Linux

NOTE: In the linux implementation, HANDLE is defined as int.

```
int OpenSerialPort(const char* device)
{
        int fd = open(device, O_RDWR | O_NOCTTY);
        struct termios options;
  tcgetattr(fd, &options);
  options.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG |
     IEXTEN);
  options.c_oflag &= ~(ONLCR | OCRNL);
  tcsetattr(fd, TCSANOW, &options);
  return fd;
}
```

4.2 Typical applications

4.2.1 Get device info

The SCIP 2.0 protocol provides some commands to get information about the laser such as range, firmware version, running state, serial number, etc. Those functions will return a null terminated char buffer that can be directly printed using operations as printf.

```
#include <stdio.h>
#include "UDG_Hokuyo.h"
int main(int nargs, char* args[])
{
#ifdef __linux
char* portName = "/dev/ttyACM0";
#else
        wchar_t *portName = L"COM7";
#endif
        HANDLE port = OpenSerialPort(portName);


        Hokuyo laser(port);
        unsigned char returnBuffer[HOKUYO_BUFFER_SIZE];

        if(laser.GetRunningState(returnBuffer))
                printf("%s\n",returnBuffer);
        if(laser.GetSpecs(returnBuffer))
                printf("%s\n",returnBuffer);
        if(laser.GetVersionDetails(returnBuffer))
                printf("%s\n",returnBuffer);


}
```

4.2.2 Reading laser data

A series of operations must be followed in order to read and use depth information from the laser. First you'll need to initialize a Hokuyo instance. You need to provide the Hokuyo::GetData() with information about the encoding (see 3.1), start and ending steps for it to behave as expected. Once the depth data has been read, it'll be sotred as raw bytes in the buffer provided

to Hokuyo::GetData(). To translate this raw depth information yo can create a LaseData instance and call LaserData::GetData() with the raw buffer as an input parameter. If succesful, this function will intialize the LaserData instance to contain information about the start and end steps, number of measurements performed and an int array (LaserData::data) containing the actual measurements. Remember that each position of the array corresponds to a measurement step, so position 0 maps to step 0,position 1 to step 1 and so on. Each position outside your measurement range (that is, smaller than the start step or greater than the end step) will be set to 0.
This example shows how to get some measurements and print them to screen.

```
#include <stdio.h>
#include "UDG_Hokuyo.h"
int main(int nargs, char* args[])
{
#ifdef __linux
char* portName = "/dev/ttyACM0";
#else
        wchar_t *portName = L"COM7";
#endif
        HANDLE port = OpenSerialPort(portName);


        Hokuyo laser(port);
        unsigned char returnBuffer[HOKUYO_BUFFER_SIZE];

        laser.GetData(HokuyoEncoding::THREE_CHARACTER,
        HOKUYO_FIRST_MEASUREMENT_POINT,
        HOKUYO_LAST_MEASUREMENT_POINT-1,
        0,0,1,returnBuffer);
        LaserData data;
        data.SetData(returnBuffer);
        for(int i = data.firstStep; i <= data.lastStep;i
            ++)
                printf("%i ",data.data[i]);

        return 0;
}
```