The Sick LIDAR Matlab/C++ Toolbox: A Quick Start Guide

Jason C. Derenick and Thomas H. Miller

Lehigh University
Computer Science and Engineering
Bethlehem, PA 18015 USA

derenick@lehigh.edu

March 10, 2008

Contents

List of Tables

1 Introduction

The Sick LIDAR Matlab/C++ Toolbox is an open-source project aimed at providing stable and easy-to-use C++ drivers for Sick LMS 2xx and Sick LD laser range finders. It is released under a flexible open–source license (see Section 10). In addition to low-level drivers, the package also features an easy to use Matlab mex interface, which allows the end-user to stream real-time range and reflectivity data directly into Matlab. This feature is especially attractive as it facilitates the rapid development of algorithms by exploiting the high-level functionality afforded by Matlab's vector-based operations. The toolbox is branched from the source code used by the Ben Franklin Racing Team, whose car – Little Ben – was one of only six vehicles to successfully complete the 2007 DARPA Urban Grand Challenge.

This document covers the basics to get you quickly up and running using the most common features of the toolbox. In particular, this document will guide you through the installation process as well as illustrate how to acquire data via the Matlab/C++ driver interfaces. Accordingly, C++ and Matlab code examples are presented to illustrate its use. It should also be noted that this document is by no means a comprehensive manual over all features of the toolbox. It is meant as a jumping off point for further use and to get the application developer/end–user up and running as quickly as possible. The user should be familiar with the operation and telegram manuals for both Sick LMS 2xx and Sick LD units. This is especially true when it comes to configuring the devices.

2 Compatible Devices

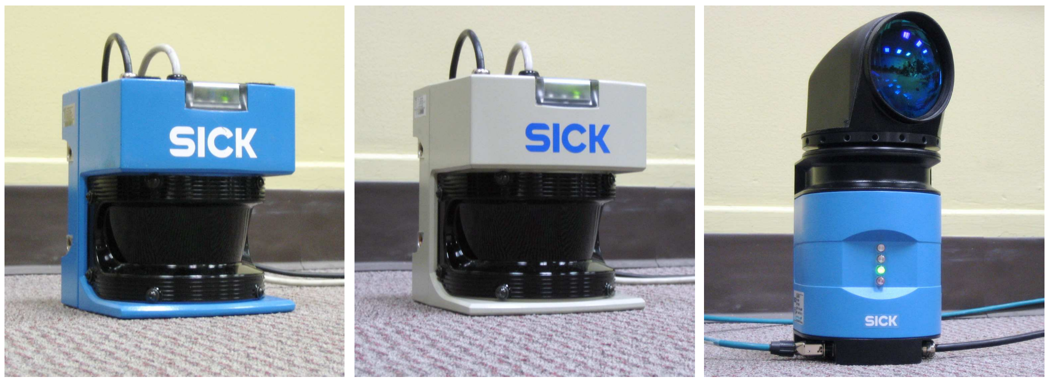


Figure 1: The Sick LIDAR Matlab/C++ Toolbox supports both the LMS 2xx and LD families of Sick LIDARs. Pictured are a few of the Sick LIDARs used to explicitly test the toolbox: (Left) A Sick LMS 200–30106 (Center) A Sick LMS 291–S05. (Right) A Sick LD–LRS 1000.

2.1 Compatible Sick LMS 2xx Units

The communication protocol implemented with the low-level Sick LMS 2xx C++ driver is compatible with LMS 200/220 firmware version: V02.30 Q501, LMS 211/221/291 firmware version: X01.27 Q501, and LMS 211/221–S19/–S20 firmware version S01.31 Q393 [1]. The code has been explicitly tested with the following units: Sick LMS 200–30106, Sick LMS 291–S05, and Sick LMS 291–S14.

2.2 Compatible Sick LD Units

The communication protocol implemented with the low–level Sick LD C++ driver is compatible with LD–LRS 1000/2100/3100 models [2]. The code has been tested explicitly with Sick LD–LRS 1000 units.

3 Tested Distibutions

3.1 Linux

The Sick LIDAR Matlab/C++ Toolbox was explicitly tested with the following Linux distributions:

- Ubuntu 6 (Feisty Fawn, i386) and 7 (Gutsy Gibbon, i386)
- Debian (amd64)
- Fedora 8 (i386)

3.2 Matlab

The toolbox was also explicitly tested against the following Matlab distributions:

- R14
- R2007a
- R2007b

4 Architectural Overview

Figure 2 shows a high–level schematic of the Sick LIDAR Matlab/C++ Toolbox. At the highest level, the toolbox is comprised of a set of Matlab mex files that directly interact with a corresponding low–level C++ driver. All drivers are comprised of two fundamental components that are derived from the shared base code over which each is built. In particular, each driver features an interface to acquire data and configure the device as well as a buffer monitor thread responsible for ensuring only the most recent scan information is returned via said interface. As expected, the interface is used to send commands to the

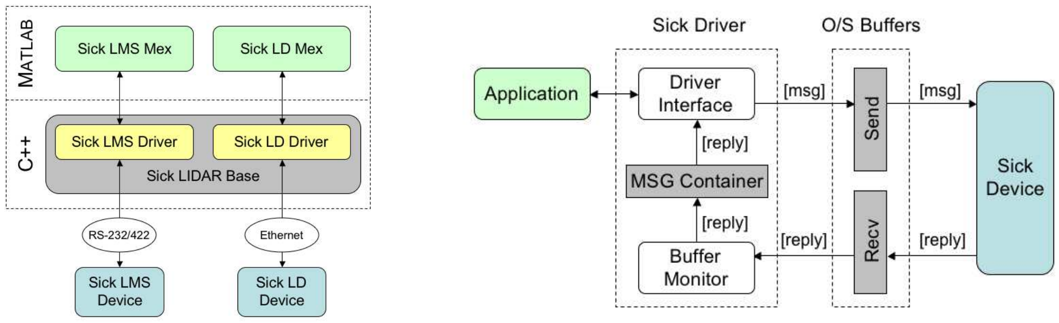


Figure 2: (Left) The high–level schematic for the Sick LIDAR Matlab/C++ Toolbox. High–level mex functions directly interact with low–level C++ drivers that share a common base framework. (Right) Schematic of a typical application using the interface provided by the low–level driver. Each driver features a buffer monitor thread that is responsible for ensuring the receive buffer never becomes saturated. The most recent Sick LMS reply is buffered in a message container where the interface can access it.

device. All replies are handled by the buffer monitor where they are encapsulated as a message object before being stored in the container where the interface can acquire them.

It is important to note that although the Sick LMS 2xx driver is RS–232/422 and the Sick LD driver is Ethernet, both share a high–level of commonality that allows them to exploit the base framework. In this case, the base code simply provides a primitive a definition of a Sick LIDAR driver over which more specialized drivers can be built. It is especially useful as it promotes modularity and emphasizes code reuse where possible.

5 Software Installation

As the toolbox is written to exploit the portability afforded by GNU Autotools, its installation is a rather straightforward process.

5.1 Building and Installing C++ Drivers

Notational Convention: For the sake of generality, we denote the Sick LIDAR Toolbox project root directory as `sicktoolbox-x` where `x` denotes the version number of the toolbox. In this document, (e.g. in example calls and directory locations) be sure to replace `x` with the correct version number. For example, if the version is 1.0 then replace references to `sicktoolbox-x` with `sicktoolbox-1.0`.

To install the C++ drivers libraries, simply change to the project's root directory (i.e. sicktoolbox-x) and do the standard three–step installation procedure for any GNU Autotools project. In particular, do the following from your command prompt:

```
    ./configure
    make
    sudo make install
```

The configure script will auto-generate a C/C++ header listing the current configuration of the platform. Executing `make` will build the Sick driver libraries and `make install` will copy the libs to `/usr/local/lib` and the headers to `/usr/local/include`. Since `make install` requires writing to protected directories, this step must be done as an administrator/root. If your Linux distribution does not support the `sudo` command or your user is not listed in the `sudoers` file then you can simply use `su` to switch to the superuser before running `make install`.

This process will install two separate libraries built as dynamically linked *shared objects*. As a result, it may be necessary to update your dynamic linker's cache if it reports that it can't resolve `libsicklms-x.so` or `libsickld-x.so`. For instance, assuming you are using the GNU `ld` linker, you might encounter the following error after running a program for the first time after installation:

```
lms_config: error while loading shared libraries: libsicklms-1.0.so.0:
cannot open shared object file: No such file or directory
```

In this case, the error occurred after running `lms_config` for the first time. To remedy this, run the following from your command line to update your linker's cache:

```
    ldconfig
```

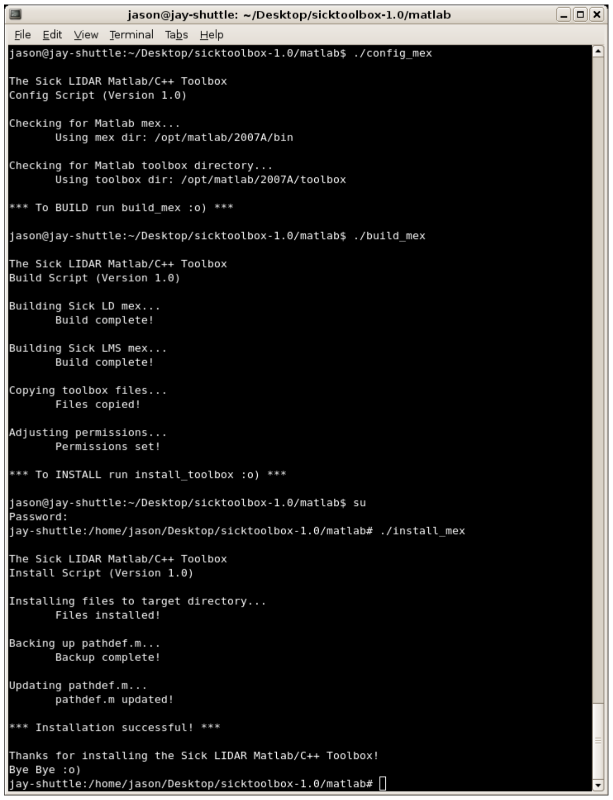
This should only ever have to be done once after the initial installation – once the cache is updated, all programs referencing the libraries will now be properly linked at run-time.

5.2 Building and Installing Mex Interface

Building the mex interface(s) requires an equally straightforward process. Assuming you are in the project's root directory (`sicktoolbox-x`) and have successfully built and installed the C++ interfaces (see Section 5.1), execute the following from your command line:

```
    cd matlab
    ./config_mex
    ./build_mex
    sudo ./install_mex
```

The three bash scripts (i.e. `config_mex`, `build_mex`, and `install_mex`) are used to respectively configure, build, and install the mex interfaces to the appropriate Matlab directory. In addition to the mex binaries, these scripts also modify certain files in your Matlab installation directory. Figure 3 illustrates the process of installing the Matlab me interfaces


jason@jay-shuttle: ~/Desktop/sicktoolbox-1.0/matlab
File Edit View Terminal Tabs Help

```
jason@jay-shuttle:~/Desktop/sicktoolbox-1.0/matlab$ ./config_mex

The Sick LIDAR Matlab/C++ Toolbox
Config Script (Version 1.0)

Checking for Matlab mex...
       Using mex dir: /opt/matlab/2007A/bin

Checking for Matlab toolbox directory...
       Using toolbox dir: /opt/matlab/2007A/toolbox

*** To BUILD run build_mex :o) ***

jason@jay-shuttle:~/Desktop/sicktoolbox-1.0/matlab$ ./build_mex

The Sick LIDAR Matlab/C++ Toolbox
Build Script (Version 1.0)

Building Sick LD mex...
       Build complete!

Building Sick LMS mex...
       Build complete!

Copying toolbox files...
       Files copied!

Adjusting permissions...
       Permissions set!

*** To INSTALL run install_toolbox :o) ***

jason@jay-shuttle:~/Desktop/sicktoolbox-1.0/matlab$ su
Password:
jay-shuttle:/home/jason/Desktop/sicktoolbox-1.0/matlab# ./install_mex

The Sick LIDAR Matlab/C++ Toolbox
Install Script (Version 1.0)

Installing files to target directory...
       Files installed!

Backing up pathdef.m...
       Backup complete!

Updating pathdef.m...
       pathdef.m updated!

*** Installation successful! ***

Thanks for installing the Sick LIDAR Matlab/C++ Toolbox!
Bye Bye :o)
jay-shuttle:/home/jason/Desktop/sicktoolbox-1.0/matlab# 
```

Figure 3: Installation example for the Sick LIDAR Matlab/C++ Toolbox mex interfaces. The steps shown correspond to configuring, building and installing the mex interfaces via the bash scripts found in the `sicktoolbox-x/matlab/` directory. In this case, `./configure` was run beforehand during the installation of the C++ libraries so it is not included as one of the shown steps.

via the these three bash scripts. In this case, the current directory was already set at `sicktoolbox-x/matlab` so the first step is omitted.

In this version of the toolbox, this installation is kept separate from the standard C++ library installation for the sake of simplicity in the GNU Autotools scripts. In a future release, we aim to integrate the entire process into a single build process.

If you want to build only one mex interface, you can specify it by passing it as an argument to `./build_mex`. For instance, to build only the Sick LMS 2xx interface, you would call `build_mex` as follows:

```
./build_mex lms
```

Similarly, you can call it as `build_mex ld` to build only the Sick LD mex wrapper. Figure 3 shows the steps being executed in a terminal under a Debian installation. In this case, `./configure` was run beforehand during the installation of the C++ libraries so it is not included as one of the shown steps.

Attention Ubuntu Users: If you plan to use a USB–COMi–M adapter under Ubuntu Feisty Fawn or newer, be sure to uninstall `brltty` and `brltty-X11` (using synaptic is the easiest way). Otherwise, you will not see a `/dev/USBx`. By default, they grab the USB serial device, preventing it from showing up under `/dev/`. Once uninstalled, reboot your machine and reconnect your USB–COMi–M adapter. You will now see the `/dev/ttyUSBx` device path associated with your Sick LMS unit.

6 Interfacing the Sick LMS 2xx

6.1 Sick LMS C++ Driver Features

The Sick LMS 2xx C++ driver provides a concise interface for conveniently acquiring measurements and configuring the unit. Following are some of the available features:

Multi-threaded Implementation: The Sick LMS 2xx C++ driver utilizes the pthreads library. While the main thread is running, the driver thread (in particular, the "buffer monitor") is busy ensuring that only the most recent complete scan is buffered for return. As threading is integrated into the driver framework, there is no need to worry about buffer management. By designing the driver this way, we alleviate as much overhead for the application developer as possible, by allowing him/her to focus on data–processing as opposed to data–management.

Supports a Variety of Measurement Modes: The Sick LMS 2xx provides a simple interface for acquiring measurements (whether they be range or reflectivity). Among other things, the driver supports streaming high–resolution partial scans as well as mean–measured values. Additionally, it supports specifying a measurement subrange in the event that the application is only interested in a subset of the data. This latter feature is especially useful to limit bandwidth consumption over low–speed connections. On standard LMS models (not LMS Fast), the driver allows the user to request a reflectivity stream instead of a range stream. For LMS Fast models, the driver supports the streaming of both range and reflectivity at full–data rate.

500Kbps Support via USB–COMi–M: In addition to supporting the standard baud rates (9600bps, 19200bps, and 38400bps), the driver also supports streaming data at a rate of 75Hz (500Kbps) from the Sick LMS 2xx. In order to achieve this data rate, a USB–COMi–M USB to RS–232/422/485 industrial adapter is required. Using this adapter (setup appropriately for RS–422) in conjunction with the driver allows data acquisition from the Sick at the fastest rate possible. This approach provides a highly cost effective way to enable high–speed communication without having to invest in a proprietary Sick RS–422 adapter.

For additional details, please see our tutorial on enabling RS–422 via the USB–COMi–M. It is included in the toolbox and can be found in the sicktoolbox–x/manuals directory. It provides a simple tutorial for setting up your own high–speed configuration.

Baud Rate Autodetection: The Sick LMS 2xx C++ driver also employs an intelligent initialization that will autodetect the Sick's baud rate. This feature is particularly nice as it allows the application developer to avoid having to always specify the *known current* baud rate of the device. For instance, suppose the device is operating at 500Kbps and then the application is suddenly killed without having uninitialized the device. As a result, the device is still operating at 500Kbps. Upon running the application again, the driver will detect that the baud rates are not synchronized and will then select the correct baud based upon the results obtained from a sequence of "pings" it sends to the device. Once the proper baud is established, initialization will then commence.

Advanced Configuration: The Sick LMS 2xx C++ driver also allows for the configuration of such device parameters as: measurement mode, sensitivity level, availability mode, and measuring units. To help make the configuration process easier, an example application called `lms_config` is included. It provides an interactive shell–like interface for easily setting these EEPROM parameters.

Easily Set Device Variant: The driver also allows the application developer to easily set the device variant. In particular, he/she can request the device to operate at a field–of–view of either 180° or 100° with a scan resolution of either 0.50° or 0.25°. See the examples for details on how this can be easily done.

6.2 Using the Sick LMS C++ Driver

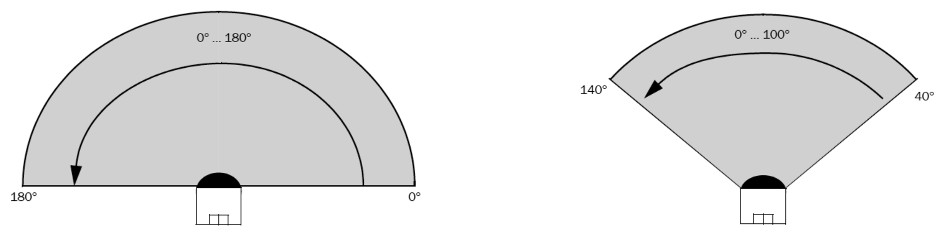


Figure 4: The Sick LMS 2xx family returns measurements as polar coordinates – scanning counter–clockwise. (Left) A Sick LMS 2xx scan using 180° field–of–view. (Right) A Sick LMS 2xx scan using 100° field–of–view. Both (Left) and (Right) are taken from the Sick LMS 2xx telegram listing [1].

6.2.1 Coordinate System

The Sick LMS 2xx LIDAR family returns measurements which can be interpreted as polar coordinates. The device scans "counter–clockwise" as shown in Figure 4.

6.2.2 Program Structure

The Sick LMS 2xx driver is written so as to provide easy data acquisition from LMS laser range finder units. In general, a program using the driver will follow the simple quaternary state machine given in Figure 5. Thus, a C++ program utilizing the driver to acquire data will perform three basic steps: initialize the device, acquire measurements and uninitialize the device. The driver source was designed to exploit the capabilities of C++ exception handling. *As a result, all calls to methods should be surrounded by try/catch statements.*

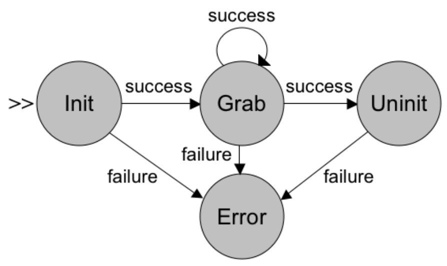


Figure 5: A quaternary state machine representing the simplest behavior of a C++ program utilizing the Sick LMS 2xx driver.

The following program shows a simple application of the driver to acquire measurements. Notice that it implements the state machine in Figure 5.

```
#include <iostream>
#include <sicklms-1.0/SickLMS.hh>

using namespace std;
using namespace SickToolbox;

int main(int argc, char *argv[]) {

  /* Specify device path and baud */
  string dev_path = "/dev/ttyUSB0";
  sick_lms_baud_t lms_baud = SickLMS::SICK_BAUD_38400;

  /* Define buffers for return values */
  unsigned int measurements[SickLMS::SICK_MAX_NUM_MEASUREMENTS] = {0};
  unsigned int num_measurements = 0;

  /* Instantiate the object */
  SickLMS sick_lms(dev_path);

  try {

    /* Initialize the device */
    sick_lms.Initialize(lms_baud);

    /* Grab some measurements */
    for(unsigned int i = 0; i < 10; i++) {
      sick_lms.GetSickScan(measurements,num_measurements);
      cout << "\t" << num_measurements << endl;
    }

    /* Uninitialize the device */
    sick_lms.Uninitialize();
  }

  catch(...) {
    cerr << "error" << endl;
    return -1;
  }

  return 0;
}
```

In this example, the three most important methods are used. Following the state machine paradigm, the program begins by calling `Initialize`. Before any data can be streamed this method must absolutely be called. Once initialization is complete, the device will then grab scans from the LIDAR by calling the *GetSickScan* method. As the receive buffer is constantly being monitored, the data returned from this call is guaranteed to be the most recent buffered scan. Once the program is done acquiring data, it then uninitializes the Sick LMS 2xx by calling *Uninitialize*, which returns the Sick LMS to a non-streaming state. It is important that each program properly initialize and uninitialize the device.

This example uses the static constant `SickLMS::SICK_MAX_NUM_MEASUREMENTS` to define its buffer size. It is provided for convenience in the SickLMS class definition. It is simply the maximum number of measurements determined by considering the maximum scan area and the highest scan resolution for the Sick LMS 2xx. It alleviates having to worry about computing buffer sizes for multiple configurations by ensuring that there is always enough storage allocated to handle the maximum number of measurements the device can return.

6.2.3 Properly Linking Programs

In order to utilize the Sick LMS 2xx driver, you must link against its associated library (e.g. `libsicklms-x`) as well as the POSIX thread library on your system (e.g. `-lpthread` or the `-pthread` flag). For instance, using `g++` (in this case version 4.1.2) on Ubuntu (as well as Debian), you can compile and link using the command:

```
g++ -o prog_name prog_name.cc -lsicklms-x -pthread
```

where `prog_name` should be replaced with the name of your program and `x` denotes the driver version number (e.g. `libsicklms-1.0`).

6.2.4 Example Programs

In addition to the C++ driver, the toolbox also comes with a variety of examples illustrating its use. All of the example projects with code can be found in the directory:

```
sicktoolbox-x/c++/examples/lms
```

In all, there are nine examples including everything from a fully-functional configuration utility to a program illstrating how to acquire high–resolution partial scans. More precisely, the following examples are provided:

- `lms_config` – A configuration utility for Sick LMS 2xx laser range finders. Allows setting: measuring units, measuring mode, availability, and more!
- `lms_mean_values` – Illustrates how to acquire mean value measurements from the Sick LMS 2xx.
- `lms_partial_scan` – Demonstrates how to acquire high–resolution partial scans from the Sick LMS 2xx.
- `lms_plot_values` (Requires `gnuplot`) – A simple program that plots range measurements using gnuplot_i++. Requires the gnuplot package to be installed.
- `lms_real_time_indices` – Illustrates how acquire real–time indices from the Sick LMS 2xx with range information.
- `lms_set_variant` (Not compatible with LMS Fast models) – Demonstrates how to properly set the Sick LMS 2xx variant (i.e. $180°/0.5°$ or $100°/0.25°$). LMS Fast models do not support the variant command.

- `lms_simple_app` – A simple program template for working with the Sick LMS 2xx C++ driver.
- `lms_stream_range_and_reflect` (Only compatible with LMS Fast models) – Illustrates how to acquire both range and reflectivity returns from LMS Fast models using the driver interface.
- `lms_subrange` – Shows how to acquire a measured value subrange from the Sick LMS.

Looking over these examples is the easiest way to quickly get yourself up and developing with the driver interface.

Running the C++ Example Programs: In addition to building the C++ driver interface, running `make` also builds each example in the example directory. If a dependency (e.g. `gnuplot`) for a certain example is not detected then the associated example is not built. The binary for each can be found in its respective `src` directory (e.g. for the `lms_config` example, look in `sicktoolbox-x/c++/examples/lms/lms_config/src`). To make things as easy as possible, all of the examples use the same command–line argument format. In particular, to call any example, simply invoke it from the command line:

```
./example_name DEVICE_PATH DESIRED_BAUD
```

where `DEVICE_PATH` denotes the associated device path for the Sick (e.g. `/dev/ttyUSB0`) and DESIRED_BAUD is the desired session baud rate. Valid values for the latter are: 9600, 19200, 38400, and 500000. For instance, to run the example `lms_simple_app` with your Sick LMS connected at `/dev/ttyUSB0` via a USB–COMi–M adapter, you would call it as follows:

```
./lms_simple_app /dev/ttyUSB0 500000
```

6.3 Using the Sick LMS Mex Interface

In a similar manner, the provided mex interface also follows the state machine given in Figure 5. Currently, the mex interface does not support high-resolution partial scans or mean measured values. Instead, it provides access to a data stream providing range and/or reflectivity information from the Sick LMS 2xx. The returned data depends upon the selected measuring mode of the device and/or the device type. For instance, if you are using an LMS Fast, the mex interface will return both range and reflectivity values.

6.3.1 Mex Commands

A call to the Sick LMS 2xx can be made by using the installed `sicklms` mex function as follows:

```
sicklms(CMD,ARGS)
```

where `CMD` is the command to issue to the device via the driver and `ARGS` denotes the argument list for the command. In particular, the interface supports the commands shown in Table 1. For example calls, see Table 2.

CMD	ARGS
`init`	`DEVICE_PATH`, `DESIRED_BAUD`
`variant`	`SCAN_ANG`, `SCAN_RES`
`grab`	N/A
`info`	N/A

Table 1: Argument list for commands supported by Sick LMS mex interface.

CMD	Example Call
`init`	`res = sicklms('init','/dev/ttyUSB0',9600);`
`variant`	`sicklms('variant',100,0.25);`
`grab`	`res = sicklms('grab');`
`info`	`sicklms('info');`

Table 2: Example calls for commands supported by the Sick LMS mex interface.

Although a command to explicitly uninitialize the device is not given, the same result can be obtained by simply clearing the mex file. More precisely, make the following call

```
clear sicklms
```

from the Matlab command line.

Return Values: Additionally, notice that two of the commands (i.e. `init` and `grab`) return values – both will likely be of interest. The init command returns a three element structure with the fields given in Table 3.

Field	Description
`lms_fast`	(Boolean) True if device is LMS Fast, false otherwise
`units_mm`	(Boolean) True if units are mm, false if units are cm
`meas_mode`	Current measurement mode of device[1]

Table 3: Structure returned from sicklms('init',DEVICE_PATH,DESIRED_BAUD)

[1] *See page 96 of the Sick LMS 2xx Telegram Listing for details.*

Field	Description
`res`	Angular scan resolution (0.25, 0.50, or 1.0)
`fov`	Angular scan angle/fov (90, 100, or 180)
`range`	$n \times 1$ vector of range values[2]
`reflect`	$n \times 1$ vector of reflectivity values[3]

Table 4: Structure returned from sicklms('grab')

The `grab` command also returns a structure. In this case, it is a four element structure having the fields given in Table 4.

6.3.2 Program Structure

As mentioned earlier, a Matlab script using the Sick LMS 2xx mex interface will follow the state machine given in Figure 5 – just as any C++ application would. To illustrate this point, we provide the following example Matlab script. It allows the application developer to acquire and process data from the unit. In this case, returned structures are displayed using the Matlab `disp` function.

```
try

    % Initialize the Sick LMS 2xx
    ret = sicklms('init','/dev/ttyUSB0',500000);
    disp(ret);

    % Grab some measurements
    for i = 1:10
        data = sicklms('grab');
        disp(data);
    end

    % Uninitialize the device
    clear sicklms;

catch
    error('An error occurred!');
end
```

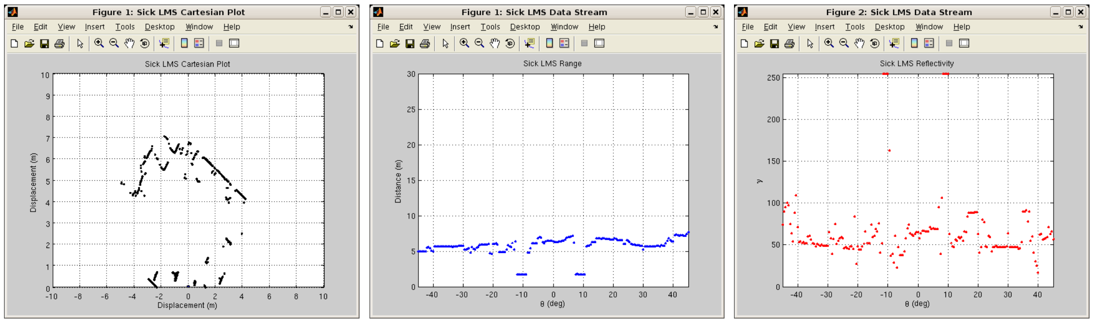


Figure 6: (Left) Visual output of *lms_cart* using a Sick LMS 291–S05. (Center) Range measurements obtained via a Sick LMS 291–S14 using *lms_stream*. (Right) Corresponding reflectivity measurements for (Center). In this case, the spikes in reflectivity are caused by a pair of retro-reflectors placed ≈ 1.7 meters from the unit.

6.3.3 Example Programs

In addition to providing the mex interface, the toolbox also comes with examples illustrating the use of the `sicklms` function. In particular, the following examples are included:

- `lms_cart` – A demo program that converts the Sick LMS 2xx measurements from polar to Cartesian coordinates and plots them in a figure window.
- `lms_stream` – Simply grabs raw data from the device and plots the values. If the device is an LMS Fast, it will display both reflectivity and range returns.
- `lms_variant` – Shows how to properly set the device variant from Matlab.

Figure 6 provides screenshots for a few of the examples using different LMS units.

Running the Matlab Example Scripts: To make things as straight forward as possible, each of the examples uses the same command line argument format. In particular, to invoke an example, call it using the following syntax (from the Matlab command prompt):

```
lms_example(DEVICE_PATH,BAUD_RATE)
```

where `DEVICE_PATH` refers to the path (e.g. `/dev/ttyUSB0`) associated with the Sick LMS unit and the `BAUD_RATE` refers to the desired session baud. Valid values for the latter are 9600, 19200, 38400, or 500000. Make sure that the working Matlab directory points to the `sicktoolbox-x/mex/examples/lms` directory. Also, be sure to replace `lms_example` with the correct name of the example that you wish to run. For instance, to call the `lms_cart` example using a Sick interfaced via a USB–COMi–M at `/dev/ttyUSB0`, simply call:

```
lms_cart('/dev/ttyUSB0',500000)
```

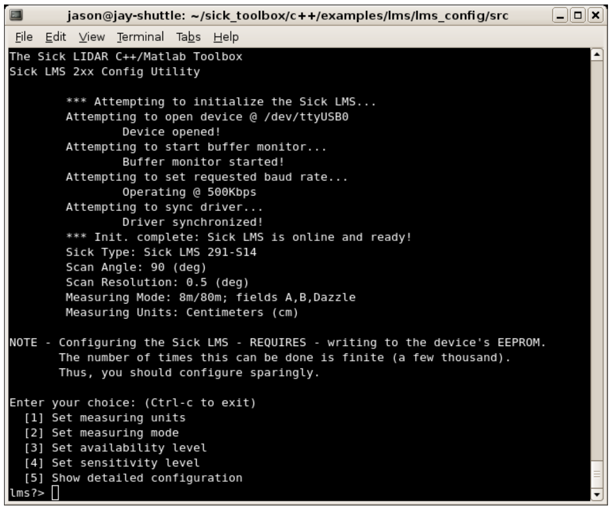
[2] *range will be empty if the device is set to stream only reflectivity values.*
[3] *reflect will be empty if the device is set to stream only range values.*

Additional Help and Documentation: For additional information with more example calls and usage, be sure to check the help documentation associated with the `sicklms` function. In particular, from within Matlab, simply call `help` as follows

```
help sicklms
```

6.4 Configuring the Sick LMS 2xx

WARNING! Configuring the Sick LMS 2xx – REQUIRES – writing to the device's EEPROM. The number of times this can be done is finite (a few thousand). Thus, configure the device only when necessary.


jason@jay-shuttle: ~/sick_toolbox/c++/examples/lms/lms_config/src
File Edit View Terminal Tabs Help

```
The Sick LIDAR C++/Matlab Toolbox
Sick LMS 2xx Config Utility

        *** Attempting to initialize the Sick LMS...
        Attempting to open device @ /dev/ttyUSB0
                Device opened!
        Attempting to start buffer monitor...
                Buffer monitor started!
        Attempting to set requested baud rate...
                Operating @ 500Kbps
        Attempting to sync driver...
                Driver synchronized!
        *** Init. complete: Sick LMS is online and ready!
        Sick Type: Sick LMS 291-S14
        Scan Angle: 90 (deg)
        Scan Resolution: 0.5 (deg)
        Measuring Mode: 8m/80m; fields A,B,Dazzle
        Measuring Units: Centimeters (cm)

NOTE - Configuring the Sick LMS - REQUIRES - writing to the device's EEPROM.
       The number of times this can be done is finite (a few thousand).
       Thus, you should configure sparingly.

Enter your choice: (Ctrl-c to exit)
  [1] Set measuring units
  [2] Set measuring mode
  [3] Set availability level
  [4] Set sensitivity level
  [5] Show detailed configuration
lms?>
```

Figure 7: `lms_config` is a shell–like configuration utility provided with the Sick LIDAR Matlab/C++ Toolbox. It provides a means to quickly reconfigure device parameters, including: measuring units, measuring mode, and availability level. It also allows the end–user to view the current device settings.

It may be desirable to reconfigure the Sick LMS 2xx EEPROM to modify its behavior for a certain task. To make such device configuration more practical, we also include an application utility called `lms_config`. Although it is provided as an example C++ program, `lms_config` is a helpful tool that allows the following parameters to be reconfigured:

- Measuring Units
- Measuring Mode
- Availability Level
- Sensitivity Level

Additionally, it allows you to quickly view the current device parameters. For additional details on what precisely each of these parameters mean, see pages 96–98 of the Sick LMS 2xx Telegram Listing that came with your unit [1].

> **Note:** `lms_config` is installed provided you ran `make` install. Thus, you should be able to call it directly as you would any other standard Linux utility. It is the only Sick LMS 2xx example that is installed.

Configuring for Dazzle Recovery: By default the Sick LMS 2xx will "lockup" upon being dazzled requiring the device to be reset. To configure the device to continue operating despite being dazzled requires simply adjusting the availability level of the unit. The easiest way to do this is to set the device to the highest availability via the `lms_config` utility (e.g. options 2, 3, 4, or 5 in `lms_config` under "set availability level"). Using this approach, dazzling will be indicated by returning an "overflow value" for the corresponding measurement.[4] As a result, each return should be filtered for overflow values. For instance, in 8m/80m mode, the overflow value for dazzling is 8190 and any measurement matching this value should be ignored. As the overflow values are a function of the device's measuring mode, be sure to see page 124 of the Sick LMS 2xx Telegram Listing to ensure you are using the correct values.

Configuring to Stream Reflectivity: Although Sick LMS units are largely used for accurate range measurements, they can also be configured to stream reflectivity information instead. Setting this up is a straight forward process using the `lms_config` utility. In particular, it simply requires adjusting to measuring mode to be "Reflectivity/Intensity values." Once this mode is set, all measurements will correspond to measured reflectivity (echo amplitude) as opposed to distance values. It should be noted that this measuring mode is not supported for LMS Fast units as they have a separate mode to stream both range and reflectivity data.

7 Interfacing the Sick LD

7.1 Sick LD C++ Driver Features

The Sick LD C++ driver provides a concise interface for conveniently acquiring measurements and configuring the Sick LD unit. Following are some of the available features:

Multi-threaded Implementation: Like the Sick LMS 2xx C++ driver, the Sick LD driver utilizes the pthreads library. While the main thread is running, the driver thread

[4] *See page 124 of the Sick LMS 2xx Telegram Listing for details regarding overflow values.*

WARNING! (UNSTABLE SICK LD NETWORK STACK): By design the Sick LD utilizes TCP/IP. In practice, we uncovered what we suspect is a serious flaw in the design of the Sick LD network stack. Putting the device unprotected on a network with excessive UDP traffic can result in device failure. This device behavior was repeatable across multiple LD units. Others have also experienced this problem independently. If using a direct connection via a Sick LD Ethernet cross–over cable is not practical, you can safely network the device if it is behind a firewall that blocks UDP traffic.

(in particular, the "buffer monitor") is busy ensuring that only the most recent complete scan is buffered for return. As threading is integrated into the driver framework, there is no need to worry about buffer management. By designing the driver this way, we alleviate as much overhead for the application developer as possible, by allowing him/her to focus on data–processing as opposed to data–management.

Support for Multiple Scan Areas: A novel feature of the Sick LD laser range finder unit is that it supports multiple (configurable) scan areas. The Sick LD C++ driver provides a simple interface for acquiring data from multiple scan sectors as well as any related information. Additionally, it supports the configuration of these scan areas. The application developer has the option of configuring the areas permanently in flash or just temporarily until the device power is reset.

Utilizes BSD Sockets Interface: Unlike the Sick LMS driver, the Sick LD C++ driver communicates over a standard BSD socket interface to the device. As a result, it supports the fastest data rate possible.

7.2 Using the Sick LD C++ Driver

7.2.1 Coordinate System

The Sick LD LIDAR family returns measurements which can be interpreted as polar coordinates. The device scans as shown in Figure 8.

7.2.2 Program Structure

The Sick LD C++ driver is designed to provide easy data acquisition as well as device configuration. Like the Sick LMS 2xx driver, its design lends itself to developing applications that emulate the quaternary state machine given in Figure 5, Section 6.2. The simplest use of the driver is given in the following code segment, which assumes a single scan sector:

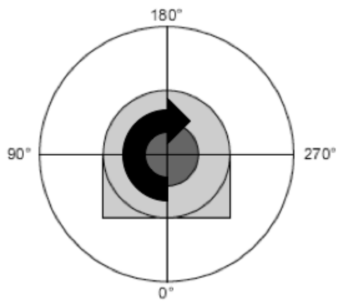


Figure 8: The Sick LD family returns measurements as polar coordinates in the illustrated direction. Multiple active (or measuring) sectors can be defined over this scan area. This figure is a modified version of that appearing in the Sick LD telegram listing [2].

```
#include <iostream>
#include <sickld-1.0/SickLD.hh>

using namespace std;
using namespace SickToolbox;

int main(int argc, char *argv[]) {

  /* Define buffers for return values */
  double measurements[SickLD::SICK_MAX_NUM_MEASUREMENTS] = {0};
  unsigned int num_measurements = 0;

  /* Instantiate the object */
  SickLD sick_ld("192.168.0.12");

  try {

    /* Initialize the device */
    sick_ld.Initialize();

    /* Grab some range measurements */
    for(unsigned int i = 0; i < 10; i++) {
      sick_ld.GetSickMeasurements(measurements,NULL,&num_measurements);
      cout << "\t" << num_measurements << endl;
    }

    /* Uninitialize the device */
    sick_ld.Uninitialize();
  }

  catch(...) {
    cerr << "error" << endl;
    return -1;
  }

  return 0;
}
```

In this example, the three most important methods are used. Following the state machine paradigm, the program begins by calling *Initialize*. Before any data can be streamed this method must absolutely be called. Once initialization is complete, the device will then grab scans from the LIDAR by calling the *GetSickMeasurements* method. As the receive buffer is constantly being monitored, the data returned from this call is guaranteed to be the most recent buffered scan. Once the program is done acquiring data, it then uninitializes the Sick LD by calling *Uninitialize*, which returns the Sick LD to a non-streaming state. It is important that each program properly initialize and uninitialize the device.

This example uses the constant `SickLD::SICK_MAX_NUM_MEASUREMENTS` to define its buffer size. This constant is provided for convenience in the SickLD class definition. It is simply the maximum number of measurements determined by considering the maximum scan area and the highest scan resolution for the Sick LD. It alleviates having to worry about computing buffer sizes for multiple configurations by ensuring that there is always enough storage allocated to handle the maximum number of measurements the device can return.

It is worth noting that the majority of applications will find this example the most relevant as they will likely require only a single active scan sector. However, in the event data from multiple sectors is required, we present the following example.

```
#include <iostream>
#include <sickld-1.0/SickLD.hh>

#define NUM_SECTORS (3)

using namespace std;
using namespace SickToolbox;

int main(int argc, char *argv[]) {

  /* Define buffers for return values */
  double measurements[SickLD::SICK_MAX_NUM_MEASUREMENTS] = {0};

  /* Define buffers to hold sector specific data */
  unsigned int num_measurements[NUM_SECTORS] = {0};
  unsigned int sector_data_offsets[NUM_SECTORS] = {0};
  unsigned int sector_ids[NUM_SECTORS] = {0};

  /* Instantiate the object */
  SickLD sick_ld("192.168.0.12");

  try {

    /* Initialize the device */
    sick_ld.Initialize();

    for(unsigned int i = 0; i < 10; i++) {

      /* Grab the measurements (from all sectors) */
      sick_ld.GetSickMeasurements(measurements,
                                  NULL,
                                  num_measurements,
                                  sector_ids,
                                  sector_data_offsets);

      /* Print the num measurements and the first
       * measured value for each measuring sector
       */
      for(unsigned int j = 0; j < NUM_SECTORS; j++) {
        cout << "\tSector: " << sector_ids[j]
             << " Num. Meas: " << num_measurements[j]
             << " First Value: " << measurements[sector_data_offsets[j]]
             << endl;
      }

      cout << endl;
    }

    /* Uninitialize the device */
    sick_ld.Uninitialize();
  }

  catch(...) {
    cerr << "error" << endl;
    return -1;
  }

  return 0;
}
```

In the latter example, we allocate a single buffer to hold the returned measurements. As the scan area associated with each sector can vary, we aggregate the measurements into a single buffer and then provide "sector data offsets" into this buffer, where the data for the associated sector can be found. The data can then be accessed via indexing accordingly as in the example or by appropriate pointer arithmetic. Additional information about each of the sectors can also be obtained. See the corresponding doxygen generated documents in the `doxygen-doc/html` directory for more details.

7.2.3 Properly Linking Programs

In order to utilize the Sick LD driver, you must link against its associated library (e.g. `libsickld-1.0`) as well as the POSIX thread library on your system (e.g. `-lpthread` or `-pthread`). For instance, using `g++` (in this case version 4.1.2) on Ubuntu (as well as Debian), you can compile and link using the command:

```
g++ -o prog_name prog_name.cc -lsickld-x -pthread
```

where `prog_name` should be replaced with the name of your program and `x` denotes the Sick LD library version (e.g. `libsickld-1.0`).

7.2.4 Example Programs

In addition to the C++ driver, the toolbox also comes with a variety of examples illustrating its use. All of the example projects with code can be found in the directory:

```
sicktoolbox-x/c++/examples/ld
```

In all, there are four examples including everything from a file–driven configuration utility to a program illustrating how to acquire data from multiple active scan areas. More precisely, the following examples are provided:

- `ld_config` – A configuration utility for Sick LD laser range finders. Allows setting: motor speed, scan resolution and scan areas.
- `ld_more_config` – Illustrates how to do additional (advanced) configuration using the Sick LD C++ driver.
- `ld_single_sector` – Demonstrates the easiest method for acquiring data from a Sick LD configured with a single sector.
- `ld_multi_sector` – Demonstrates acquiring data from a Sick LD configured with multiple sectors.

Looking over these examples is the easiest way to quickly get yourself up and developing with the driver interface.

Running the C++ Example Programs: In addition to building the C++ driver interface, running `make` also builds each of the examples in the example directory. If a dependency for a certain example is not detected then the associated example is not built. The binary for each can be found in its respective `src` directory (e.g. for the `ld_config` example, look in `sicktoolbox-x/c++/examples/ld/ld_config/src`). To make things as easy as possible, all of the examples use the same command–line argument format. In particular, to call any example, simply invoke it from the command line as follows:

```
./example_name [DEVICE_IP]
```

where `DEVICE_IP` (default: "192.168.1.10") denotes the associated IP address for the Sick. For instance, to run the example `ld_single_sector` with your Sick LD using the factory default IP address of 192.168.1.10, simply call it as follows:

```
./ld_single_sector
```

7.3 Using the Sick LD Mex Interface

7.3.1 Mex Commands

A call to the Sick LD can be made by using the installed *sickld* mex function as follows:

```
sickld(CMD,ARGS)
```

where `CMD` is the command to issue to the device via the driver and `ARGS` denotes the argument list for the command. In particular, the interface supports the commands shown in Table 5. For example calls, see Table 6.

Although a command to explicitly uninitialize the device is not given, the same result can be obtained by simply clearing the mex file. More precisely, make the following call

CMD	ARGS
`init`	`DEVICE_IP`
`range`	N/A
`range+reflect`	N/A
`info`	N/A

Table 5: Commands supported by Sick LD mex interface.

CMD	Example Call
`init`	`sickld('init','192.168.1.10');`
`range`	`res = sickld('range');`
`range+reflect`	`res = sickld('range+reflect');`
`info`	`sickld('info');`

Table 6: Example calls for commands supported by the Sick LD mex interface.

```
clear sickld
```

from the Matlab command line.

Return Values: Additionally, notice that two of the commands (i.e. `range` and `range+reflect`) return values. Both commands return an array of eight element structures (one for each measuring sector) each with the fields given in Table 7.

Field	Description
`id`	Sector ID number
`res_ang`	Angular resolution over sector
`beg_ang`	Beginning angle of sector (included in scan)
`end_ang`	Ending angle of sector (included in scan)
`beg_time`	Time at which first measurement was obtained
`end_time`	Time at which last measurement was obtained
`range`	$n \times 1$ vector of range measurements
`reflect`	$n \times 1$ vector of reflectivity Measurements

Table 7: Structure returned from sickld('init',DEVICE_IP)

7.3.2 Program Structure

As mentioned earlier, a Matlab script using the Sick LD mex interface will follow the state machine given in Figure 5, Section 6.2 – just as any C++ application would. To illustrate this point, we provide the following example Matlab script. It is a simple implementation allowing the application developer to acquire and process data.

```
try

    % Initialize the Sick LD
    sickld('init','192.168.0.12');

    for i=1:10

        % Grab the most recent values
        data = sickld('range+reflect');

        % Print the ID of each sector
        for j = 1:length(data)
            disp(data(j).id);
        end

    end

    % Uninitialize the device
    clear sickld;

catch
    error('An error occurred!');
end
```

Attention: The Sick LD mex interface by design will return an array of structures – one for each active sector. To access the data structure associated with the i^{th} scan area sub–index it. For example, if the device is configured with two active scan areas then data(1) will correspond to the data structure corresponding to the first and data(2) will correspond to the second. If the device is only using a single scan sector then sub-indexing is not required (e.g. use `data` instead of `data(1)`).

7.3.3 Example Programs

In addition to providing the mex interface, the toolbox also comes with examples illustrating the use of the *sickld* function. In particular, the following examples are included:

- `ld_cart` – A demo program that converts the Sick LD measurements from polar to Cartesian coordinates.
- `ld_stream` – Simply grabs raw data from the device and plots the values obtained from each sector in separate figures (e.g. if there are three active/measuring sectors then three figures will be generated).

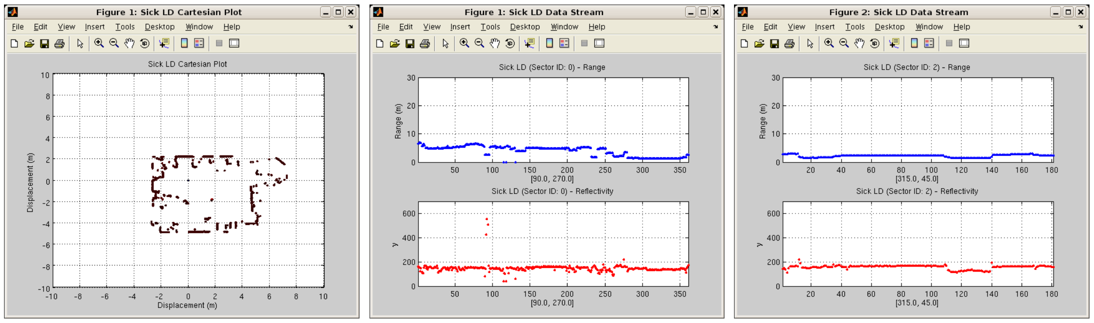


Figure 1: Sick LD Cartesian Plot
Sick LD Cartesian Plot
Displacement (m)
Displacement (m)
Figure 1: Sick LD Data Stream
Sick LD (Sector ID: 0) - Range
Range (m)
[90.0, 270.0]
Sick LD (Sector ID: 0) - Reflectivity
Figure 2: Sick LD Data Stream
Sick LD (Sector ID: 2) - Range
Range (m)
[315.0, 45.0]
Sick LD (Sector ID: 2) - Reflectivity

Figure 9: (Left) Visual output of ld_cart using a Sick LD–LRS 1000 scanning over a single scan area of $[0, 359.5]$ at $0.5°$ scan resolution. (Center, Right) Range and reflectivity measurements obtained for two separate scan areas defined respectively over: $[90, 270]$ and $[315, 45]$.

Figure 9 provides screenshots for the examples using a Sick LD–LRS 1000.

Running the Matlab Example Scripts: To make things as straight forward as possible, each example uses the same command line argument format. In particular, to invoke an example, call it using the following syntax (from the Matlab command prompt):

```
ld_example(DEVICE_IP,ARGS)
```

where `DEVICE_IP` refers to the IP address (e.g. '192.168.1.10') associated with the Sick LD unit and `ARGS` is any additional optional arguments for the example. To learn what arguments are accepted for a particular example, just type `help` followed by the example name. Make sure that the working Matlab directory points to the `sicktoolbox-x/mex/examples/ld` directory. As an example, to call `ld_cart` simply invoke it from the Matlab command line it as follows:

```
ld_cart('192.168.1.10')
```

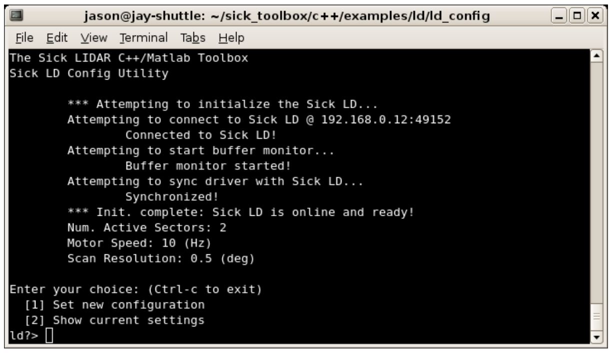
Note that the IP address for these examples must be specified as the m–files make no assumptions regarding the default address of the device.

Additional Help and Documentation: For additional information with more example calls and usage, be sure to check the help documentation associated with the `sickld` function. In particular, from within Matlab, simply call:

```
help sickld
```

7.4 Configuring the Sick LD

It may be desirable to reconfigure the Sick LD to modify its parameters for a certain task. To make such device configuration more practical, we also include an application utility


jason@jay-shuttle: ~/sick_toolbox/c++/examples/ld/ld_config
File Edit View Terminal Tabs Help

```
The Sick LIDAR C++/Matlab Toolbox
Sick LD Config Utility

        *** Attempting to initialize the Sick LD...
        Attempting to connect to Sick LD @ 192.168.0.12:49152
                Connected to Sick LD!
        Attempting to start buffer monitor...
                Buffer monitor started!
        Attempting to sync driver with Sick LD...
                Synchronized!
        *** Init. complete: Sick LD is online and ready!
        Num. Active Sectors: 2
        Motor Speed: 10 (Hz)
        Scan Resolution: 0.5 (deg)

Enter your choice: (Ctrl-c to exit)
  [1] Set new configuration
  [2] Show current settings
ld?> 
```

Figure 10: *ld_config* is a configuration utility provided with the Sick LIDAR Matlab/C++ Toolbox. It provides a means to reconfigure device parameters using a simple configuration file. It allows the end–user to set: motor speed, scan resolution, and active/measuring scan areas. Additionally, the end–user can view the current device settings.

called `ld_config`. Although it is provided as an example C++ program, `ld_config` is a helpful tool that allows the following parameters to be reconfigured:

- Motor Speed
- Scan Resolution
- Scan Areas

Additionally, it allows you to quickly view the current device parameters. Note that the `ld_config` utility is installed provided you ran `make install`. Thus, you can call it directly from your shell as you would any other standard Linux utility. It is the only example that is installed.

Due to the large number of possible scan configurations, we decided to implement `ld_config` to use a configuration file. To adjust the configuration parameters for the device, you can simply modify the file and then run `ld_config`. The utility will give you the option of displaying the current configuration or setting a new one by supplying the source file name. Figure 10 shows what the main screen looks like.

An Example Configuration File: Each configuration file must define three settings for the device: motor speed, resolution, and scan areas. These parameters are set by assigning value to one of the following variables in the configuration file:

- `SICK_LD_MOTOR_SPEED`
- `SICK_LD_SCAN_RESOLUTION`
- `SICK_LD_SCAN_AREAS`

It should be noted that all three variables *must* be set in the file otherwise an error will be thrown. Following is a sample configuration file contained in the `sicktoolbox-x/c++/examples/ld/ld_confi` directory.

```
################################################################
#
# The Sick LIDAR Matlab/C++ Toolbox
#
# File: sickld.conf
# Auth: Jason Derenick and Thomas Miller at Lehigh University
# Cont: derenick(at)lehigh(dot)edu
# Date: 20 July 2007
#
# Desc: Sample config file for ld_config utility.
#
################################################################

# Define the Sick LD motor speed (Hz)
SICK_LD_MOTOR_SPEED = 10

# Define the Sick LD scan res. (angle step) in degrees
SICK_LD_SCAN_RESOLUTION = 0.5

# Define the active scan areas for the device
SICK_LD_SCAN_AREAS = [90 270] [315 45]
```

This configuration file indicates that the device should be set for two active scan areas (at most there can only be four) – each is defined as a closed set in the configuration file. Additionally, it specifies the motor to operate at 10Hz (the slowest motor speed is 5Hz with a max being 20Hz for LD models excluding the LD–LRS 1000) and the scan resolution to be a half–degree (valid values are multiples of 0.125° that do not exceed 1.0°). Additionally, please note that the scan resolution *must* evenly divide into the boundaries defining each of the scan areas. To learn more about additional configurational parameters and for a detailed description on what constitutes a valid configuration be sure to see [2].

8 Uninstalling the Toolbox

8.1 Uninstalling the C++ Drivers

Uninstalling the C++ driver libraries is simply a matter of running *make uninstall*.

8.2 Uninstalling the Mex Interfaces

Uninstalling the mex interface is also a straightforward. Just delete the directory:

```
$MATLAB/toolbox/sick
```

where `$MATLAB` is the root directory of your Matlab install.

To complete the uninstallation, just remove the corresponding directory from your Matlab path. To do this, simple run

```
pathtool
```

and then click and delete the `$MATLAB/toolbox/sick` path listing.

9 Doxygen Documentation

All of the C++ contained in the toolbox is commented using Doxygen commenting standards. Assuming Doxygen is installed on your machine, simply run:

```
make doxygen-doc
```

from the sicktoolbox–x directory. This will generate the project's source code documentation and place it in the sicktoolbox–x/doxygen–doc directory. To view it, simply open sicktoolbox–x/doxygen–doc/html/index.html in your favorite web browser. Additionally, you can browse the Doxygen generated comments on the project's website at:

```
http://vader.cse.lehigh.edu/sicktoolbox
```

10 Open–Source Software License

The Sick LIDAR Matlab/C++ Toolbox is offered under a BSD Open-Source License.

References

[1] SICK AG Waldkirch. *Telegrams for Configuring and Operating the LMS 2xx Laser Measurement System*, 2006.

[2] SICK AG Waldkirch. *User Protocol Services for Operating/Configuring the LD–OEM/LD–LRS Laser Measurement System*, 2006.