


CONTENT 目录

Version Control 修订记录	2
1. 概述	4
2. 系统要求	4
3. 编译使用	4
3.1. Linux 下编译.....	4
3.2. Windows 下编译.....	6
4. 代码介绍	8
4.1. 代码目录结构.....	8
4.2. 代码接口文件.....	9
5. 接口函数说明	9
5.1. 网络参数.....	9
5.1.1. IP 地址配置和查询.....	9
5.1.2. 端口号配置和查询.....	10
5.2. 工作状态参数.....	10
5.2.1. 扫描频率配置和查询.....	10
5.2.2. 点云滤波等级配置和查询.....	10
5.2.3. 时间戳配置和查询.....	10
5.2.4. 外同步状态查询.....	11
5.3. 设备的基本操作.....	11
5.3.1. 设备连接.....	11
5.3.2. 设备开关.....	11
5.3.3. 点云数据传输开关.....	11
5.3.4. 参数配置信息保存.....	11
5.3.5. 设备复位.....	11
5.4. 版本标识信息.....	12
5.4.1. 固件版本号查询.....	12
5.4.2. 硬件版本号查询.....	12
5.4.3. SDK 版本号查询.....	12
5.5. 实时监测信息.....	12
5.5.1. 电机瞬时旋转频率.....	12
5.5.2. 内部监测温度.....	12
5.6. 点云信息获取.....	13
5.6.1. 获取一个 block 点云数据.....	13
5.6.2. 获取一帧点云数据.....	14
5.7. 超时设置.....	15
5.8. 点云滤波功能.....	15
5.9. 注意事项.....	15
5.9.1. 错误提醒.....	15
5.9.2. 接口使用限制.....	16
6. 快速使用指南	16
7. 表目录	18

1. 概述

本文档描述了 Oradar MS500 Lidar SDK 的功能与使用方法，与 Oradar MS500 Lidar 同时发布。

该 SDK 具有如下特点：

- 1) 采用 C++编写，API 函数以 C++类库的形式供用户调用和开发；
- 2) 内部使用异步模式实现高效数据 IO，外部 API 接口则采用同步式设计以简化使用；
- 3) 支持主流操作系统平台，可在 Windows 和 Linux 系统下编译和使用；
- 4) 所需的核心第三方库均已随 SDK 一起打包发布，编译时无需安装额外支持文件。

本 SDK 不能用于对其他设备或系统的控制，请勿移作他用。

2. 系统要求

本 SDK 为源码发布，请用户自行集成到目标系统。系统环境应满足如下要求：

支持 C++11 的编译器(必选):SDK 及其依赖的部分第三方库使用了 C++11 引入的特性，因此编译器应提供完整的 C++11 支持；

CMake (必选): SDK 使用 CMake 作为构建工具，要求系统中 CMake 的版本号为 2.8.3 或更高。

3. 编译使用

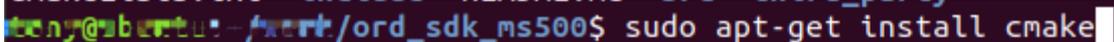
ord_sdk_ms500 代码可以独立编译，代码工程由 Cmake 构建，CMake 会根据 CMakeLists.txt 文件中的命令，为指定的开发环境生成工程文件并设置各项参数（例如源文件的路径以及各种编译选项）。CMake 运行完毕后，会在当前目录下生成包含编译指令的 Makefile。此时执行 make 命令即可对项目进行编译。

3.1. Linux 下编译

Linux 以 X86 的 Ubuntu-16.04 为例：

1、安装 cmake:

```
sudo apt-get install cmake
```



```
keny@ubuntu:~/work/ord_sdk_ms500$ sudo apt-get install cmake
```

```
cmake --version
```

```

tony@ubuntu:~/src/ord_sdk_ms500$ cmake -version
cmake version 3.10.2

CMake suite maintained and supported by Kitware (kitware.com/cmake).
    
```

2、查看 gcc 版本:

```

tony@ubuntu:~/src/ord_sdk_ms500$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
    
```

make 命令默认会调用系统的 gcc 命令,必须确保系统支持 gcc , 如果没有请先安装。

3、代码编译

unzip ord_sdk_ms500.zip

```

tony@ubuntu:~/work$ cd ord_sdk_ms500/
tony@ubuntu:~/work/ord_sdk_ms500$ mkdir build
tony@ubuntu:~/work/ord_sdk_ms500$ cd build/
tony@ubuntu:~/work/ord_sdk_ms500/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tony/work/ord_sdk_ms500/build
tony@ubuntu:~/work/ord_sdk_ms500/build$

```

默认是 Release 版本，如果要编译调试版本，`cmake -DCMAKE_BUILD_TYPE=Debug ..`
`make -j8` 编译后生成 `ord_sdk.a`

```

tony@ubuntu:~/work/ord_sdk_ms500/build$ make -j8
Scanning dependencies of target ord_sdk
[ 20%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver_impl.cpp.o
[ 60%] Building CXX object CMakeFiles/ord_sdk.dir/src/lidar_address.cpp.o
[ 60%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver.cpp.o
[ 80%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver_net.cpp.o
[100%] Linking CXX static library ord_sdk.a
[100%] Built target ord_sdk
tony@ubuntu:~/work/ord_sdk_ms500/build$

```

3.2. Windows 下编译

这里以 Windows10 为例：

- 1、从官网 <https://cmake.org/download/> 下载最新版本的 `cmake-3.15.5-win64-x64.msi` 安装。
- 2、安装后按 Shift 键和鼠标右键打开 powershell，`cmake -version` 确认 cmake 已安装。

```

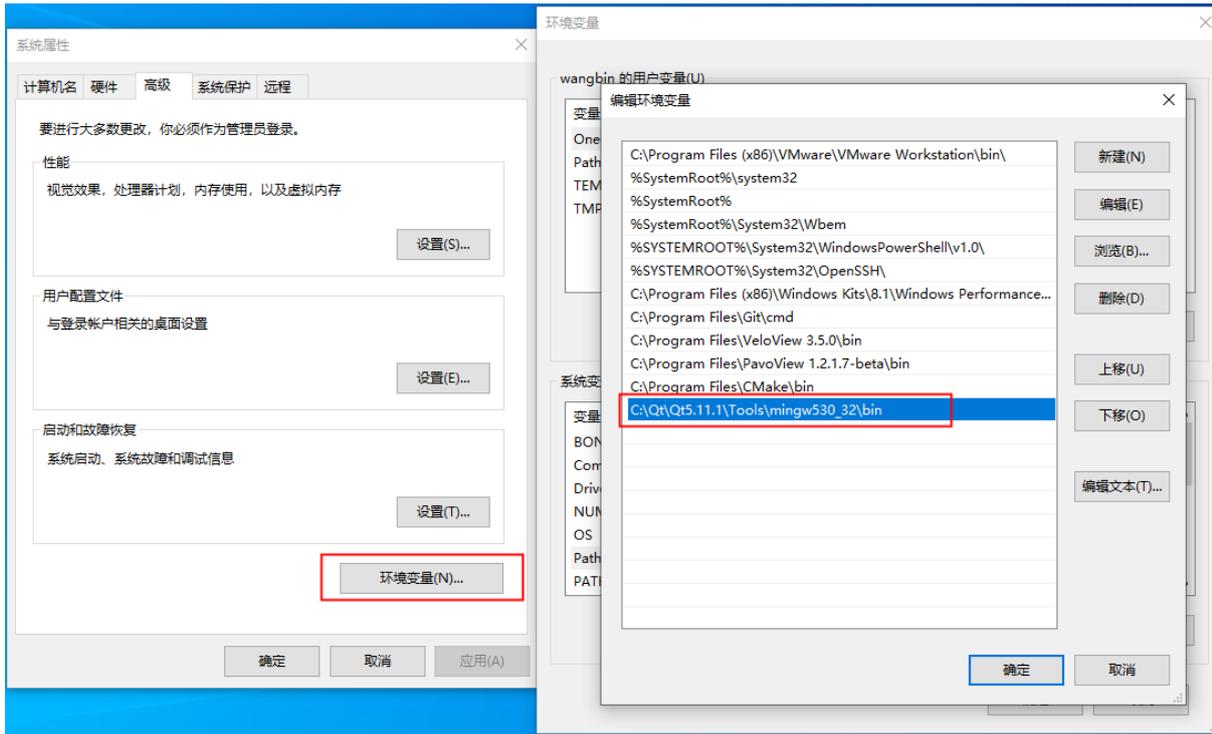
PS Y:\ord_sdk_ms500\build> cmake -version
cmake version 3.15.5

CMake suite maintained and supported by Kitware (kitware.com/cmake).
PS Y:\ord_sdk_ms500\build>

```

- 3、安装 gcc 编译器，这里以 QT 的 MinGW 为例，官网 <http://www.mingw-w64.org/doku.php/download> 下载 `MinGW-5.1.6.exe` 安装。

4、将编译器的安装路径导入系统环境变量中。



5、PowerShell 中输入以下命令确认编译器正确安装：

```
PS Y:\ord_sdk_ms500\build> cmake -version
cmake version 3.15.5

CMake suite maintained and supported by Kitware (kitware.com/cmake).
PS Y:\ord_sdk_ms500\build> mingw32-make -version
GNU Make 4.2.1
Built for i686-w64-mingw32
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
PS Y:\ord_sdk_ms500\build> gcc -v
Using built-in specs.
COLLECT_GCC=C:\Qt\Qt5.14.2\Tools\mingw730_32\bin\gcc.exe
COLLECT_LTO_WRAPPER=C:\Qt\Qt5.14.2\Tools\mingw730_32\bin\..\libexec/gcc/i686-w64-mingw32/7.3.0/lto-wrapper.exe
Target: i686-w64-mingw32
Configured with: ../../../../src/gcc-7.3.0/configure --host=i686-w64-mingw32 --build=i686-w64-mingw32 --target=i686-w64-mingw32 --prefix=/mingw32 --with-sysroot=/c:/mingw730/i686-730-posix-dwarf-rt_v5-rev0/mingw32 --enable-shared --enable-static --disable-multilib --enable-languages=c,c++,fortran,lto --enable-libstdcxx-time=yes --enable-threads=posix --enable-libgomp --enable-libatomic --enable-lto --enable-graphite --enable-checking=release --enable-fully-dynamic-string --enable-version-specific-runtime-libs --enable-libstdcxx-filesystem-ts=yes --disable-sjlj-exceptions --with-dwarf2 --disable-libstdcxx-pch --disable-libstdcxx-debug --enable-bootstrap --disable-rpath --disable-win32-registry --disable-mls --disable-werror --disable-symvers --with-gnu-as --with-gnu-ld --with-arch=i686 --with-tune=generic --with-libiconv --with-system-zlib --with-gmp=/c:/mingw730/prerequisites/i686-w64-mingw32-static --with-mpfr=/c:/mingw730/prerequisites/i686-w64-mingw32-static --with-mpc=/c:/mingw730/prerequisites/i686-w64-mingw32-static --with-isl=/c:/mingw730/prerequisites/i686-w64-mingw32-static --with-pkgversion=' i686-posix-dwarf-rev0, Built by MinGW-W64 project' --with-bugurl=https://sourceforge.net/projects/mingw-w64 CFLAGS=' -O2 -pipe -fno-ident -I/c:/mingw730/i686-730-posix-dwarf-rt_v5-rev0/mingw32/opt/include -I/c:/mingw730/prerequisites/i686-zlib-static/include -I/c:/mingw730/prerequisites/i686-w64-mingw32-static/include' CXXFLAGS=' -O2 -pipe -fno-ident -I/c:/mingw730/i686-730-posix-dwarf-rt_v5-rev0/mingw32/opt/include -I/c:/mingw730/prerequisites/i686-zlib-static/include -I/c:/mingw730/prerequisites/i686-w64-mingw32-static/include' CPPFLAGS=' -I/c:/mingw730/i686-730-posix-dwarf-rt_v5-rev0/mingw32/opt/include -I/c:/mingw730/prerequisites/i686-zlib-static/include -I/c:/mingw730/prerequisites/i686-w64-mingw32-static/include' LDFLAGS=' -pipe -fno-ident -L/c:/mingw730/i686-730-posix-dwarf-rt_v5-rev0/mingw32/opt/lib -L/c:/mingw730/prerequisites/i686-zlib-static/lib -L/c:/mingw730/prerequisites/i686-w64-mingw32-static/lib -Wl,--large-address-aware'
Thread model: posix
gcc version 7.3.0 (i686-posix-dwarf-rev0, Built by MinGW-W64 project)
PS Y:\ord_sdk_ms500\build>
```

6、cmake 和编译器安装完成后开始编译

```

PS Y:\ord_sdk_ms500\build> cmake -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release ..
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: C:/Qt/Qt5.14.2/Tools/mingw730_32/bin/gcc.exe
-- Check for working C compiler: C:/Qt/Qt5.14.2/Tools/mingw730_32/bin/gcc.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: C:/Qt/Qt5.14.2/Tools/mingw730_32/bin/g++.exe
-- Check for working CXX compiler: C:/Qt/Qt5.14.2/Tools/mingw730_32/bin/g++.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: Y:\ord_sdk_ms500\build
PS Y:\ord_sdk_ms500\build>
    
```

如果要编译调试版本，`cmake -DCMAKE_BUILD_TYPE=Debug ..`
`mingw32-make -j8` 编译后生成 `ord_sdk.a`

```

PS Y:\ord_sdk_ms500\build> mingw32-make -j8
Scanning dependencies of target ord_sdk
[ 20%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver.cpp.obj
[ 60%] Building CXX object CMakeFiles/ord_sdk.dir/src/lidar_address.cpp.obj
[ 60%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver_impl.cpp.obj
[ 80%] Building CXX object CMakeFiles/ord_sdk.dir/src/ord_driver_net.cpp.obj
[100%] Linking CXX static library ord_sdk.a
[100%] Built target ord_sdk
PS Y:\ord_sdk_ms500\build>
    
```

4. 代码介绍

4.1. 代码目录结构

代码目录结构如下：

```

ord_sdk_ms500
├── include
├── src
├── third_party
├── samples
├── doc
├── CMakeLists.txt
└── README.md
    
```

include：SDK 的头文件目录；

src：SDK 的源文件目录；

third_party:存放 SDK 中用到的核心第三方库；

samples: 示例代码

doc: 相关说明文档

CMakeLists.txt：编译 SDK 的 CMake 命令文件；

README.md：SDK 的编译使用简介。

4.2. 代码接口文件

```

ord_sdk_ms500
├── include
│   └── ord
│       ├── lidar_address.h
│       ├── ord_driver.h
│       ├── ord_driver_impl.h
│       ├── ord_driver_net.h
│       ├── ord_types.h
│       └── utils.h
├── src
│   ├── lidar_address.cpp
│   ├── ord_driver.cpp
│   ├── ord_driver_impl.cpp
│   └── ord_driver_net.cpp
├── third_party
│   └── Asio
│       └── asio-1.18.2
├── samples
│   ├── demo_test.cpp
│   └── CMakeLists.txt
├── doc
├── CMakeLists.txt
└── README.md
    
```

ord_driver.h : SDK 的顶层接口函数 ;
 ord_driver.cpp : SDK 的顶层接口函数实现 ;
 ord_types.h : 点云数据块、数据帧格式。
 其它文件为内部实现 , 可以不关注。

5. 接口函数说明

5.1. 网络参数

包含激光雷达的 IP 地址 , 端口号等。对以上参数提供了设置和查询的接口。设置完成后需调用参数配置信息保存接口 applyConfigs() 才可对配置进行保存 , 且需要重启设备才可生效。

5.1.1. IP 地址配置和查询

```

error_t setLidarIPAddress(in_addr_t address);
error_t getLidarIPAddress(in_addr_t& address);
    
```

in_addr_t address , 即需要连接的设备的 IP 地址。该参数使用的数据类型为 in_addr_t , 实际上是 32 位无符号整数表示的 IP 地址 , 且使用网络字节序 (大尾端)。通常可通过 inet_addr() 或其它类似函数将字符串形式的 IP 地址转换为 in_addr_t 类型 , 反过来可通过 inet_ntoa 将 in_addr_t 类型转为字符串类型。MS500 雷达的出厂默认 IP 地址为 192.168.1.100 , 用户可以通过我们 SDK 提供的 API 接口函数进行修改和查询。

5.1.2. 端口号配置和查询

```
error_t setLidarNetPort(in_port_t port);
error_t getLidarNetPort(in_port_t& port);
```

in_port_t port, 即需要连接的设备的端口号。该参数的数据类型为 in_port_t。MS500 对外开放的默认端口号为 2007, 用户可以通过我们 SDK 提供的 API 接口函数进行修改和查询。

5.2. 工作状态参数

包含扫描频率、点云滤波等级、时间戳等。对以上参数提供了设置和查询的接口。

5.2.1. 扫描频率配置和查询

```
error_t setScanSpeed(uint32_t speed);
error_t getScanSpeed(uint32_t& speed);
```

参数 uint32_t speed 占用 4 个字节, 为网络字节序即大端模式。扫描频率值有效范围为 10、15、20、25 和 30。MS500 雷达支持多级调速功能, 其扫描频率可在 10 Hz 到 30 Hz 的范围内, 以 5 Hz 为间隔进行调整。由于雷达内部测距单元每秒所能测量的次数是固定的 (约 30 KHz), 当扫描频率发生变化时, 雷达的角分辨率和扫描数据帧中测量值的个数也会随之变动, 如下表所示:

表 5-1 转速与开火角分辨的关系

激光雷达扫描速率 (Hz)	角分辨 (°)	一个扫描周期开火总数 (次)
10	0.12	2268
15	0.18	1512
20	0.23	1152
25	0.30	900
30	0.36	756

5.2.2. 点云滤波等级配置和查询

```
error_t setTailFilterLevel(uint32_t level);
error_t getTailFilterLevel(uint32_t& level);
```

参数 uint32_t level 占用 4 个字节, 为网络字节序即大端模式。参数为点云滤波等级值, 有效范围为 0、1、2、3、4、5。MS500 雷达共支持 5 级滤波等级设置。其中为 0 时, 表示关闭点云滤波功能。用户可以通过我们 SDK 提供的 API 接口函数进行点云滤波等级的修改和查询。

5.2.3. 时间戳配置和查询

```
error_t setTimestamp(uint32_t timestamp);
error_t getTimestamp(uint32_t& timestamp);
```

参数 uint32_t timestamp 占用 4 个字节, 为网络字节序即大端模式。时间戳的范围: $0 \sim 3600 \times 10^6$, 即 1h, 单位为 us。用户可以通过我们 SDK 提供的 API 接口函数配置激光雷达的时间戳信息, 以便与主设备间保持同步, 也可以通过接口函数获取当前的时间戳信息。

5.2.4. 外同步状态查询

```
error_t getSyncStatus(uint32_t& sync_status);
```

参数 uint32_t work_status 占用 4 个字节，为网络字节序即大端模式。MS500 激光雷达外同步状态指示时间戳的同步模式，有两种，分别是：01 自由运行状态、02 外同步状态。所以参数 uint32_t sync_status 的取值范围为：0x01, 0x02。用户可以通过我们 SDK 提供的 API 接口函数对外同步状态进行查询。

5.3. 设备的基本操作

包括设备连接、设备开关、点云传输开关、工作状态、参数保存、设备复位等。其中设备开和关、点云传输开和关分别提供独立的操作接口，工作状态提供配置和查询接口。

5.3.1. 设备连接

```
error_t trackConnect();
```

由于 MS500 激光雷达与上位机的通讯采用 UDP 模式，UDP 属于无连接方式，该接口调用后主机会向激光雷达发送固定的测试报文，雷达收到报文后做出应答，主机据此确认是否与雷达建立有效连接。

5.3.2. 设备开关

```
error_t enableMeasure();  
error_t disableMeasure();
```

该接口调用后主机按照约定的雷达通讯协议发送测距开关控制命令，MS500 激光雷达接收到报文后作出应答，如果是开启命令，将开启开火及测距处理，如果是关闭命令，将关闭开火及测距处理。

5.3.3. 点云数据传输开关

```
error_t enableDataStream();  
error_t disableDataStream();
```

该接口调用后主机按照约定的雷达通讯协议发送点云上报开关控制命令，MS500 激光雷达接收到报文后作出反应，如果是开启操作，将开启点云数据上报，如果是关闭操作，将停止点云数据上报。

5.3.4. 参数配置信息保存

```
error_t applyConfigs();
```

MS500 的所有参数信息，在通过 SDK 提供的接口进行配置修改后，都必须调用该接口函数，才能将配置信息保存到 Flash 当中，重启后得到保持。如果参数修改，但是没有调用该接口，那该参数将不会保存到 Flash 当中，重启后恢复修改前的状态。

5.3.5. 设备复位

```
error_t deviceReboot();
```

调用该接口，主机将按照约定协议向激光雷达发送复位命令，MS500 接收到报文后作出应答，如果命令正确，将执行复位重启操作。

5.4. 版本标识信息

包含设备序列号、产品型号、固件版本号、硬件版本号、SDK 版本号。其中设备序列号、产品型号提供了配置和查询接口，固件版本号、硬件版本号、SDK 版本号提供查询接口。

5.4.1. 固件版本号查询

```
error_t getFirmwareVersion(std::string& firmware_version);
```

参数 std::string firmware_version 最大支持 16 字节的字符长度，具体以查询到的实际长度为准，MS500 激光雷达出厂有烧录对应的固件，用户可以通过我们 SDK 提供的 API 接口函数查询当前固件的版本号。

5.4.2. 硬件版本号查询

```
error_t getHardwareVersion(std::string& hardware_version);
```

参数 std::string hardware_version 最大支持 16 字节的字符长度，具体以查询到的实际长度为准，MS500 激光雷达出厂的硬件版本号可以通过我们 SDK 提供的 API 接口函数进行查询。

5.4.3. SDK 版本号查询

```
error_t getSDKVersion(std::string& sdk_version);
```

参数 std::string sdk_version 最大支持 16 字节的字符长度，具体以查询到的实际长度为准，当前使用的 SDK 软件版本号可以通过我们 SDK 提供的 API 接口函数进行查询。

5.5. 实时监测信息

包括电机瞬时旋转频率、内部监测温度、探测器 APD 偏置高压等，为以上实时监测信息提供了查询接口。

5.5.1. 电机瞬时旋转频率

```
error_t getMotorSpeed(uint32_t& motor_speed);
```

参数 uint32_t& motor_speed 占用 4 个字节，它的单位为 RPM，即每分钟的旋转周期数。用户可以通过我们 SDK 提供的 API 接口获取激光雷达的电机的瞬时旋转频率。

5.5.2. 内部监测温度

```
error_t getInternalTemperature(float& inter_temp);
```

参数 float&inter_temp 为浮点型数据，取小数点后两位有效。用户可以通过我们 SDK 提供的 API 接口获取激光雷达的内部的温度信息。

5.6. 点云信息获取

MS500 的水平视场角为 270° ，覆盖了机体正前方两侧各 135° 的区域。雷达沿逆时针方向对这部分区域进行重复旋转扫描，以 15 Hz 扫描频率为例，从雷达转动到水平视场角的开始位置起，每隔约 0.18° 即进行一次距离测量，直到水平视场角的结束位置处停止测距，共得到 1512 组测量数据，形成一个扫描数据帧，其中每组测量数据均含有相应角度的距离和强度值。

由激光雷达设备将点云数据通过上行网络通讯协议传送到 SDK 软件，数据通讯协议说明如下表所示：

表 5-2 点云数据通讯协议

顺序	名称	数据名称	数据内容	数据长度 (Byte)
1-6	帧头信息	固定部分	4D 53 01 F4 EB 90	6
7-8		帧长度 (整个数据包长度)	10Hz 时:0x05FC 15Hz 时:0x0404 20Hz 时:0x0314 25Hz 时:0x026C 30Hz 时:0x020C	2
9		保留	-	-
10		保留	-	-
11		数据信息类型	04 : 10Hz 时点云数据帧 05 : 15Hz 时点云数据帧 06 : 20Hz 时点云数据帧 07 : 25Hz 时点云数据帧 08 : 30Hz 时点云数据帧	1
12	数据块信息	数据块编号	01: 45~90°扫描范围点云帧 02: 90~135°扫描范围点云帧 03: 135~180°扫描范围点云帧 04: 180~225°扫描范围点云帧 05: 225~270°扫描范围点云帧 06: 270~315°扫描范围点云帧	1
13-14		块计数	1~65535	2
15-18	时间戳信息	时间戳	范围为 0~3600e6, 单位为 us	4
19		时间戳同步模式	0 : 自由运行模式 ; 1 : 外同步模式	1
20		备用	备用	1
21-22		点云 1 距离	单位为 2mm	2
23		点云 1 强度	0~255	1
24-25		点云 2 距离	单位为 2mm	2
26		点云 2 强度	0~255	1
...		点云 N-1 距离	单位为 2mm	2
...		点云 N-1 强度	0~255	1
18+N*3- 19+N*3		点云 N 距离	单位为 2mm	2
20+N*3		点云 N 强度	0~255	1

一个 UDP 包显然无法承载共 1512 组测量数据的 1 帧点云，所以，将 1 帧点云数据分成了 6 个 block，每个 UDP 包传输 252 (1512/6) 组测量数据。在 SDK 中，提供两种获取点云数据的函数接口。

5.6.1. 获取一个 block 点云数据

```
error_t GrabOneScan(ScanBlockData& scan_block_data); //非阻塞式
```

```
error_t GrabOneScanBlocking(ScanBlockData& scan_block_data); //阻塞式
```

MS500 水平视场角 270°范围的点云数据从水平视场角的开始位置 (45°) 起到结束位置 (315°) 被均匀的分成 6 个数据块, 每块的视场角为 45°, 从起始位置到结束位置依次排序为 1~6。用户可以通过我们 SDK 提供的 API 接口函数获取一块 45°范围视场角的点云数据。

该函数需要用户提供一个 ScanBlockData 类的实例作为参数, 当函数成功返回时, 该实例即包含读取到的扫描数据块的内容。ScanBlockData 类的定义如下:

```
class ScanBlockData
{
public:
    class BlockData
    {
    public:
        std::vector<uint16_t> ranges;
        std::vector<uint8_t> intensities;
    };

    uint8_t block_id;
    uint32_t timestamp;
    uint8_t sync_mode;
    std::vector<BlockData> layers;
};
```

其数据成员均为公共变量, 以方便用户代码访问, 这些成员包括:

- uint8_t block_id: 记录当前数据块的排序号, 占用一个字节, 从起始位置到结束位置依次排序为 1~6;
- uint32_t timestamp: 开始测量数据块内第一组数据时设备内部时间戳计数器的值, 占用 4 个字节。该计数器为 32 位, 最小间隔为 1 微秒;
- uint8_t sync_mode: 标记此数据块采样结束时刻内部时间戳模式, 其取值范围为 0 (自由运行模式), 1 (外同步模式), 占用一个字节;
- std::vector<BlockData> layers: 由 BlockData 构成的列表, 其中每个成员都含有某个扫描平面 45 度范围内的测量数据。BlockData 结构包含以下数据:
 - std::vector<uint16_t> ranges: 当前数据块内测量点的距离值列表, 其单位为 2 毫米, 且各距离值沿逆时针方向依次排列;
 - std::vector<uint8_t> intensities: 当前数据块内测量点的强度值列表, 其中每个值的有效范围为 0 至 255。

5.6.2. 获取一帧点云数据

```
error_t GrabFullScan(ScanFrameData& scan_frame_data); //非阻塞式
error_t GrabFullScanBlocking(ScanFrameData& scan_frame_data); //阻塞式
```

MS500 水平视场角 270°范围的点云数据从水平视场角的开始位置起 (45°) 到结束位置 (315°) 被分成 6 个点云数据块送出, 用户可以通过我们 SDK 提供的 API 接口函数一次性获取 270°完整视场角范围内的点云数据帧。

该函数需要用户提供一个 ScanFrameData 类的实例作为参数, 当函数成功返回时, 该实例即含有读取得到的扫描数据帧的内容。ScanFrameData 类的定义如下:

```
class ScanFrameData
{
public:
```

```
class FrameData
{
public:
    std::vector<uint16_t> ranges;
    std::vector<uint8_t> intensities;
};

uint32_t timestamp;
std::vector<FrameData> layers;
};
```

- uint32_t timestamp : 开始测量数据帧内第一组数据时设备内部时间戳计数器的值，占用 4 个字节。该计数器为 32 位，最小间隔为 1 微秒；
- std::vector<ScanFrameData> layers : 由 ScanFrameData 构成的列表，其中每个成员都含有某个扫描平面 270 度范围内的测量数据。ScanFrameData 结构包含以下数据：
 - std::vector<uint16_t> ranges : 当前数据帧内测量点的距离值列表，其单位为 2 毫米，且各距离值沿逆时针方向依次排列；
 - std::vector<uint8_t> intensities : 当前数据帧内测量点的强度值列表，其中每个值的有效范围为 0 至 255。

5.7. 超时设置

```
void setTimeout(int timeout);
```

为了方便用户使用，SDK 提供的 API 全部使用同步调用的方式，即只有函数执行完毕或发生超时后才会返回；在执行函数的过程中，调用者处于阻塞等待状态。由于计算机与激光雷达之间通过以太网连接，API 的执行时间是无法预测的，因此 SDK 允许用户设置 API 调用的超时值，以灵活处理网络通信延时的不确定性。

参数 int timeout 占用 4 个字节，超时的最小设置单位为 1 毫秒，当 timeout 为 5000 时，表示超时时间设置为 5 秒即 5000 毫秒。如果不设置，系统默认的超时时间为 3000 毫秒。

5.8. 点云滤波功能

```
void point_cloud_filter(uint32_t FilterLevel, uint32_t MotorSpeed,
std::vector<uint16_t>& distances);
```

为了解决点云数据出现拖尾或者飞点问题，SDK 中加入了点云滤波接口 point_cloud_filter，接口参数 FilterLevel 代表滤波等级值，有效范围为 1、2、3、4、5；MotorSpeed 代表雷达电机转速，有效范围值为 10、15、20、25、30，需填入当前雷达实际的转速；distances 代表一圈的点云数据距离值数组。

5.9. 注意事项

5.9.1. 错误提醒

```
enum error_t {
    no_error = 0,
    operation_failure,
    timed_out,
```

```
address_in_use,
md5_check_failure,
unknown
};
```

按协议约定，目前 SDK 提供的上述接口函数提供统一的返回错误信息码。可以通过错误信息码判断协议报文通讯的错误类型。所有接口函数的正确返回都是 no_error。

5.9.2. 接口使用限制

```
virtual error_t open();
virtual void close();
```

open 和 close 接口函数必须成对出现，接口函数 open 调用后，必须先调用 close 才允许再一次 open。

```
bool isOpened() const;
```

如果 open 后，不确定是否已 close，可以通过 isOpened 判断是否已处于 open 状态，如果 isOpened 返回 true 表示当前处于 open 状态，如果返回 false 表示当前处于 close 状态。

6. 快速使用指南

新建 sdk_test 文件目录，将 ord_sdk_ms500 中的 include 和 ord_sdk.a 拷贝到 sdk_test 中，并将 ord_sdk.a 重命名为 libord_sdk.a。

```
ubuntu@ubuntu:~$ mkdir sdk_test
ubuntu@ubuntu:~$ cd sdk_test/
ubuntu@ubuntu:~/sdk_test$ cp -rf ../ord_sdk_ms500/include/ .
ubuntu@ubuntu:~/sdk_test$ cp ../ord_sdk_ms500/build/ord_sdk.a ./libord_sdk.a
ubuntu@ubuntu:~/sdk_test$ vim sdk_demo.cpp
ubuntu@ubuntu:~/sdk_test$ vim Makefile
ubuntu@ubuntu:~/sdk_test$ make
g++ -c -g -std=c++11 sdk_demo.cpp -o sdk_demo.o -I ./include/
g++ sdk_demo.o -o sdk_demo -L . -lord_sdk -lpthread
ubuntu@ubuntu:~/sdk_test$ ls
include libord_sdk.a Makefile sdk_demo sdk_demo.cpp sdk_demo.o
ubuntu@ubuntu:~/sdk_test$
```

新建 sdk_demo.cpp 和 Makefile 文件，就可以编译使用我们提供的 SDK 的 API 接口了。
sdk_demo.cpp :

```
#include <iostream>
#include "ord/ord_driver.h"
#include "ord/lidar_address.h"
#include "ord/ord_types.h"
#include <string.h>
#ifdef __linux__
#include <arpa/inet.h>
#include <unistd.h>
#endif

int main()
```

```
{
    printf("this is oradar lidar sdk test demo \n");
    //ms500 lidar factory default IP:192.168.1.100,Port:2007
    in_addr_t ip_addr = inet_addr("192.168.1.100");
    in_port_t port = htons(2007);
    ord_sdk::LidarAddress sensor(ip_addr, port);
    ord_sdk::OrdDriver drv(sensor);
    ord_sdk::ScanFrameData scan_frame_data;
    if (drv.open() != ord_sdk::no_error){
        std::cerr << "unable to open device" << std::endl;
        return -1;
    }
    if(drv.trackConnect() != ord_sdk::no_error){
        std::cerr << "unable to connect ms500 lidar" << std::endl;
        return -1;
    }
    while (true) {
        if (drv.getScanFrameData(scan_frame_data) == ord_sdk::no_error){
            int count = scan_frame_data.layers[0].ranges.size();
            std::cout << "scan_frame_data count size is " << \
                scan_frame_data.layers[0].ranges.size() << std::endl;
            std::cout << "scan_frame_data.layers.szie = " << \
                scan_frame_data.layers.size() << std::endl;
            std::cout << "scan_frame_data.layers[0].ranges.szie = " << \
                scan_frame_data.layers[0].ranges.size() << std::endl;
            std::cout << "scan_frame_data.layers[0].intensities.szie = " << \
                scan_frame_data.layers[0].intensities.size() << std::endl;
            std::cout << "timestamp is " << scan_frame_data.timestamp << std::endl;
        }else{
            std::cerr << "unable to get point cloud data\n";
            break;
        }
    }
    drv.close();
    return 0;
}
```

Makefile :

```
CC = g++
CFLAGS = -g -std=c++11
SOURCES = $(wildcard *.cpp)
INCLUDE_DIRS = -I ./include/
LIB_DIRS = -L .
ifeq ($(LANG),)
CLIBS = -lord_sdk -lpthread -lws2_32 -lws2_32
else
CLIBS = -lord_sdk -lpthread
```

```
endif

TARGET = sdk_demo
OBJECTS = $(patsubst %.cpp,%.o,$(SOURCES))

$(TARGET):$(OBJECTS)
    $(CC)$@$(LDFLAGS) $^ -o $@ $(LIB_DIRS) $(CLIBS)
$(OBJECTS):%.o:%.cpp
    $(CC) -c $(CFLAGS) $^ -o $@ $(INCLUDE_DIRS)

.PHONY:clean
clean:
    ifeq ($(LANG),)

        del $(TARGET).exe $(OBJECTS)

    else

        rm -rf $(TARGET) $(OBJECTS)

    endif
```

7. 表目录

表 5-1 转速与开火角分辨的关系	10
表 5-2 点云数据通讯协议	13