
ibeo
automotive

ibeo automotive

Introduction to

ibeoSDK Demoprojects

Ibeo Automotive Systems GmbH
Merkurring 60-62
D - 22143 Hamburg
Phone: +49 - (0) 40 298 676 - 0
Fax: +49 - (0) 40 298 676 - 10
E-mail: info@ibeo-as.com

Version History

Date	Version	Changes
19-Feb-2019	1.0	Initial Version of this manual

Table 0.1: Version History

Contents

1 Introduction

After explaining the basic functionality and structure in the IbeoSDK introduction, it is important to see how this knowledge can be applied. This document will show the practical use of the IbeoSDK and explain every single step.

To cover all possible sources of data, we will use the 3 demos *GeneralFileDemo*, *FileDemo* and *WriterDemo*.

The *GeneralFileDemo* and *FileDemo* processes data from IDC Files.

The *WriterDemo* shows the connection to an IbeoDevice and the processing of that live data into an IDC File.

2 GeneralFileDemo

The GeneralFileDemo is a demo project for reading IDC files and process the DataContainers with general data types.

2.1 Arguments

The following program arguments shows how to start the GeneralFileDemo properly:

```
.../IbeoSdkGeneralFileDemo.cpp InputFileName LogFile
```

Argument	Purpose	Example
InputFileName	Name of the IDC File to use as input.	.../TestInputFile.idc
LogFile	Name of the log file (optional).	.../TestInputLogFile.log

2.2 Source

2.2.1 Version

```
1  int main(const int argc, const char** argv)
2  {
3      std::cerr << argv[0] << " (" << appName << ")"
4                << " Version " << appVersion.toString();
5      std::cerr << "  using IbeoSDK " << ibeoSDK.getVersion().toString() << std::endl;
```

The first part of the source code prints the current IbeoSDK version.

2.2.2 Parameter check and parsing

```
6      bool hasLogFile;
7      const int checkResult = checkArguments(argc, argv, hasLogFile);
8      if (checkResult != 0)
9          exit(checkResult);
10     int currArg = 1;
```

The next step is to check the parsed program argument syntax and whether the file has a log file or not.

```
11     std::string filename = argv[currArg++];
```

Assigns the first argument to the string filename and increments the current argument.

2.2.3 Logging

The IbeoSDK offers a special framework, the IbeoSDK Logging. It allows to log every debug, warning and error message coming from the SDK itself in a specified log file. This is a excellent source of information to understand the system and analyze any upcoming issues. For further information regarding the IbeoSDK Logging: See IbeoSDK Logging manual.

```
12      if (hasLogFile)
13      {
14          ibeo::common::logging::FileLoggerBackendSPtr fileLoggerBackend
15              = std::dynamic_pointer_cast<ibeo::common::logging::FileLoggerBackend>(
16                  ibeo::common::logging::LogManager::getInstance().getBackendById(
17                      ibeo::common::logging::FileLoggerBackend::getBackendId()));
18          fileLoggerBackend->setFilePath(argv[currArg++]);
19      }
```

The logging system itself provides two backends which can be used by loggers: console logging and file logging. If a log file has been parsed, it sets the backend to log to a file.

```
20      ibeo::common::logging::LogManager::getInstance().setDefaultLogLevel("Debug");
```

Sets the default log level to debug, which is the least important level.

```
21      if (hasLogFile)
22      {
23          LOGINFO(logger,
24                  argv[0] << " Version " << appVersion.toString() << "  using IbeoSDK "
25                          << ibeoSDK.getVersion().toString());
26      }
```

If a log file has been parsed, then it writes the current IbeoSDK version into it.

```
27      generalFileDemoContainer(filename);
28  }
```

Calls the generalFileDemoContainer method with the filename parameter.

2.2.4 Listener

There are two different specializations of DataContainerListeners. **GeneralDataContainerListeners** and **DataContainerListener**.

DataContainerListeners are defined by giving a DataContainer and a DataTypeID of a serialization.

GeneralDataContainerListener are defined by giving only a general DataContainer.

The general DataContainer can lose the preciseness of some data within or even lose the data completely due to the conversion. The user has to decide for himself which information content makes more sense for the respective task that has to be fulfilled.

For each DataContainer mentioned in one of the DataContainerListeners specializations, an onData method has to be implemented with this DataContainer type as parameter. This onData method will be called with a filled DataContainer instance if the Device has filled out a container of this DataType.

2.2.5 Reading and processing data out of an IDC File

```
1  void generalFileDemoContainer(const std::string& filename)
2  {
3      IdcFile file;
4      file.open(filename);
```

Instantiates file as IdcFile and opens the IDC file(that previously has been assigned to filename) for read access.

```
5      if (file.isOpen())
6      {
7          AllListener allListener;
8          file.registerContainerListener(&allListener);
```

If the file has been successfully opened, an AllListener will be instantiated and registered to the IDC file. In this case the Listeners are implemented as general DataContainer for a DataTypeID(e.g. DataContainerListener<Scan, DataTypeId::DataType_Scan2202>).

```
9          uint32_t nbOfMessages = 0; //# of messages we parsed
```

Instantiates the number of parsed messages with a 0.

```
10         while (file.isGood())
11         {
12             const IdcFile::ContainerImporterPairs p = file.getNextDataContainer();
13             file.notifyContainerListeners(p);
14             ++nbOfMessages;
15             ...
16         }
17     }
18 }
```

The code block will be executed repeatedly until an error bit is set.

- **Line 12**: Reads and assigns the next DataContainer as **ContainerImporterPair**. The **ContainerImporterPair** is a vector with pairs of pointer to the filled DataContainers and the Importer used to fill them.
- **Line 13**: Calls all registered ContainerListener listening to the combination of DataContainer and Import for all pairs in the given vector.
- **Line 14**: Increments the number of parsed messages.

```
1  void onData(const VehicleState& vs) override
2  {
3      ...
4      LOGINFO(logger, "Vehicle state container received: ");
5      LOGINFO(logger, "  position: " << vs.getRelativeXPosition() << "/" << vs.getRelativeYPosition());
6      LOGINFO(logger, "  yaw rate: " << vs.getYawRate());
7      LOGINFO(logger, "  speed: " << vs.getLongitudinalVelocity());
8      LOGINFO(logger, "=======================================================");
9      ...
10 }
```

In this example the method parameters indicates that a general DataContainer VehicleState is being used for every serialization (Can be checked in the AllListener). So whenever the next DataContainer is a VehicleState2805, VehicleState2806, VehicleState2807 or VehicleState2808 it will log the position(line 2) and yaw rate (line 3) and speed(line 4).

3 FileDemo

The FileDemo is a demo project for reading IDC files and process the DataContainers with special data types.

3.1 Arguments

The following program arguments shows how to start the FileDemo properly:

```
.../IbeoSdkFileDemo.cpp InputFileName LogFile
```

Argument	Purpose	Example
InputFileName	Name of the IDC File to use as input.	.../TestInputFile.idc
LogFile	Name of the log file (optional).	.../TestInputLogFile.log

3.2 Source

3.2.1 Version

```
1  int main(const int argc, const char** argv)
2  {
3      std::cerr << argv[0] << " (" << appName << ")"
4                << " Version " << appVersion.toString();
5      std::cerr << "  using IbeoSDK " << ibeoSDK.getVersion().toString() << std::endl;
```

The first part of the source code prints the current IbeoSDK version.

3.2.2 Parameter check and parsing

The next step is to check the parsed program argument syntax and whether the file has a log file or not.

```
6      std::string filename = argv[currArg++];
```

Assigns the first argument to the string filename and increments the current argument.

3.2.3 Logging

The IbeoSDK offers a special framework, the IbeoSDK Logging. It allows to log every debug, warning and error message coming from the SDK itself in a specified log file. This is a excellent source of information to understand the system and analyze any upcoming issues. For further information regarding the IbeoSDK Logging: See IbeoSDK Logging manual.

```
7      if (hasLogFile)
8      {
9          ibeo::common::logging::FileLoggerBackendSPtr fileLoggerBackend
10         = std::dynamic_pointer_cast<ibeo::common::logging::FileLoggerBackend>(
11             ibeo::common::logging::LogManager::getInstance().getBackendById(
```

```
12                  ibeo::common::logging::FileLoggerBackend::getBackendId()));
13          fileLoggerBackend->setFilePath(argv[currArg++]);
14      }
```

The logging system itself provides two backends which can be used by loggers: console logging and file logging. If a log file has been parsed, it sets the backend to log to a file.

```
15      ibeo::common::logging::LogManager::getInstance().setDefaultLogLevel("Debug");
```

Sets the default log level to debug, which is the least important level.

```
16      if (hasLogFile)
17      {
18          LOGINFO(logger,
19                  argv[0] << " Version " << appVersion.toString() << "  using IbeoSDK "
20                          << ibeoSDK.getVersion().toString());
21      }
```

If a log file has been parsed, then it writes the current IbeoSDK version into it.

```
22      file_demo_container(filename);
23  }
```

Calls the file_demo_container method with the filename parameter.

3.2.4 Listener

There are two different specializations of DataContainerListeners. **GeneralDataContainerListeners** and **DataContainerListener**.

DataContainerListeners are defined by giving a DataContainer and a DataTypeID of a serialization.

GeneralDataContainerListener are defined by giving only a general DataContainer.

The general DataContainer can lose the preciseness of some data within or even lose the data completely due to the conversion. The user has to decide for himself which information content makes more sense for the respective task that has to be fulfilled.

For each DataContainer mentioned in one of the DataContainerListeners specializations, an onData method has to be implemented with this DataContainer type as parameter. This onData method will be called with a filled DataContainer instance if the Device has filled out a container of this DataType.

3.2.5 Reading and processing data out of an IDC File

```
1  void file_demo_container(const std::string& filename)
2  {
3      IdcFile file;
4      file.open(filename);
```

Instantiates file as IdcFile and opens the IDC file(that previously has been assigned to filename) for read access.

```
5      if (file.isOpen())
6      {
7          AllListener allListener;
8          file.registerContainerListener(&allListener);
```

If the file has been successfully opened, an AllListener will be instantiated and registered to the IDC file. In this case the Listeners are implemented as special DataContainer for a DataTypeID(e.g. DataContainerListener<Scan2202, DataTypeId::DataType_Scan2202>).

```
9          uint32_t nbOfMessages = 0; // # of messages we parsed
```

Instantiates the number of parsed messages with a 0.

```
10         while (file.isGood())
11         {
12             const IdcFile::ContainerImporterPairs p = file.getNextDataContainer();
13             file.notifyContainerListeners(p);
14             ++nbOfMessages;
15             ...
16         }
17     }
18 }
```

The code block will be executed repeatedly until an error bit is set.

- **Line 12**: Reads and assigns the next DataContainer as **ContainerImporterPair**. The **ContainerImporterPair** is a vector with pairs of pointer to the filled DataContainers and the Importer used to fill them.
- **Line 13**: Calls all registered ContainerListener listening to the combination of DataContainer and Import for all pairs in the given vector.
- **Line 14**: Increments the number of parsed messages.

```
1  void onData(const VehicleState2808& vs) override
2  {
3      LOGTRACE(logger, "VS (0x2808) container received: time: "
4                       << tc.toString(vs.getTimestamp().toPtime()));
5      LOGTRACE(logger, "=======================================================");
6  }
```

In this example the method parameters indicates that a special DataContainer VehicleState2808 is being used for the 0x2808 serialization. So whenever the next DataContainer is a VehicleState2808 it will log the timestamp at trace level(line 1).

4 WriterDemo

The WriterDemo project is for connecting to a IbeoDevice and process the received DataContaienrs.

4.1 Arguments

The following program arguments shows how to start the WriterDemo properly:

```
.../IbeoSdkWriterDemo.cpp -e Ip OutputFileName LogFile
```

Argument	Purpose	Example
-e	If this parameter is given and the port is set to 12002. The sensor is an ECU instead of a Lux.	-e
Ip	Ip address and port of the IbeoDevice.	192.168.0.1:12004
OutputFileName	Name of the IDC File to use as out.	TestOutputFile
LogFile	Name of the log file (optional).	.../TestInputLogFile.log

4.2 Source

4.2.1 Version

```
1  int main(const int argc, const char** argv)
2  {
3      std::cerr << argv[0] << " (" << appName << ")"
4                << " Version " << appVersion.toString();
5      std::cerr << "  using IbeoSDK " << ibeoSDK.getVersion().toString() << std::endl;
```

The first part of the source code prints the current IbeoSDK version.

4.2.2 Parameter check and parsing

```
6      bool useEcu;
7      bool hasLogFile;
8      const int checkResult = checkArguments(argc, argv, useEcu, hasLogFile);
9      if (checkResult != 0)
10     {
11         exit(checkResult);
12     }
```

The next step is to check the parsed program argument syntax and whether the file has a log file or not.

```
13     int currArg = 1 + (useEcu ? 1 : 0);
```

Instantiates currArg(short for current argument) as 1 if there is no ECU or 2 if there is an ECU.

```
14     std::string ip                = argv[currArg++];
15     const uint16_t port           = getPort(ip, 12002);
16     const std::string outFilename = argv[currArg++];
```

Line 15: Sets the IP address as string.

Line 16: Sets the port as uint16_t.

Line 17: Sets the outputfilename as string.

4.2.3 Logging

The IbeoSDK offers a special framework, the IbeoSDK Logging. It allows to log every debug, warning and error message coming from the SDK itself in a specified log file. This is a excellent source of information to understand the system and analyze any upcoming issues. For further information regarding the IbeoSDK Logging: See IbeoSDK Logging manual.

```
17      if (hasLogFile)
18      {
19          ibeo::common::logging::FileLoggerBackendSPtr fileLoggerBackend
20              = std::dynamic_pointer_cast<ibeo::common::logging::FileLoggerBackend>(
21                  ibeo::common::logging::LogManager::getInstance().getBackendById(
22                      ibeo::common::logging::FileLoggerBackend::getBackendId()));
23          fileLoggerBackend->setFilePath(argv[currArg++]);
24      }
```

The logging system itself provides two backends which can be used by loggers: console logging and file logging. If a log file has been parsed, it sets the backend to log to a file.

```
25      ibeo::common::logging::LogManager::getInstance().setDefaultLogLevel("Debug");
```

Sets the default log level to debug, which is the least important level.

```
26      if (hasLogFile)
27      {
28          LOGINFO(logger,
29                  argv[0] << " Version " << appVersion.toString() << "  using IbeoSDK "
30                          << ibeoSDK.getVersion().toString());
31      }
```

If a log file has been parsed, then it writes the current IbeoSDK version into it.

4.2.4 Selecting a device

```
32      switch (port)
33      {
34          case 12002:
35              if (useEcu)
36              {
37                  idcWriterEcu_demo(ip, outFilename);
38              }
39              else
40              {
41                  idcWriterLux_demo(ip, outFilename);
42              }
43              break;
44          case 12004:
45              idcWriterScala_demo(ip, outFilename);
46              break;
47          case 12006:
48              idcWriterMiniLux_demo(ip, outFilename);
49              break;
50          case 12008:
51              idcWriterLuxHr_demo(ip, outFilename);
52              break;
53          default:
54              LOGERROR(appLogger, "Port " << port << " is not supported by any known device.");
55              break;
56      }
57  }
```

Depending on the parsed port the program will call the specific idcWriter method with the ip and output file name parameters.

Port 12002: Can either be ECU or Lux

Port 12004: Scala

Port 12006: MiniLux

Port 12008: LuxHr

Anything else: Error

4.2.5 Streamer

IdcWriter is a streamer class within the WriterDemo that allows to create and write the received data from a device into an IDC File. Streamers are classes whose data is derived from DataStreamer. They have to implement the onData method, which needs a IbeoDataHeader and a byte array (char*) as arguments.
Streamer can be used for forwarding DataTypes from the Device to a destination that does not need deserialized DataContainers. In our demo the destination is an IDC File.

4.2.6 Connecting to a device and process data

```
1  void idcWriterEcu_demo(const std::string& ip, const std::string& outFilename)
2  {
3      IbeoEcu ecu(ip);
```

Instanties an ECU with the given ip address.

```
4      IdcWriter idcWriter(outFilename, 1000000000);
5      if (idcWriter.isOpen())
6      {
7          ecu.registerStreamer(&idcWriter);
8      }
9      ecu.getConnected();
10 }
```

- **Line 4**: Creates an IdcWriter with the output file name and file size limit as parameters.
- **Line 7**: If the IDC File is open, registers the IdcWriter as streamer.
- **Line 8**: Establish the connection to the hardware.

A device in the Ibeo SDK receives a stream of data packets from the sensor and converts them to an Ibeo SDK DataContainer. If multiple packets are needed, the device is responsible for collecting and storing these packets before creating a data container.
The onData method of a streamer class gets an Ibeo data header and a plain byte buffer, that means that the received DataType is not serialized.

NOTE

The following code is just to show what can be done with the onData method of a streamer class. There is no particular reason why some DataTypes are treated different than other DataTypes.

```
1  void IdcWriter::onData(const IbeoDataHeader& dh, const char* const bodyBuf)
2  {
3      LOGDEBUG(appLogger, "Received message 0x" << std::hex << dh.getDataType() << std::dec);
4      switch (dh.getDataType())
5      {
6          case DataTypeId::DataType_Scan2202:
7          case DataTypeId::DataType_Scan2205:
8          case DataTypeId::DataType_Scan2208:
9              this->splitFileIfNeeded();
10             break;
11         case DataTypeId::DataType_IdcTrailer6120:
12         case DataTypeId::DataType_FrameIndex6130:
13             LOGDEBUG(appLogger, "Skip writing datatype " << toHex(dh.getDataType()));
14             return;
15         default:
16         break;
17     }
```

- **Line 3**: Logs at debug level that a DataType has been received.
- **Line 4**: Switch case based on the received DataType
- **Line 6-8**: In case of a scan (0x2202, 0x2205, or 0x2208), the file will be split.
- **Line 11 and 12**: In case of a trailer (0x6120) or frame index (0x6130), there will be an debug log about skipping it and after that it will leave the onData method.