Data Reader Implementation

Contents

Problem Definition

"

The problem to solve is following:

You monitor devices, which are sending data to you.
Each device have a unique name.
Each device produces measurements.

The challenge is:

Compute number of messages you got or read from the devices.

The main outcome of your task will describe the concept / architecture of proposed solution (UML, documentation, work split, …).
The scope is open, you must decide how the "devices" will work in your system.
The solution should be posted on GitHub or a similar page for a review.
The solution should contain proof of concept (preferably in C++).
Please add documentation explaining us how to run your code

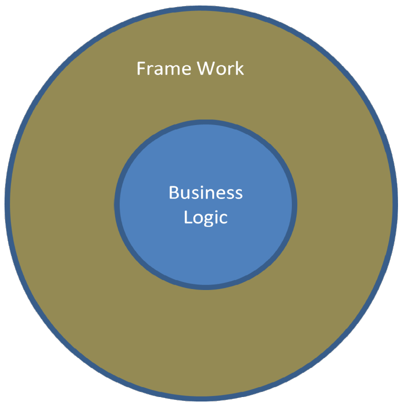
"

Conclusion due to the problem definition

- Each device might have different protocol, any available protocol shall be implementable [point to point, one to many network topologies might be possible]
- The "deviceID" is guaranteed to be unique. So can be used as key in any hash map.
- Each device produces measurements. I am not sure about the purpose of this explanation. As it does not matter what the data is read for, it is a stream of bytes. It can be a transfer of a single byte or 2048 byte data structure transfer. The read data can be forwarded to any parser.
- This document is to present the overall architecture and class diagrams, …

Overall Architecture

For basic SOLID principles the business logic shall be the highest layer, and it shall not depend on any frame-work layer elements.



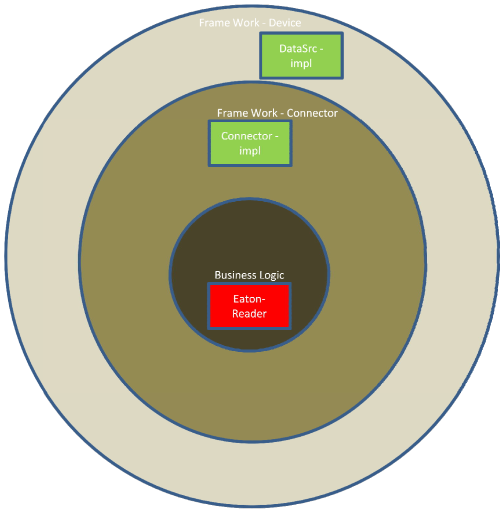
In this example the business logic is: to traverse through devices and to record num of readings.

It is important to detect points of change and handle them in design. Following list the points of change;

1. Business logic: instead of counting number of reads any other algorithm may be required to run on read data
2. Instead of read write might be required in the system
3. Adding new devices shall be easy [extensible]
4. Adding new Connectors shall be easy [extensible]

So at the business level interfaces that will be used through framework will be defined. These interfaces will be used for “Dependency Inversion”. No dependency shall exist on concrete classes. No dependency to FrameWork elements shall occur in Business Logic.

Levels of relations resembles onion and it looks like below:



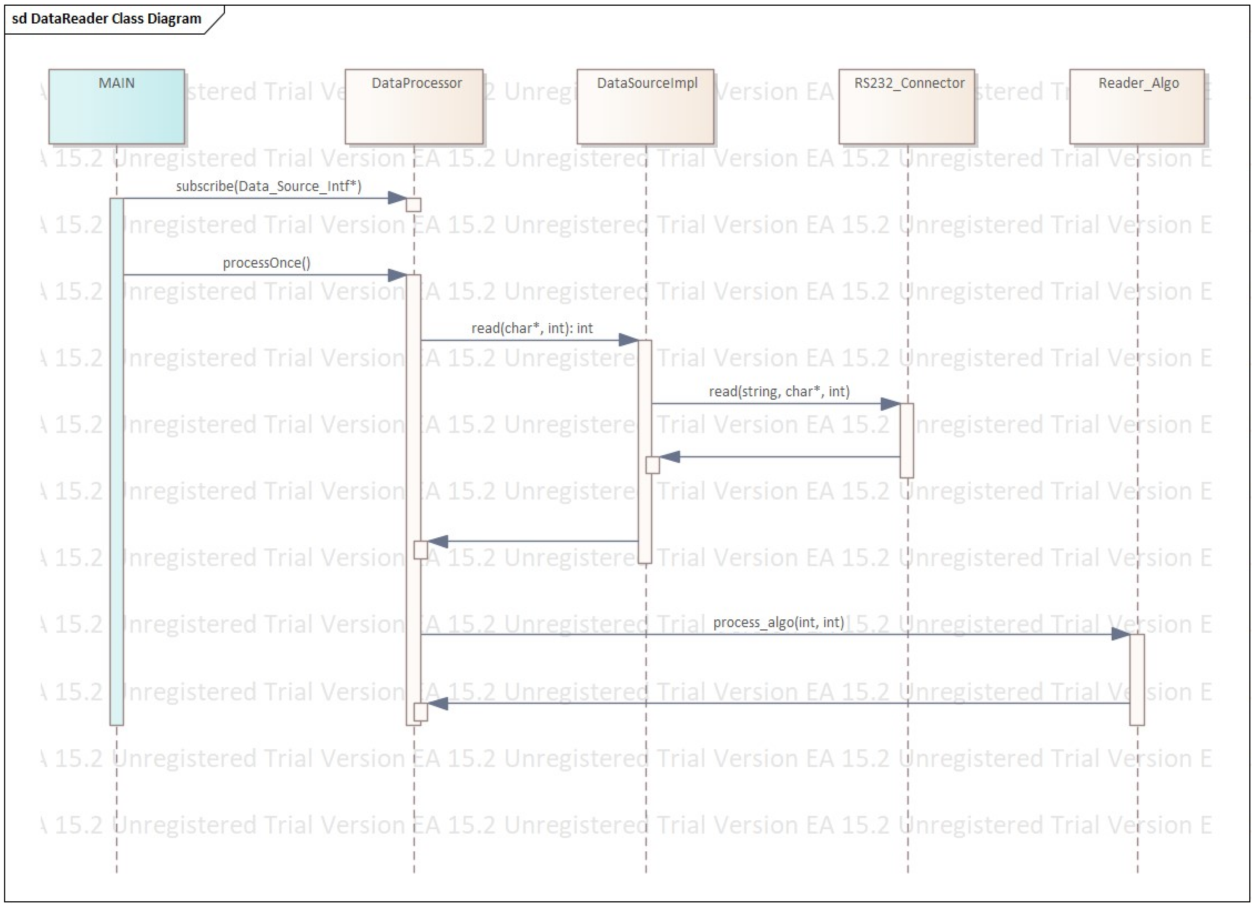
Here there is one core and 2 layers of frame work.

In the eaton_reader core layer there are 5 different interface definitions. Brief explanation about each is given below:

- **Algo_Intf:** this interface defines the algorithm that will be used by “eaton_reader”.
- **Connector_Intf:** this interface defines the interface that is to be implemented by connector classes. Connector classes are location where low level details of a connection are handled. It might have different implementations. For example
  - An RS232 connection is a point to point connection. And it is convenient to perform read operation in the context of caller thread. When a read method is invoked the related drivers read implementation will be called and retrieved data can be returned to caller.
  - An RS485 is one-to-many connection. Data will come from different devices in rs485 network sources. In this case the Connector implementation might have a thread that reads the connection and pushes collected data to separate fifos for each device. And when “eaton_reader” calls for a read on RS485 device Connector shall return already collected data to “eaton_reader”.
- **Data_Source_Intf:** this interface defines the devices that can answer read calls by “eaton_reader”. It is easy to add a new interface named “Data_Destination_Intf” and add *write* capability through.
- **Processor_Intf:** this is the interface where processing implementation is provided.
- **Subscription_Intf:** this is the interface which enables “eaton_reader” to accept devices for processor. Each device shall subscribe to processor and then when processor runs it will be read and evaluated by the algorithm assigned.

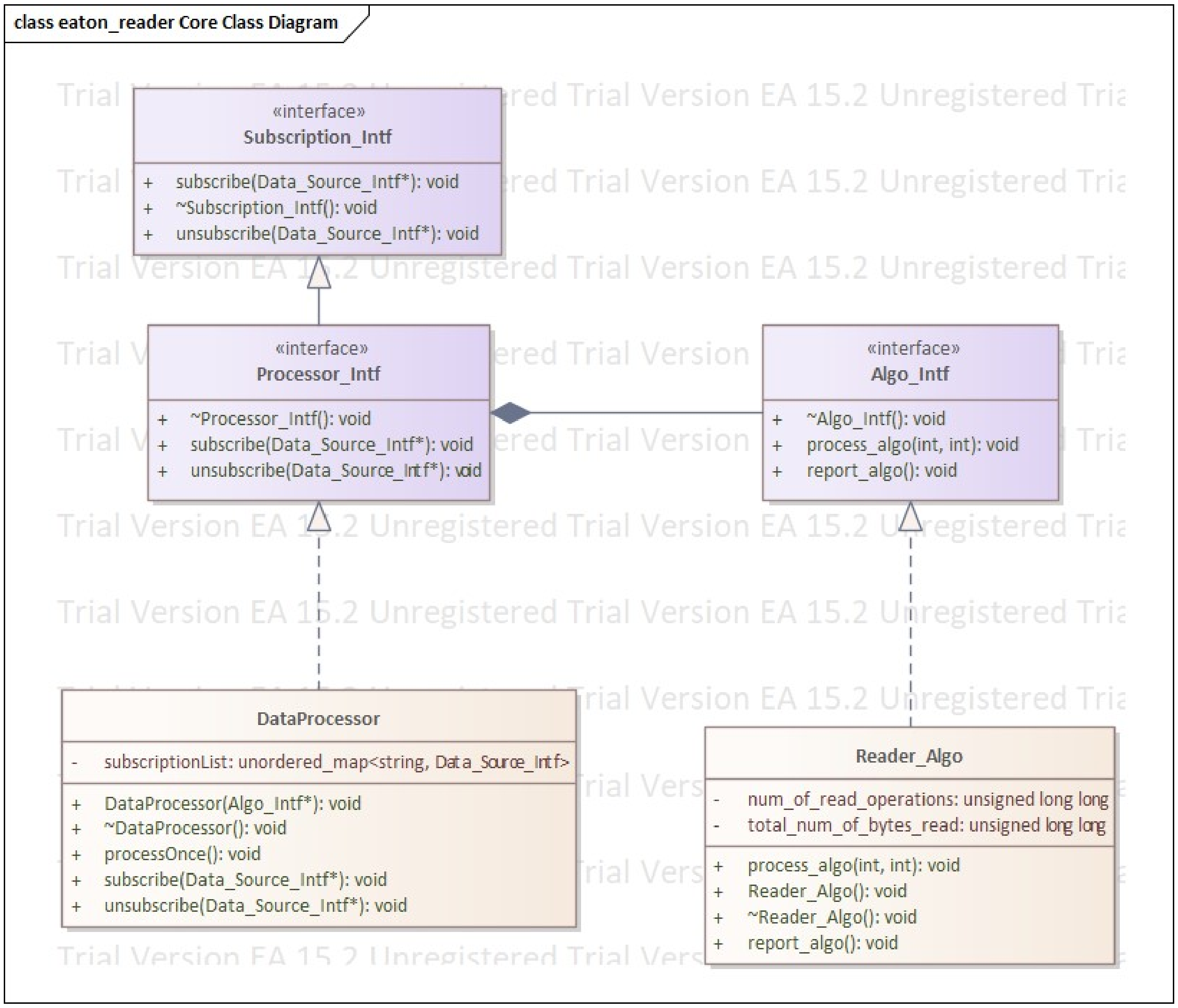
Sequence Diagrams

Basic flow is given for one data source.



Core Class Diagram

Core Layer Class Diagram can be found below.


class eaton_reader Core Class Diagram
«interface»
Subscription_Intf
+ subscribe(Data_Source_Intf*): void
+ ~Subscription_Intf(): void
+ unsubscribe(Data_Source_Intf*): void
«interface»
Processor_Intf
+ ~Processor_Intf(): void
+ subscribe(Data_Source_Intf*): void
+ unsubscribe(Data_Source_Intf*): void
«interface»
Algo_Intf
+ ~Algo_Intf(): void
+ process_algo(int, int): void
+ report_algo(): void
DataProcessor
- subscriptionList: unordered_map<string, Data_Source_Intf>
+ DataProcessor(Algo_Intf*): void
+ ~DataProcessor(): void
+ processOnce(): void
+ subscribe(Data_Source_Intf*): void
+ unsubscribe(Data_Source_Intf*): void
Reader_Algo
- num_of_read_operations: unsigned long long
- total_num_of_bytes_read: unsigned long long
+ process_algo(int, int): void
+ Reader_Algo(): void
+ ~Reader_Algo(): void
+ report_algo(): void

DataProcessor class is the heart of "eaton_reader". Every device is to subscribe to this class. Each DataProcessor shall have an algorithm that is assigned during initialization.

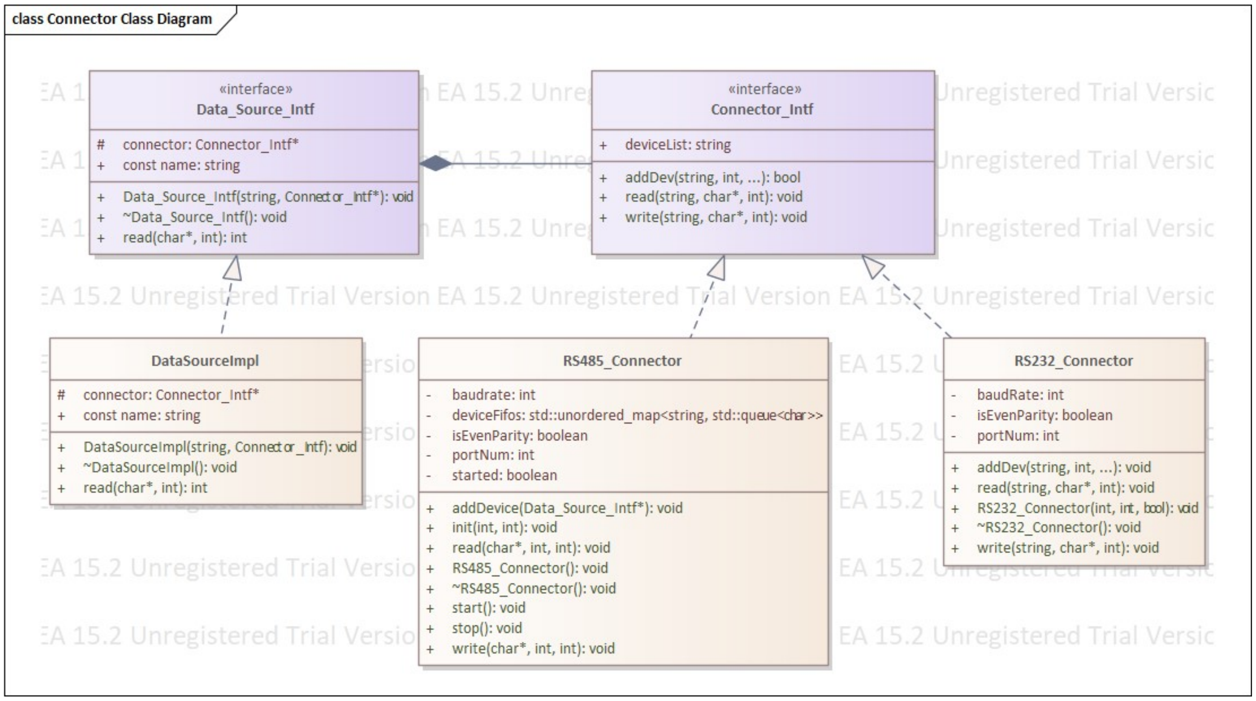
DataProcessor implementation in this case is to be called from an external thread for execution [that is read data from each source and process algo on it].

Also "DataProcessor" class takes an Algo_Intf*. It is a method for **dependency injection**. DataProcessor depends on Algo_Intf. Through dependency injection, unit test writing is quite simplified.

Dependency injection is used throughout the design.

Frame-work Class Diagram

Frame-work class diagram is given below:


class Connector Class Diagram
«interface»
Data_Source_Intf
# connector: Connector_Intf*
+ const name: string
+ Data_Source_Intf(string, Connector_Intf*): void
+ ~Data_Source_Intf(): void
+ read(char*, int): int
«interface»
Connector_Intf
+ deviceList: string
+ addDev(string, int, ...): bool
+ read(string, char*, int): void
+ write(string, char*, int): void
DataSourceImpl
# connector: Connector_Intf*
+ const name: string
+ DataSourceImpl(string, Connector_Intf): void
+ ~DataSourceImpl(): void
+ read(char*, int): int
RS485_Connector
- baudrate: int
- deviceFifos: std::unordered_map<string, std::queue<char>>
- isEvenParity: boolean
- portNum: int
- started: boolean
+ addDevice(Data_Source_Intf*): void
+ init(int, int): void
+ read(char*, int, int): void
+ RS485_Connector(): void
+ ~RS485_Connector(): void
+ start(): void
+ stop(): void
+ write(char*, int, int): void
RS232_Connector
- baudRate: int
- isEvenParity: boolean
- portNum: int
+ addDev(string, int, ...): void
+ read(string, char*, int): void
+ RS232_Connector(int, int, bool): void
+ ~RS232_Connector(): void
+ write(string, char*, int): void

“DataSourceImpl” class is an implementation of a data source, namely *device*, according to naming of the problem definition. In the actual proof of concept there are many different versions of DataSourceImpl is implemented. As each device might have different requirements.

Connector is the class which holds low level connectivity code, driver calls, physical config for the connection.

In its implementation it is even possible to keep fifo for each device connected through the connector.

In RS485_Connector deviceFifo and start/stop operations are defined to emphasize that a thread might run to collect data from each device and write it to respective fifo. And when “eaton_reader” reads data for a device, its fifo content will be returned. But this implementation is skipped. As this much detail adds much work for a proof of concept.

Proof of Concept

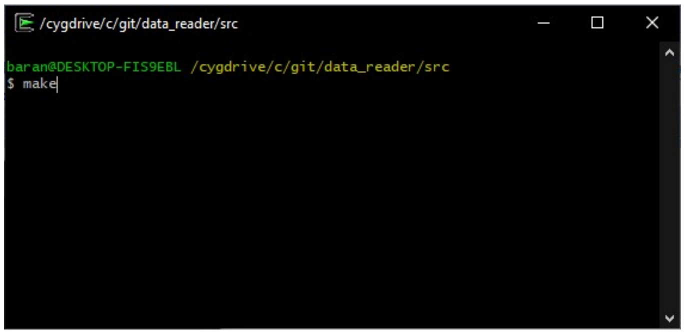
Tools and Environment

Proof of concept is developed in a Windows 10 machine using **cygwin**. No IDE is used, only vi and gcc and make utility is used as tool.

Build

The project is both built in windows 10 – cygwin and Ubuntu 20.04 LTS.

Project can be built with the following command in "*src*" folder:



```
/cygdrive/c/git/data_reader/src

baran@DESKTOP-FIS9EBL /cygdrive/c/git/data_reader/src
$ make
```

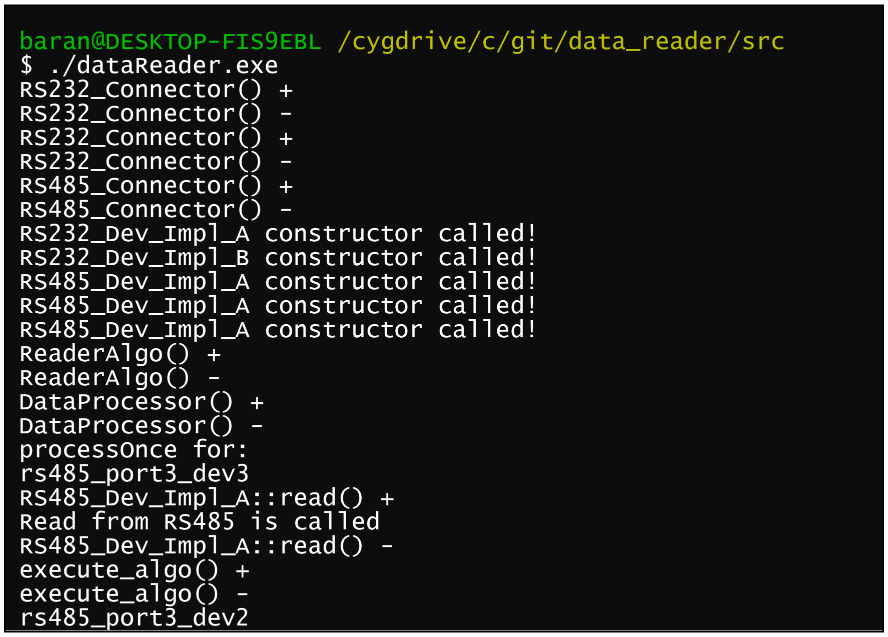
Test

The output of the make command is executed in both windows 10 and Ubuntu 20.04 LTS separately.

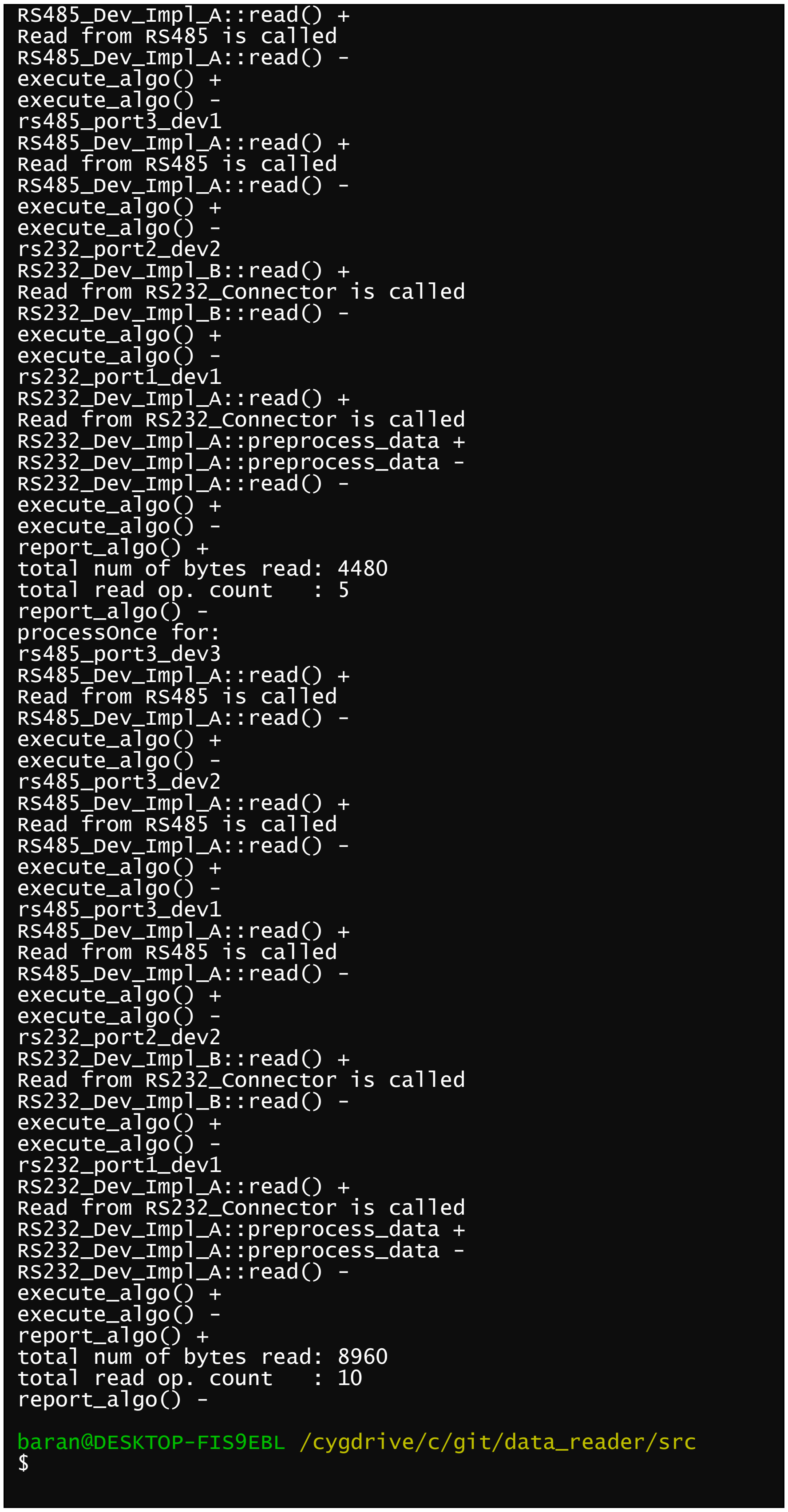
Some log outputs are left on, to trace program execution.

Sample of screen shots are given below:

For cygwin:



```
baran@DESKTOP-FIS9EBL /cygdrive/c/git/data_reader/src
$ ./dataReader.exe
RS232_Connector() +
RS232_Connector() -
RS232_Connector() +
RS232_Connector() -
RS485_Connector() +
RS485_Connector() -
RS232_Dev_Impl_A constructor called!
RS232_Dev_Impl_B constructor called!
RS485_Dev_Impl_A constructor called!
RS485_Dev_Impl_A constructor called!
RS485_Dev_Impl_A constructor called!
ReaderAlgo() +
ReaderAlgo() -
DataProcessor() +
DataProcessor() -
processOnce for:
rs485_port3_dev3
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev2
```

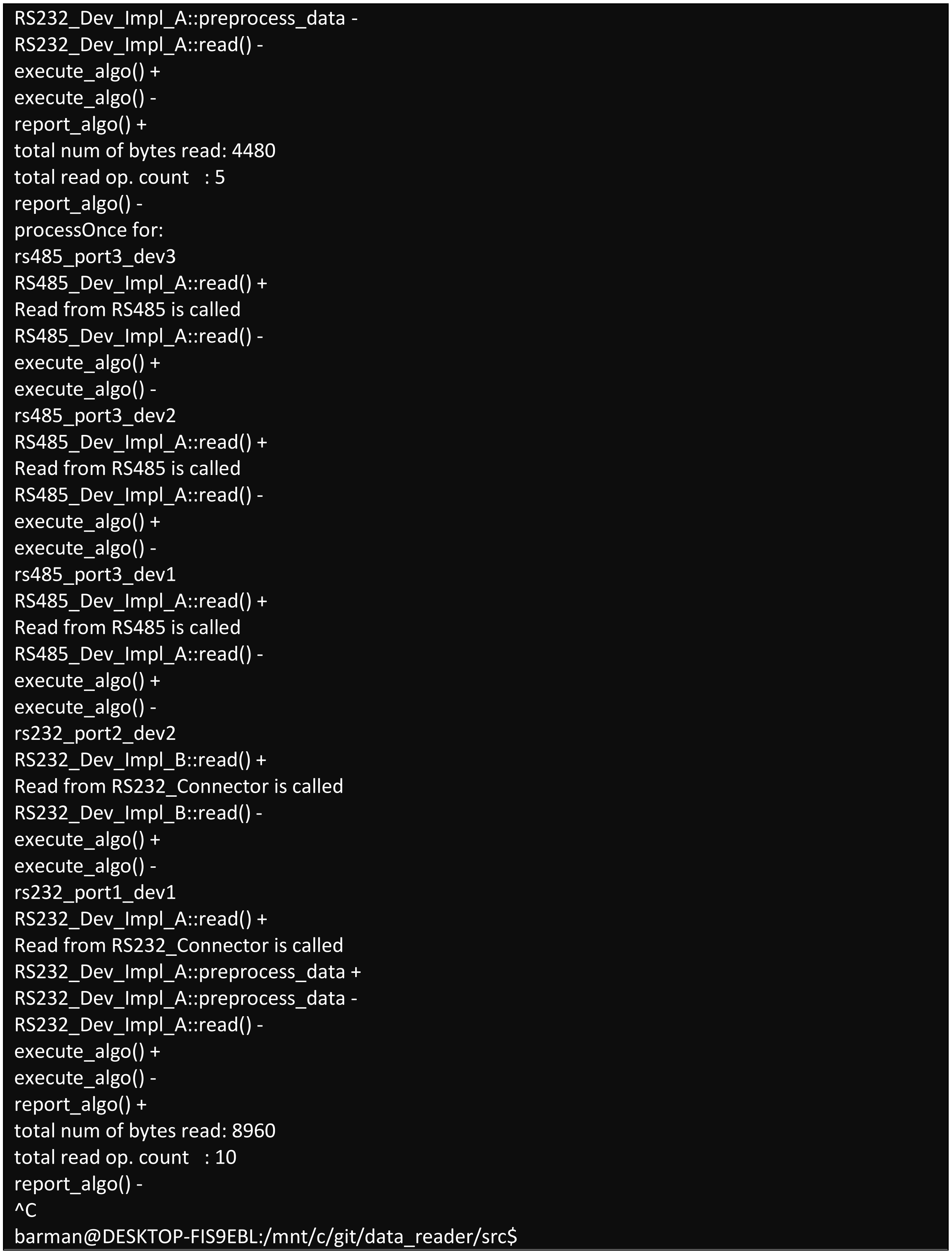


```
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev1
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs232_port2_dev2
RS232_Dev_Impl_B::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_B::read() -
execute_algo() +
execute_algo() -
rs232_port1_dev1
RS232_Dev_Impl_A::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_A::preprocess_data +
RS232_Dev_Impl_A::preprocess_data -
RS232_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
report_algo() +
total num of bytes read: 4480
total read op. count    : 5
report_algo() -
processOnce for:
rs485_port3_dev3
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev2
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev1
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs232_port2_dev2
RS232_Dev_Impl_B::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_B::read() -
execute_algo() +
execute_algo() -
rs232_port1_dev1
RS232_Dev_Impl_A::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_A::preprocess_data +
RS232_Dev_Impl_A::preprocess_data -
RS232_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
report_algo() +
total num of bytes read: 8960
total read op. count    : 10
report_algo() -

baran@DESKTOP-FIS9EBL /cygdrive/c/git/data_reader/src
$
```

For UBUNTU

```
barman@DESKTOP-FIS9EBL:/mnt/c/git/data_reader/src$
./dataReader.exe
RS232_Connector() +
RS232_Connector() -
RS232_Connector() +
RS232_Connector() -
RS485_Connector() +
RS485_Connector() -
RS232_Dev_Impl_A constructor called!
RS232_Dev_Impl_B constructor called!
RS485_Dev_Impl_A constructor called!
RS485_Dev_Impl_A constructor called!
RS485_Dev_Impl_A constructor called!
ReaderAlgo() +
ReaderAlgo() -
DataProcessor() +
DataProcessor() -
processOnce for:
rs485_port3_dev3
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev2
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev1
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs232_port2_dev2
RS232_Dev_Impl_B::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_B::read() -
execute_algo() +
execute_algo() -
rs232_port1_dev1
RS232_Dev_Impl_A::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_A::preprocess_data +
```



```
RS232_Dev_Impl_A::preprocess_data -
RS232_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
report_algo() +
total num of bytes read: 4480
total read op. count   : 5
report_algo() -
processOnce for:
rs485_port3_dev3
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev2
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs485_port3_dev1
RS485_Dev_Impl_A::read() +
Read from RS485 is called
RS485_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
rs232_port2_dev2
RS232_Dev_Impl_B::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_B::read() -
execute_algo() +
execute_algo() -
rs232_port1_dev1
RS232_Dev_Impl_A::read() +
Read from RS232_Connector is called
RS232_Dev_Impl_A::preprocess_data +
RS232_Dev_Impl_A::preprocess_data -
RS232_Dev_Impl_A::read() -
execute_algo() +
execute_algo() -
report_algo() +
total num of bytes read: 8960
total read op. count   : 10
report_algo() -
^C
barman@DESKTOP-FIS9EBL:/mnt/c/git/data_reader/src$
```