Gateway Refactoring

Learning focus:
Class Design, Class Diagrams, Sequence Diagrams, Refactoring, Configuration, Error Handling

Your task is to refactor a first version of a network gateway application.

Marking

Project Analysis	2 points
Refactoring driver	4 points
Adding configuration and error handling	4 points

You need 4 points in order to pass the lab!

Upload the code in Moodle and bring a handwritten solution together with your code to the lab.

Structure will be more important for marking than behavior!

1 Analysis

1.1 Requirements

Your task is to develop a gateway application, which reads data from a port and sends it out through another port. In between, the data is packaged according the port requirements. Let's assume the receiver port is a UART (Packagesize 1) and the sender is a CAN (Packagesize 8)

- The data is received bytewise (== packagesize 1) from the UART (in the rx_isr)
- All bytes are stored in the UART port RX ringbuffer
- The data is read out by the gateway transmit method
- And stored in the CAN port TX ringbuffer
- The port TX isr reads out the data from the ringbuffer, creates packages (packagesize 8) and sends the packages to the hardware driver

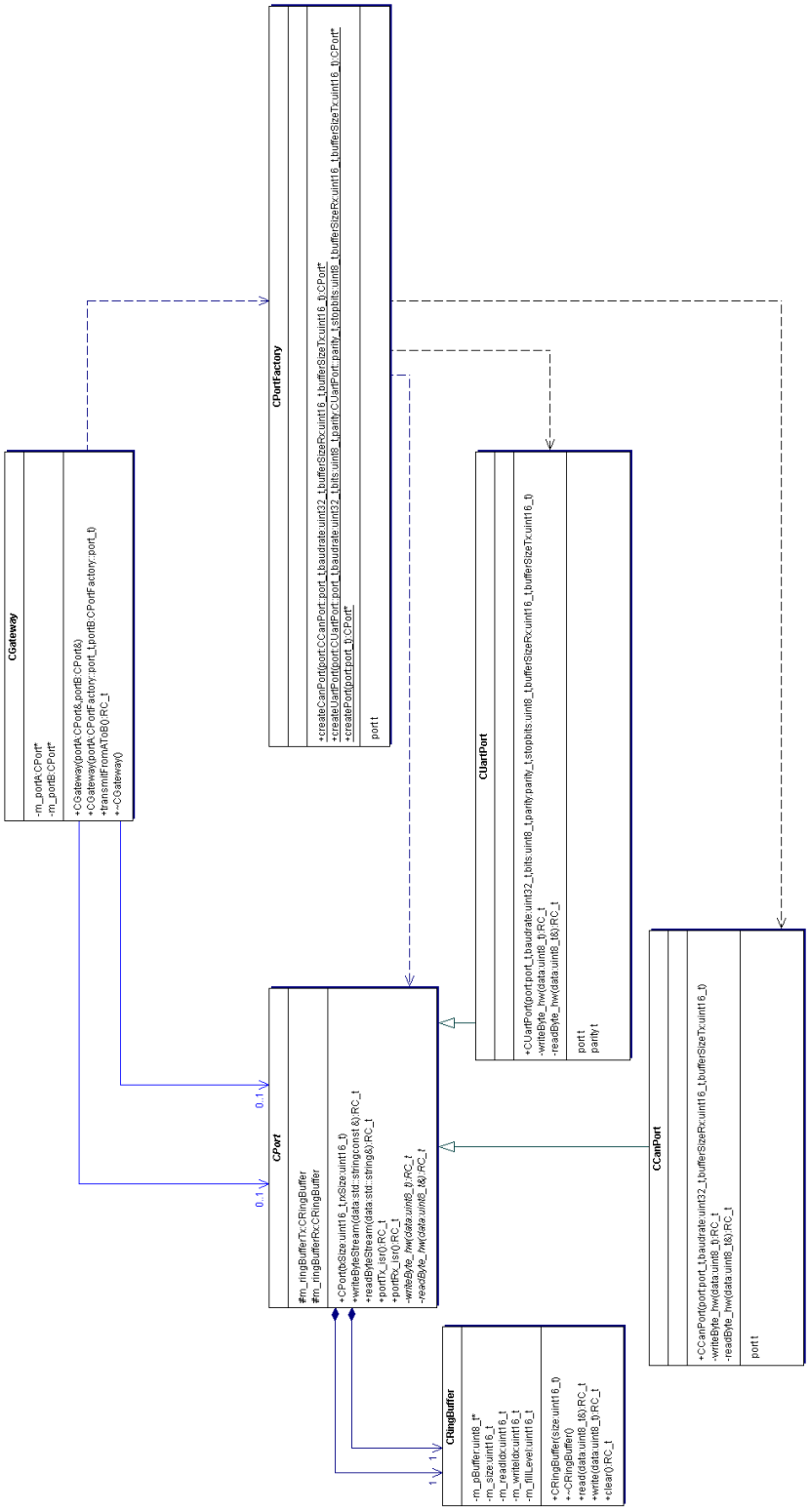
Technical requirements

- The implementation shall be generic, i.e. any combination (set during runtime) of ports shall be supported
  - UART to CAN
  - CAN to UART
  - UART to UART
  - CAN to CAN
  - Any new additional port
- Provide DoxyGen Style comments for all entities and create a website for your project, which shall also be uploaded.

1.2 Class Diagram

The current class design still contains uint8_t interfaces for the hardware driver. To support dynamic data sizes, you decide to use ringbuffer objects for passing the data.

Mark all methods which need to be changed to implement the intended behavior.



1.3 Sequence Diagram

Draw a sequence diagram showing the logic of the NEW transmit method of the gateway.

1.4 Interface Design

Refactor the system to implement a driver specific packagesize, taking a ringbuffer object as parameter. Complete the interfaces of the new methods. Design a robust and fast interface:

```
/**
 * \brief Sends one Package to the hardware
 *
 * \return RC_t:
 *              RC_SUCCESS - byte was transmitted
 *              Device specific ErrorCode - in case of error
 */
virtual RC_t writePackage_hw(_____________________ data) = 0;

/**
 * \brief Receive one Package from the hardware
 *
 *              RC_SUCCESS - byte was received
 *              Device specific ErrorCode - in case of error
 */
virtual RC_t readPackage_hw(_____________________ data) = 0;

/**
 * Get the size of a package for the peripheral
 * \return packagesize in byte
 */
virtual uint16_t getDriverPackageSize(_____________________) = 0;
```

1.5 Dynamic versus static design

Compare the current design with a static design in terms of quality goals. For every design goal, mark the design which you consider to better fulfill the goal and explain why. Hint: Complete the implementations before answering this question.

Quality goal	Dynamic Design (as is)	Static design (of your choice)	Justification
Functionality			
Reliability			
Usability			
Efficiency			
Maintainability			
Portability			

2 Refactor Driver

Implement the new driver interface. Try to modify as few code as possible and do not change the existing elementary architecture. Remember: Adding code does not hurt, changing code might hurt.

The new output should look similar to:

```
Reading 1 byte packages from UART:

Just read from UART hardware: Package: [ a ]
Just read from UART hardware: Package: [ b ]
Just read from UART hardware: Package: [ c ]
Just read from UART hardware: Package: [ d ]
Just read from UART hardware: Package: [ e ]
Just read from UART hardware: Package: [ f ]
Just read from UART hardware: Package: [ g ]
Just read from UART hardware: Package: [ h ]
Just read from UART hardware: Package: [ i ]
Just read from UART hardware: Package: [ j ]
Just read from UART hardware: Package: [ k ]
Just read from UART hardware: Package: [ l ]
Just read from UART hardware: Package: [ m ]
Just read from UART hardware: Package: [ n ]
Just read from UART hardware: Package: [ o ]
Just read from UART hardware: Package: [ p ]
Just read from UART hardware: Package: [ q ]
Just read from UART hardware: Package: [ r ]
Just read from UART hardware: Package: [ s ]

Writing 8 byte packages to CAN:

Just wrote to CAN hardware: Package: [ abcdefgh ]
Just wrote to CAN hardware: Package: [ ijklmnop ]
Just wrote to CAN hardware: Package: [ qrs ]
```

Hints:

- You might need to add some methods / overloaded operators to the Ringbuffer class.
- The UART and CAN read and write methods are stubs. Modify them to comply to the new interface.
- The packaging logic should be added to a single class.
- You need to extend / change less than a handful of methods and a few handful lines of code only.

3 Configuration and Error Handling

Add a configuration file to the system, which allows you to limit the maximum numbers of ports per system, i.e.

```
#define PORT_ASCLINMAX 4
#define PORT_CANNMAX   2
```

The following (slightly weird) error handling shall be implemented:

- If the gateway objects use more ports then allowed, an error shall be escalated to a central error handler, which will terminate the execution of the application.
- Nullpointers shall also be handled by the central error handler.
- If the user accidentally occupies a port, which already is in use, a C++ exception handler shall be called.

Hint:

- You may use the error handlers presented in the lecture on error handling as starting point.
- It might be helpful to provide another constructor to the Gateway class accepting pointers to ports.
- The central error handler may be a global object or passed as reference to the different classes.