Система управления тренажером состоит из несколько связанных узлов, обеспечивающих отработку всех необходимых алгоритмов. Такими узлами являются «**ядро**», перечень шаблонов для **моделей** и **форматов**, **менеджер** объектов, модули **сетевого** взаимодействия, база данных с **генератором** исходных кодов и основное приложение, обеспечивающее взаимодействие всех узлов и предоставляющее графический интерфейс для взаимодействия с пользователем.

**Ядро**

Все переменные в виде **приватных** полей и команды в виде констант описываются в классе **ядра “CoreQ”**. Для каждой переменной определяются методы **private set(i)** и **public get()** для взаимодействия с ними из моделей и форматов. Помимо методов для доступа к переменным, описаны методы **упаковки** всех полей в единый пакет байт, который в последствие из серверной части приложения пересылается через сеть в клиентскую, где происходит его **распаковка** и установка всех значений полей в соответствии с порядком упаковки. Таким образом ядро серверной части и все ядра клиентских частей приложения **синхронизируются** с определённым интервалом.

Все команды, которыми обмениваются модели и форматы, описаны в файлах **commandsq.cpp** и **commandsq.h** в виде констант и могут быть доступны через указатель ядра как, например, **core->commands.command\_3.**

Все типы данных, структуры, шаблоны и методы приведения типов, которыми оперируют ядро, модели и форматы и прочие модули, описаны в файле **“include/common.h”**

**Модели и форматы**

Для модели и формата описаны базовые классы **ModelQ** и **FormatQ** соответственно. В них определены стандартные (**виртуальные**) методы для взаимодействия, а также стандартный набор данных, присущий для всех наследников (ID модели, имя, принадлежность). Все модели и форматы, которые в будущем будут реализованы для конечного продукта, наследуются от **класса-родителя**, и наследуют все его методы, которые в будущем должны быть переопределены (**виртуальные**) при реализации модуля.

Указатель на объект класса **ядра** передаётся в конструкторы «**родителей»** шаблонных моделей и форматов как **CoreQ \*core**. Таким образом любой формат или модель могут получить значение конкретной переменной через определённый метод **get(),** например, **core->getVar1** или **core->getArray3(4)** для массивов. Методы **set()** доступны **только** для моделей (описаны как дружественные классы в классе ядра) и предназначены только для установки конечных значений **через** виртуальный метод **outputToCore(),** за исключением каких-то особенных случаев. Форматы имеют возможность лишь получать данные, не изменяя их значение напрямую. Для удобства оперирования самими переменными в моделях, создаются их **локальные** версии, с которыми происходит взаимодействие в ***рабочих***методах **init(), set(),** **update()** и **ReadCommand().** Принцип работы подразумевает автоматическую подгрузку моделью переменных из ядра, установка значений в локальные переменные, работа с ними, выгрузка локальных переменных в переменные ядра:

**Модель**

inputFromCore()

уровень работы с

данными

init()

set()

update()

ReadCommand()

*прочие пользовательские*

outputToCore()

update()

*прочие пользовательские*

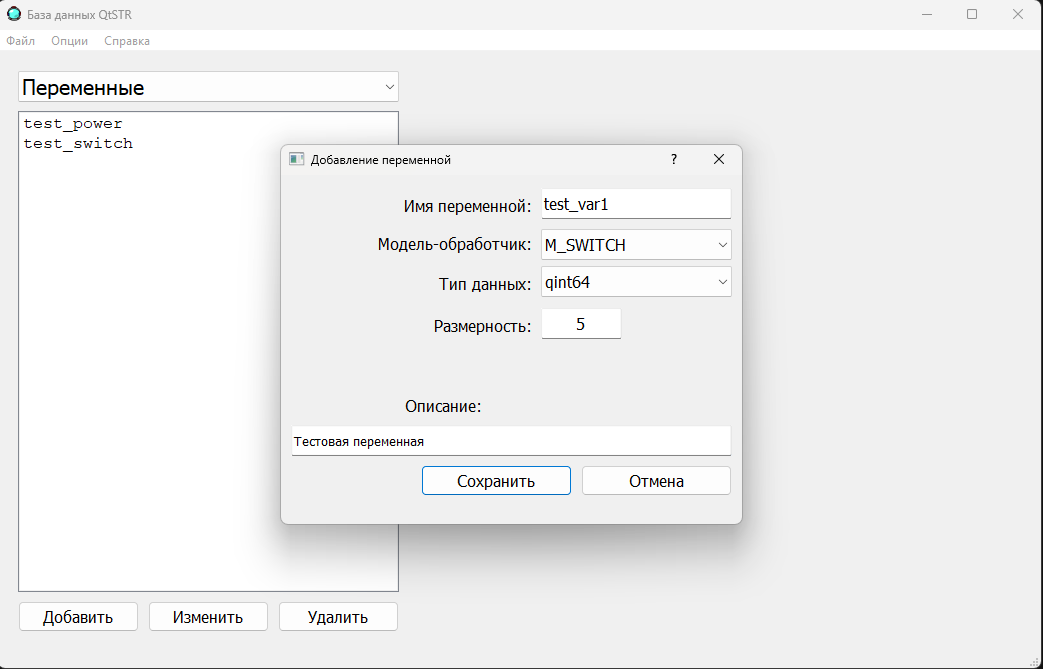
**Формат**

**Менеджер объектов**

Менеджер объектов представлен в виде класса, который включает в себя перечень всех объектов классов моделей и форматов в виде закрытых полей класса, помещённых в соответствующий контейнер. Служит прослойкой между самими моделями и форматами и основным приложением, обеспечивающим взаимодействие между всеми узлами системы.

**База данных и генератор кода**

Менеджер объектов, пользовательские форматы и модели и непосредственно само ядро **не допускают** вмешательство разработчика в их базовые методы и структуру объекта в целом. Все новые переменные, команды, модели и форматы **должны** быть описаны в **базе данных** с расставлением связей и зависимостей.



Когда все необходимые данные внесены, база данных производит **генерацию** исходных кодов с помещением их в директорию. Далее разработчик посредством редактирования определённых участков кода, обозначенных специальными **тегами** и пометками в комментариях, реализует логику работы конкретного модуля (модели или формата) и осуществляет его компиляцию. При перестроении исходных кодов модули ядра и менеджера объектов также требуют отдельной компиляции. Внутри файлов исходных кодов база данных генерирует в определённом порядке строки с тегом «**/\*<::db\_generated::>\*/**», необходимые для компиляции и корректной отработки системы. Изменение этих строк вручную может негативно повлиять на стабильность работы модуля и системы в целом.

Список методов, которые доступны для редактирования:

|  |  |
| --- | --- |
| **Для моделей:** | |
| init() | Отработка условий при инициализации модели |
| set() | Отработка начальных условий |
| update() | Основной цикл при старте системы по команде ПУСК |
| ReadCommand() | Обработка внутренних команд |
| ReadCommandFromNA() | Обработка команд от нестандартных абонентов |
| ReadPackageFromNA() | Обработка пакетов от нестандартных абонентов |
| *пользовательские* | При необходимости разработчик может описать любое количество прочих пользовательских методов и функций, необходимых для работы конкретных алгоритмов |

|  |  |
| --- | --- |
| **Для форматов:** | |
| update() | Основной цикл для обновления данных на формате |
| *пользовательские* | При необходимости разработчик может описать любое количество прочих пользовательских методов и функций, необходимых для работы конкретных алгоритмов |

«Менеджер» и «ядро» полностью генерируются и заполняются генератором исходных кодов и не подлежат редактированию разработчиком. Генератор кода при помощи ключевых слов в тексте шаблонов заполняет нужные участки кода объявлениями и определениями всех необходимых данных, которые были указаны в базе данных.

База данных и генератор исходных кодов

Ядро

Менеджер

Контейнер моделей

Переменные

Контейнер форматов

Команды

Упаковщик/ распаковщик

Формат 1

Модель 1

Формат n

Модель n

**Клиент и Сервер (сокеты)**

Классы **NetworkClient** и **NetworkServer** обеспечивают взаимодействие между серверной и клиентской частью приложения посредством реализации определённого сетевого протокола. После установки соединения при помощи стандартных **Qt** классов **QTcpSocket** происходит обмен пакетами (32 байта), которые имеют следующую структуру:

|  |  |
| --- | --- |
| Заголовок | 8 байт |
| Команда | 4 байта |
| Параметр 1 | 8 байт |
| Параметр 2 | 8 байт |
| Модельное время | 4 байта |

Заголовок определяет тип пакета и алгоритм распаковки, который будет применяться для преобразования потока байт в нужный вид. В зависимости от значения определённых байтов, используются те или иные методы разбора пакета. Описание заголовка имеет следующий вид:

|  |  |
| --- | --- |
| 1 байт | флаг пакета **ядра**. |
| 2 байт | **длина** текущего фрагмента в байтах, включая заголовок |
| 3 байт | ID получателя |
| 4 байт | ID отправителя |
| 5 байт | **флаг** текущего фрагмента (0 – сервер, 1 – клиент) |
| 6 байт | тип **кадра** |
| 7 байт | (резерв) |
| 8 байт | (резерв) |

Если **1-й** байт пакета имеет значение «1», значит, получен пакет для **синхронизации** ядра в клиентской части. Вне зависимости от значения других байтов, заголовок **отсекается**, а первые **8 байт** оставшегося пакета определяют его длину. Весь пакет без заголовка и первых 8-ми байт передаётся в ядро для **распаковки** и **синхронизации**.

**2-й** байт заголовка показывает длину текущего фрагмента в байтах. Для межмашинного обмена в пределах клиентской и серверной части он всегда должен быть равен **32** байт − если нет, то произошла ошибка при упаковке или случилась потеря при передаче по сети, и такой фрагмент считается некорректным и должен быть игнорирован.

Остальные байты определяют параметры абонента, необходимые для корректной отработки внутренних алгоритмов модулей взаимодействия.

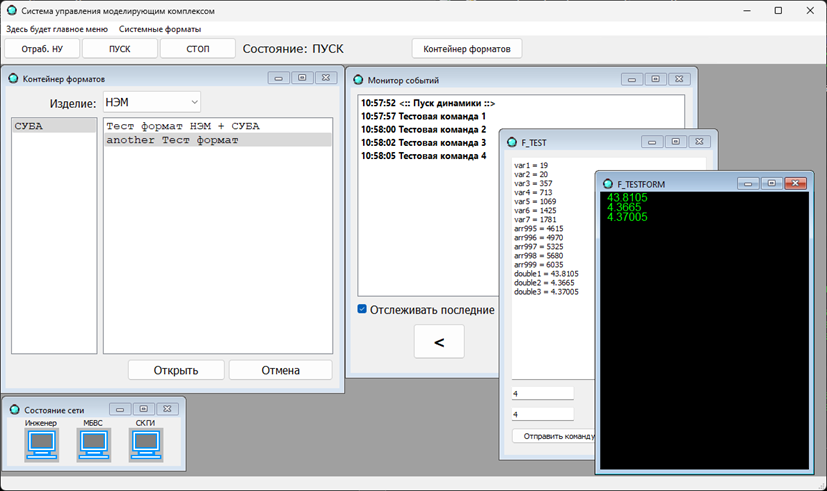
В конструктор класса **NetworkClient** может быть передан флаг **NA\_mode**, равный ***true***. В данном случае межмашинный обмен будет происходить не 32-байтовыми, а 24-байтовыми пакетами (аналогично стандартному обмену **ТРИО**):

|  |  |
| --- | --- |
| Заголовок | 8 байт |
| Команда | 4 байта |
| Параметр 1 | 4 байт |
| Параметр 2 | 4 байт |
| Модельное время | 4 байта |

Заголовок имеет аналогичное описание за исключением первого байта, который всегда устанавливается в «0». Данный формат реализован для возможности обмена с **нестандартными абонентами (МБВС, УСО и прочие),** которые в параметрической части имеют лишь 4 байта, что не позволяет обмениваться данными типа **double**. Новый внутренний обмен на основе параметрической части в 8 байт позволяет производить сетевую синхронизацию вне зависимости от типа.

**Основное приложение STR**

Приложение имеет стандартный тип и наследуется от класса **QMainWindow** с возможностью открытия произвольного количества дочерних (MDI) окон-форматов.



Основные параметры приложения описаны в конфигурационном файле «config.ini»:

[MAIN\_SETTINGS]

server=true

host=192.168.1.4

port=2001

name=Система управления моделирующим комплексом

frameType=0

mainNode=2

nodeID=2

multiScreen=false

[NODE-9]

nodeID=9

host=127.0.0.1

port=2002

title=Инженер

frameType=9

modelName=M\_TEST

Первый параметр «server» секции [MAIN\_SETTINGS] определяет режим работы – при значении **true**, приложение STR запускается в режиме **сервера**, инициализирует работу всех моделей и ядра и готово к подключению произвольного количества **клиентов** STR. При значении **false** приложение запускается в режиме **клиента** STR, **НЕ** активизирует работу моделей, но инициализирует ядро для синхронизации данных с **сервером** STR.

Параметры «host» и «port» определяют настройки сети вычислительной машины, где запускается **сервер** STR. «name» передаётся в заголовок главного окна. «frameType», «mainNode» и «nodeID» необходимы для внутренней отработки алгоритмов взаимодействия. «mainNode» относится к ID **сервера** STR, «nodeID» является **уникальным** и относится к каждому из **клиентов**. «frameType» определяется идентификационный тип сетевого кадра (пакета) в обмене – при несоответствии его в сервере и клиенте, такие пакеты будут **игнорироваться**.

Параметр «multiscreen» определяет тип отображения главного окна – при значении false приложение запускается на **одном** главном мониторе, даже если их в системе несколько; при значении true приложение **разворачивается** на всё доступное пространство.

Далее следуют секции [NODE-ID] с **уникальным** ID в названии, определяющие всех нестандартных абонентов, которые будут взаимодействовать с приложением 24-байтовыми фреймами. Все параметры **аналогичны** параметрам из секции [MAIN\_SETTINGS], за исключением параметра «modelName». Данный параметр содержит символьное название модели, которая будет обрабатывать входящие команды от абонента. Параметр «nodeID» должен **соответствовать** ID из заголовка секции.

Приложение имеет три системных дочерних окна:

− **контейнер форматов**: содержит перечень всех форматов системы, распределённых по подсистемам;

− **монитор событий**: отображает все отработанные команды и прочие события;

− **состояние сети**: отображает всех нестандартных абонентов и их статус.

При наличии подключения клиентов STR, их количество отображается в заголовке основного окна приложения.

Рабочая панель приложения имеет четыре кнопки:

− **Отраб. НУ**: вызывает метод set() у всех моделей, возвращая в исходное состояние тренажер;

− **ПУСК**: запускает модельный цикл и отработку метода update() у моделей;

− **СТОП**: останавливает модельный цикл и отработку метода update() у моделей;

− **Контейнер форматов**: вызывает формат контейнера форматов :).

**Общая схема приложения и сетевого взаимодействия**

Переменные

Команды

Формат n

Формат 1

Модель n

Модель 1

Менеджер

Контейнер форматов

Контейнер моделей

Ядро

Упаковщик

NetworkServer

Сервер STR

Менеджер

Контейнер форматов

Клиент STR

NetworkClient

Ядро

Переменные

Команды

Упаковщик/ распаковщик

Формат 1

Формат n