Общество с ограниченной ответственностью

«Электронные технологии и метрологические системы»

Цифровые датчики семейства ZETSENSOR

Руководство программиста

ЭТМС.421400.000 33

Оглавление

1. Описание протоколов

1.1. Формат структуры данных и структуры описателей

Параметры и настройки датчиков ZETSENSOR хранятся в его внутренней памяти в виде структур языка С. Каждой из структур, доступных для чтения и записи, соответствует структура-описатель. Формат структуры данных имеет следующий вид (см. Рисунок 1)

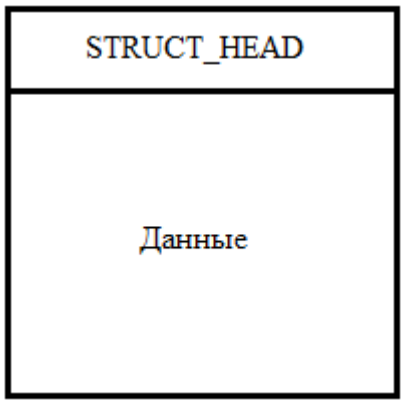

Рисунок 1. Формат структуры данных ZETSENSOR

STRUCT_HEAD имеет следующий вид:

```
typedef struct {
   unsigned int size : 12;
   unsigned int struct_type : 10;
   unsigned int status : 10;
   unsigned int write_enable : 16;
   unsigned int usCRC16 : 16;
} STRUCT_HEAD;
```

где:

size – размер структуры;

struct_type – идентификатор структуры;

status – идентификатор ошибки;

write_enable – статус при записи данных;

usCRC16 – контрольная сумма.

Наиболее важными являются структуры DEV_PAR (структура с идентификационными данными датчика, идентификатор структуры 0x18C) и CHANNEL_PAR (структура с настройками измерительного канала датчика, идентификатор структуры 0x0D0)

Структура DEV_PAR имеет вид:

```
typedef struct {
   STRUCT_HEAD head;
```

```
    long type;
    unsigned long long serial_num;
    long compile_time;
    long edition_time;
    unsigned long dev_addr;
} DEV_PAR;
```

где:

`type` – идентификатор типа датчика;

`serial_num` – серийный номер датчика;

`compile_time` – версия программного обеспечения датчика;

`edition_time` – время последнего изменения настроек датчика;

`dev_addr` – адрес устройства (от 1 до 63).

Структура `CHANNEL_PAR` имеет вид:

```
typedef struct {
    STRUCT_HEAD head;
    float value;
    float frequency;
    char measure[8];
    char channel_name[32];
    float min_level;
    float max_level;
    float reference;
    float sense;
    float resolution;
} CHANNEL_PAR;
```

где:

`value` – текущее измеренное значение, ед. изм.;

`frequency` – частота обновления данных в канале, Гц;

`measure` – единица измерения по каналу;

`channel_name` –название канала;

`min_level` – минимальное измеряемое значение по каналу, ед. изм.;

`max_level` – максимальное измеряемое значение по каналу, ед. изм.;

`reference` – опорное значение, ед. изм.;

`sense` – чувствительность, В/ед. изм.;

`resolution` – порог чувствительности, ед. изм.;

Структуры расположены в памяти датчика друг за другом.

1.2. Протокол Modbus для RS485

В устройствах ZETSENSOR реализован открытый коммуникационный протокол Modbus со стандартным набором команд:

- 0x3 - для чтения регистров с настройками и текущего измеренного значения;
- 0x4 – для чтения измеренных потоковых данных;
- 0x6 – для чтения описателей структур;
- 0x10 – для записи регистров с настройками (для изменения настроек).

Контроллеры на шине Modbus взаимодействуют, используя master-slave модель, основанную на транзакциях, состоящих из запроса и ответа. Каждый датчик имеет свой уникальный адрес в сети. Для master-устройств (преобразователей интерфейсов) этот адрес - 1. Для slave-устройств этот адрес может принимать любое значение от 2 до 63. При передаче данных используется порядок байт little-endian.

Для чтения регистров с настройками и текущего измеренного значения используется команда 0x3. Формат запроса такой командой представлен в Таблица 1.

Таблица 1. Структура запроса командой 0x3.

Тип поля	Размер (в байтах)	Номер байта
Адрес устройства	1	0
Команда запроса	1	1
Адрес регистра	2	2, 3
Количество данных для чтения (в словах)	2	4,5
Контрольная сумма (CRC16)	2	6, 7

Формат ответа на запрос с командой 0x3 представлен в Таблица 2.

Таблица 2. Структура ответа на команду 0x3.

Тип поля	Размер (в байтах)	Номер байта
Адрес устройства	1	0
Команда запроса	1	1

Размер считанных данных (в байтах)	1	2
Данные	N	3 – N+2
Контрольная сумма (CRC16)	2	N+3, N+4

Чтение значений регистров для устройства с адресом 0x4 по адресу регистра 0x0 размером 120 слов (240 байт) выглядит так:

- запрос:

0x04 0x03 0x00 0x00 0x00 0x78 0x45 0xbd

- ответ:

0x04 0x03 0xf0 0xc0 0x20 0x00 0x58 0x00 0x00 0xe5 0x4f 0x00 0x03 0x00 0x00 0x03 0xdf 0x52 0x45 0x23 0x12 0x2b 0x17 0xbd 0xa8 0x55 0x66 0xd8 0x98 0x4e 0x6d 0x00 0x04 0x00 0x00 0x00 0x4c 0x00 0x4d 0x00 0x00 0xc4 0x3b 0x44 0x64 0xc3 0xdd 0x00 0x00 0x42 0xfa 0x00 0xf2 0x6a 0x00 0x00 0x68 0x00 0x00 0x45 0x5a 0x37 0x54 0x31 0x30 0x00 0x30 0x54 0x00 0x30 0x37 0x30 0x31 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x44 0x64 0xc3 0xdd 0x44 0x64 0x43 0xdd 0x00 0x00 0x3f 0x80 0x00 0x00 0x3f 0x80 0xc5 0xac 0x37 0x27 0xc0 0x3c 0x00 0x59 0x00 0x00 0x2e 0xbc 0x00 0x01 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0xcc 0xcd 0x3d 0xcc 0x00 0xf2 0x6a 0x00 0x00 0x68 0x00 0x00 0x20 0xe5 0xee 0xef 0xe5 0xeb 0x31 0x5f 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xa0 0x14 0x00 0x74 0x00 0x00 0x09 0x46 0x00 0x01 0x00 0x00 0x00 0x00 0x42 0x48 0xf8 0xa0 0x3f 0x61 0xa0 0x10 0x00 0x76 0x00 0x00 0xf1 0x77 0x00 0x00 0x00 0x00 0x00 0x00 0x3f 0x80 0xa0 0x10 0x00 0x77 0x00 0x00 0x2d 0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xa0 0x14 0x00 0x47 0x00 0x00 0x94 0xbd 0xe1 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x0f 0xd5 0x57 0x55 0x54 0x02

Для запроса текущего измеренного значения в качестве адреса регистра используется адрес поля `value` структуры `CHANNEL_PAR`.

Для чтения описателей структур используется команда 0x6. Формат запросов и ответов такой командой аналогичен команде 0x3.

Для чтения измеренных потоковых данных используется команда 0x4. Формат запросов и ответов такой командой аналогичен команде 0x3. Для запроса потоковых данных в качестве адреса регистра используется адрес поля `value` структуры `CHANNEL_PAR`.

Для записи регистров с настройками (для изменения настроек) используется команда (0x10). Формат запроса такой командой представлен в Таблица 3.

Таблица 3. Структура запроса командой 0x10.

Тип поля	Размер (в байтах)	Номер байта
Адрес устройства	1	0
Команда запроса	1	1
Адрес регистра	2	2, 3
Количество данных для записи (в словах)	2	4, 5
Количество данных для записи (в байтах)	1	6
Данные	N	7 – N+6
Контрольная сумма (CRC16)	2	N+7, N+8

Формат ответа на запрос с командой 0x10 представлен в Таблица 4.

Таблица 4. Структура ответа на команду 0x10.

Тип поля	Размер (в байтах)	Номер байта
Адрес устройства	1	0
Команда запроса	1	1
Адрес регистра	2	2, 3
Количество записанных данных (в словах)	2	4, 5
Контрольная сумма (CRC16)	2	6, 7

1.2.1. Особенности при изменении настроек в ZETSENSOR

Процедура записи данных от внешнего устройства в память датчика является очень ответственной задачей, так как при некорректной записи данных в память, датчик перестанет корректно работать. Причины некорректной записи данных:

1) в сети находится 2 датчика с одинаковыми адресами. При попытке записи данных в один датчик, те же самые данные запишутся и на другой датчик, что приведёт к порче настроек датчика.

2) внешнее устройство начало записывать данные на датчик. Но ввиду каких-либо причин, связь случайно оборвалась (отсоединили устройство, пропало питание и т.д.). Таким образом, в память датчика могут быть записаны некорректные данные.

Возникает задача, построить алгоритм безопасной записи данных на датчик. Алгоритм должен решить следующие проблемы:

1) Датчик должен принимать только те данные, которые предназначены для него. Для идентификации принимаемых данных можно использовать 64-битный серийный номер датчика.

2) Использование принятых данных разрешено только после принятия всех данных и проверки корректности этих данных.

Первая задача решается путём добавления контрольной суммы CRC16 в каждую из подструктур основной структуры датчика. Благодаря этой контрольной суммы мы всегда можем проверить корректность текущих данных и принадлежность этих данных конкретно этому устройству.

Для решения второй задачи необходимо организовать целостность передаваемых данных. Для этого мы будем использовать поле `write_enable` структуры. Когда данные в структуре корректны это поле имеет статус «Корректные данные». Перед записью данных мы должны выставить статус «Начало передачи данных», обозначающий то, что значения данной структуры будут изменены. Далее записать данные. (После получения первого пакета, состояние структуры изменяется в состояние «Активная передача данных»). После окончания записи данных выставить поле `write_enable` в состояние «Передача данных завершена». Как только выставлен статус «Передача данных завершена», датчик начинает проверять корректность полученных данных и в случае успеха записывает их в flash-память. Далее выставляется статус «Корректные данные». Расшифровка статуса `write_enable` отражена в Таблица 5.

Таблица 5. Расшифровка статуса `write_enable`

Статус структуры (значение поля `write_enable`)	Определение статуса
0 – «Корректные данные»	Структура содержит полностью корректные данные, эти данные можно использовать и обрабатывать датчиком.
1 – «Начало передачи данных»	В данном состоянии запрещается использовать данные, находящиеся в структуре датчика. Только в этом

	состоянии можно начинать передавать данные датчику.
2 – «Активная передача данных»	В данном состоянии запрещается использовать данные, находящиеся в структуре датчика.
3 – «Конец передачи данных»	В данном состоянии датчик должен проверить все переданные ему данные. И сбросить своё состояние в состояние “Корректные данные”.

Изменения статусов возможно только в следующем порядке: «Корректные данные» → «Начало передачи данных» → «Активная передача данных» → «Конец передачи данных» → «Корректные данные»

Примечание:

1) Необходимость выставления статуса «Начало передачи данных» заключается в обозначении того, что данные изменяются извне, а не самим датчиком.

2) Необходимость выставления статуса «Конец передачи данных» заключается в том, чтобы датчик обработал полученную информацию, а не сразу перешёл в состояние «Корректные данные». Статус «Корректные данные» выставляет только датчик (выставление данного статуса извне запрещено прошивкой устройства).

3) Возможен вариант выставления статуса «Начало передачи данных» но сами данные не были переданы (в связи с обрывом соединения или каких-либо других причин). В данном случае статус не сможет сброситься в состояние «Корректные данные». Данную ситуацию должен обрабатывать компьютер. Так как датчик по определению не может определить были переданы все данные или нет, обрыв соединения это или задержки передачи данных. Поэтому для однозначности эту ситуацию будем обрабатывать компьютером.

4) Возможен вариант что статус «Начало передачи данных» не выставился по каким-либо причинам. В данном случае все последующие данные будут игнорироваться

5) Компьютеру разрешается выставлять только статусы «Начало передачи данных» или «Конец передачи данных».

В итоге алгоритм записи данных в датчик сводится к следующей последовательности действий:

1) Выставить статус «Начало передачи данных», в структуре, данные которой изменяются.

2) Начать передачу данных. Для оптимальности передачи данных, передавать нужно не всю структуру, а только изменённые поля.

3) Передать пересчитанное значение CRC16 для передаваемой структуры.

4) Выставить статус «Конец передачи данных», в структуре, данные которой изменяются.

Для расчета контрольной суммы CRC16 используется следующий алгоритм с начальным значением для расчета 0xffff:

```
static const unsigned char g_uiCRC16tableHi[] = {
	0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x8
0, 0x41, 0x00, 0xC1, 0x81,
	0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC
1, 0x81, 0x40, 0x01, 0xC0,
	0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x0
0, 0xC1, 0x81, 0x40, 0x01,
	0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x4
1, 0x01, 0xC0, 0x80, 0x41,
	0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x8
1, 0x40, 0x00, 0xC1, 0x81,
	0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC
0, 0x80, 0x41, 0x01, 0xC0,
	0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x0
1, 0xC0, 0x80, 0x41, 0x01,
	0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x4
1, 0x00, 0xC1, 0x81, 0x40,
	0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x8
0, 0x41, 0x00, 0xC1, 0x81,
	0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC
1, 0x81, 0x40, 0x01, 0xC0,
	0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x0
0, 0xC1, 0x81, 0x40, 0x01,
	0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x4
0, 0x01, 0xC0, 0x80, 0x41,
	0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x8
0, 0x41, 0x00, 0xC1, 0x81,
	0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC
1, 0x81, 0x40, 0x01, 0xC0,
	0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x0
1, 0xC0, 0x80, 0x41, 0x01,
	0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x4
0, 0x01, 0xC0, 0x80, 0x41,
	0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x8
0, 0x41, 0x00, 0xC1, 0x81,
	0x40
};
static const unsigned char g_uiCRC16tableLo[] = {
	0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x0
7, 0xC7, 0x05, 0xC5, 0xC4,
```

```
        0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xC
A, 0xCB, 0x0B, 0xC9, 0x09,
        0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1
E, 0xDE, 0xDF, 0x1F, 0xDD,
        0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD
6, 0xD2, 0x12, 0x13, 0xD3,
        0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF
2, 0x32, 0x36, 0xF6, 0xF7,
        0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3
F, 0x3E, 0xFE, 0xFA, 0x3A,
        0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xE
B, 0x2B, 0x2A, 0xEA, 0xEE,
        0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE
5, 0x27, 0xE7, 0xE6, 0x26,
        0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x6
1, 0xA1, 0x63, 0xA3, 0xA2,
        0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xA
C, 0xAD, 0x6D, 0xAF, 0x6F,
        0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x7
8, 0xB8, 0xB9, 0x79, 0xBB,
        0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7
C, 0xB4, 0x74, 0x75, 0xB5,
        0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x7
0, 0xB0, 0x50, 0x90, 0x91,
        0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x9
5, 0x94, 0x54, 0x9C, 0x5C,
        0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x9
9, 0x59, 0x58, 0x98, 0x88,
        0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4
F, 0x8D, 0x4D, 0x4C, 0x8C,
        0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x4
3, 0x83, 0x41, 0x81, 0x80, 0x40
};
//------------------------------------------------------------------
unsigned short CRC16(unsigned short usCRC, const void *pBuffer,
unsigned int uiSize)
{
        const unsigned char* pucBuffer =
                reinterpret_cast<const unsigned char*>(pBuffer);

        unsigned char ucHi = (usCRC >> 8) & 0xFF;
        unsigned char ucLo = usCRC & 0xFF;
        unsigned uIndex;

        while (uiSize--)
        {
                uIndex = ucHi ^ *pucBuffer++;
                ucHi = ucLo ^ g_uiCRC16tableHi[uIndex];
                ucLo = g_uiCRC16tableLo[uIndex];
        }
        return (ucHi << 8 | ucLo);
```

```
}
//---------------------------------------------------------------------
```

1.3. Внутренний протокол для CAN

1.3.1. Параметры CAN

Сеть CAN представляет собой шину – каждый пакет передается сразу всем участникам. Модуль сам решает, какие пакеты он будет обрабатывать.

Скорость передачи данных:

– 100 кбит/сек;

– 300 кбит/сек;

– 960 кбит/сек.

От скорости передачи зависит общая пропускная способность сети, а также максимально допустимая длина кабелей. Точка сэмплирования бита (Sample point) – в диапазоне от 60% до 85%.

1.3.2. Формат пакета

Каждый пакет CAN в модулях ZETSENSOR идентифицируется 11-битным идентификатором (см. Таблица 6).

Таблица 6. Значение идентификатора CAN-пакета

Бит	Поле
10	CAN_SHORT
9	CAN_BODY
8	CAN_BEGIN
7	CAN_CHAIN
6	CAN_TO_MASTER
5	node
4	node
3	node
2	node
1	node
0	node

Младшие шесть битов числа (маска 0x3f) определяют адрес узла (node) от 0 до 63. Старшие пять битов содержат флаги для различения пакетов по типу:

– CAN_TO_MASTER (0x40) — флаг направления;

– CAN_CHAIN (0x80) — флаг потоковых данных;

– CAN_BEGIN (0x100), CAN_BODY(0x200), CAN_END(0x300) — флаги составных пакетов;

– CAN_SHORT (0x400) — флаг типа данных.

Флаги 0x40..0x400 не должны быть все одновременно установлены в единицу. Каждый модуль может занимать в сети CAN от одного и более адресов, по одному на каждый поддерживаемый им канал. В одной сети CAN не должно быть двух одинаковых адресов.

В семействе ZETSENSOR по сети CAN передаются следующие виды пакетов:

– широковещательные пакеты;

– потоковые данные;

– командные пакеты (команды и ответы);

– сервисные пакеты.

Мы условно выделяем роль мастера CAN — это модуль, который отправляет команды и принимает ответы. Мастер CAN может быть в сети только один, его адрес установлен в 1.

В модулях ZETSENSOR реализована функция автоматического определения скорости CAN. После включения только мастер может отправлять пакеты в CAN на выбранной скорости, а все остальные модули сначала слушают сеть, но ничего не отправляют. При успешном получении любого пакета модули определяют выбранную скорость и только после этого начинают отправку собственных пакетов. Таким образом, мастер CAN выполняет роль стартера сети CAN.

1.3.3. Широковещательные пакеты

В общем смысле, в сети CAN все пакеты широковещательные. Но мы будем называть широковещательными только пакеты с идентификатором 0, так как каждый модуль ZETSENSOR обязательно должен быть настроен на прием таких пакетов. Широковещательные пакеты имеют самый высокий приоритет.

1.3.4. Пакеты активности узла

Пакет активности узла — это широковещательный пакет с одним байтом данных. Единственный байт данных содержит адрес (номер узла). Пакет активности узла играет роль heartbeat (сердцебиение), то есть служит для оповещения других участников сети CAN о присутствии этого узла в сети.

Пакет активности рассылается каждым узлом. Если модуль имеет несколько каналов, то для каждого канала рассылается собственный пакет активности.

Период рассылки пакета активности составляет 1 сек + (адрес узла) мс. Если с какого-то адреса нет пакетов активности в течение достаточно длительного времени (10 сек и более), то можно считать данного участника сети отсутствующим (был отключен, завис, сменил адрес и так далее).

1.3.5. Пакеты синхронизации времени

Синхронизация времени основана на том, что модули запоминают время фактической отправки и приема CAN пакета (точнее, время отправки и приема символа SOF).

Задатчик времени (это либо модуль ZET 7175, либо мастер CAN, если ZET 7175 отсутствует) раз в секунду рассылает пакеты синхронизации. Каждый пакет содержит восемь байтов данных с информацией о времени отправки предыдущего пакета синхронизации:

```
long seconds; // время в секундах, начиная с 1970-01-01 00:00:00 UTC
long nanoseconds; // кол-во наносекунд, прошедших с указанной секунды
```

Точность снятия временных меток модулем довольно низкая — 1..10 мкс. Однако итоговое качество синхронизации обычно имеет более высокую точность (в районе 100..2000 нс) за счет фильтрации показаний и ПИ-регуляции.

В сети CAN может быть только один задатчик времени.

1.3.6. Потоковые пакеты

Потоковые пакеты – это пакеты с какими-то цифровыми данными, сгенерированными модулем. Это либо данные по каналу в формате float или short, либо событие. Принимать потоковые пакеты может как мастер CAN для передачи на ПК или записи на SD карту, так и любой модуль по желанию (например, для индикации текущего значения).

Потоковые данные отправляются модулем автоматически по мере их готовности, но только после подачи данному узлу команды запуска выдачи данных.

Командой выдачи служит любой командный пакет с кодом 0x03. Обычно достаточно отправить узлу пакет, содержащий следующие шесть байтов: 0x03 0x00 0x77 0x77 0x77 0x77.

В пакетах с потоковыми данными всегда установлены флаги CAN_CHAIN и CAN_TO_MASTER. Флаги CAN_SHORT, CAN_BEGIN и CAN_END используются при передаче данных, формат которых отличается от float.

Каждый пакет содержит до восьми байтов. В случае с форматом float это одно или два значения IEEE 754. Все значения передаются в порядке LSB first (или little-endian).

1.3.7. Командные пакеты

Командные пакеты (или пакеты Modbus) служат для передачи пакетов протокола Modbus между модулями и ПК.

Пакеты Modbus делятся на запросы и ответы. Запросы обычно исходят от ПК и адресованы самому мастеру CAN или любому другому модулю. Модуль с несколькими адресами принимает команды на каждый адрес.

Каждый модуль (включая мастера) умеет разбивать пакет Modbus на серию пакетов CAN для передачи их по шине CAN, а также собирать принятые пакеты CAN в пакет Modbus для его обработки. При этом мастер CAN передает по сети CAN запросы Modbus и принимает ответы Modbus, а остальные модули, наоборот, принимают запросы и отправляют ответы.

Запрос Modbus, в зависимости от кода команды, может быть преобразован в единичный пакет CAN или в серию пакетов:

– адрес — node (адрес узла, которому адресован запрос);

– если это единичный пакет, то все флаги сброшены;

– если это составной пакет, то первый пакет имеет флаг CAN_BEGIN, последний – CAN_END, а промежуточные пакеты – CAN_BODY;

–данные содержат 6 или более байтов (в зависимости от кода команды), разбитые на пакеты по 8 байтов:

– два байта — command (код команды Modbus в LSB);

– два байта — addr (адрес регистра Modbus в LSB);

– два байта — quantity (кол-во регистров Modbus в LSB);

– после этого могут идти данные.

Формат пакетов с ответами Modbus соответствуют формату пакетов с запросами, но в данных отсутствуют первые шесть байтов (command, addr и quantity).

Инициатором запросов является мастер CAN (обычно по команде от ПК). В редких случаях, инициатором запросов может выступать другой модуль.

1.3.8. Сервисные пакеты

Сервисные пакеты имеют адрес с номером узла 1 и установленным флагом CAN_SHORT. Флаг CAN_SHORT добавлен с целью сохранения обратной совместимости,

чтобы модули со старыми прошивками смогли пропустить такие пакеты. Формат данных в сервисном пакете (всегда восемь байтов).

```
#define CAN_SERVICE_MESS_ID (CAN_SHORT | 0x001)
#define CAN_SERVICE_ID_FROM_MASTER (0x0080)
#define CAN_SERVICE_ID_TOGGLE (0x0040)
#define CAN_SERVICE_ID_NODE_MASK (0x003f)
typedef struct
{
        unsigned short service_id;
        unsigned short service;
        unsigned long param;
} CAN_SERVICE_DATA;
```

Идентификатор сервисного пакета состоит из следующих частей:

– флаг направления: сброшен, если пакет содержит команду от указанного узла мастеру, и установлен, если пакет содержит ответ от мастера указанному узлу;

– флаг четности для проверки в случае отправки одинаковых запросов: в команде флаг четности просто меняется перед каждой новой отправкой, а в ответе дублируется из команды;

– номер узла всегда содержит номер узла, сделавшего запрос, независимо от направления пакета;

Код сервисной функции определяет тип запроса. Параметр содержит какое-то число, смысл которого зависит от кода функции. Модули отправляют только команды и принимают только ответы. Мастер CAN принимает только команды и отправляет только ответы. Модуль может в любой момент времени отправить новую команду. Перед отправкой команды меняется флаг четности. При получении ответа, если в идентификаторе пакета совпадают флаг четности и номер узла, то вызывается обработчик ответа. Обработчик ответа реализуется в каждом модуле отдельно. Сервисные функции не имеют времени ожидания, то есть ответ может прийти через большой промежуток времени. Если в запросе предполагается таймаут, то он должен быть реализован модулем самостоятельно. Модуль может отправить новую команду, не дожидаясь ответа. При этом ответы на предыдущие команды будут проигнорированы. Мастер CAN должен быть готов к приему новой команды в любой момент времени. Выполнение команды должно производиться как можно быстрее. После выполнения команды мастер может отправить ответ (а может и не отправлять). Только модули могут быть инициаторами сервисных запросов. Даже если

мастер CAN отправит какой-нибудь пакет без получения команды, то все такие пакеты должны быть проигнорированы модулями.

1.3.9. Получение данных с цифровых датчиков ZETSENSOR по CAN

Настройка и получение данных с цифровых датчиков ZETSENSOR (далее датчики) с интерфейсом CAN производится с помощью преобразователя интерфейсов ZET 7174 (CAN→USB) или ZET 7176 (CAN→Ethernet) и ПО ZETLAB.

Имеется возможность считывания значений с датчиков напрямую, то есть с использованием собственного устройства с интерфейсом CAN (далее мастер).

Датчики должны быть предварительно настроены, адреса (node) каналов и частоты выдачи значений по ним должны быть известны.

Мастер должен иметь следующие параметры CAN:

– скорость передачи данных составляет 960 кбит/с;

– sample point в диапазоне от 60% до 85%;

– используются базовые пакеты с 11-битными идентификаторами.

Чтобы активировать выдачу значений в датчиках, мастер должен отправить на каждый адрес (node) пакет с командой запуска:

– идентификатор – адрес канала (например, 0x00B для адреса 11);

– содержимое – 6 байтов данных: 0x03 0x00 0x77 0x77 0x77 0x77.

Отправку пакетов с командой запуска рекомендуется периодически повторять, например, раз в 10 или 60 секунд. После получения команды запуска, датчики начинают самостоятельно отправлять по каждому каналу пакеты со значениями (по мере их готовности, в соответствии с настроенной частотой выдачи):

– идентификатор – адрес канала с установленными битами 0x0C0 (например, идентификатор для адреса 11 будет равен 0x00B + 0x0C0 = 0x0CB);

– содержимое – 4 или 8 байтов, каждые четыре байта содержат очередное значение.

Значения передаются в формате float (IEEE 754, одинарная точность, порядок байтов little-endian).

1.3. Протокол OPC

Программа «Сервисная работа с ZET7xxx» (SensorWork) предоставляет возможность работы в режиме OPC-сервера. При активации данного режима программа формирует дерево тегов на основе найденных в измерительной линии устройств и раз в секунду обновляет значение тега. По умолчанию дерево тегов формируется только из регистров текущего значения измеряемой величины. При необходимости можно

сконфигурировать программу так, чтобы из цифрового датчика читались и другие регистры с последующей записью в соответствующий тег OPC-сервера.

Для активации режима OPC-сервера отметьте символом “галочка” идентификаторы преобразователей интерфейсов, которые должны работать в этом режиме. В окне “Сервисная работа с ZET7xxx” в меню “Действия” в разделе “Работа с OPC” выберите “Запуск OPC-сервера” (см. Рисунок 2), при этом программа сформирует дерево OPC-тегов и начнет чтение нужных регистров из датчиков. Процесс работы OPC-сервера будет отображаться в окне “Работа с OPC-сервером” (см. Рисунок 3)

Примечание: *После первой активации режима автоматически сформируется конфигурация работы OPC-сервера, которая будет использоваться при последующих активациях данного режима. При этом даже если какой-либо преобразователь или цифровой датчик будет отсутствовать в системе, программа сформирует для него тег, но качество тега будет плохое, т.е. тег с недостоверными данными. Сохраненную конфигурацию OPC-сервера можно при желании изменять под нужды своей системы.*

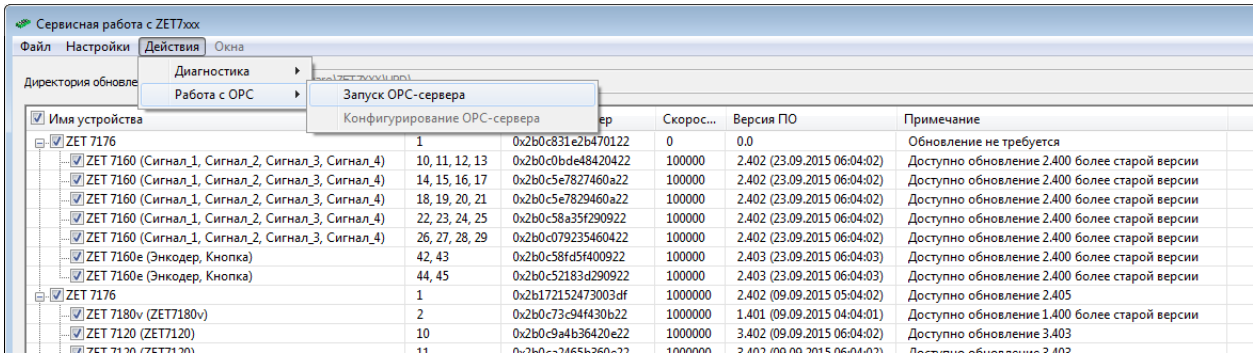

Рисунок 2. Запуск OPC-сервера

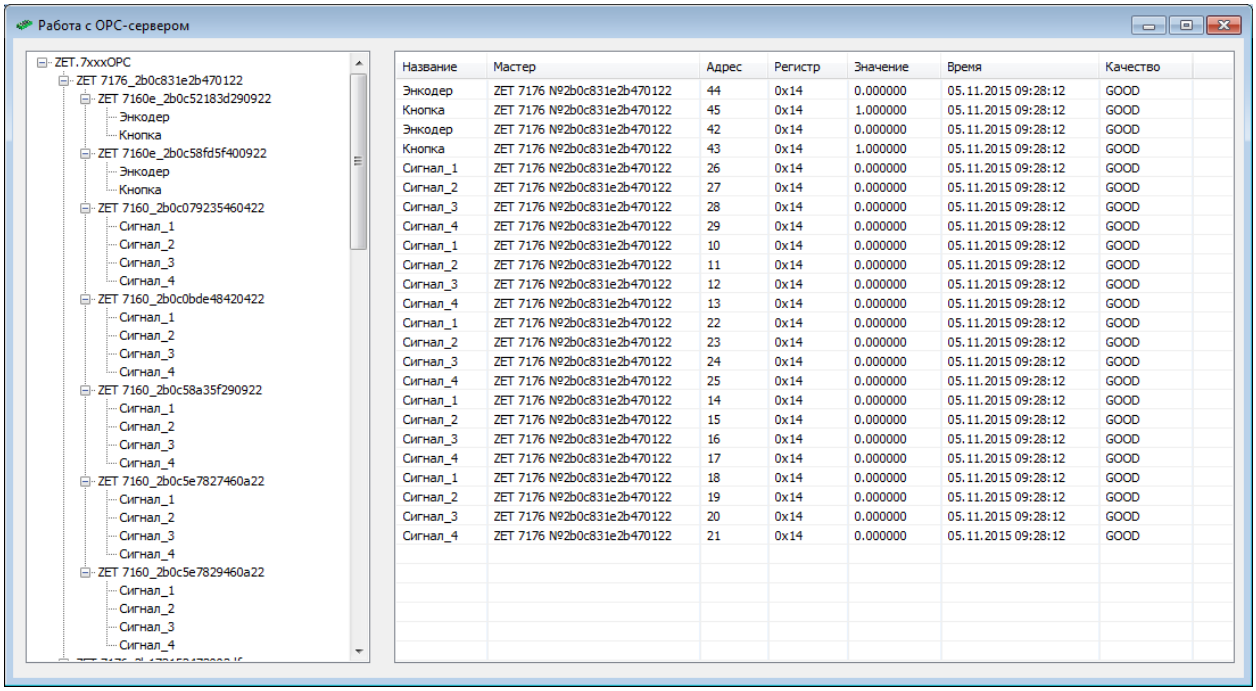

Рисунок 3. Дерево OPC-тегов

2. Работа с утилитой SensorWork

2.1. Обновление внутреннего ПО ZETSENSOR

Внимание: для обновления внутреннего ПО ZETSENSOR, компьютер, к которому подключен цифровой датчик, должен иметь доступ в глобальную сеть интернет.

Для обновления внутреннего ПО ZETSENSOR необходимо выполнить следующие действия:

1) Запустить программу “Сервисная работа с ZET7xxx” и убедиться в том, что открыт доступ к серверу с файлами обновления. В графе “Директория обновления ПО” должен отображаться путь к серверу с файлами обновлений *file.zetlab.ru* (см. Рисунок 4). Если доступ к серверу отсутствует, то следует проверить подключение компьютера к глобальной сети интернет.

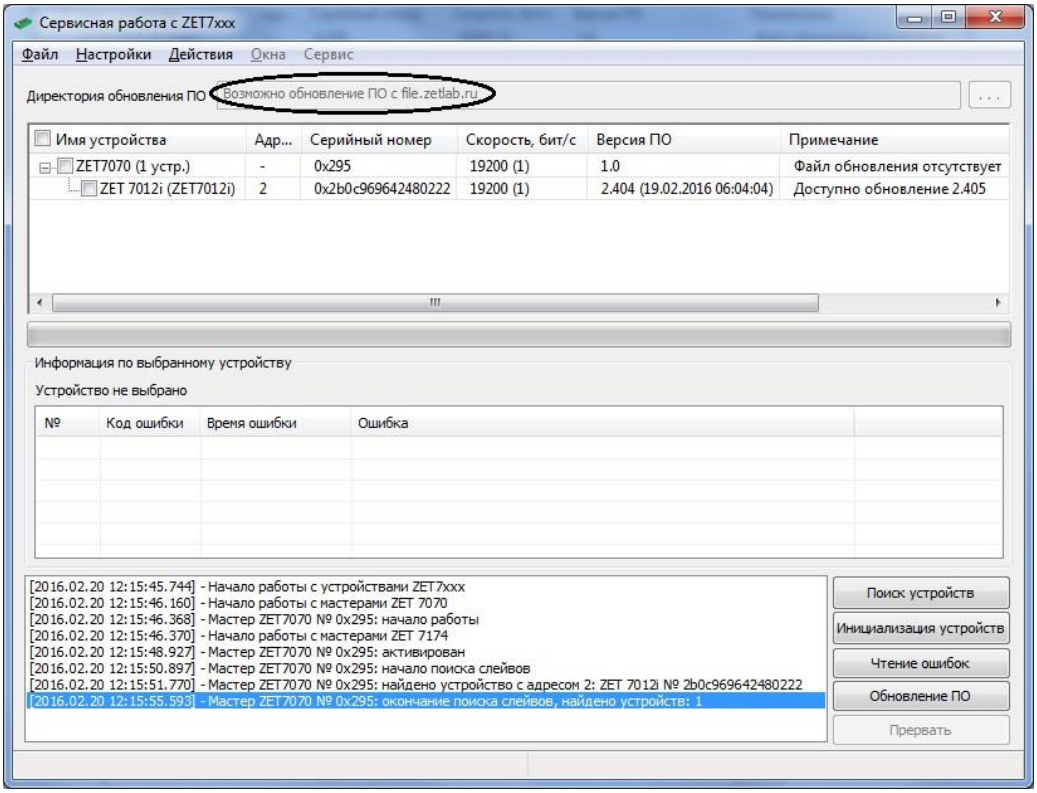

Рисунок 4.

2) Следует убедиться в наличии новой версии ПО устройства. Справа от имени цифрового датчика, который следует обновить, в столбце “Примечание” должно отображаться значение “Доступно обновление X”, где X - номер версии ПО (см. Рисунок 5).

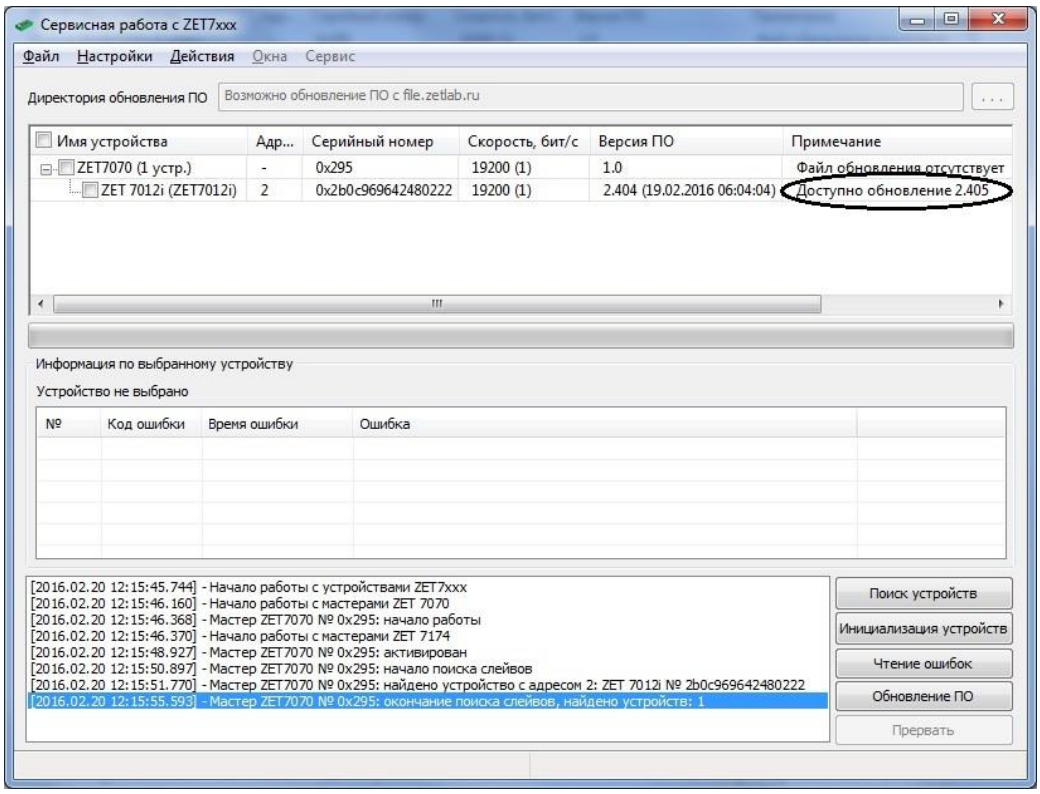

Рисунок 5.

3) Выбрать цифровой датчик, ПО которого необходимо обновить, установив галочку слева от имени цифрового датчика (см. Рисунок 6).

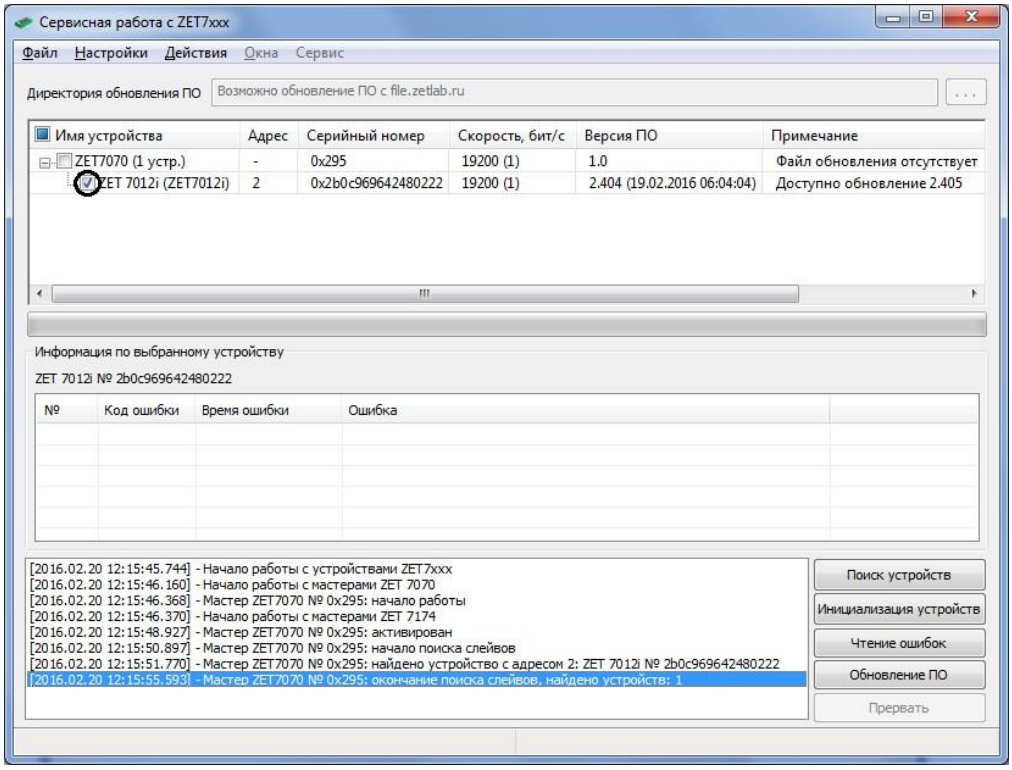

Рисунок 6.

4) Нажать кнопку “Обновление ПО” (см. Рисунок 7).

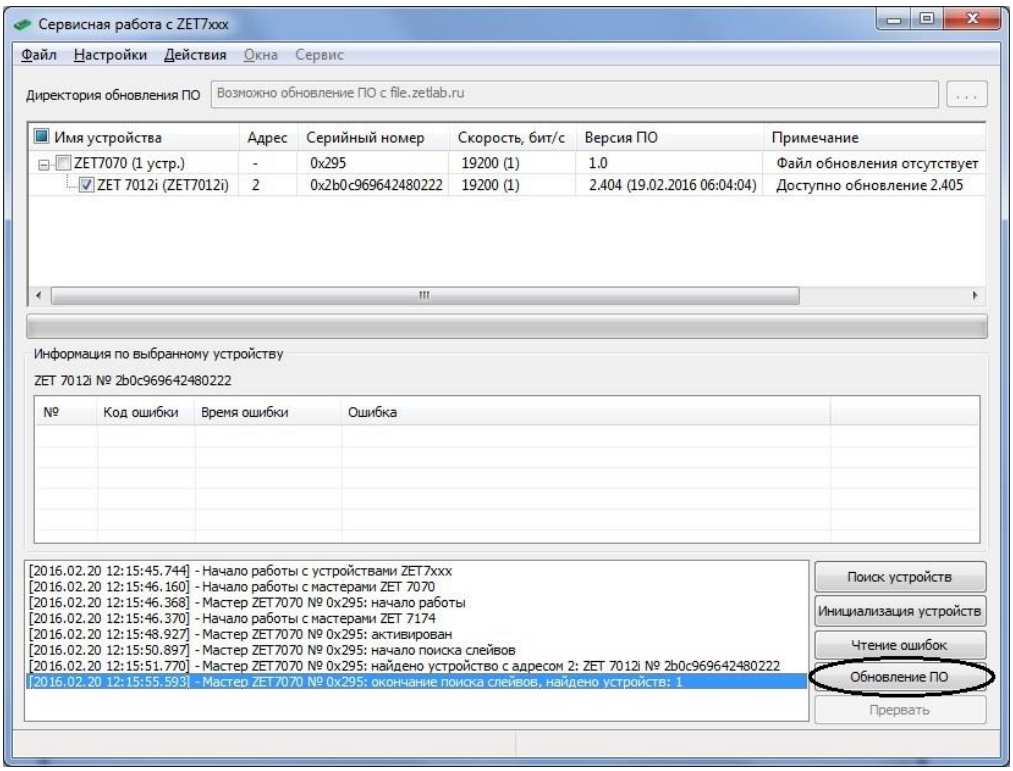

Рисунок 7.

5) Дождаться окончания обновления ПО устройства. В столбце “Состояние” отображается текущее состояние обновления ПО цифрового датчика (см. Рисунок 8). При необходимости требуется следовать инструкциям программы.

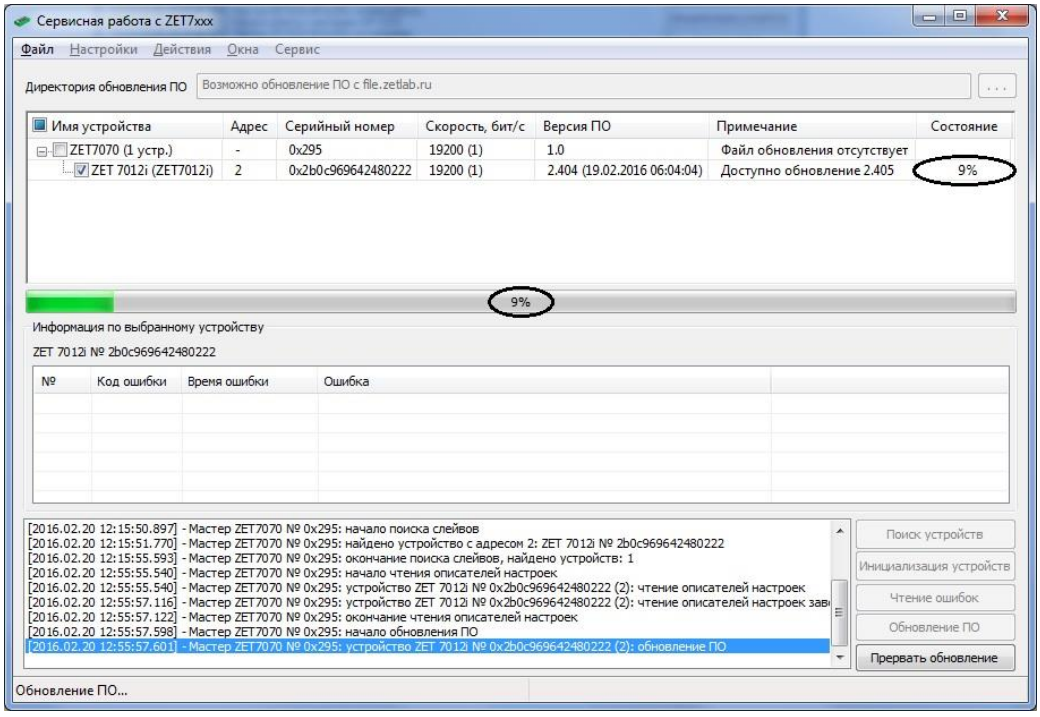

Рисунок 8.

6) После обновления следует убедиться, что на цифровой датчик установилась последняя версия ПО (см. Рисунок 9).

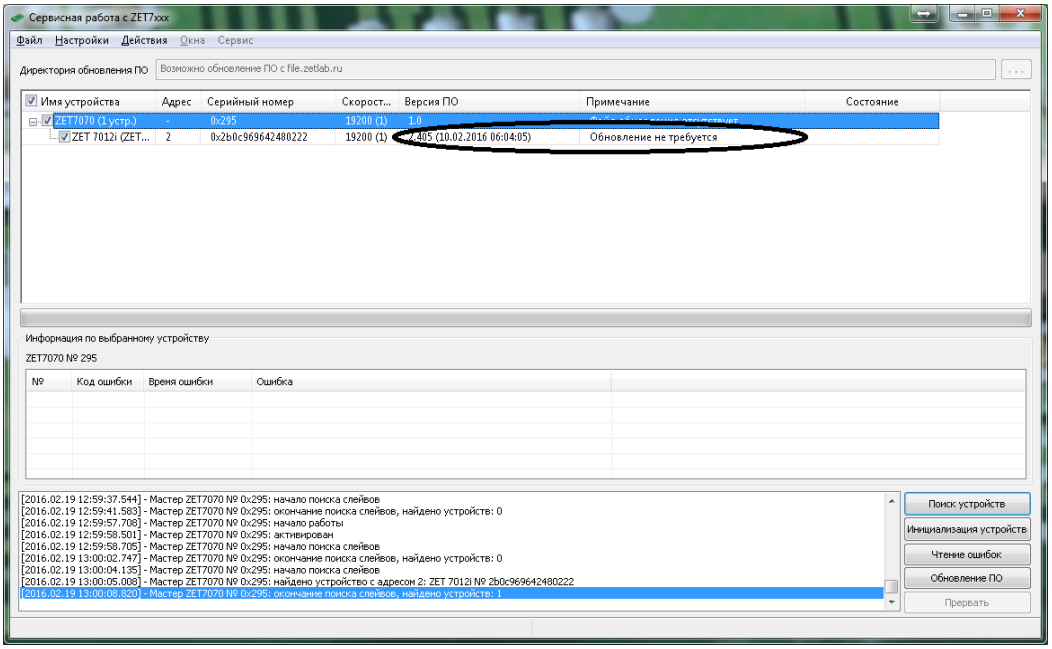

Рисунок 9.

2.2. Диагностика обмена данными в измерительной линии

Диагностика обмена данными в измерительной линии выполняется посредством посылки в линию команд ReadHoldingRegisters и/или ReadInputRegisters с последующим анализом ответа или его отсутствия. По умолчанию запросы идут с максимально возможной частотой. Целью диагностики является выявление неисправностей и тонких мест в измерительной линии с точки зрения обмена данными между преобразователем и цифровым датчиком.

Отметьте символом “галочка” идентификаторы преобразователей интерфейсов, для которых необходимо произвести диагностику формируемых ими измерительных линий. В окне “Сервисная работа с ZET7xxx” в меню “Действия” в разделе “Диагностика” выберите “Диагностика обмена данными” (см. Рисунок 10), при этом программа начнет тестирование измерительных линий результаты которого будут отображаться в окне «Диагностика обмена данными в линии» (см. Рисунок 11)

Примечание: *В окне “Диагностика обмена данными в линии” отображается диагностическая информация для измерительной линии относящаяся к первому выбранному в списке преобразователю интерфейса, для просмотра диагностической информации по другим измерительным линиям в окне “Сервисная работа с ZET7xxx” в меню “Окна” выберите соответствующие идентификаторы преобразователя интерфейса.*

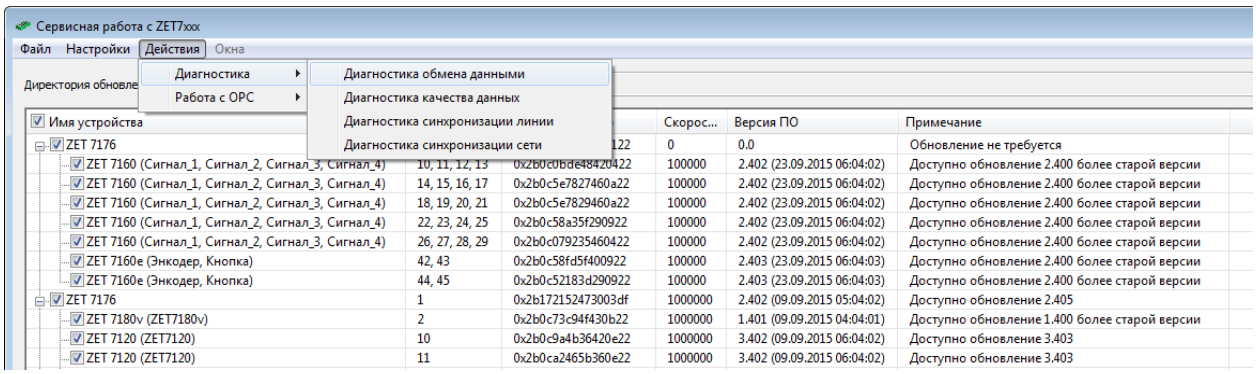

Рисунок 10.

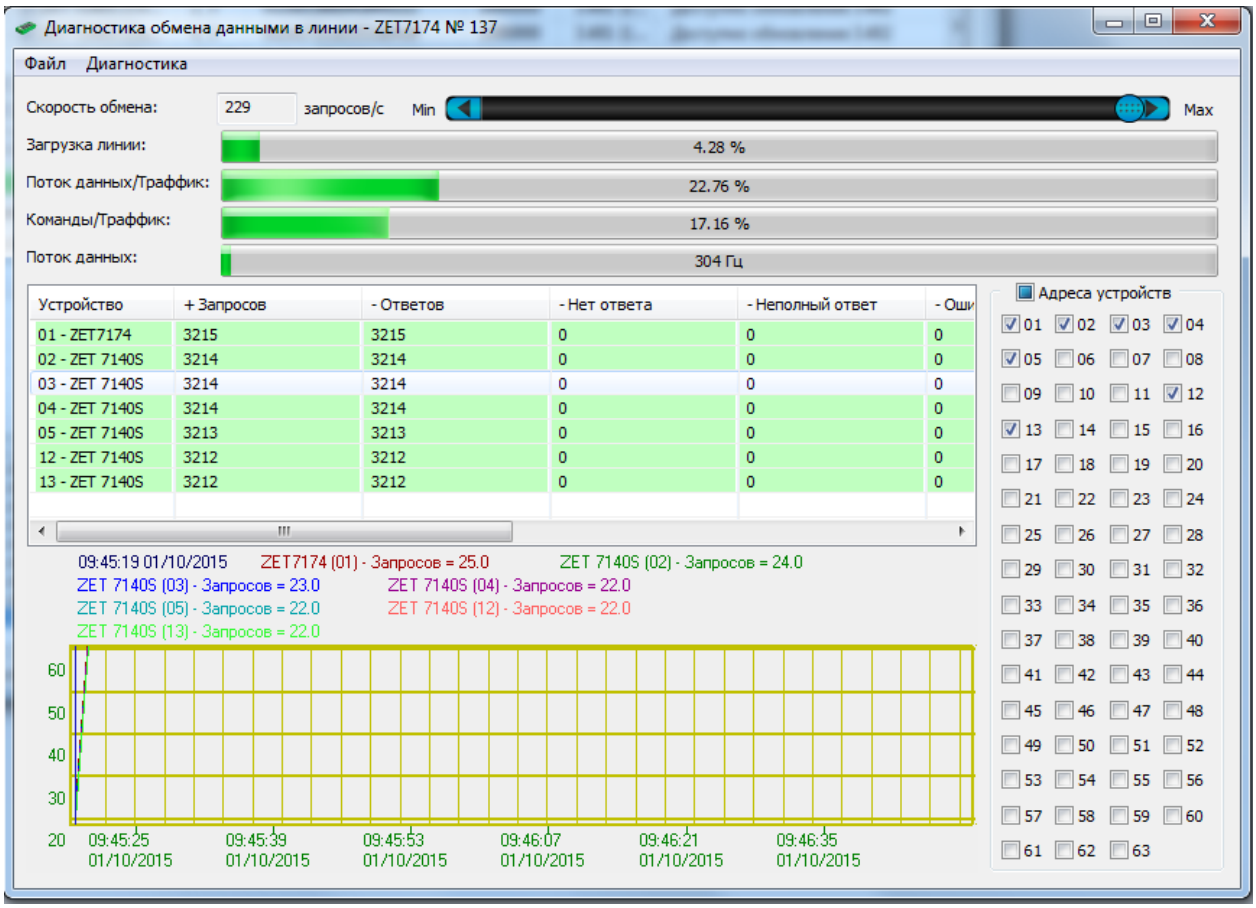

Рисунок 11.

В окне «Диагностическая информация» имеются следующие области:

- скорость обмена – отображается текущая скорость обмена на цифровом канале между преобразователями интерфейса и цифровыми модулями

- загрузка линии – отображается (в процентном соотношении от максимальной) текущая загруженность цифровой линии

- область списка устройств – отображается список устройств, начинающихся с номеров, означающих адрес устройства на цифровой линии

- область адреса устройств – отображает адреса, по которым программа производит диагностические запросы (выбранные адреса отмечены “галочкой”)

На Рисунок 12 проиллюстрирована неисправность, диагностированную на измерительном канале которая связана с дублированием назначенных цифровым модулям адресов №2.

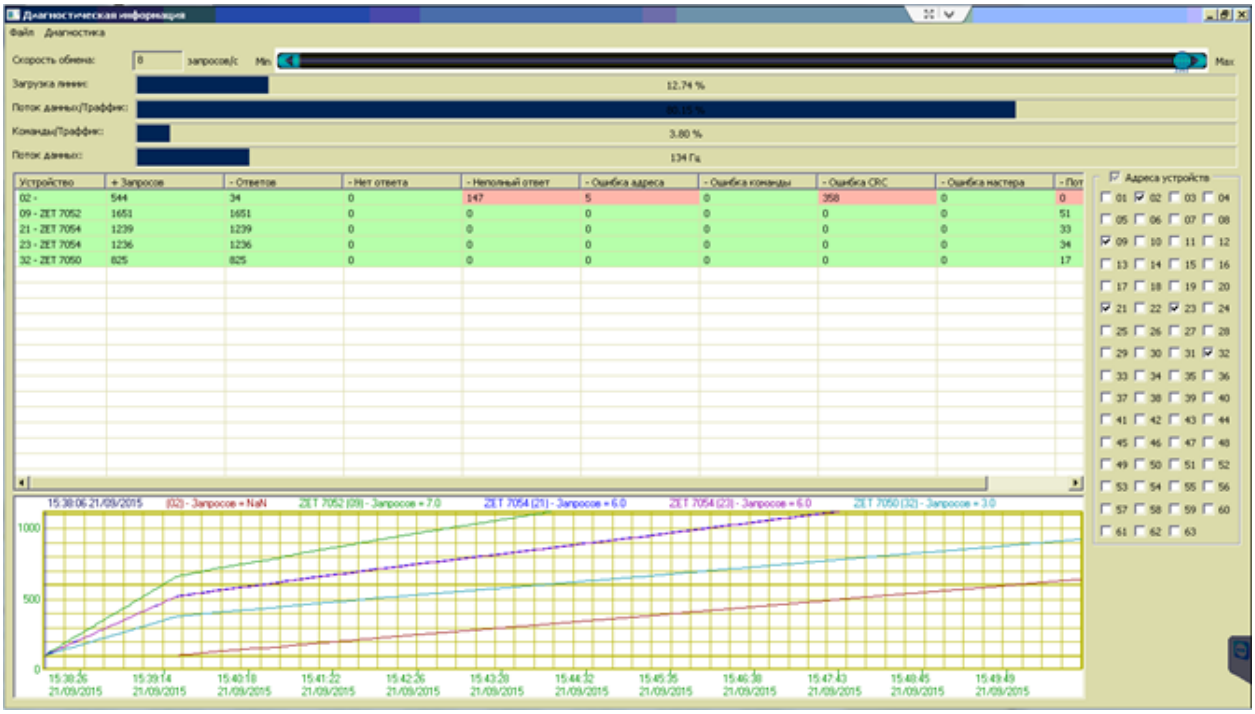

Рисунок 12.

Проиллюстрированная неисправность имеет характерное отражение в окне “Диагностическая информация”. При совпадении адресов устройств в сети MODBUS на каждый запрос устройства с одинаковым адресом отвечают одновременно, в результате чего образуются коллизии при обмене данными, а следовательно, возникают некорректные ответы на запросы, и тогда в колонках “Неполный ответ”, “Ошибка адреса”, “Ошибка команды” и “Ошибка CRC” инкрементируются значения. Кроме этого, падает общая скорость обмена в линии, и величина, отображаемая в поле “Скорость обмена” будет отличаться от нормальной для данной линии.

Примечание: *Нормальная скорость обмена в линии определяется при максимальной загруженности запросами (регулятор скорости обмена находится в крайнем правом “максимальном” положении) при посылке команды ReadHoldingRegisters (команда запроса данных - ReadInputRegisters - должна быть отключена через меню “Диагностика”) только существующим в линии устройствам. Так при правильном построении измерительной линии должны выполняться следующие соотношения:*

1. *для линии с преобразователем ZET7070:*
   - *2400 бит/с ≈ 10 запросов/с*
   - *4800 бит/с ≈ 15 запросов/с*
   - *9600 бит/с ≈ 30 запросов/с*

- *14400 бит/с ≈ 40 запросов/с*
- *19200 бит/с ≈ 50 запросов/с*
- *38400 бит/с ≈ 75 запросов/с*
- *57600 бит/с ≈ 90 запросов/с*
- *115200 бит/с ≈ 110 запросов/с*

2. *для линии с преобразователем ZET7174:*
   - *100 кбит/с ≈ 175 запросов/с*
   - *300 кбит/с ≈ 210 запросов/с*
   - *1 Мбит/с ≈ 245 запросов/с*
3. *для линии с преобразователем ZET7076:*
   - *2400 бит/с – не поддерживается ZET7076*
   - *4800 бит/с ≈ 10 запросов/с*
   - *9600 бит/с ≈ 20 запросов/с*
   - *14400 бит/с ≈ – не поддерживается ZET7076*
   - *19200 бит/с ≈ 30 запросов/с*
   - *38400 бит/с ≈ 45 запросов/с*
   - *57600 бит/с ≈ 50 запросов/с*
   - *115200 бит/с ≈ 55 запросов/с*
4. *для линии с преобразователем ZET7176:*
   - *100 кбит/с ≈ 110 запросов/с*
   - *300 кбит/с ≈ 140 запросов/с*
   - *1 Мбит/с ≈ 180 запросов/с*

В некоторых случаях количество запросов в секунду может отличаться от нормальных показателей, однако она должна оставаться стабильной. Если стабильность с течением времени не наблюдается, значит в линии скорее всего есть проблемы.

Внимание! *Для корректной работы измерительной линии категорически запрещается дублирование адресов располагающихся на ней цифровых датчиков*

Примечание: *Адреса цифровых датчиков всегда соотносятся с их измерительными каналами поэтому следует учитывать, что некоторые цифровые датчики такие как например ZET7152 или ZET7154 имеют в своем составе более одного измерительного канала. При конфигурировании цифровых датчиков, имеющих больше одного адреса, указывается только адрес первого из его измерительных каналов, однако при этом следует помнить о том, что следующие по списку адреса (в зависимости от количества измерительных каналов в составе датчика) также будут задействованы и не должны быть назначены другим цифровым датчикам, устанавливаемым на ту же измерительную*

линию. Пример: на измерительную линию установлен цифровой датчик ZET7152 сконфигурированный на адрес №5. Так как цифровой датчик ZET7152 имеет в своем составе три измерительных канала то адреса №6 и №7 не могут быть назначены другим цифровым датчикам на данной измерительной линии.

2.3. Диагностика качества данных в измерительной линии

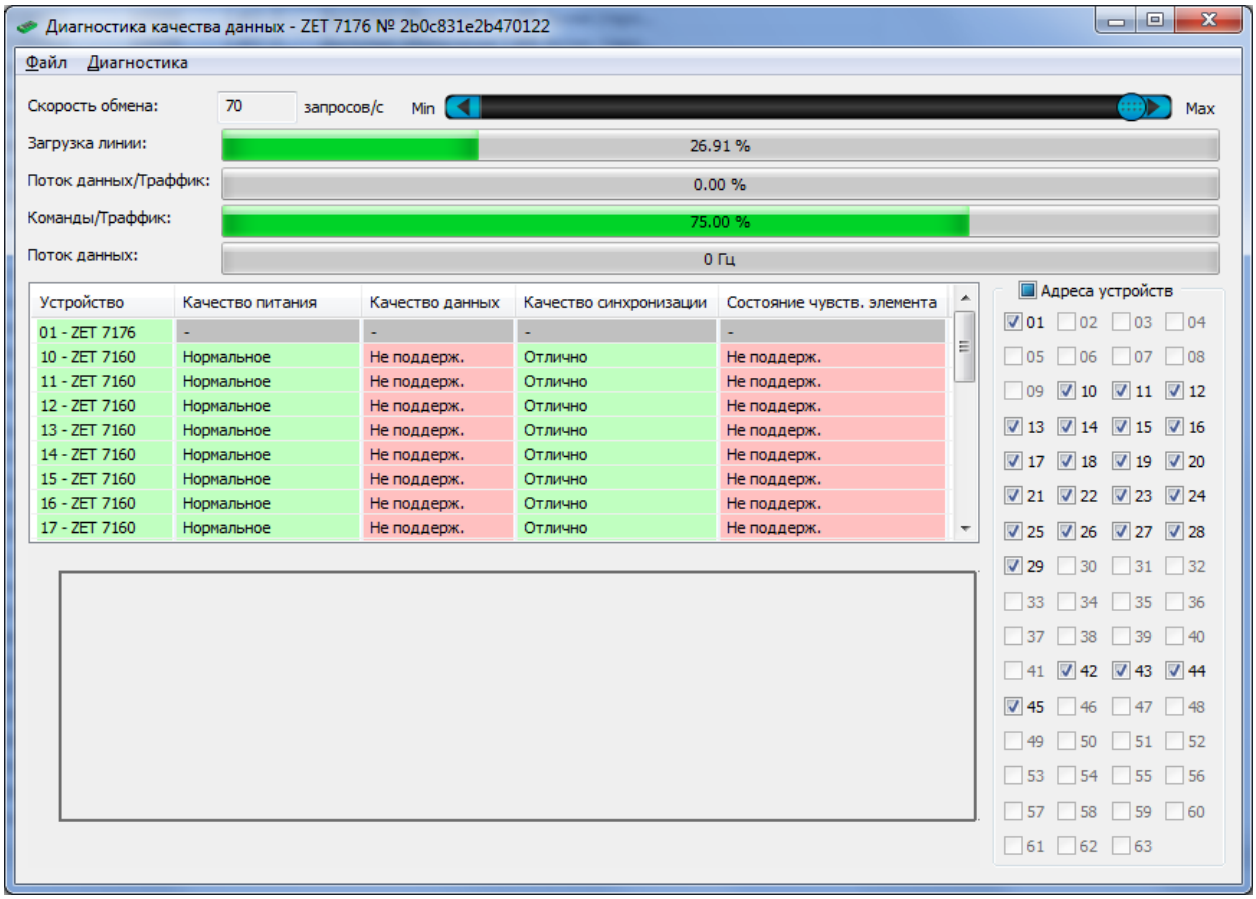

Рисунок 13.

Диагностика качества данных в измерительной линии заключается в диагностике цифрового датчика по нескольким параметрам:

1. Параметр “Качество питания”

Параметр “Качество питания” информирует о состоянии напряжения питания цифрового датчика. Существует несколько состояний индикации параметра “Качество питания”:

- Не поддерживается - цифровой датчик не обладает функцией контроля качества питания.
- Низкое - напряжение питания цифрового датчика менее 9 В.
- Нормальное - напряжения питания цифрового датчика находится в диапазоне от 9 В до 24 В.
- Высокое - напряжение питания цифрового датчика более 24 В.

2. Параметр “Качество данных”

Параметр “Качество данных” информирует о неполадках в линии передачи данных цифрового датчика. Существует несколько состояний параметра “Качество данных”:

- Не поддерживается - цифровой датчик не обладает функцией контроля качества данных.
- Плохо - в линии передачи данных существуют проблемы. Качество данных неудовлетворительное, текущим данным верить нельзя.
- Хорошо - возможно в линии передачи данных существуют проблемы. Следует обратить внимание на данные.
- Отлично - в линии передачи данных проблемы отсутствуют.

3. Качество синхронизации

Параметр “Качество синхронизации” информирует о состоянии синхронизации цифрового датчика. Функция действительна только для цифровых датчиком работающих по интерфейсу CAN. Существует несколько состояний параметра “Качество синхронизации”:

- Не поддерживается - цифровой датчик не обладает функцией контроля качества синхронизации.
- Плохо - синхронизация отсутствует. Проверьте GPS-антенну, либо модуль синхронизации ZET 7175.
- Хорошо - ????????????
- Отлично - ?????????????

4. Состояние чувствительного элемента

Параметр “Состояние чувствительного элемента” информирует о состоянии первичного преобразователя, подключенного к конкретному цифровому датчику. Существует несколько состояний параметра “Состояние чувствительного элемента”:

- Не поддерживается - цифровой датчик не обладает функцией контроля за состоянием чувствительного элемента.
- Ошибка - чувствительный элемент не подключен, не исправен, либо подключен неправильно.
- Хорошо - чувствительный элемент подключен корректно.

2.4. Диагностика синхронизации измерительной линии

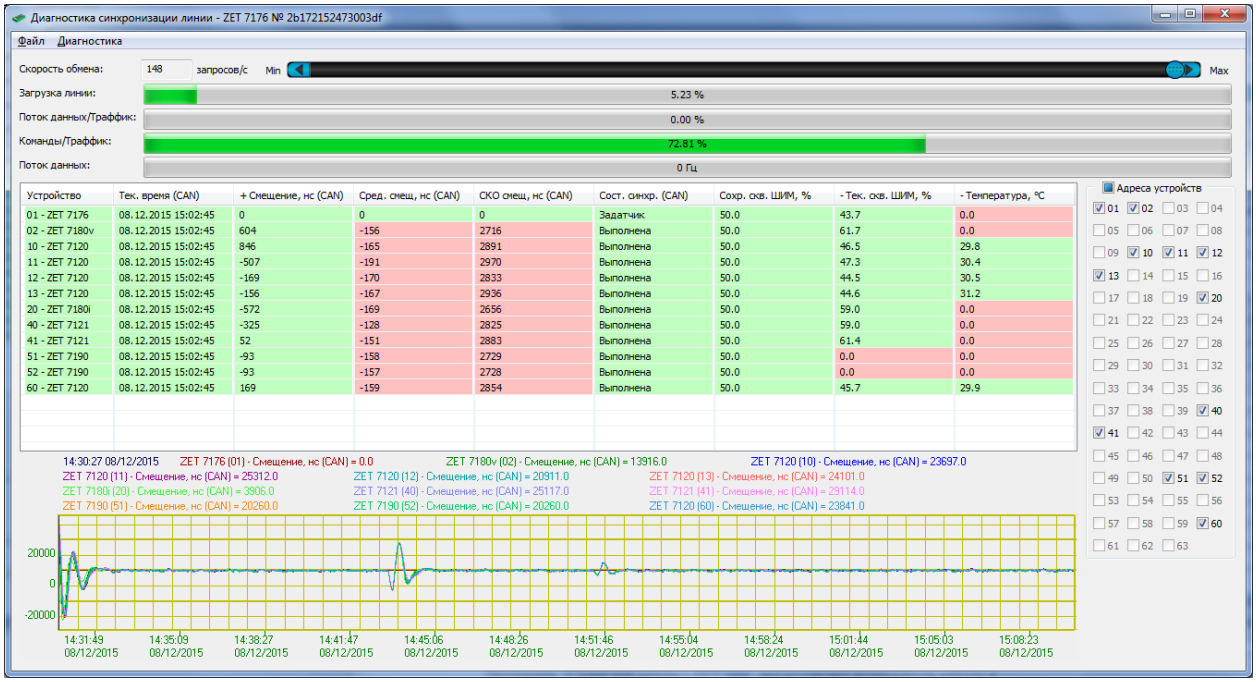

Рисунок 14.

Диагностика синхронизации измерительной линии заключается в диагностике по нескольким параметрам:

1. Текущее время (CAN)

Внутренние часы цифрового датчика

2. Смещение, нс (CAN)

Мгновенное значение вычисленного смещения внутренних часов относительно часов задатчика (мастера).

3. Среднее смещение, нс (CAN)

Среднестатистическое значение смещения за определенный период времени.

4. СКО смещение, нс (CAN)

Среднеквадратичное отклонение смещения за определенный период времени.

5. Состояние синхронизации (CAN)

Состояние синхронизации цифрового датчика. Существует несколько состояний:

- Выполнена
- Выполняется

6. Сохраненная скважность ШИМ, %

Значение скважности ШИМ, заданная на этапе поверки. Выражается в %.

7. Текущая скважность ШИМ, %

Текущая скважность ШИМ, измеренная в настоящий момент времени. Нормальное значение должно находиться в диапазоне 25-75%.

8. Температура, ^{0}C

Температура платы.

2.5. Диагностика синхронизации измерительной сети

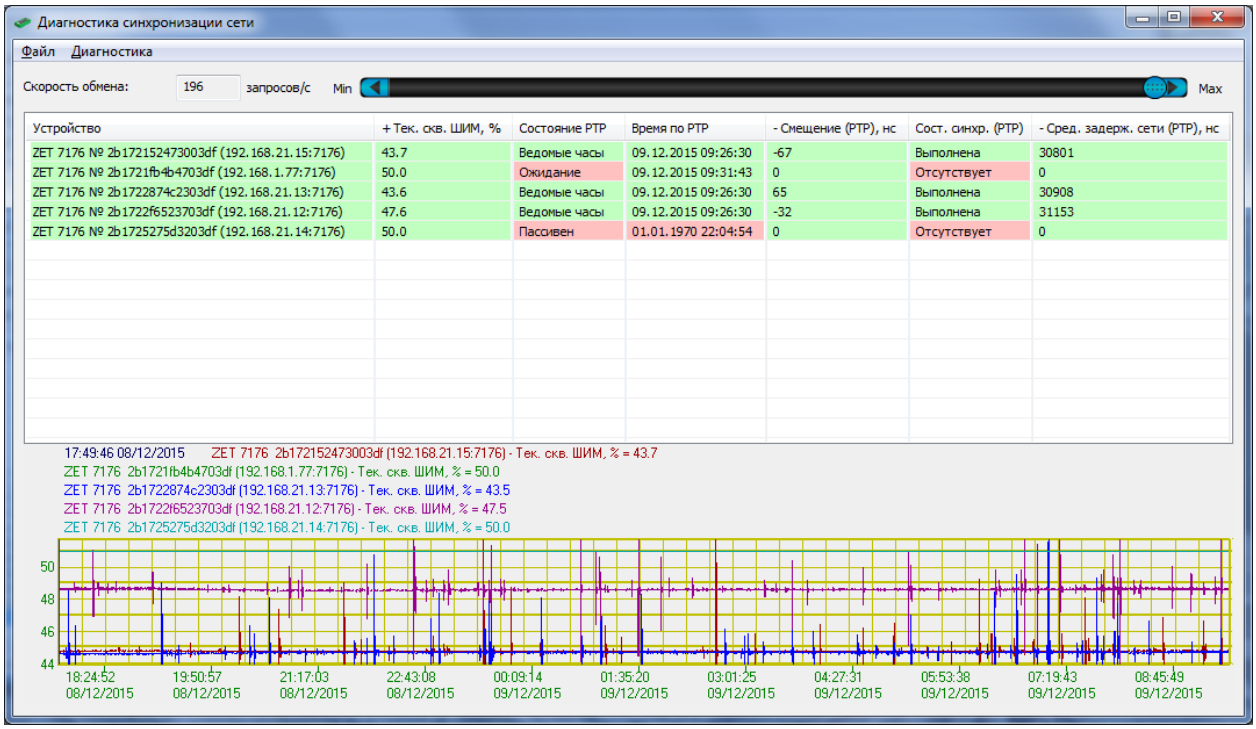

Рисунок 15.

3. Примеры интеграции в сторонние системы

Основные схемы интеграции ZETSENSOR в сторонние системы представлены на Рисунок 16, Рисунок 17, Рисунок 18, Рисунок 19.

Рисунок 16. Чтение информации с цифровых датчиков ZETLAB при использовании ПО сторонних производителей

Рисунок 17. Чтение информации с цифровых датчиков ZETLAB при использовании контроллеров и ПО сторонних производителей

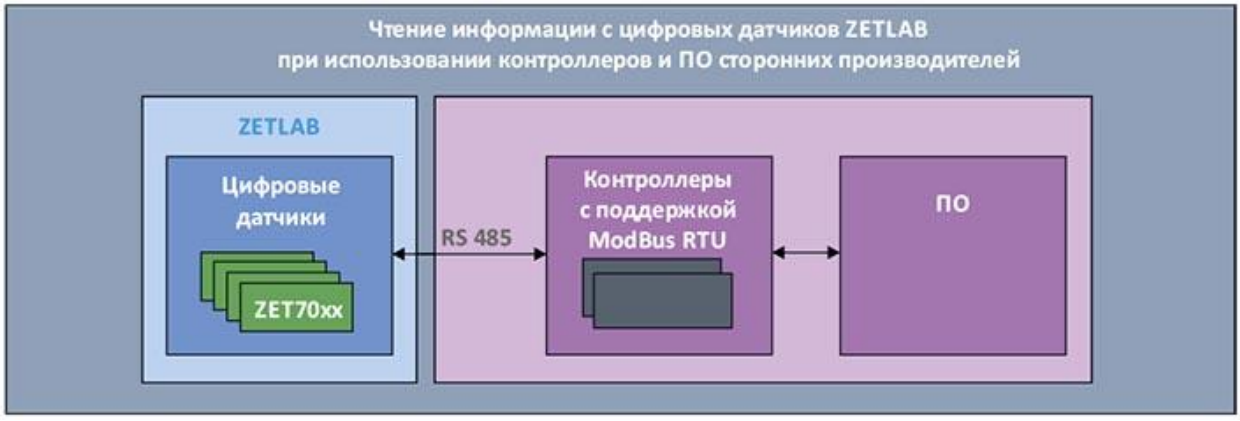

Рисунок 18. Чтение информации с цифровых датчиков сторонних производителей при использование преобразователей интерфейса и ПО ZETLAB

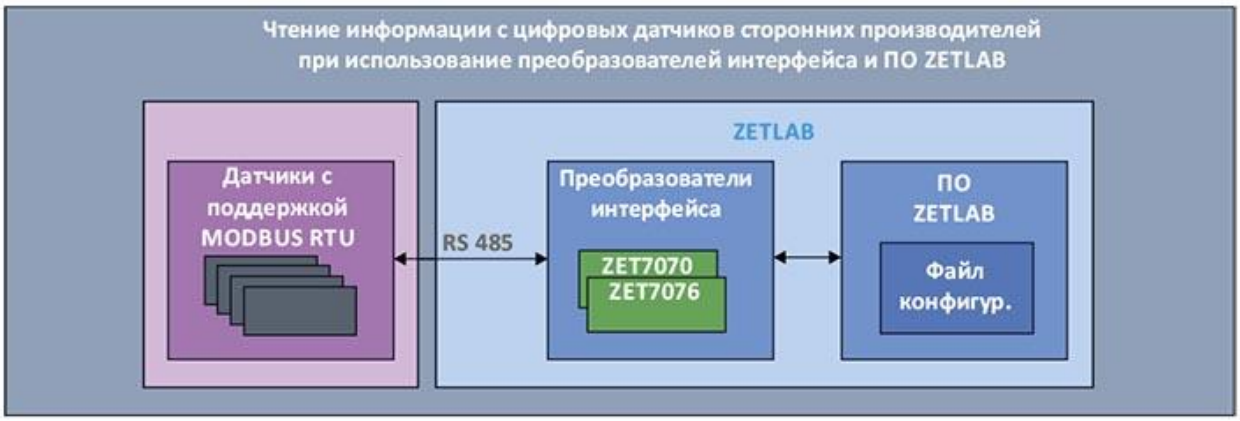

Рисунок 19. Конфигурирование режимов работы цифровых датчиков ZETLAB под управлением ПО сторонних производителей

3.1. Работа с ZET 7070 в качестве COM-порта (Windows)

Необходимо зайти в *Диспетчер устройств* → *Контроллер USB* выбрать соответствующее устройство. В нашем случае это ZETSENSOR USB (см. Рисунок 20), открыть его свойства и во вкладке «Дополнительно» выставить галочку напротив «*Загрузить VCP*» (см. Рисунок 21).

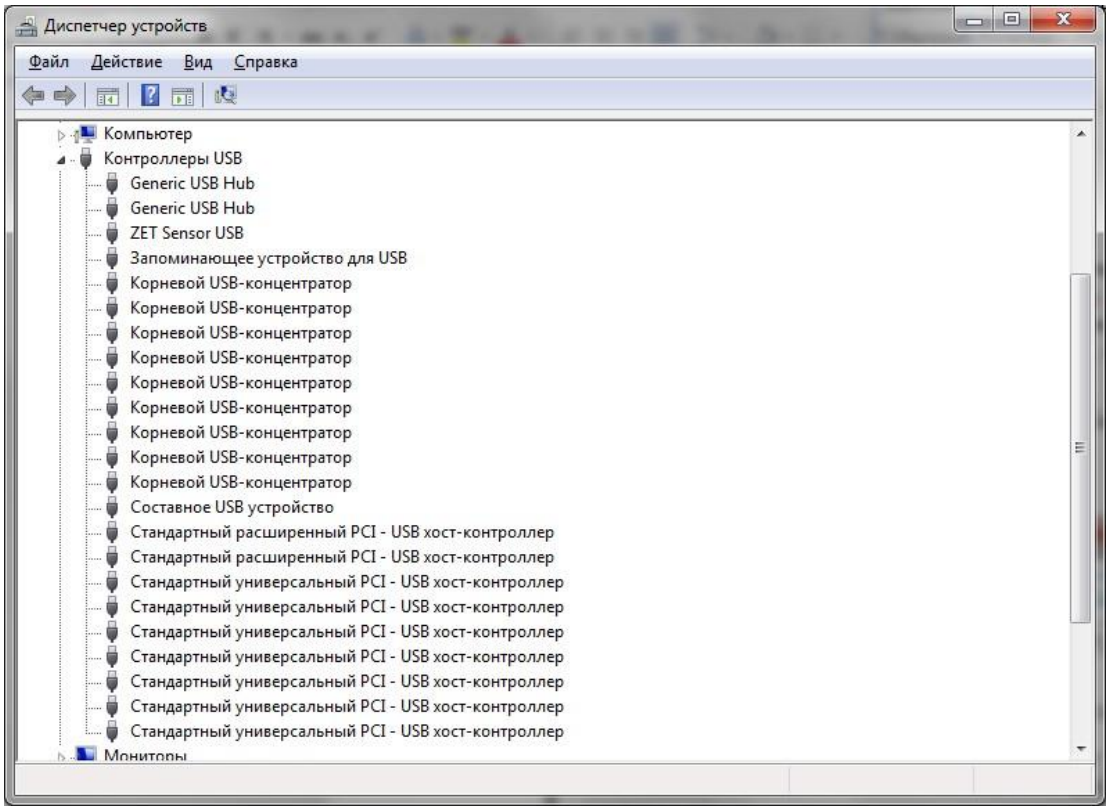

Рисунок 20.

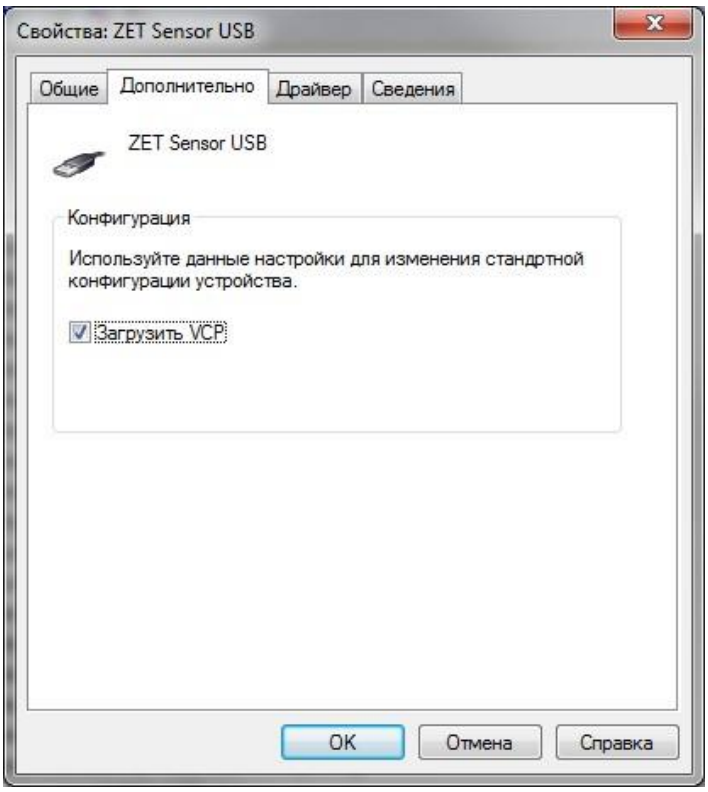

Рисунок 21.

Далее необходимо выключить и включить заново контроллер/преобразователь интерфейса к ПК.

3.2. Работа с ZET 7070 в качестве COM-порта (Linux)

Чтобы подключить ZET 7070 к системе с Linux, требуется настроить его VID и PID номера. Например, в Ubuntu 14.04 LTS это можно сделать с помощью следующей команды (с правами администратора): echo 10e8 3820 > /sys/bus/usb-serial/drivers/ftdi_sio/new_id. После этого, если подключить ZET 7070 к USB порту на ПК с Linux, то в списке драйверов должно появиться устройство /dev/ttyUSB0 (или с другим номером, например, /dev/ttyUSB2). Проверить подключение можно с помощью команды dmesg.

3.3. Подключение ZETSENSOR к Modbus Poll по протоколу Modbus.

После запуска программы необходимо ее настроить должным образом. Для этого зайти в верхнее меню ***Connection***, выбрать пункт *Connect…*, т.е. подключение к контроллеру, который преобразует USB в COM-порт. Если программа в бесплатном режиме, сперва откроется окно с просьбой ввести регистрационный ключ, следует пропустить данную операцию посредством нажатия кнопки ОК. Далее откроется окно с возможностью выбора устройства (см. Рисунок 22). Настройте как показано на Рисунок 22.

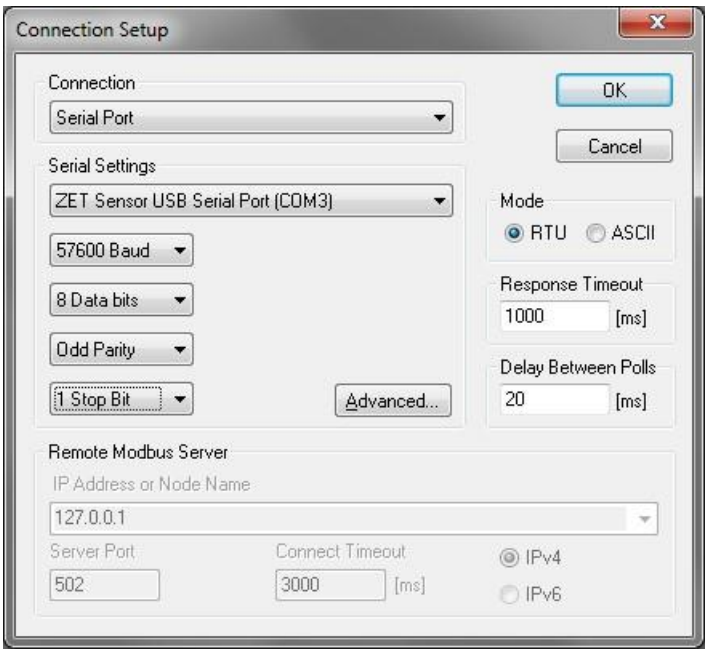

Рисунок 22.

В поле (см. Рисунок 23) идет опрос модуля, сканирование любого цифрового датчика (RS-485) ZET 70XX. Как видно на рисунке – имеется ошибка Timeout Error, которая возникает из-за того, что опрашивается устройство с адресом 1, нам же нужно выставить адрес существующего модуля. Узнать адрес модуля можно с помощью программы ***ZETLAB → Время ZETServer***, предварительно необходимо в Modbus Poll выполнить Disconnect, на Рисунок 24 показано, где отображается адрес устройства в ZETLAB.

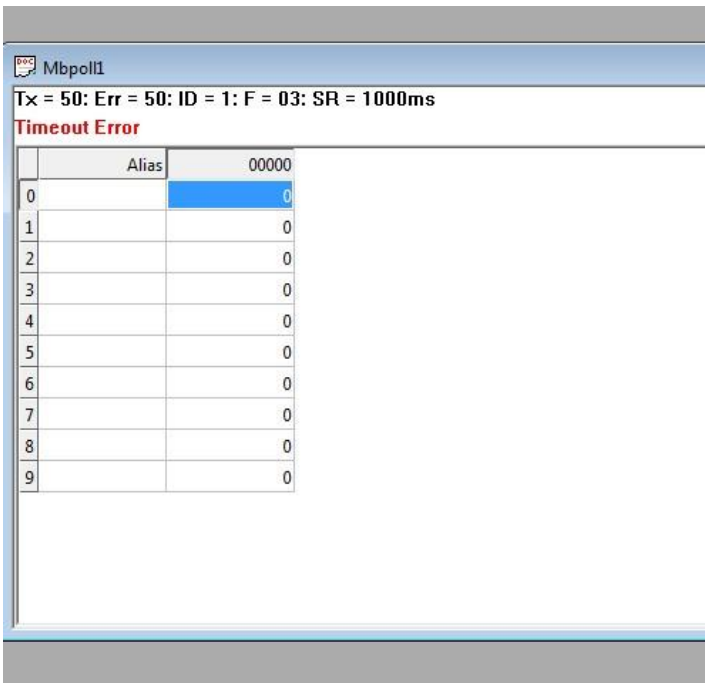

Рисунок 23.

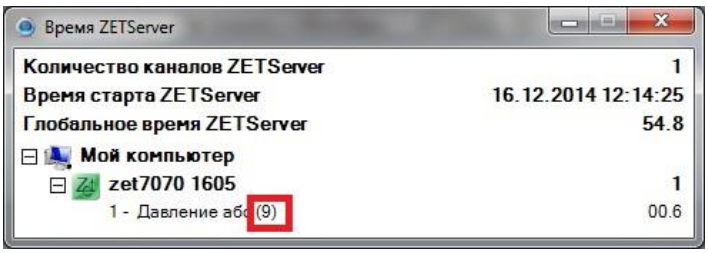

Рисунок 24.

Возвращаемся вновь к программе **Modbus Poll**, выполняем ***Connection→Connect***. Далее следует выставить адрес модуля и другие настройки в окне ***Setup→Read/Write Definition*** как показано на Рисунок 25. В ячейке *Address* выставляем значение, указанное в таблице, в нашем случае используется датчик абсолютного давления ZET 7012 и его адрес «20».

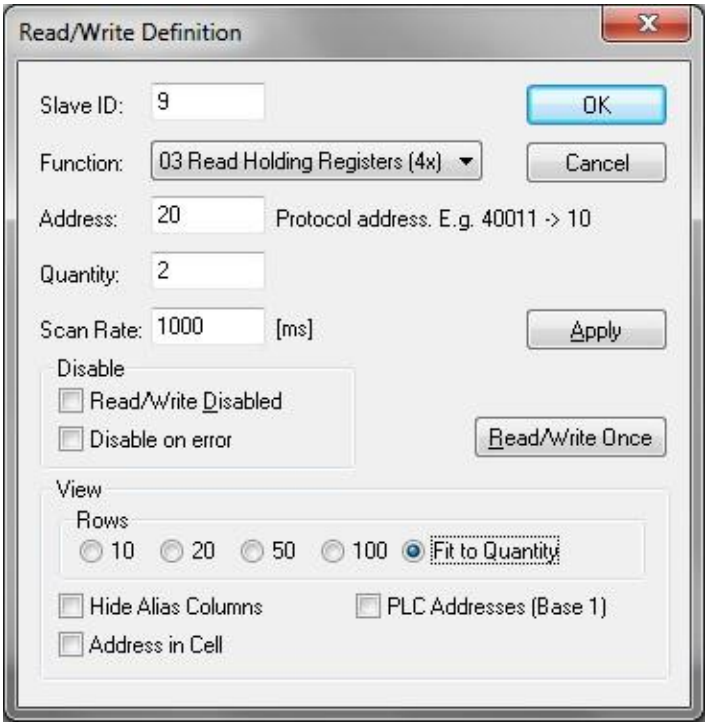

Рисунок 25.

Чтобы получать данные в нужном формате, также необходимо выставить ряд настроек. Необходимо зайти «Display» и выбрать тип переменных, в нашем случае это Float CD AB (см. Рисунок 26).

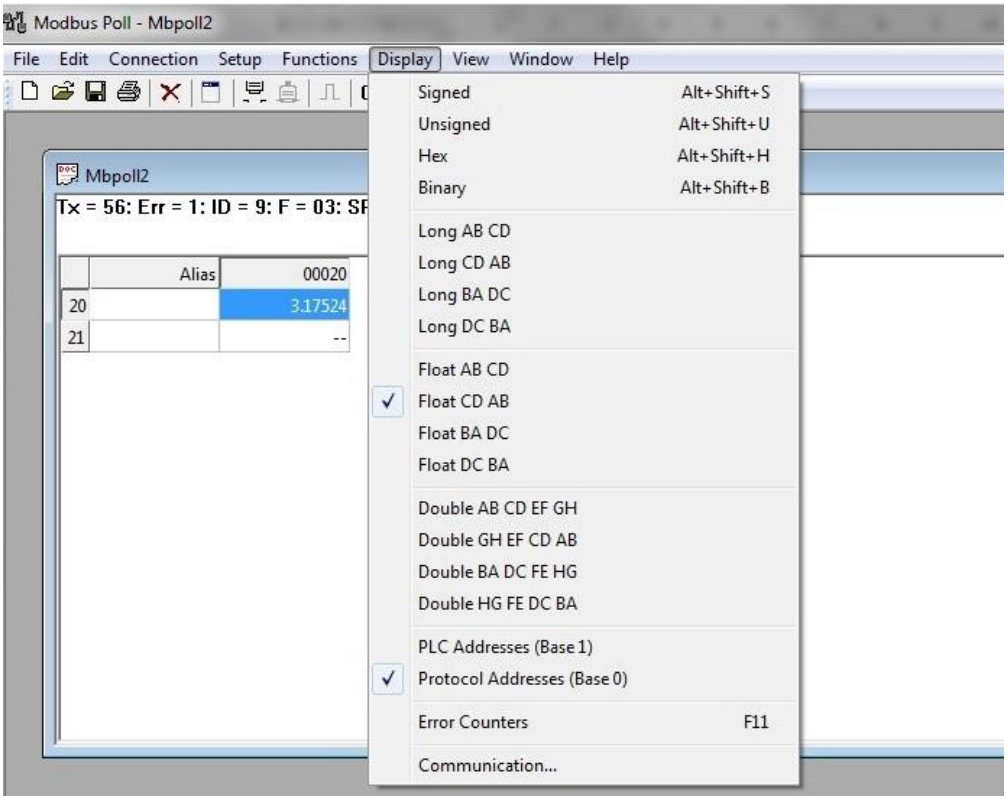

Рисунок 26.

Если устройств несколько, то для каждого устройства следует открыть свое окно посредством функции New из меню File и выполнить все вышеописанные настройки для опроса модуля. В результате после все настроек в окне программы вы сможете увидеть данные, поступающие с интеллектуального датчика и убедиться в корректности его работы.

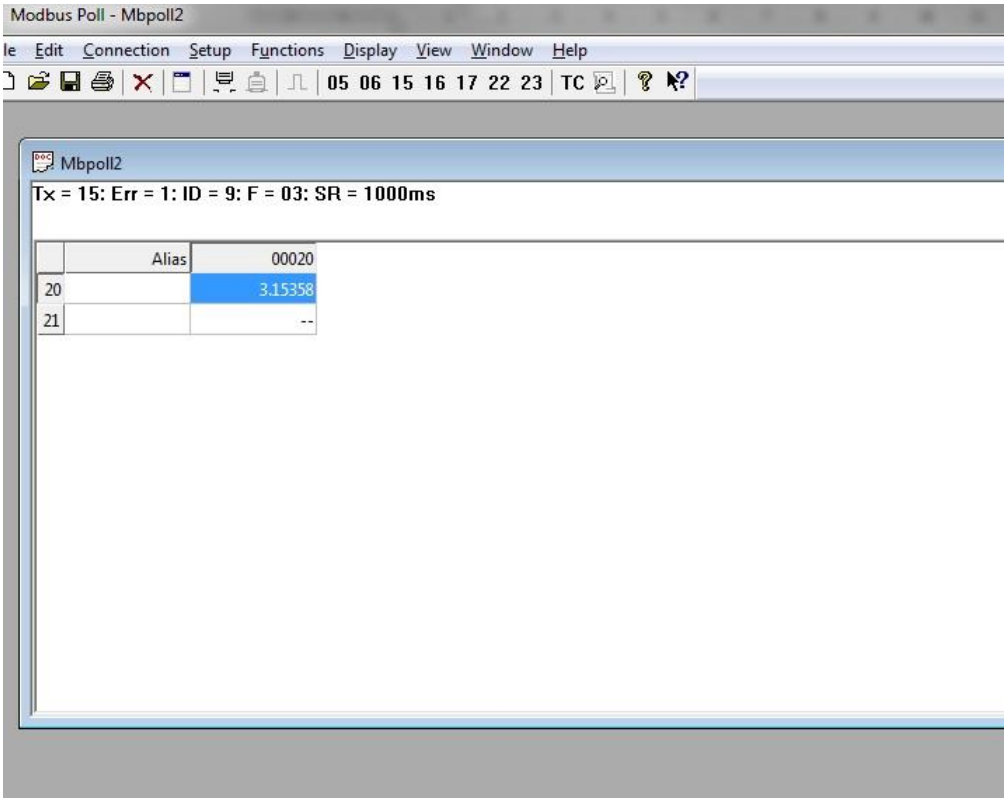

Рисунок 27.

3.4. Подключение ZETSENSOR к Simply Modbus по протоколу Modbus.

Программа Simply Modbus позволяет сформировать запрос цифровому датчику ZETSENSOR (см. Рисунок 28).

Посмотреть номер COM порта или настроить скорость передачи данных можно с помощью программы "Диспетчер устройств", находящуюся во вкладке «Сервисные» панели ZETPanel. У всех датчиков **ZET 70XX** 0x14 — это адрес в памяти датчика, содержащий значения канала (возможно лишь считывание).

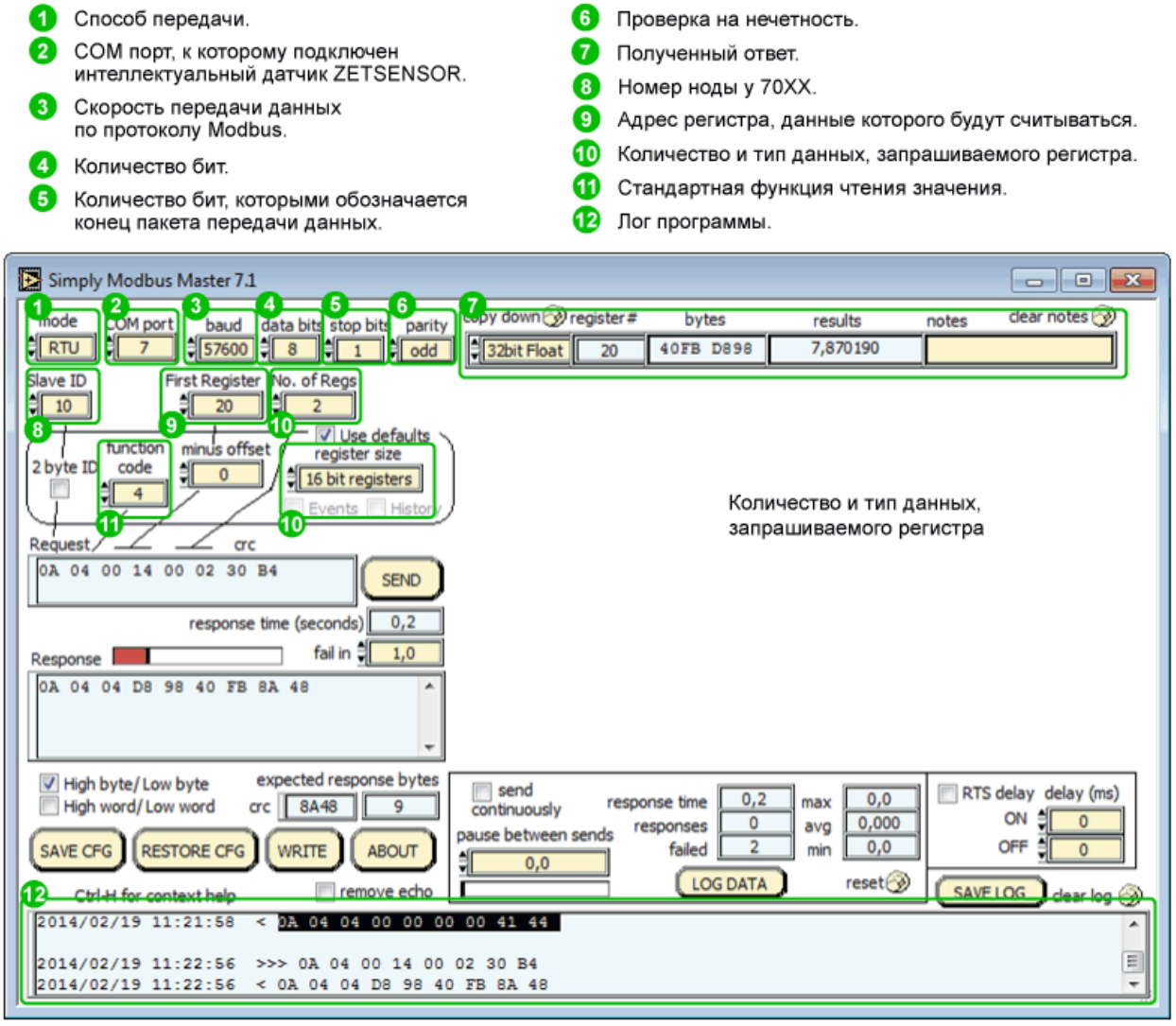

Рисунок 28. Главное окно программы Simply Modbus

3.5. Подключение ZETSENSOR к TRACE MODE по протоколу Modbus

Рассмотрим подключение цифровых датчиков ZETSENSOR по протоколу Modbus с помощью преобразователя интерфейса ZET7070 (USB-RS485) в интегрированной среде разработки TRACE MODE. В качестве примера возьмем следующую связку: ZET 7070 (преобразователь интерфейса USB-RS485) + ZET 7052 (цифровой трехкомпонентный датчик линейного ускорения), как показано на Рисунок 29.

Рисунок 29.

Выполняем действия в соответствии с учебным фильмом, демонстрирующим подключение оборудования через последовательный порт RS 232/485 по протоколу MODBUS RTU (http://www.adastra.ru/products/drivers/modbus/).

1) В «Источники/Приемники» добавляем группу Modbus и в ней создаем три компонента (для осей X, Y и Z датчика линейного ускорения) Rout_Float(3) для чтения 4 байт с приведением к float командой ReadHoldingRegisters. Настраиваем каждый из компонентов (см. Рисунок 30, Рисунок 31, Рисунок 32). В качестве имени берем название оси датчика. Номер порта для каждого выставляем 0x8 (COM9). Адрес устройства в сети MODBUS 0x2. Адрес регистра для чтения данных (канал) выставляем в соответствии с таблицей регистров для датчика ZET7052.

Примечание: таблицы адресов регистров для датчиков генерируются при помощи утилиты SensorWork при подключении к компьютеру. Фрагмент сгенерированной таблицы представлен на Рисунок 33.

Для оси X это адрес 0x14, для оси Y – 0x3a, для оси Z – 0x60. Все остальные настройки оставляем без изменений.

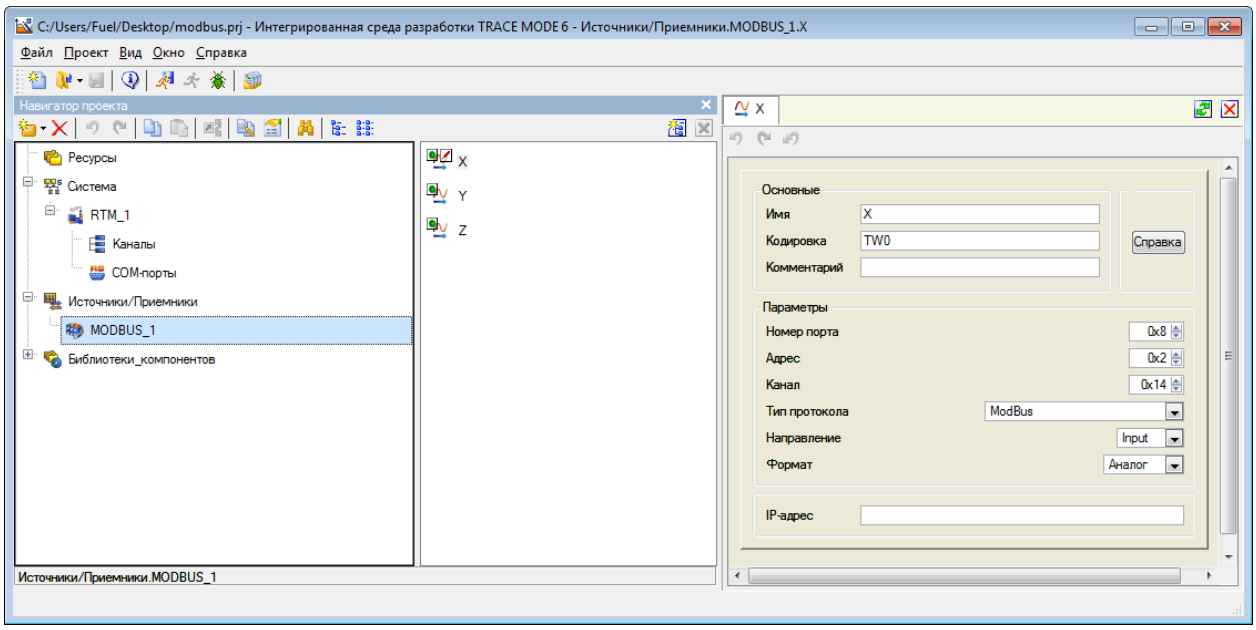

Рисунок 30.

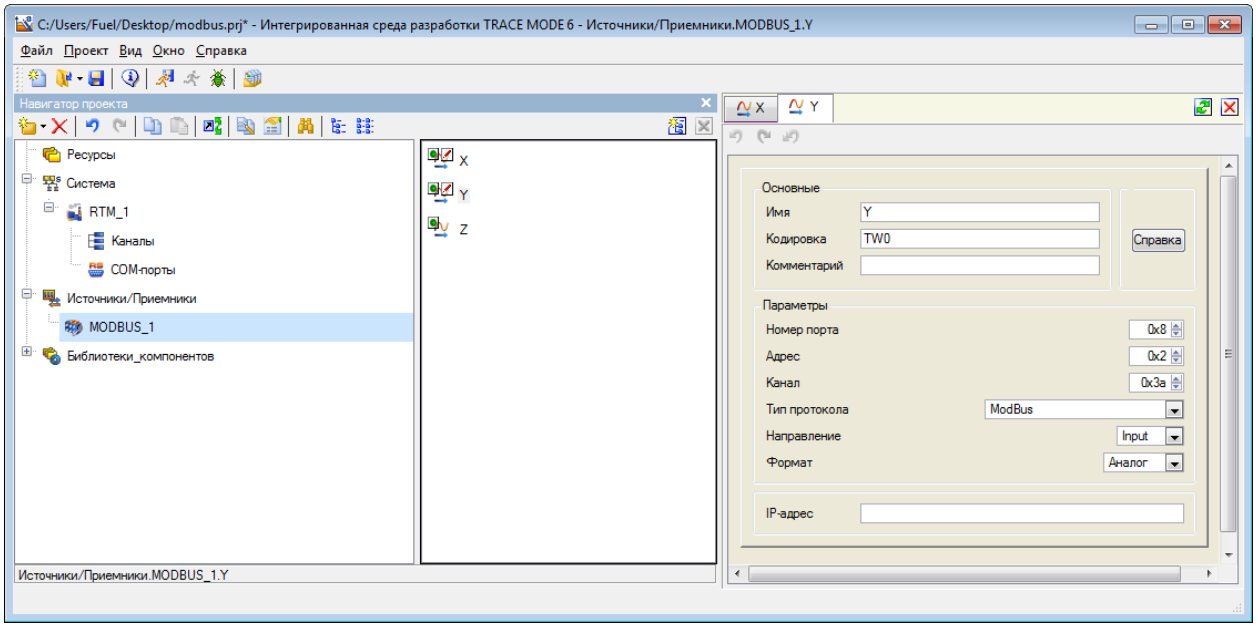

Рисунок 31.

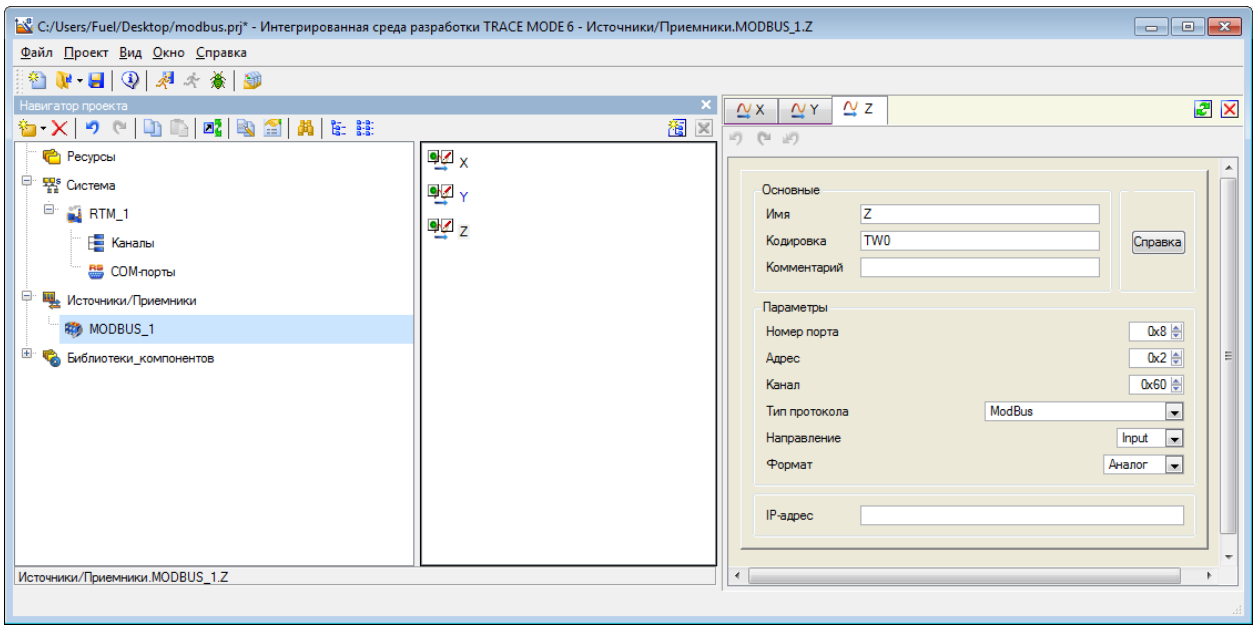

Рисунок 32.

Название параметра	Адрес, WORD hex (WORD dec)	Адрес в структуре, WORD hex (WORD dec)	Тип данных	Количество регистров (в словах)	Принимаемые значения
Информация (Настройка адреса устройства), ID = 0x18c, адрес = 0x00 (00)					
Тип устройства	0x04 (04)	0x04 (04)	int (тип 17)	2	Произвольное значение (только чтение)
Серийный номер устройства	0x06 (06)	0x06 (06)	longlong (тип 14)	4	Произвольное значение (только чтение)
Дата выпуска программного обеспечения	0x0a (10)	0x0a (10)	time (тип 11)	2	Произвольное значение (только чтение)
Дата выпуска аппаратной части	0x0c (12)	0x0c (12)	time (тип 11)	2	Произвольное значение (только чтение)
Адрес устройства от 1 до 63	0x0e (14)	0x0e (14)	int (тип 3)	2	Произвольное значение
Канал 1 (Настройка входного канала), ID = 0xd0, адрес = 0x10 (16)					
Текущее значение канала (в ед. изм.)	0x14 (20)	0x04 (04)	float (тип 6)	2	Произвольное значение (только чтение)
Частота обновления выходного сигнала, Гц	0x16 (22)	0x06 (06)	float (тип 6)	2	Произвольное значение (только чтение)
Единица измерения	0x18 (24)	0x08 (08)	char[8] (тип 1)	4	Произвольное значение (только чтение)
Наименование канала	0x1c (28)	0x0c (12)	char[32] (тип 1)	16	Произвольное значение
Минимальный уровень (в ед. изм.)	0x2c (44)	0x1c (28)	float (тип 6)	2	Произвольное значение (только чтение)
Максимальный уровень (в ед. изм.)	0x2e (46)	0x1e (30)	float (тип 6)	2	Произвольное значение (только чтение)

Рисунок 33.

2) Три созданных компонента перемещаем в группу «Каналы» узла «RTM_1» (см. Рисунок 34) Настройки каждого из трех каналов TRACE MODE оставляем по умолчанию.

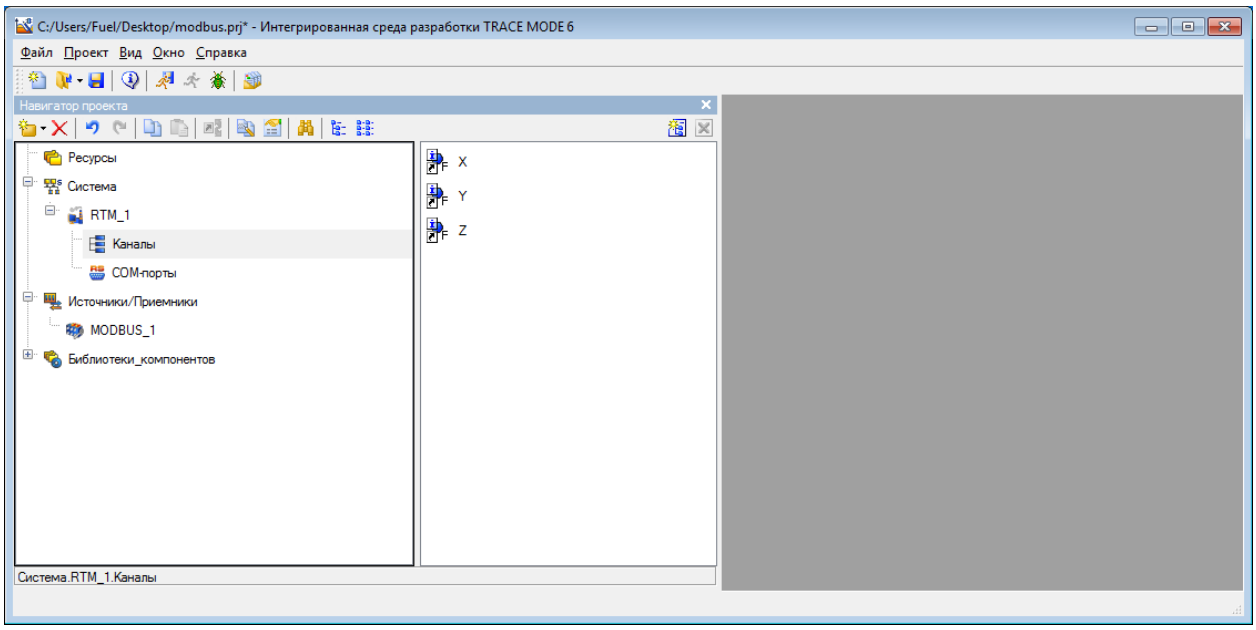

Рисунок 34.

3) В узле «RTM_1» создаем группу COM-порты. Созданный COM-порт открываем на редактирование и настраиваем (см. Рисунок 35). Номер порта присваиваем COM9, скорость 19200 бит/с, контроль четности 8-1-о. Галочка CRC должна быть включена. Все остальные настройки остаются без изменений.

Примечание: такие настройки датчика как скорость и контроль четности можно считать из датчика при помощи утилиты SensorWork. Настройка параметров в датчике осуществляется через стандартное ПО ZETLAB SENSOR, которое входит в комплект поставки датчика.

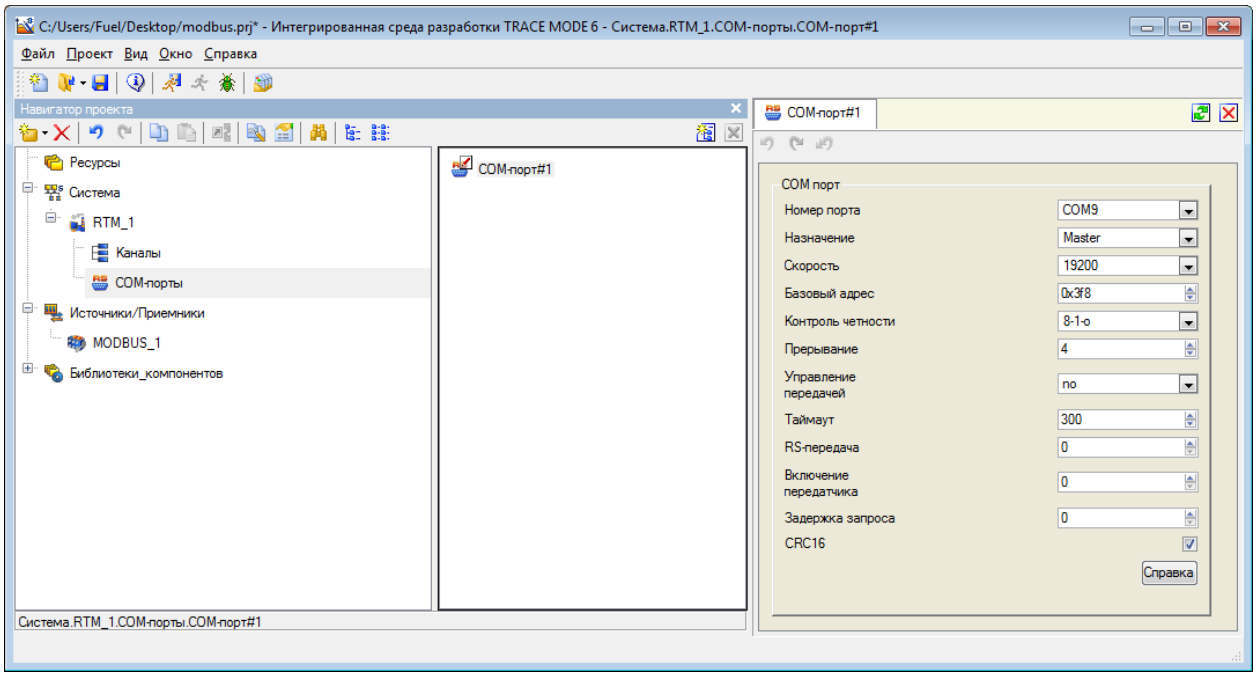

Рисунок 35.

4) Оформляем рабочий экран так, чтобы на нем получилось три текстовых поля, куда будет выдаваться текущее показание с датчика, а также тренд для отображения изменения значений ускорения во времени (см. Рисунок 36).

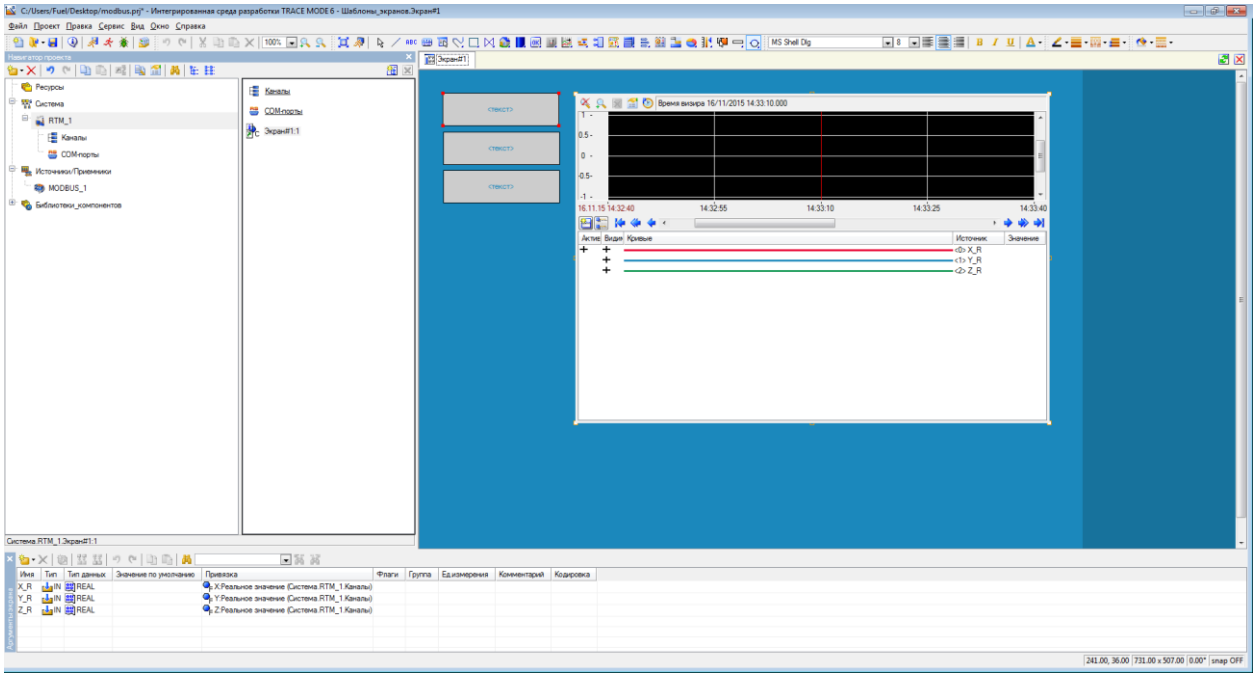

Рисунок 36.

5) Сохраняем проект на жесткий диск и для монитора реального времени. Запускаем профайлер. Результат работы отображен на Рисунок 37 (в процессе записи тренда датчику меняли положение в пространстве).

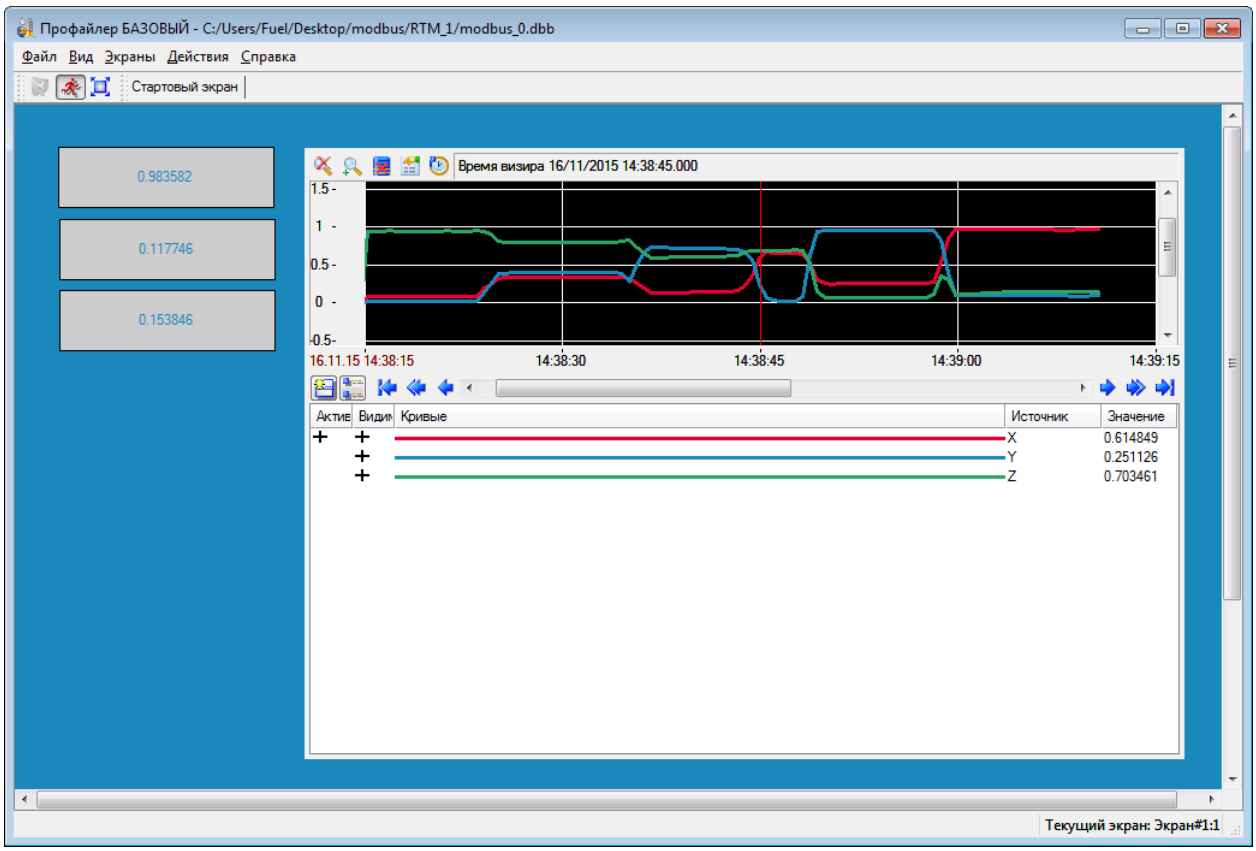

Рисунок 37.

3.6. Подключение ZETSENSOR к TRACE MODE через OPC-сервер

В качестве примера возьмем следующую связку: ZET 7070 (преобразователь интерфейса USB-RS485) + ZET 7052 (цифровой трехкомпонентный датчик линейного ускорения), как показано на Рисунок 38.

Рисунок 38.

Настраиваем OPC-сервер ZET.7xxxOPC для работы с выбранными устройствами. Для этого запускаем программу SensorWork, которая входит в комплект поставки датчиков, осуществляем поиск устройств (см. Рисунок 39), активируем режим OPC-сервера (см. Рисунок 40), убеждаемся в наличии данных в OPC-тегах (см. Рисунок 41), закрываем окно «Работа с OPC-сервером», настраиваем программу для запуска в режиме OPC-сервера при последующих стартах программы (см. Рисунок 42).

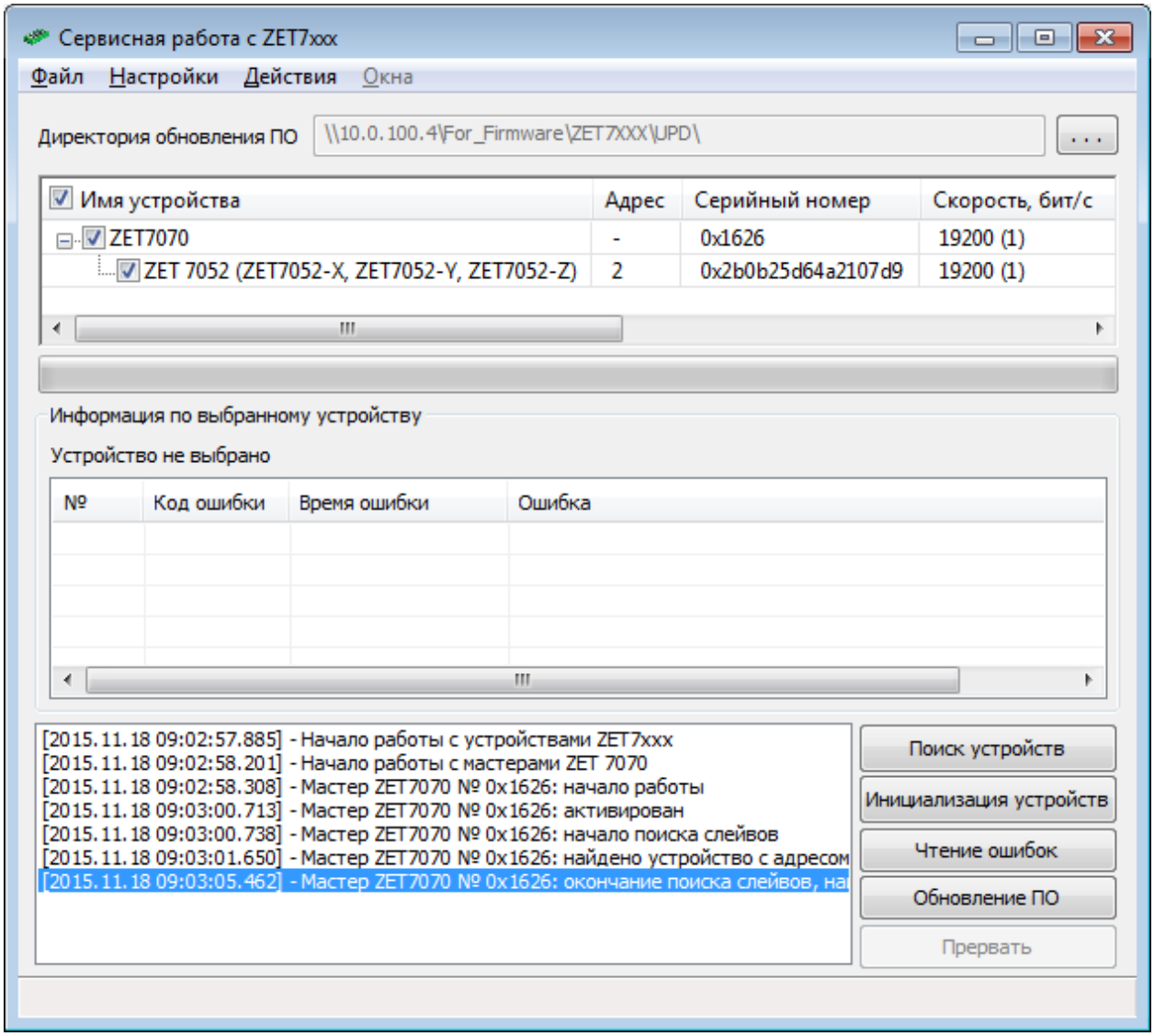

Рисунок 39.

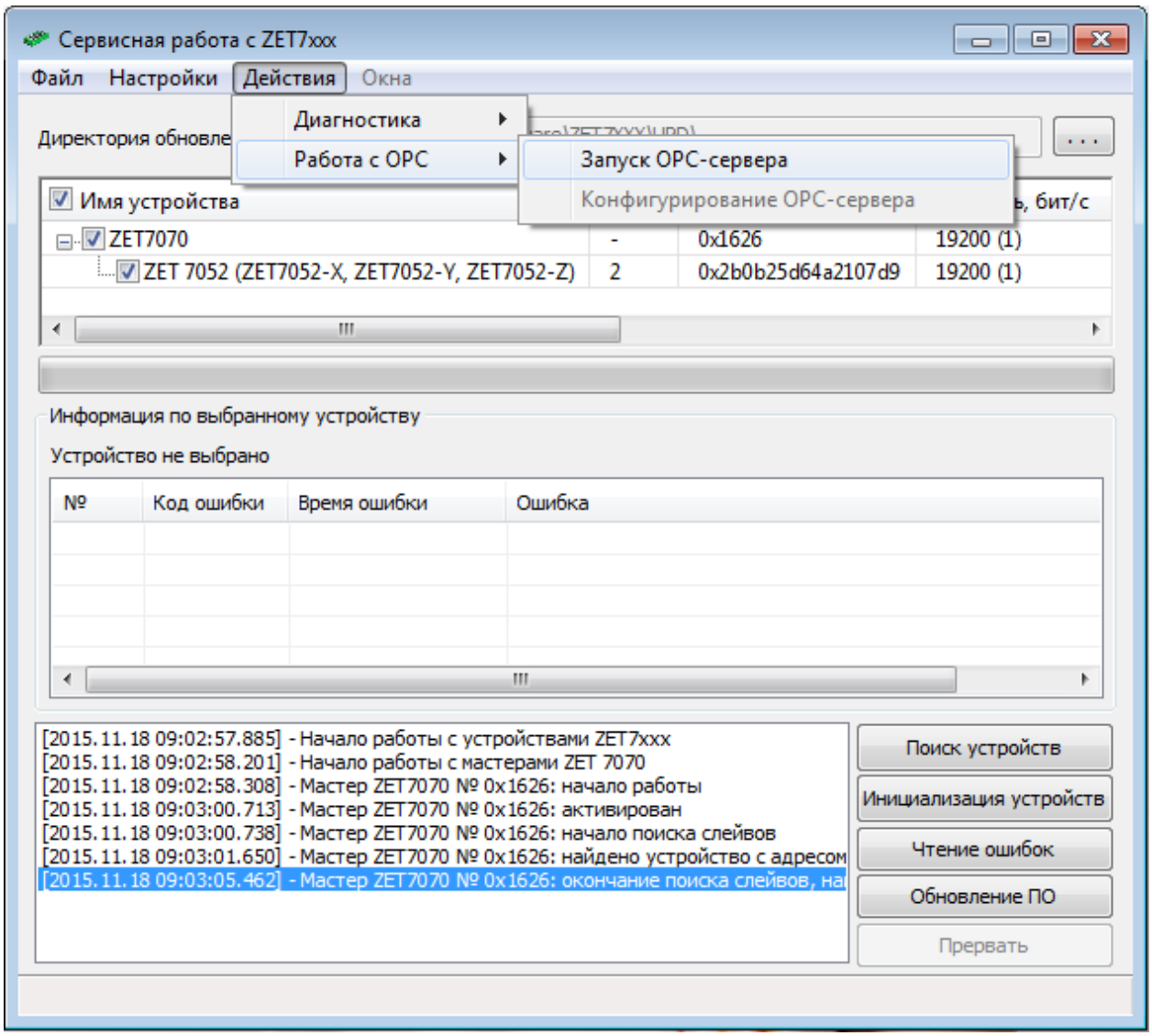
Сервисная работа с ZET7xxx
Файл
Настройки
Действия
Окна
Диагностика
Работа с OPC
Запуск OPC-сервера
Конфигурирование OPC-сервера
Директория обновле
Имя устройства
ZET7070
ZET 7052 (ZET7052-X, ZET7052-Y, ZET7052-Z)
0x1626
0x2b0b25d64a2107d9
19200 (1)
Информация по выбранному устройству
Устройство не выбрано
№
Код ошибки
Время ошибки
Ошибка
[2015.11.18 09:02:57.885] - Начало работы с устройствами ZET7xxx
[2015.11.18 09:02:58.201] - Начало работы с мастерами ZET 7070
[2015.11.18 09:02:58.308] - Мастер ZET7070 № 0x1626: начало работы
[2015.11.18 09:03:00.713] - Мастер ZET7070 № 0x1626: активирован
[2015.11.18 09:03:00.738] - Мастер ZET7070 № 0x1626: начало поиска слейвов
[2015.11.18 09:03:01.650] - Мастер ZET7070 № 0x1626: найдено устройство с адресом
[2015.11.18 09:03:05.462] - Мастер ZET7070 № 0x1626: окончание поиска слейвов, на
Поиск устройств
Инициализация устройств
Чтение ошибок
Обновление ПО
Прервать

Рисунок 40.

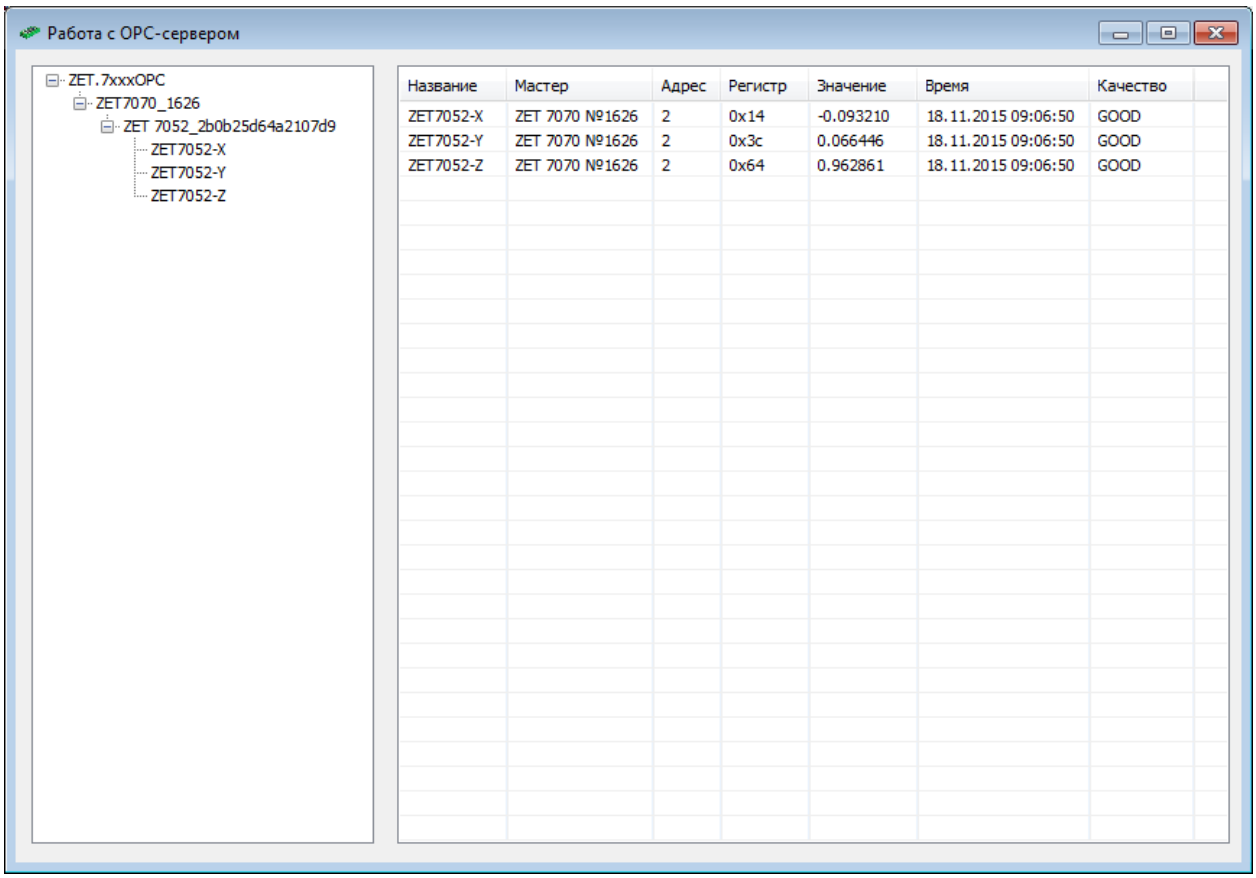
Работа с OPC-сервером
ZET.7xxxOPC
ZET7070_1626
ZET 7052_2b0b25d64a2107d9
ZET7052-X
ZET7052-Y
ZET7052-Z
Название
Мастер
Адрес
Регистр
Значение
Время
Качество
ZET7052-X
ZET 7070 №1626
2
0x14
-0.093210
18.11.2015 09:06:50
GOOD
ZET7052-Y
ZET 7070 №1626
2
0x3c
0.066446
18.11.2015 09:06:50
GOOD
ZET7052-Z
ZET 7070 №1626
2
0x64
0.962861
18.11.2015 09:06:50
GOOD

Рисунок 41.

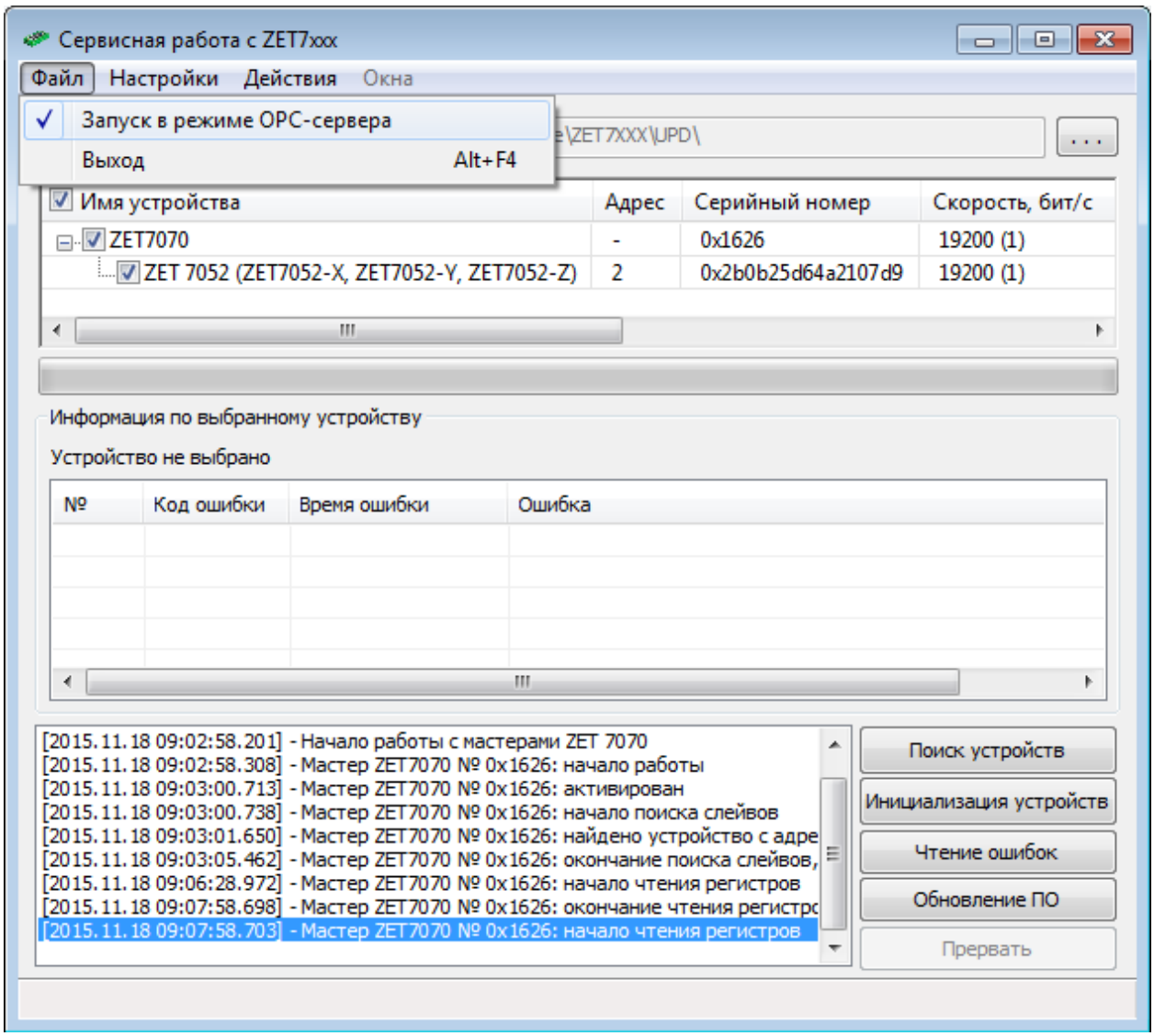

Рисунок 42.

Далее выполняем действия в соответствии с учебным фильмом, демонстрирующим подключение оборудования через встроенный OPC-клиент TRACE MODE (www.adastra.ru/products/drivers/opc_connect/):

1) В Источники/Приемники добавляем группу OPC, в ней создаем группу OPC_сервер. В группе OPC_сервер создаем три компонента OPC (для осей X, Y и Z датчика линейного ускорения) для чтения тегов. Настраиваем каждый из компонентов (см. Рисунок 43,Рисунок 44, Рисунок 45). В качестве имени берем название оси датчика. В строке «Сервер» нажимаем кнопку «Обзор», выбираем сервер ZET.7xxxOPC из списка доступных OPC-серверов и соответствующий тег для чтения. Режим меняем на ADVISE. Все остальные настройки оставляем без изменений.

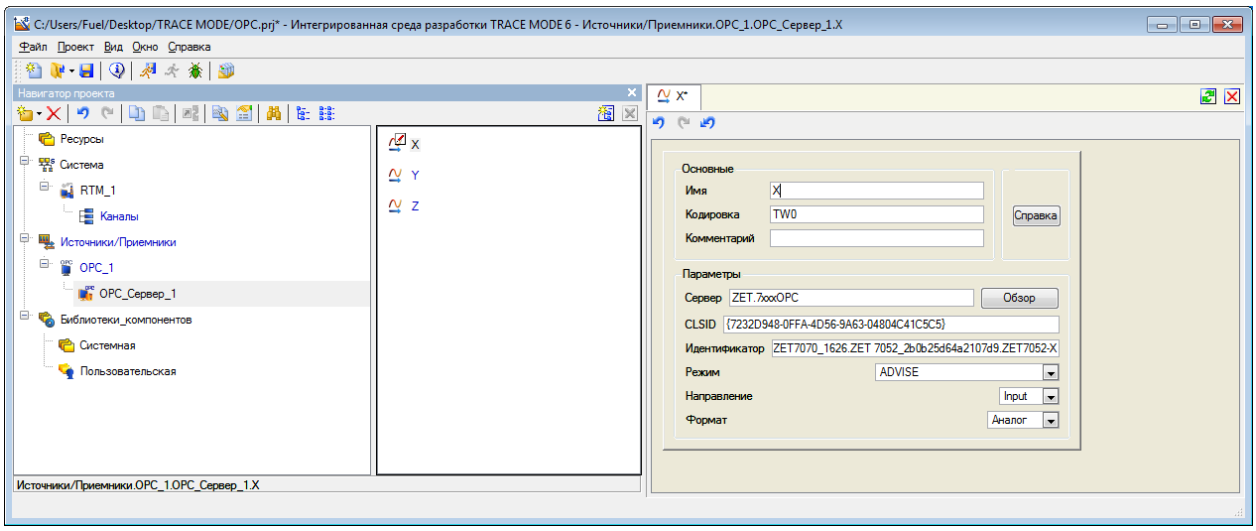

Рисунок 43.

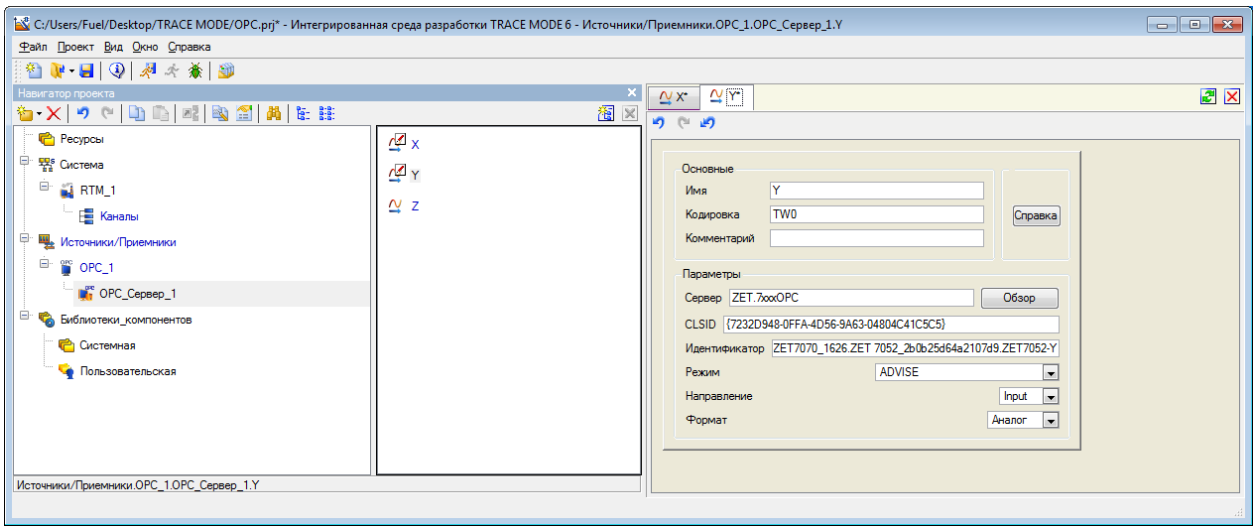

Рисунок 44.

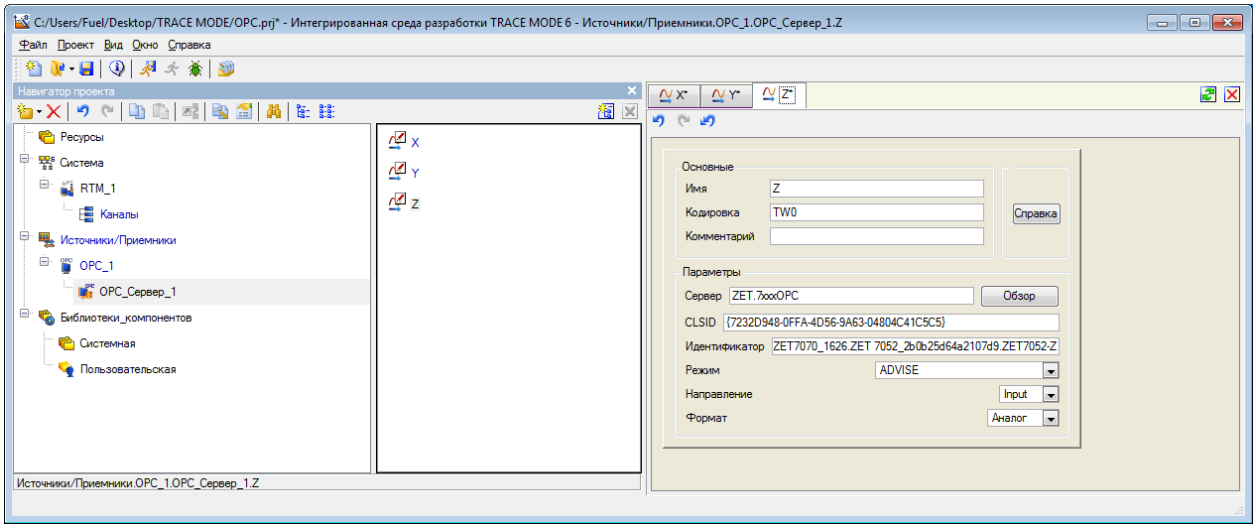

Рисунок 45.

2) Три созданных компонента перемещаем в группу «Каналы» узла «RTM_1» (см. Рисунок 46) Настройки каждого из трех каналов TRACE MODE оставляем по умолчанию.

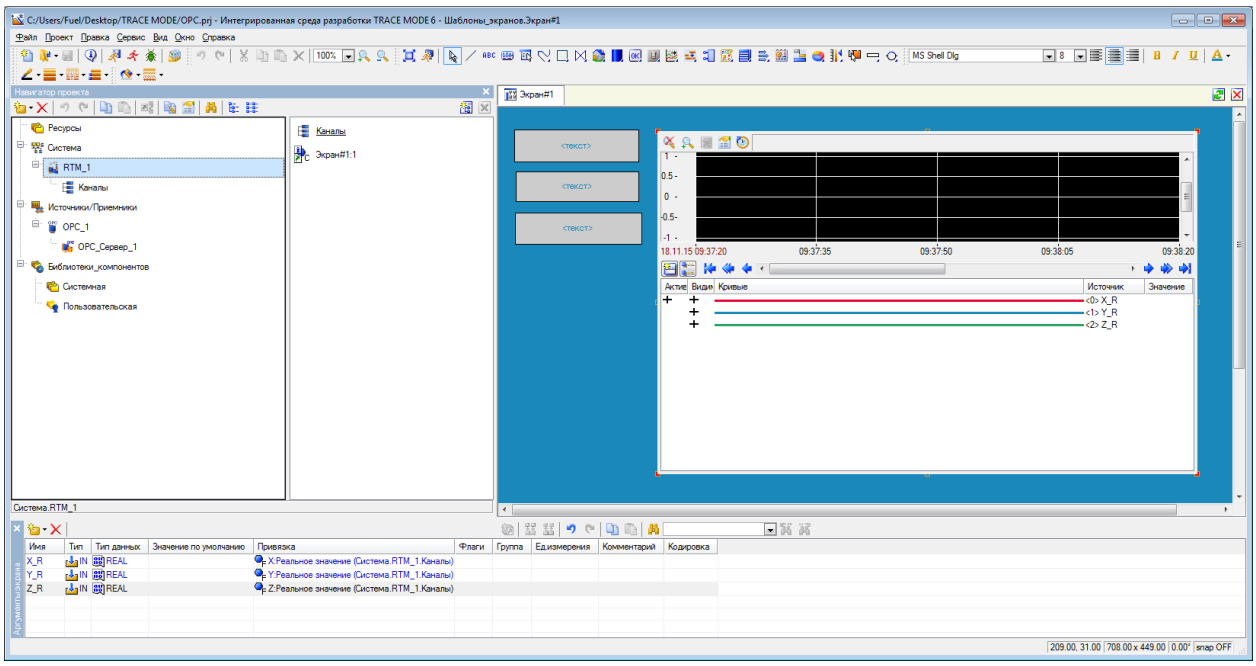

Рисунок 46.

3) Оформляем рабочий экран так, чтобы на нем получилось три текстовых поля, куда будет выдаваться текущее показание с датчика, а также тренд для отображения изменения значений ускорения во времени (см. Рисунок 47)

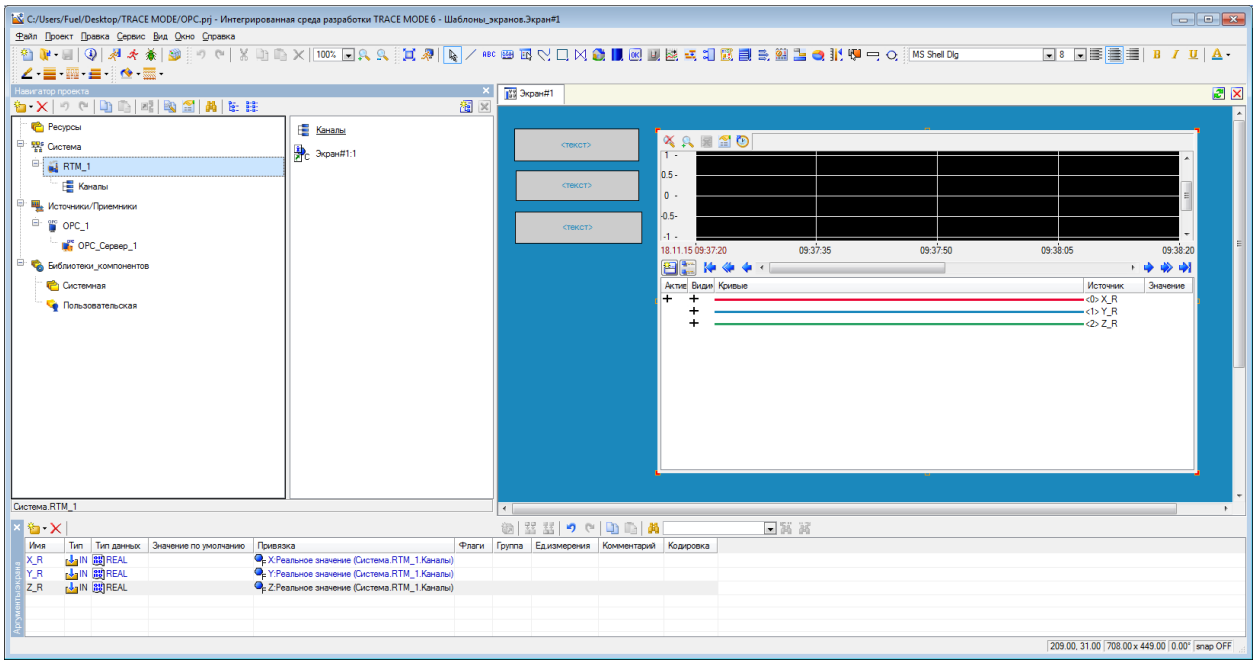

Рисунок 47.

4) Сохраняем проект на жесткий диск и для монитора реального времени. Запускаем профайлер. Результат работы отображен на Рисунок 48 (в процессе записи тренда датчику меняли положение в пространстве)

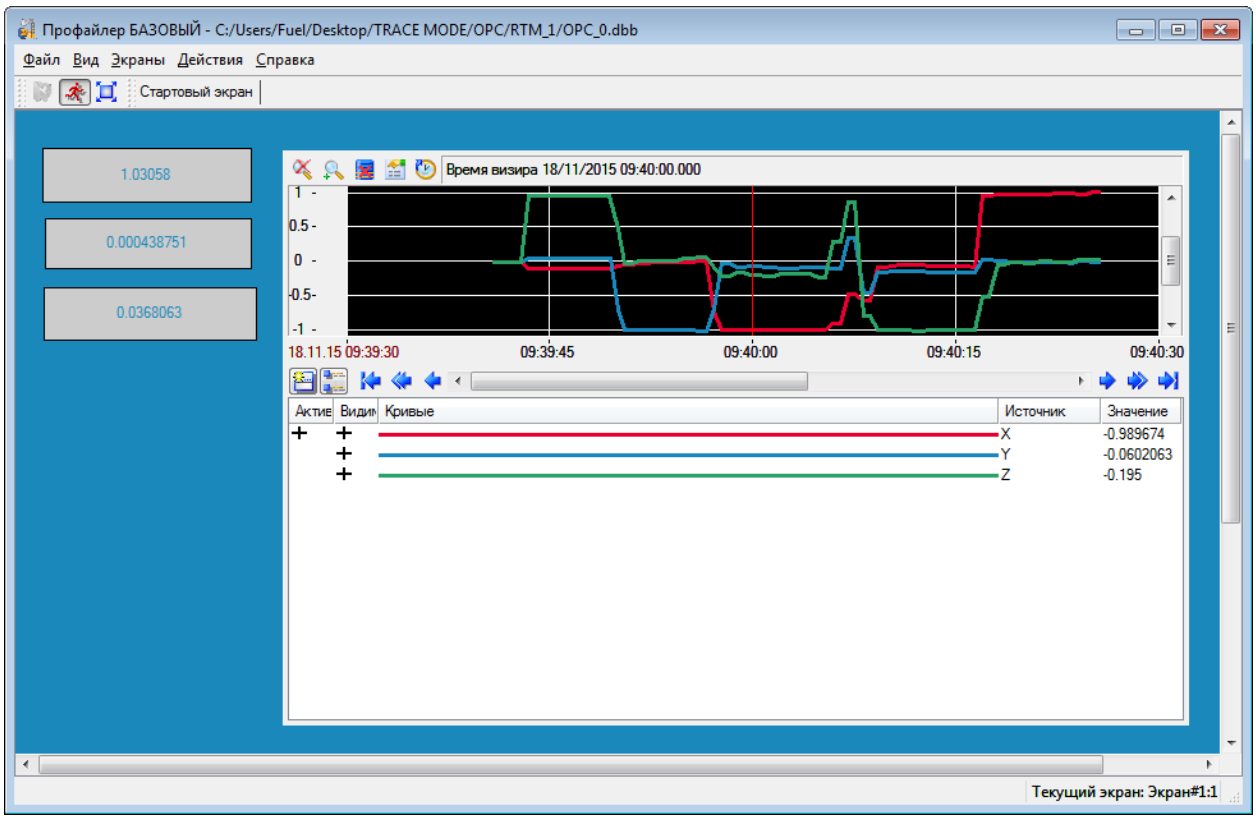

Рисунок 48.

3.7. Подключение ZETSENSOR к MasterSCADA через OPC-сервер

Рассмотрим подключение цифровых датчиков ZETSENSOR через OPC-сервер в SCADA-системе MasterSCADA на примере ZET7052 (цифровой трехкомпонентный датчик линейного ускорения) + ZET7070 (преобразователь интерфейса USB-RS485).

Рисунок 49.

Настраиваем OPC-сервер ZET.7xxxOPC для работы с выбранными устройствами. Для этого запускаем программу **SensorWork**, которая входит в комплект поставки датчиков, осуществляем поиск устройств (см. Рисунок 50), активируем режим OPC-сервера (см. Рисунок 51), убеждаемся в наличии данных в OPC-тегах (см. Рисунок 52), закрываем окно «Работа с OPC-сервером», настраиваем программу для запуска в режиме OPC-сервера при последующих стартах программы (см. Рисунок 53).

Примечание: более подробное описание работы с программным обеспечением **SensorWork** представлено в разделе Сервисная работа с ZETSENSOR.

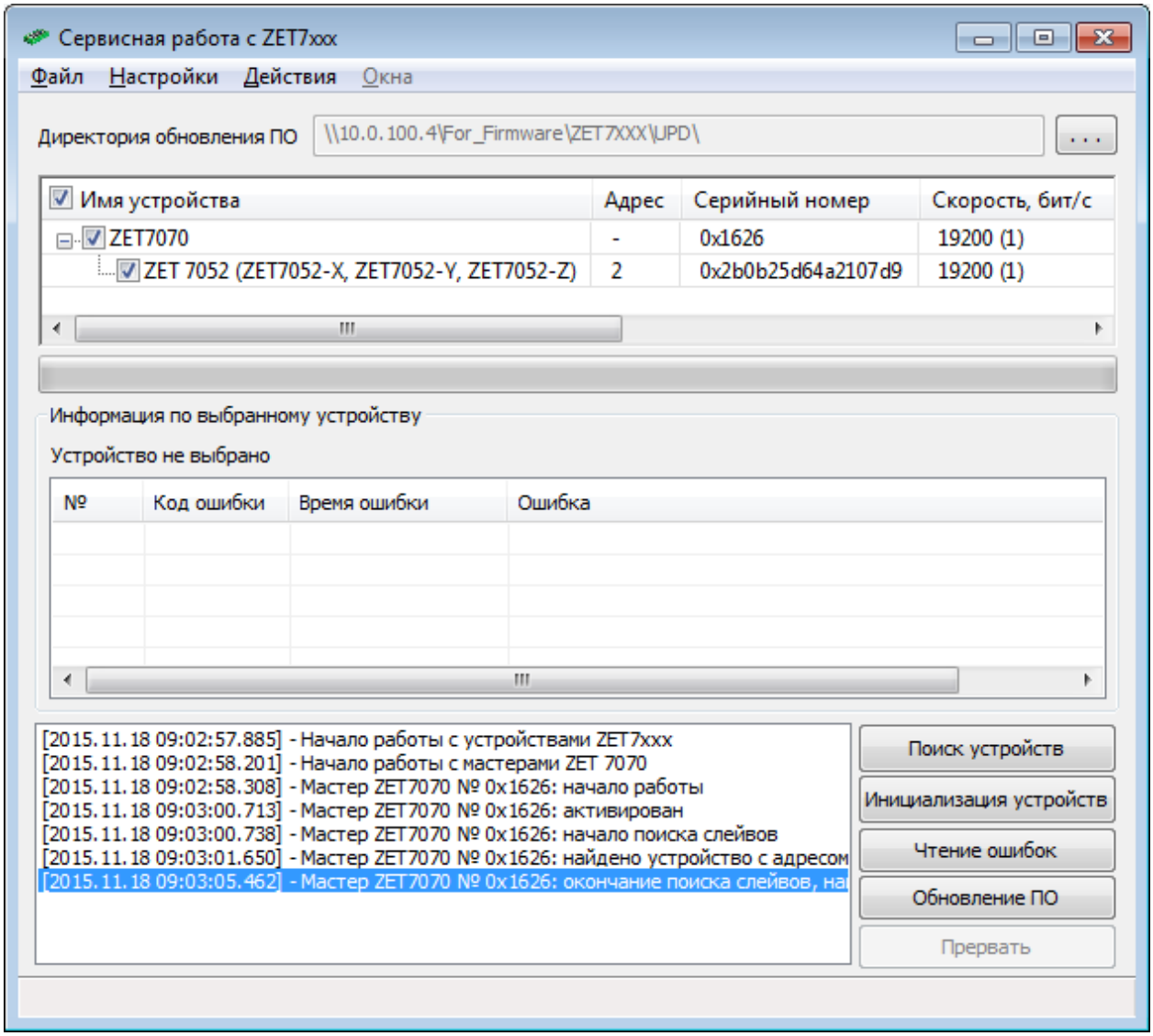

Рисунок 50.

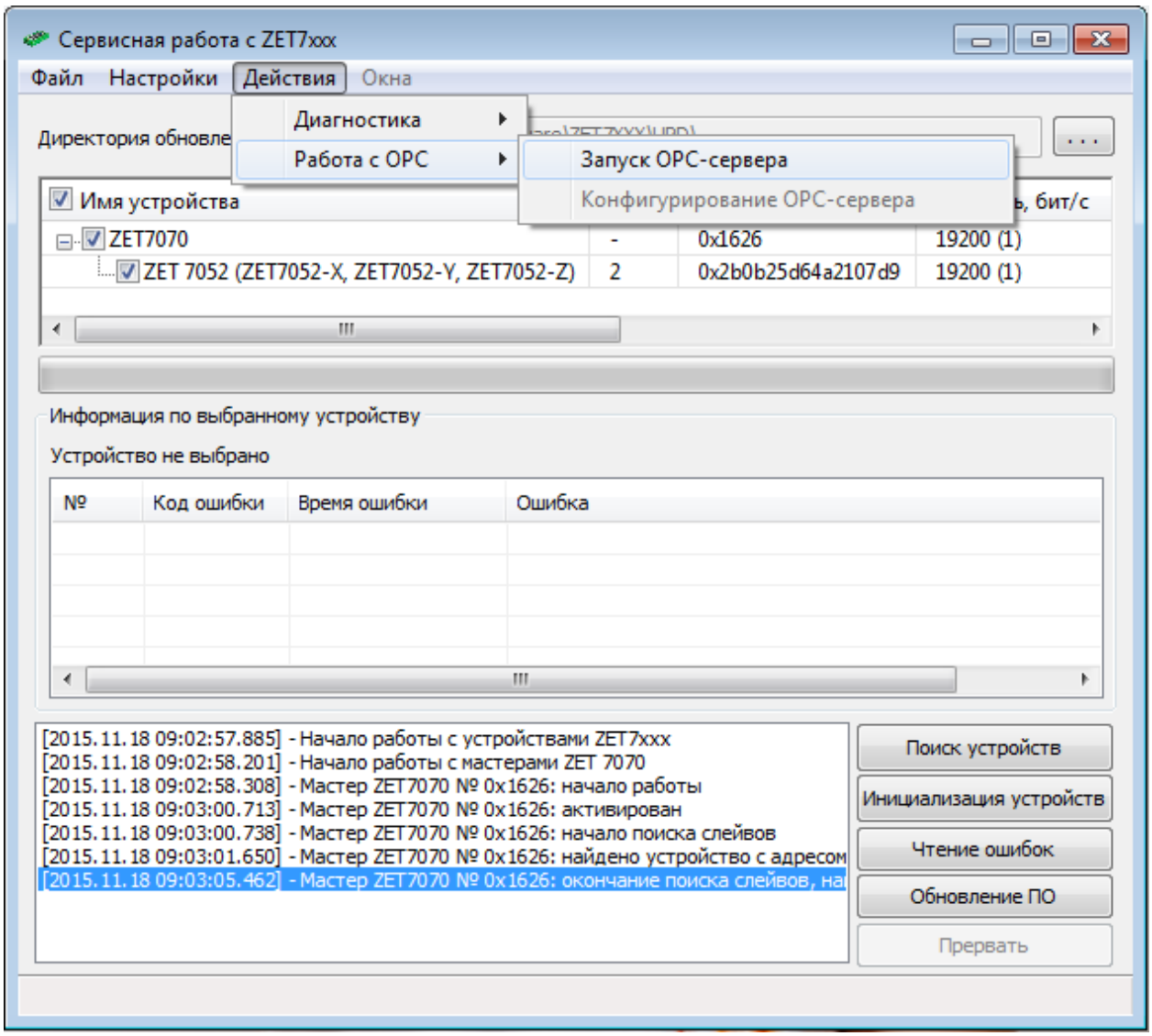
Сервисная работа с ZET7xxx
Файл
Настройки
Действия
Окна
Диагностика
Работа с OPC
Запуск OPC-сервера
Конфигурирование OPC-сервера
Директория обновле
Имя устройства
ZET7070
ZET 7052 (ZET7052-X, ZET7052-Y, ZET7052-Z)
0x1626
0x2b0b25d64a2107d9
19200 (1)
Информация по выбранному устройству
Устройство не выбрано
№
Код ошибки
Время ошибки
Ошибка
[2015.11.18 09:02:57.885] - Начало работы с устройствами ZET7xxx
[2015.11.18 09:02:58.201] - Начало работы с мастерами ZET 7070
[2015.11.18 09:02:58.308] - Мастер ZET7070 № 0x1626: начало работы
[2015.11.18 09:03:00.713] - Мастер ZET7070 № 0x1626: активирован
[2015.11.18 09:03:00.738] - Мастер ZET7070 № 0x1626: начало поиска слейвов
[2015.11.18 09:03:01.650] - Мастер ZET7070 № 0x1626: найдено устройство с адресом
[2015.11.18 09:03:05.462] - Мастер ZET7070 № 0x1626: окончание поиска слейвов, на
Поиск устройств
Инициализация устройств
Чтение ошибок
Обновление ПО
Прервать

Рисунок 51.

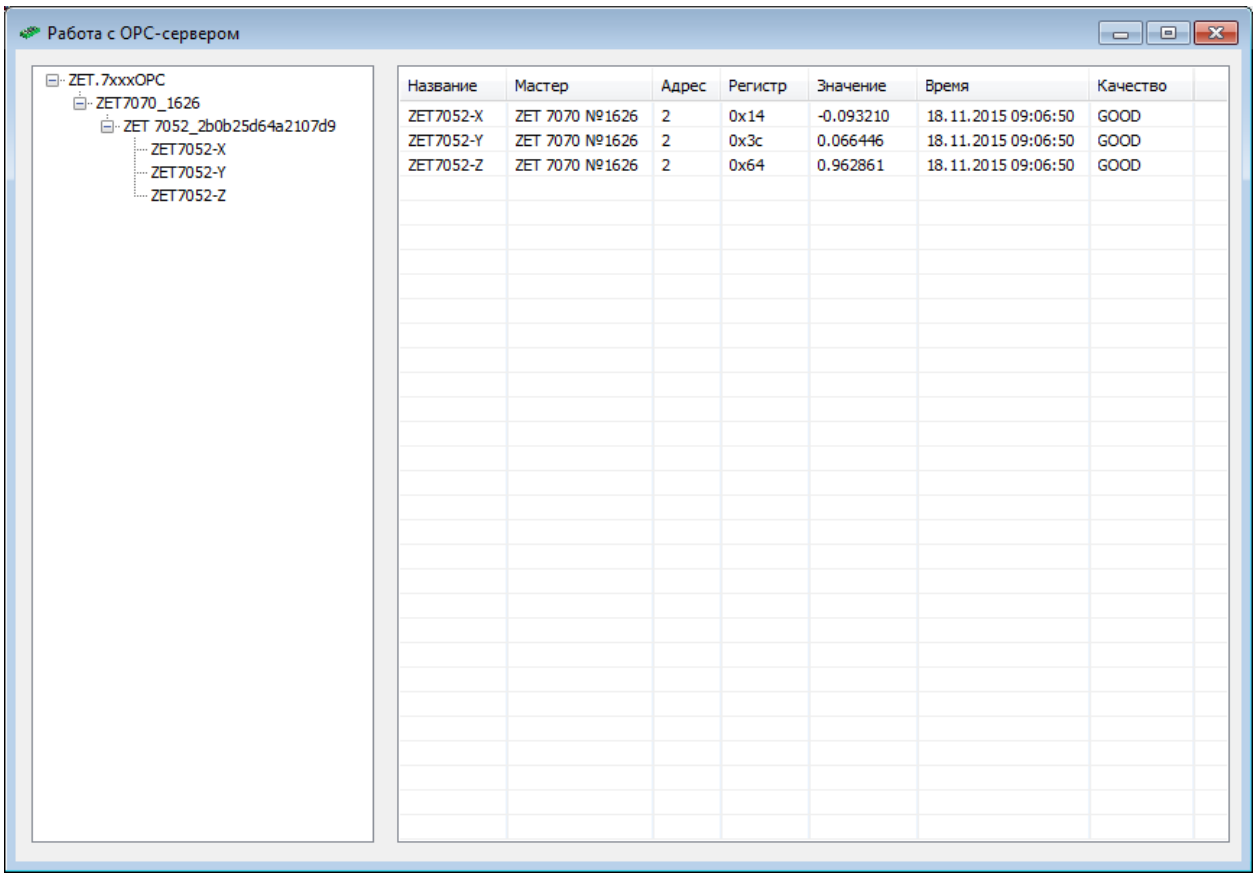
Работа с OPC-сервером
ZET.7xxxOPC
ZET7070_1626
ZET 7052_2b0b25d64a2107d9
ZET7052-X
ZET7052-Y
ZET7052-Z
Название
Мастер
Адрес
Регистр
Значение
Время
Качество
ZET7052-X
ZET 7070 №1626
2
0x14
-0.093210
18.11.2015 09:06:50
GOOD
ZET7052-Y
ZET 7070 №1626
2
0x3c
0.066446
18.11.2015 09:06:50
GOOD
ZET7052-Z
ZET 7070 №1626
2
0x64
0.962861
18.11.2015 09:06:50
GOOD

Рисунок 52.

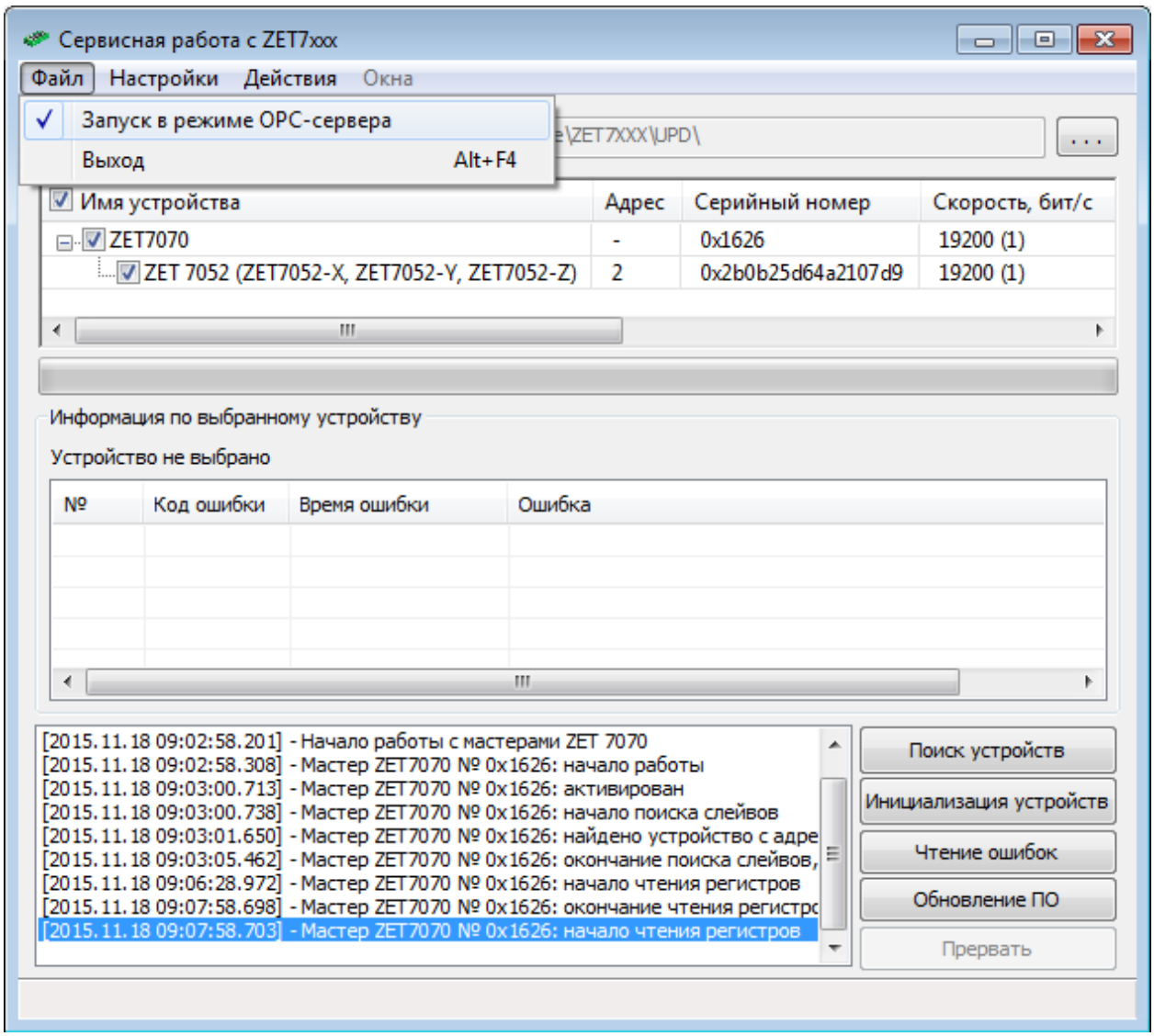

Рисунок 53.

Далее выполняем действия в соответствии с учебным фильмом, демонстрирующим подключение оборудования через встроенный OPC-клиент MasterSCADA (http://www.youtube.com/watch?v=a4Hk_a_BbGQ):

1) Добавляем в дерево «Система» узел «Компьютер» с названием «Компьютер 1». В узел «Компьютер 1» добавляем OPC-сервер ZET.7xxxOPC и для него добавляем все переменные и группы (см. Рисунок 54)

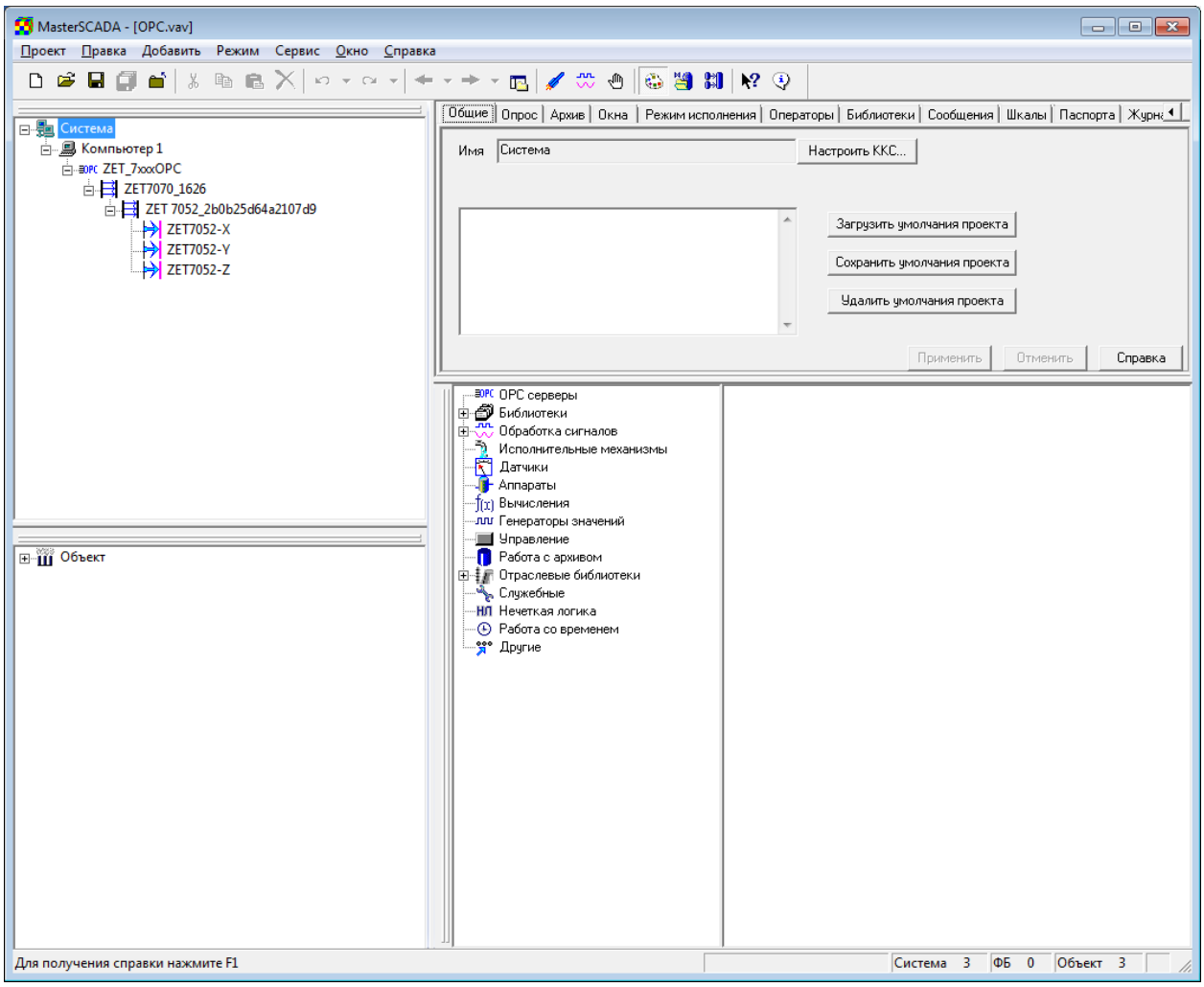

Рисунок 54.

2) Формируем дерево объектов, для этого создаем в дереве «Объект» узел «Ускорение» и назначаем ему выполнение на всех компьютерах (см. Рисунок 55). В узле «Ускорение» добавляем «Значение» для каждого из трех OPC-тегов и называем их «X», «Y» и «Z» соответственно (см. Рисунок 56). Добавляем связь значений «X», «Y» и «Z» узла объектов с соответствующими им OPC-тегами узла системы (см. Рисунок 57).

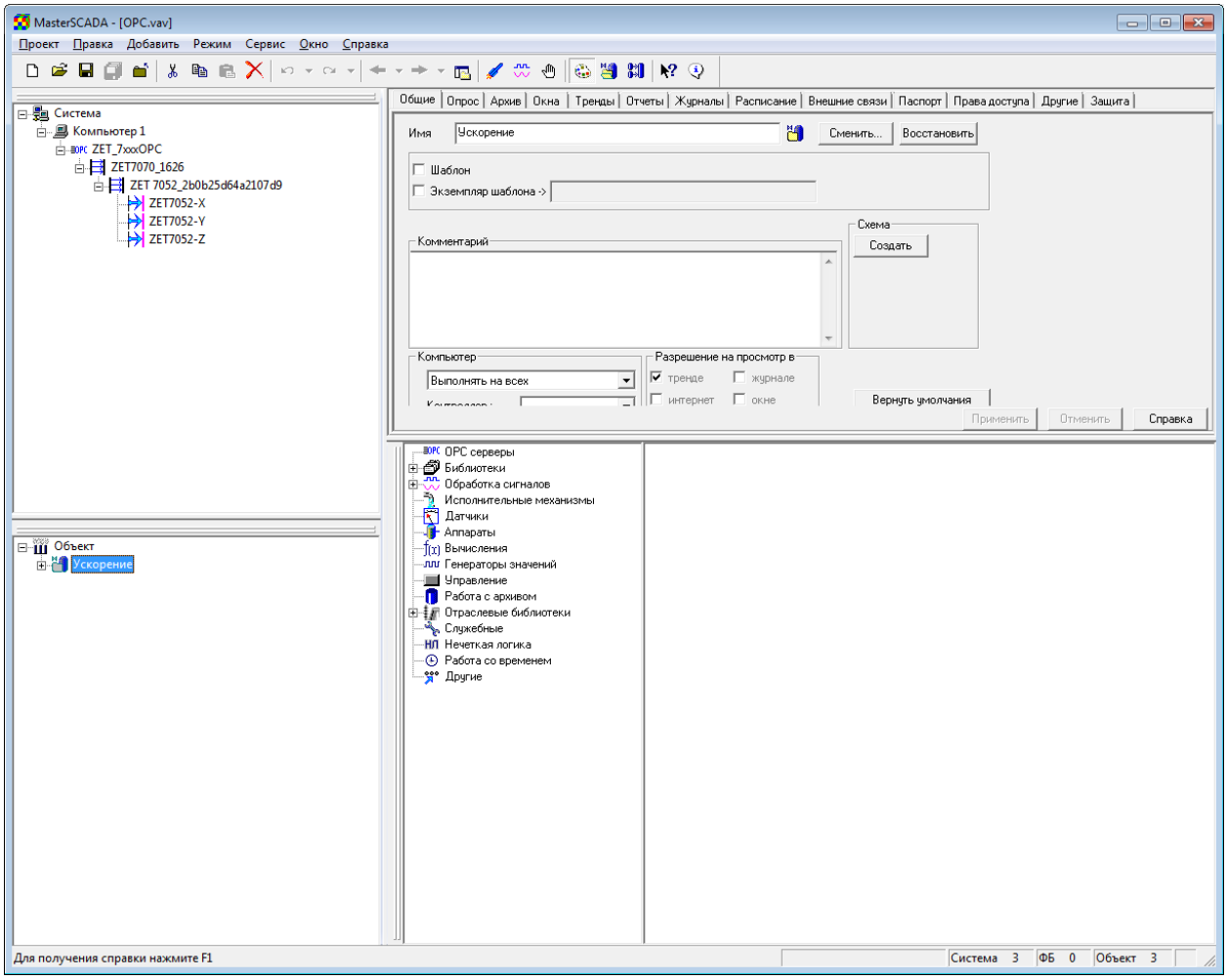
MasterSCADA - [OPC.vav]
Проект Правка Добавить Режим Сервис Окно Справка
Система
Компьютер 1
ZET_7xxxOPC
ZET7070_1626
ZET 7052_2b0b25d64a2107d9
ZET7052-X
ZET7052-Y
ZET7052-Z
Объект
Ускорение
Общие Опрос Архив Окна Тренды Отчеты Журналы Расписание Внешние связи Паспорт Права доступа Другие Защита
Имя
Ускорение
Сменить...
Восстановить
Шаблон
Экземпляр шаблона ->
Комментарий
Схема
Создать
Компьютер
Выполнять на всех
Разрешение на просмотр в
тренде
журнале
интернет
окне
Вернуть умолчания
Применить
Отменить
Справка
OPC серверы
Библиотеки
Обработка сигналов
Исполнительные механизмы
Датчики
Аппараты
Вычисления
Генераторы значений
Управление
Работа с архивом
Отраслевые библиотеки
Служебные
Нечеткая логика
Работа со временем
Другие
Для получения справки нажмите F1
Система 3
ФБ 0
Объект 3

Рисунок 55.

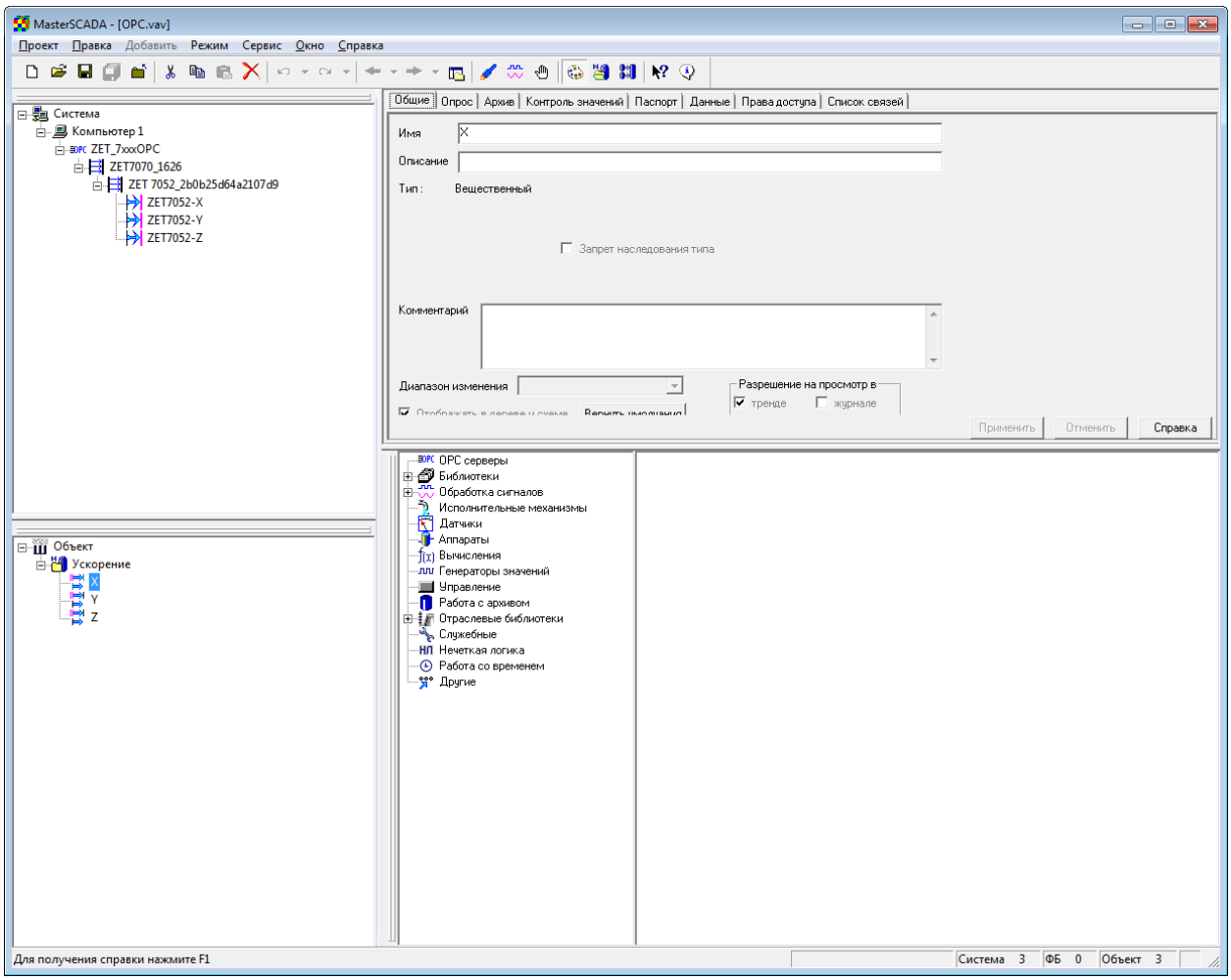

Рисунок 56.

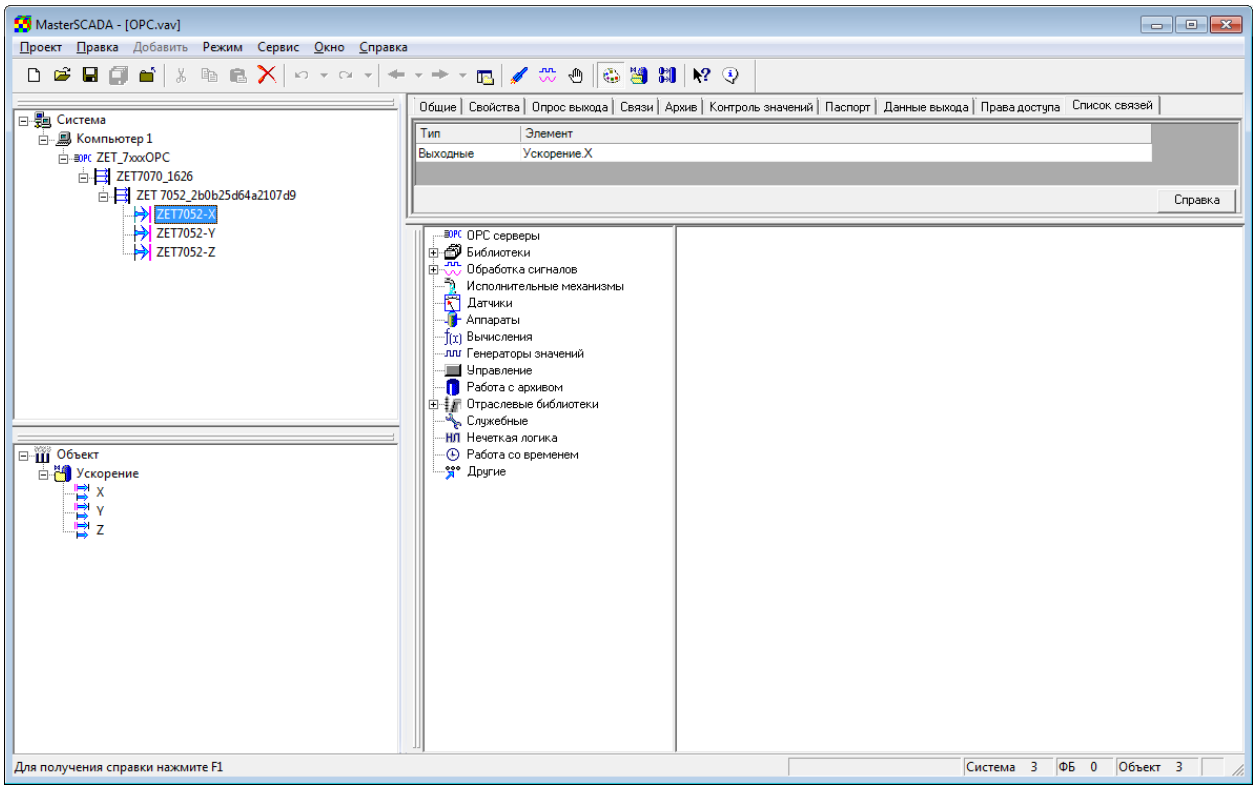

Рисунок 57.

3) Оформляем мнемосхему узла, чтобы на ней отображались тренды для каждого из значений ускорения по осям X, Y и Z (см. Рисунок 58)

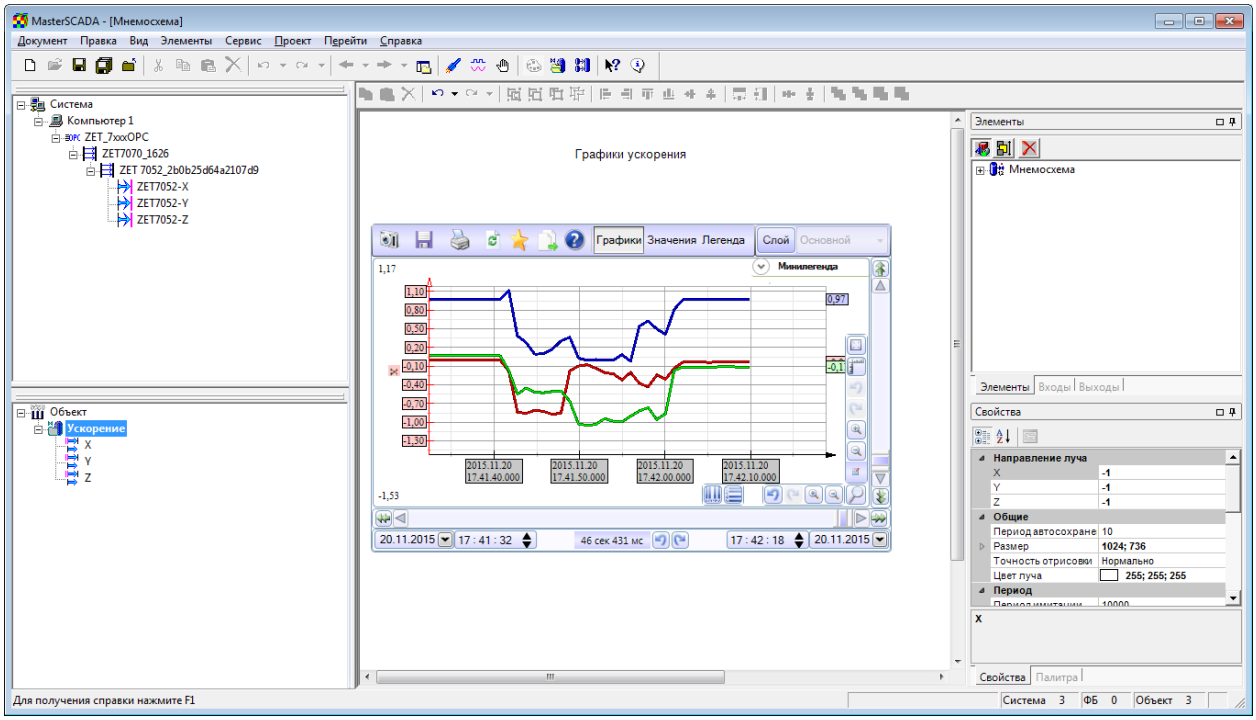

Рисунок 58.

4) Сохраняем проект на жесткий диск. Запускаем проект. Результат работы отображен на Рисунок 59 (в процессе записи тренда датчику меняли положение в пространстве)

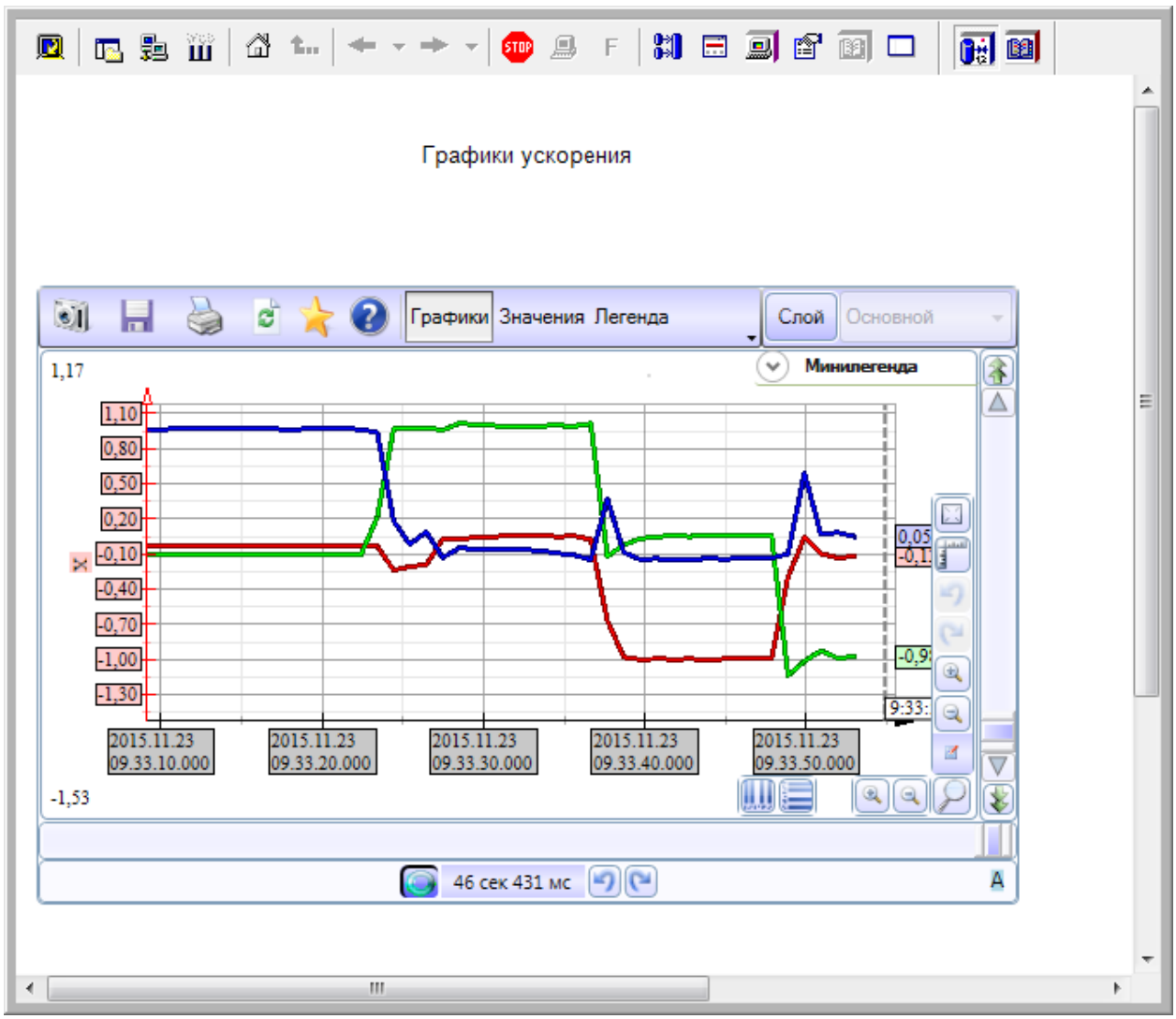

Рисунок 59.

3.8. Подключение ZETSENSOR к Network Enabler Administrator

Преобразователь интерфейсов Ethernet↔RS-485 ZET7076 позволяет подключать датчики серии ZET7xxx к системе по сети Ethernet.

Для работы с данным преобразователем интерфейсов можно использовать утилиту Network Enabler Administrator, доступную для скачивания по следующей ссылке: http://www.moxa.com/support/download.aspx?type=support&id=959

Утилита Network Enabler Administrator подключается к ZET7076 по сети Ethernet и создает в системе привязанный к преобразователю интерфейсов локальный COM-порт, через который можно передавать данные по протоколу Modbus.

После установки и запуска утилиты, необходимо выполнить сканирование доступных в сети Ethernet устройств (см. Рисунок 60).

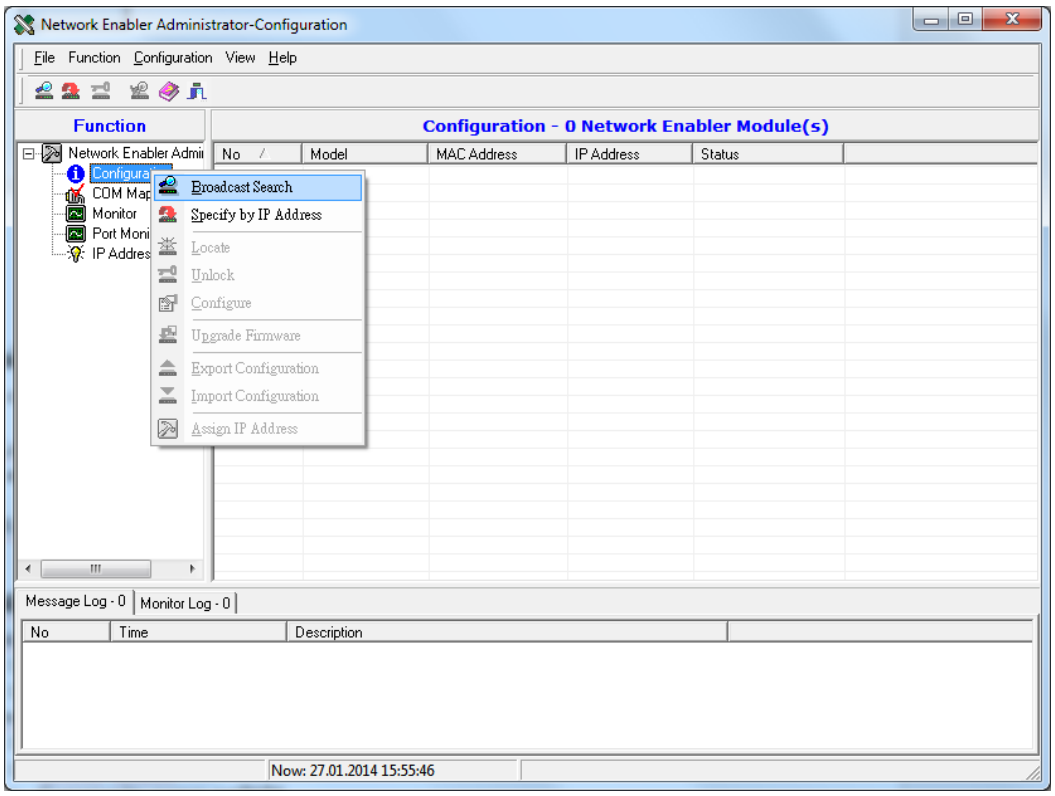

Рисунок 60.

Найденное устройство необходимо переключить в режим «Real COM Mode» (Configuration → Configure → Operating Mode → Real COM Mode), как показано на Рисунок 61.

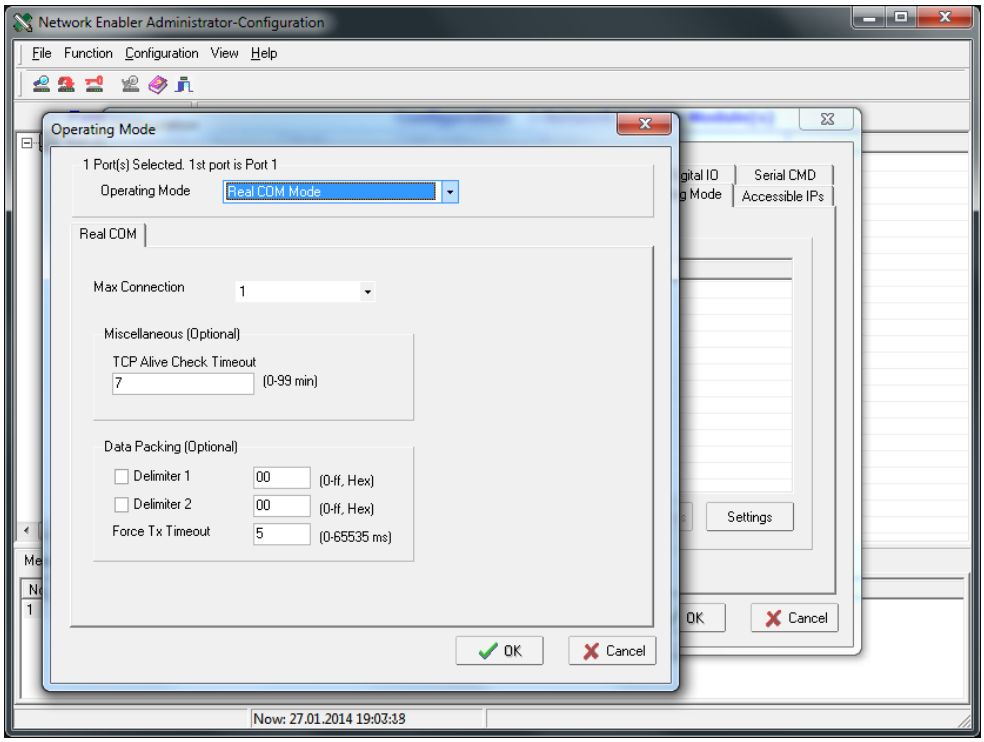

Рисунок 61.

Затем необходимо включить эмуляцию COM-порта для данного устройства с помощью меню «COM Mapping» (команды «Add Target», «Apply Changes» и «Enable»), как показано на Рисунок 62.

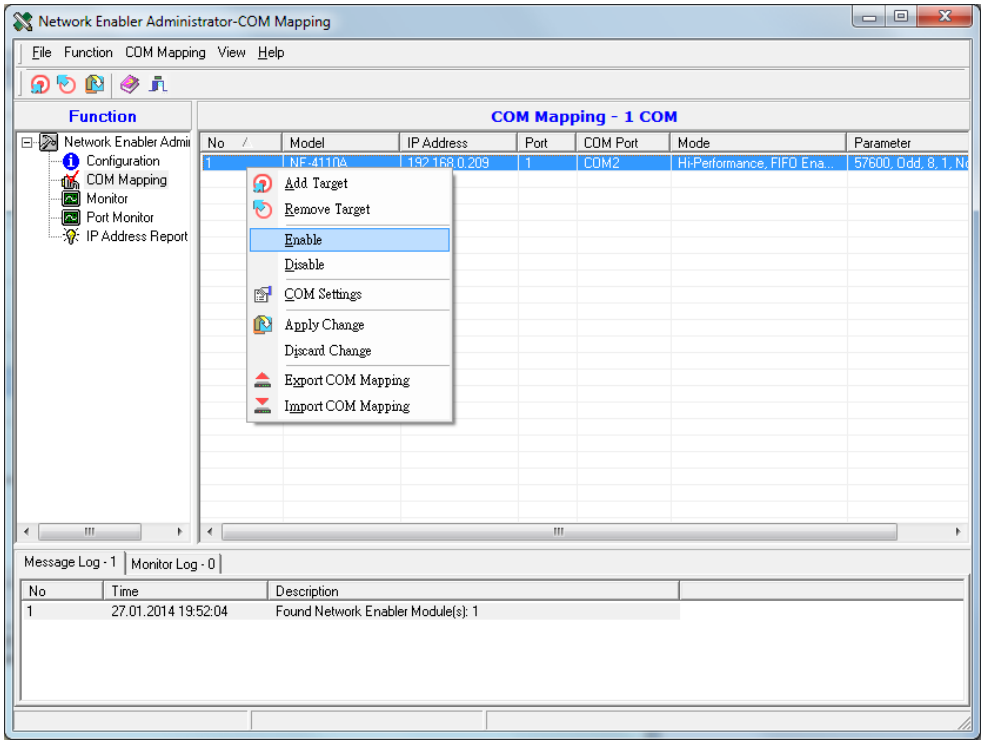

Рисунок 62.

После успешной настройки в системе должен появиться локальный COM-порт, который можно использовать для работы с датчиками. Протокол обмена в данном случае будет таким же, как и в случае подключения через другие преобразователи интерфейсов, а именно Modbus RTU (не Modbus TCP).

При работе с ZET7076 следует учитывать одну особенность — данное устройство выдает эхо передаваемых данных, т.е. в ответ на отправленный запрос сначала будет принят сам посланный запрос (эхо), а затем уже ответ датчика.

Для приведенного выше примера с датчиком ZET7010 последовательность запросов и ответов примет вид, показанный в Таблица 7:

Таблица 7.

1	Запрос	0x0A 0x03 0x00 0x00 0x00 0x04 0x45 0x72
	Ответ	0x0A 0x03 0x00 0x00 0x00 0x04 0x45 0x72 0x0A 0x03 0x08 0xC0 0x20 0x00 0x58 0x00 0x00 0xFA 0xAF 0xBE 0x70
	Эхо	8 байтов

	Структура	Тип 396, размер 36 байтов
2	Запрос	0x0A 0x03 0x00 0x10 0x00 0x04 0x44 0xB7
	Ответ	0x0A 0x03 0x00 0x10 0x00 0x04 0x44 0xB7 0x0A 0x03 0x08 0x00 0x4C 0x00 0x4D 0x00 0x00 0x1A 0x36 0x9A 0x4F
	Эхо	8 байтов
	Структура	Тип 208, размер 76 байтов

Таким образом, в системе, использующей ZET7076, должно быть реализовано программное подавление эха.

4. Примеры программирования

4.1. Пример 1

В примере показано взаимодействие с ZET 7010 посредством программы, работающей с устройством ICP CON I-7561 как с COM-портом. Язык C++.

```
#include "stdafx.h"
#include <windows.h>
#include <conio.h>
#include <vector>
//структура заголовка
typedef struct _STRUCT_HEAD
{
  unsigned int uiSize : 12; //Размер текущей структуры (в словах)
  unsigned int uiStructType : 10; //Тип текущей структуры
  unsigned int uiStatus : 10; //Статус канала (ошибка)
  unsigned int uiWriteEnable : 16; //Разрешение записи в структуру
  unsigned int uiCRC16 : 16; //Контрольная сумма
} STRUCT_HEAD, *pSTRUCT_HEAD;

HANDLE hCOMPort(NULL); //дескриптор порта
HANDLE g_hEvent[2]; //дескрипторы событий
std::vector<char> g_cReadData; //данные, прочитанные из порта

//функция расчета CRC16
unsigned short Crc16(unsigned char ucByte, unsigned short usCRC)
{
  usCRC ^= (ucByte) & 0xff;
  for( int i = 0; i < 8; ++i )
    usCRC = (usCRC & 0x01) ? (usCRC >> 1) ^ 0xA001 : usCRC >> 1;
  return usCRC;
}

//функция создания посылки
void CreatePackage(unsigned char ucDeviceAddress, unsigned
char ucCommand, unsigned shortusDataAddress, unsigned short usSize,
std::vector<char> &cData)
{
  //очистка массива данных
  if (cData.size())
  cData.clear();
  //первый байт - адрес устройства
  cData.push_back(ucDeviceAddress);
  //второй байт - команда
  cData.push_back(ucCommand);
  //третий и четвертый байты - адрес обращения
  cData.push_back(HIBYTE(usDataAddress));
  cData.push_back(LOBYTE(usDataAddress));
  //пятый и шестой байты -  размер запроса в словах
  cData.push_back(HIBYTE(usSize));
  cData.push_back(LOBYTE(usSize));
```

```
  //рассчет контрольной суммы CRC16
  unsigned short usCRC16 = 0xffff;
  for (int i = 0; i < cData.size(); ++i)
  usCRC16 = Crc16(cData[i], usCRC16);
  //седьмой и восьмой байты - контрольная сумма CRC16
  cData.push_back(LOBYTE(usCRC16));
  cData.push_back(HIBYTE(usCRC16));
}

//структура для асинхронных операций записи
OVERLAPPED structOVERLAPPEDWrite;
//функция записи в порт
void WritePackage(std::vector<char> &cData)
{
  DWORD dwTemp;
  structOVERLAPPEDWrite.hEvent = CreateEvent(NULL, true, true, NULL);
  WriteFile(hCOMPort, cData.data(), cData.size(), &dwTemp,
&structOVERLAPPEDWrite);
  WaitForSingleObject(structOVERLAPPEDWrite.hEvent, INFINITE);
}

//структура для асинхронных операций чтения
OVERLAPPED structOVERLAPPEDRead;
//функция потока чтения из порта
DWORD WINAPI ReadThread(LPVOID lpParam)
{
  COMSTAT structCOMSTAT;
  DWORD dwReceivedSize, dwEventMask, dwTemp;
  char cBufferRead[512];

  //создание события для прерывания потока чтения
  g_hEvent[1] = CreateEvent(NULL, TRUE, FALSE, NULL);
  while(1)
  {
    //создание события для асинхронной операции чтения
    structOVERLAPPEDRead.hEvent = CreateEvent(NULL, true, true, NULL);
    SetCommMask(hCOMPort, EV_RXCHAR);
    WaitCommEvent(hCOMPort, &dwEventMask, &structOVERLAPPEDRead);
    g_hEvent[0] = structOVERLAPPEDRead.hEvent;
    //ожидание одного из двух событий
    DWORD dwResult = WaitForMultipleObjects(2, g_hEvent, FALSE,
INFINITE);
    //произошло событие в порте
    if(dwResult == WAIT_OBJECT_0)
    {
      if(GetOverlappedResult(hCOMPort, &structOVERLAPPEDRead, &dwTemp,
true))
      {
        //проверка прихода байта
        if((dwEventMask & EV_RXCHAR) != 0)
        {
          //чтение из порта
```

```
          ClearCommError(hCOMPort, &dwTemp, &structCOMSTAT);
          dwReceivedSize = structCOMSTAT.cbInQue;
          if(dwReceivedSize)
          {
            ReadFile(hCOMPort, cBufferRead, dwReceivedSize, &dwTemp,
&structOVERLAPPEDRead);
            for (int i = 0; i < dwReceivedSize; ++i)
              g_cReadData.push_back(cBufferRead[i]);
          }
        }
      }
    }
    else
    //произошло событие прерывания потока чтения
    if(dwResult == WAIT_OBJECT_0 + 1)
    {
      break;
    }
  }

  printf("End reading thread...\n");

  return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
  wchar_t wcPortName[10];
  wcscpy_s(wcPortName, L"\\\\.\\COM7");

  //открытие порта
  hCOMPort = CreateFile(wcPortName, GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
  if(hCOMPort == INVALID_HANDLE_VALUE)
  {
    hCOMPort = NULL;
    printf("COM port open error!!!\n\r");
    printf("Press any key for exit...");
    getch();
    return 0;
  }
  printf("COM port open successfull\n\r");

  //считывание структуры DCB
  DCB structDCB;
  structDCB.DCBlength = sizeof(DCB);
  if(!GetCommState(hCOMPort, &structDCB))
  {
    if (hCOMPort != NULL)
    {
      printf("DCB read error!!!\n\r");
      CloseHandle(hCOMPort);
```

```
      hCOMPort = NULL;
      printf("COM port close successfull\n\r");
      printf("Press any key for exit...");
      getch();
    }
    return 0;
  }
  printf("DCB read successfull\n\r");
  //установка параметров обмена данными
  structDCB.DCBlength = sizeof(DCB);
  structDCB.fBinary = TRUE;
  structDCB.BaudRate = CBR_57600;
  structDCB.ByteSize = 8;
  structDCB.Parity = ODDPARITY;
  structDCB.StopBits = ONESTOPBIT;

  //установка структуры DCB
  if(!SetCommState(hCOMPort, &structDCB))
  {
    if (hCOMPort != NULL)
    {
      printf("DCB write error!!!\n\r");
      CloseHandle(hCOMPort);
      hCOMPort = NULL;
      printf("COM port close successfull\n\r");
      printf("Press any key for exit...");
      getch();
    }
    return 0;
  }
  printf("DCB write successfull\n\r");

  //Установка таймаутов передачи данных
  COMMTIMEOUTS structCOMMTIMOUTS;
  structCOMMTIMOUTS.ReadIntervalTimeout = 1000. / double(CBR_57600)
* 10. * 10. + 5.5;
  structCOMMTIMOUTS.ReadTotalTimeoutMultiplier = 0;
  structCOMMTIMOUTS.ReadTotalTimeoutConstant = 0;
  structCOMMTIMOUTS.WriteTotalTimeoutMultiplier = 0;
  structCOMMTIMOUTS.WriteTotalTimeoutConstant = 0;

  if(!SetCommTimeouts(hCOMPort, &structCOMMTIMOUTS))
  {
    if (hCOMPort != NULL)
    {
      printf("Timeouts write error!!!\n\r");
      CloseHandle(hCOMPort);
      hCOMPort = NULL;
      printf("COM port close successfull\n\r");
      printf("Press any key for exit...");
      getch();
    }
```

```
return 0;
}
printf("Timeouts write successfull\n\r");

//установка размеров очередей приёма и передачи
if (!SetupComm(hCOMPort, 512, 512))
{
  if (hCOMPort != NULL)
  {
    printf("SetupComm error!!!\n\r");
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    printf("COM port close successfull\n\r");
    printf("Press any key for exit...");
    getch();
  }
  return 0;
}
printf("SetupComm successfull\n\r");

//очистка принимающего буфера порта
if (!PurgeComm(hCOMPort, PURGE_RXCLEAR))
{
  if (hCOMPort != NULL)
  {
    printf("PurgeComm error!!!\n\r");
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    printf("COM port close successfull\n\r");
    printf("Press any key for exit...");
    getch();
  }
  return 0;
}
printf("PurgeComm successfull\n\r");

//дескриптор потока чтения
HANDLE hReadThread(NULL);
//идентификатор потока чтения
DWORD dwThreadID;
//создание потока чтения из порта
hReadThread = CreateThread(NULL, 0, ReadThread, 0, 0, &dwThreadID);

//буфер данных для записи в порт
std::vector<char> cWriteData;

//адрес местоположения данных в датчике
unsigned short usMainDataAddress(0x0);

//адрес датчика
unsigned char ucAddressWrite(0xa);
//команда
unsigned char ucCommandWrite(0x3);
```

```
  //адрес обращения
  unsigned short usDataAddressWrite(0x0);
  //размер запроса в словах
  unsigned short usSizeWrite(sizeof(STRUCT_HEAD) / 2);

  //цикл поиска адреса местоположения данных
  while(1)
  {
    //создание посылки
    CreatePackage(ucAddressWrite, ucCommandWrite, usDataAddressWrite,
usSizeWrite, cWriteData);
    //отправка запроса в порт
    WritePackage(cWriteData);

    //флаг выхода из цикла поиска адреса местоположения данных
    bool bEnd(false);
    //цикл чтения ответа на запросы
    while (1)
    {
      //размер данных в ответе
      int iSize(g_cReadData.size());
      if (iSize)
      {
        //флаг того, что можно отправлять новый запрос
        bool bNextPacket(false);
        //адрес датчика в ответе
        unsigned char ucAddressRead(0);
        //команда в ответе
        unsigned char ucCommandRead(0);
        //контрольная сумма в ответе
        unsigned short usCRC16(0xffff);
        for (int i = 0; i < iSize; ++i)
        {
          //выход из циклов
          if (bNextPacket || bEnd)
            break;

          switch(i)
          {
            //байт адреса устройства
            case 0:
            {
              ucAddressRead = g_cReadData[i];
              usCRC16 = Crc16(ucAddressRead, usCRC16);
              break;
            }
            //байт команды
            case 1:
            {
              ucCommandRead = g_cReadData[i];
              usCRC16 = Crc16(ucCommandRead, usCRC16);
              break;
```

```
            }
            //байт размера данных
            case 2:
            {
              unsigned char ucSizeRead(g_cReadData[i++]);
              usCRC16 = Crc16(ucSizeRead, usCRC16);
              //если данные пришли все
              if (iSize >= i + ucSizeRead + sizeof(unsigned short))
              {
                //проверка размера данных
                if (ucSizeRead > 0)
                {
                  //чтение и преобразование данных ответа
                  std::vector<short> sData(ucSizeRead / 2);
                  char* pData = reinterpret_cast<char*>(sData.data());
                  int iEndian(1);
                  for (int j = 0; j < ucSizeRead; ++i, ++j)
                  {
                  usCRC16 = Crc16(g_cReadData.data()[i], usCRC16);
                  pData[j + iEndian] = g_cReadData.data()[i];
                  iEndian *= -1;
                  }
                  //рассчет контрольной суммы
                  unsigned short usCRC16Read = unsigned
short(g_cReadData.data()[i++] &0xff);
                  usCRC16Read += ((unsigned
short(g_cReadData.data()[i++]) & 0xff) << 8);
                  //проверка на совпадение контрольных сумм
                  if (usCRC16 == usCRC16Read)
                  {
                    //проверка на совпадение адресов устройства
                    if (ucAddressRead == ucAddressWrite)
                      printf("Get answer: address %d\n",
ucAddressWrite);
                    else
                      bNextPacket = true;
                    //проверка на совпадение команд
                    if (ucCommandRead == ucCommandWrite)
                      printf("Get answer: command %d\n",
ucCommandWrite);
                    else
                      bNextPacket = true;
                    printf("Get answer: size %d byte\n", ucSizeRead);

                    //преобразование в структуру заголовка
                    pSTRUCT_HEAD pStructRead
= reinterpret_cast<pSTRUCT_HEAD>(sData.data());
                    printf("Get answer: struct: size %d, type %d, \n",
pStructRead->uiSize, pStructRead->uiStructType);
                    //проверка на условие нахождения нужного типа
структуры
                    if (pStructRead->uiStructType == 0xd0)
```

```
                    {
                      usMainDataAddress = usDataAddressWrite
+ sizeof(STRUCT_HEAD) / 2;
                      printf("Data address detected: %d\n",
usMainDataAddress);
                      //окончание поиска адреса местоположения данных
                      bEnd = true;
                    }
                    else
                    {
                      //инкремент адреса обращения
                      usDataAddressWrite += pStructRead->uiSize / 2;
                      //запрос следующего пакета
                      bNextPacket = true;
                    }
                  }
                  else
                    //запрос следующего пакета
                    bNextPacket = true;
                }
                else
                {
                  //окончание поиска адреса местоположения данных (все
структуры считали, а нужную не нашли)
                  bEnd = true;
                }
              }
              else
                //продолжение обработки ответа
                continue;
              break;
            }
          }
        }
        if (bNextPacket)
        {
          printf("Get next package...!\n");
          g_cReadData.clear();
          break;
        }
        if (bEnd)
          break;
      }
      Sleep(20);
    }
    if (bEnd)
    {
      printf("End of reading...\n");
      g_cReadData.clear();
      break;
```

```
    }
  }

  //чтение измеренных данных
  //проверка на предмет того, что нашелся адрес местоположения
измеренных значений в устройстве
  if (usMainDataAddress != 0)
  {
    //переинициализация для новых запросов
    ucCommandWrite = 0x4;
    usDataAddressWrite = usMainDataAddress;
    usSizeWrite = 0x7777;
    //цикл опроса данных
    while(!kbhit())
    {
      //создание посылки
      CreatePackage(ucAddressWrite, ucCommandWrite,
usDataAddressWrite, usSizeWrite, cWriteData);
      //отправка запроса в порт
      WritePackage(cWriteData);
      //цикл чтения ответ на запросы
      while(1)
      {
        //размер данных в ответе
        int iSize(g_cReadData.size());
        if (iSize)
        {
          //флаг того, что можно отправлять новый запрос
          bool bNextPacket(false);
          //адрес датчика в ответе
          unsigned char ucAddressRead(0);
          //команда в ответе
          unsigned char ucCommandRead(0);
          //контрольная сумма в ответе
          unsigned short usCRC16(0xffff);
          for (int i = 0; i < iSize; ++i)
          {
            //выход из цикла
            if (bNextPacket)
              break;

            switch(i)
            {
              //байт адреса устройства
              case 0:
              {
                ucAddressRead = g_cReadData[i];
                usCRC16 = Crc16(ucAddressRead, usCRC16);
                break;
              }
              //байт команды
              case 1:
```

```
                {
                  ucCommandRead = g_cReadData[i];
                  usCRC16 = Crc16(ucCommandRead, usCRC16);
                  break;
                }
                //байт размера
                case 2:
                {
                  unsigned char ucSizeRead(g_cReadData[i++]);
                  usCRC16 = Crc16(ucSizeRead, usCRC16);
                  //если данные пришли все
                  if (iSize >= i + ucSizeRead + sizeof(unsigned short))
                  {
                    //проверка размера данных
                    if (ucSizeRead > 0)
                    {
                    //чтение и преобразование данных ответа
                    std::vector<float> fData(ucSizeRead / 2 / 2);
                    char* pData = reinterpret_cast<char*>(fData.data());
                    int iEndian(1);
                    for (int j = 0; j < ucSizeRead; ++i, ++j)
                    {
                      usCRC16 = Crc16(g_cReadData.data()[i], usCRC16);
                      pData[j + iEndian] = g_cReadData.data()[i];
                      iEndian *= -1;
                    }
                    //рассчет контрольной суммы
                    unsigned short usCRC16Read = unsigned
short(g_cReadData.data()[i++] &0xff);
                    usCRC16Read += ((unsigned
short(g_cReadData.data()[i++]) & 0xff) << 8);
                    //проверка на совпадение контрольных сумм
                    if (usCRC16 == usCRC16Read)
                    {
                      //проверка на совпадение адресов устройства
                      if (ucAddressRead == ucAddressWrite)
                        printf("Get answer: address %d\n",
ucAddressWrite);
                      else
                        bNextPacket = true;
                      //проверка на совпадение комманд
                      if (ucCommandRead == ucCommandWrite)
                        printf("Get answer: command %d\n",
ucCommandWrite);
                      else
                        bNextPacket = true;
                        //вывод измеренных значений
                        printf("Get answer: size %d, byte or %d, data:",
ucSizeRead, fData.size());
                      for (int j = 0; j < fData.size(); ++j)
                        printf(" %f", fData[j]);
```

```
                    printf("\n");
                    //запрос следующего пакета
                    bNextPacket = true;
                  }
                  else
                    //запрос следующего пакета
                    bNextPacket = true;
                }
                else
                {
                  printf("Get answer: size 0 byte\n");
                  //запрос следующего пакета
                  bNextPacket = true;
                }
              }
              else
                //продолжение обработки ответа
                continue;
              break;
            }
          }
        }
        if (bNextPacket)
        {
          printf("Get next package...!\n");
          g_cReadData.clear();
          break;
        }
      }
      Sleep(500);
    }
  }
  }
  if (hCOMPort != NULL)
  {
    //взвод события для завершения потока чтения
    SetEvent(g_hEvent[1]);
    WaitForSingleObject(hReadThread, INFINITE);
    //закрытие порта
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    printf("COM port close successfull\n\r");
  }
  printf("Press any key for exit...");
  getch();

  return 0;
}
```

4.2. Пример 2

Программа осуществляет поиск устройств в измерительной линии датчиков с цифровым выходом RS-485. В качестве мастера используется любое устройство типа преобразователь USB-RS485, подключенное к компьютеру и создающее в нем виртуальный COM-порт. Программа задействует указанный пользователем COM-порт и, перебирая стандартные скорости обмена, виды проверки бита четности и адреса ведомых MODBUS-устройств, шлет запросы и ожидает на них ответ. После прохода каждого цикла программа отображает количество удачных передач т.е. тех запросов, на которые был ответ от ведомого устройства. Язык C++.

```
#include "stdafx.h"
#include <windows.h>
#include <conio.h>
#include <vector>
#include <algorithm>
static const unsigned char crc16TableHi[] = {
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80,
0x41, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00,
0xC1, 0x81, 0x40, 0x01,
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x01, 0xC0, 0x80, 0x41,
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0,
  0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x01,
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40,
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80,
0x41, 0x00, 0xC1, 0x81,
  0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00,
0xC1, 0x81, 0x40, 0x01,
  0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41,
  0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80,
0x41, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1,
0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x01,
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40,
0x01, 0xC0, 0x80, 0x41,
```

```
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80,
0x41, 0x00, 0xC1, 0x81,
  0x40
};
static const unsigned char crc16TableLo[] = {
  0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07,
0xC7, 0x05, 0xC5, 0xC4,
  0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA,
0xCB, 0x0B, 0xC9, 0x09,
  0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E,
0xDE, 0xDF, 0x1F, 0xDD,
  0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6,
0xD2, 0x12, 0x13, 0xD3,
  0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2,
0x32, 0x36, 0xF6, 0xF7,
  0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F,
0x3E, 0xFE, 0xFA, 0x3A,
  0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB,
0x2B, 0x2A, 0xEA, 0xEE,
  0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5,
0x27, 0xE7, 0xE6, 0x26,
  0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61,
0xA1, 0x63, 0xA3, 0xA2,
  0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC,
0xAD, 0x6D, 0xAF, 0x6F,
  0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78,
0xB8, 0xB9, 0x79, 0xBB,
  0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C,
0xB4, 0x74, 0x75, 0xB5,
  0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70,
0xB0, 0x50, 0x90, 0x91,
  0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95,
0x94, 0x54, 0x9C, 0x5C,
  0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99,
0x59, 0x58, 0x98, 0x88,
  0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F,
0x8D, 0x4D, 0x4C, 0x8C,
  0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43,
0x83, 0x41, 0x81, 0x80,
  0x40
};
unsigned short calcCrc16(unsigned short crc16, const
void *pBuffer, unsigned int size)
{
 const unsigned char* pData = reinterpret_cast<const unsigned
char*>(pBuffer);
 unsigned char hiByte = HIBYTE(crc16);
 unsigned char loByte = LOBYTE(crc16);
 unsigned index;
  while (size--)
  {
```

```
    index = hiByte ^ *pData++;
    hiByte = loByte ^ crc16TableHi[index];
    loByte = crc16TableLo[index];
  }
 return MAKEWORD(loByte, hiByte);
}
void createRequest(unsigned char node, std::vector<unsigned char>
&data)
{
  //очистка массива данных
  if (!data.empty())
    data.clear();
  //первый байт - адрес устройства
  data.push_back(node);
  //второй байт - команда
  data.push_back(3);
  //третий и четвертый байты - адрес обращения
  data.push_back(0);
  data.push_back(0);
  //пятый и шестой байты - размер запроса в словах
  data.push_back(0);
  data.push_back(4);
  //рассчет контрольной суммы CRC16
  unsigned short crc16 = calcCrc16(0xffff, data.data(), data.size());
  //седьмой и восьмой байты - контрольная сумма CRC16
  data.push_back(HIBYTE(crc16));
  data.push_back(LOBYTE(crc16));
}
bool checkResponse(unsigned char node, std::vector<unsigned char>
&data)
{
  unsigned int size(data.size());
  unsigned int index(0);
  unsigned char symbol;
  unsigned short crc16(0xffff);
  if (index >= size)
    return false;
  //проверка совпадения адресов устройства запроса и ответа
  symbol = data[index++];
  crc16 = calcCrc16(crc16, &symbol, 1);
  if (symbol != node)
    return false;
  if (index >= size)
    return false;
  //проверка совпадения команды запроса и ответа
  symbol = data[index++];
  crc16 = calcCrc16(crc16, &symbol, 1);
  if (symbol != 3)
    return false;
  if (index >= size)
    return false;
```

```
  //определения размера пришедших данных
  symbol = data[index++];
  crc16 = calcCrc16(crc16, &symbol, 1);
  if (index >= size)
    return false;
  unsigned short dataSize = symbol;
  for (unsigned short i = 0; i < dataSize; ++i)
  {
    symbol = data[index++];
    crc16 = calcCrc16(crc16, &symbol, 1);
    if (index >= size)
      return false;
  }
  //проверка контрольной суммы
  unsigned short crc16FromResponse;
  symbol = data[index++];
  crc16FromResponse = unsigned short(symbol & 0xff);
  crc16FromResponse <<= 8;
  if (index >= size)
    return false;
  symbol = data[index++];
  crc16FromResponse += unsigned short(symbol & 0xff);
  if (crc16FromResponse != crc16)
    return false;
  return true;
}
unsigned int getSleepTimeout(unsigned int length, DWORD baudRate)
{
  long sleepTime(1);
  //время ожидания при скоростях выше 19200 бит/с (в миллисекундах)
  if (baudRate > CBR_19200)
    sleepTime = long(1.75 + 1000. * double(length) * 11.
/ double(baudRate) + 0.5);
  //время ожидания при скоростях ниже 19200 бит/с (в миллисекундах)
  else
    sleepTime = long(1000. * (3.5 + double(length)) * 11.
/ double(baudRate) + 0.5);
  if (sleepTime < 1)
    sleepTime = 1;
  return sleepTime;
}
double getTime()
{
  LARGE_INTEGER frequency, counter;
  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter(&counter);
  return double(counter.QuadPart) / double(frequency.QuadPart);
}
int _tmain(int argc, _TCHAR* argv[])
{
  //скорости, по которым будет проходить программа
```

```
  static const DWORD baudRates[] = {
    CBR_2400,
    CBR_4800,
    CBR_9600,
    CBR_19200,
    CBR_38400,
    CBR_57600,
    CBR_115200
  };
  const unsigned short baudRateCount = sizeof(baudRates)
/ sizeof(DWORD);
  //проверки бита четности, по которым будет проходить программа
  static const BYTE parities[] = {
    NOPARITY,
    ODDPARITY,
    EVENPARITY
  };
  const unsigned short parityCount = sizeof(parities) / sizeof(BYTE);
  unsigned int portNumder(1);
  wchar_t sPortNumber[10];
   wprintf(L"Enter COM port number: ");
  _getws_s(sPortNumber);
  swscanf_s(sPortNumber, L"%d", &portNumder);
  wchar_t sComPort[10];
  swprintf_s(sComPort, L"\\\\.\\COM%d", portNumder);
  //открытие порта
  HANDLE hCOMPort = CreateFile(sComPort,
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        NULL,
        NULL);
  if (hCOMPort == INVALID_HANDLE_VALUE)
  {
    hCOMPort = NULL;
    wprintf(L"COM port open error!!!\n\rPress any key for exit...");
    _getch();
    return 0;
  }
  wprintf(L"COM port open successfull\n\r");
  //установка таймаутов передачи данных
  COMMTIMEOUTS commtimouts;
  commtimouts.ReadIntervalTimeout = 0;
  commtimouts.ReadTotalTimeoutMultiplier = 0;
  commtimouts.ReadTotalTimeoutConstant = 0;
  commtimouts.WriteTotalTimeoutMultiplier = 0;
  commtimouts.WriteTotalTimeoutConstant = 0;
  if (!SetCommTimeouts(hCOMPort, &commtimouts))
  {
    wprintf(L"Timeouts write error!!!\n\r");
```

```
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    wprintf(L"COM port close successfull\n\rPress any key for
exit...");
    _getch();
    return 0;
  }
  wprintf(L"Timeouts write successfull\n\r");
  //установка размеров очередей приёма и передачи
  if (!SetupComm(hCOMPort, 512, 512))
  {
    wprintf(L"SetupComm error!!!\n\r");
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    wprintf(L"COM port close successfull\n\rPress any key for
exit...");
    _getch();
    return 0;
  }
  wprintf(L"SetupComm successfull\n\r");
  //очистка принимающего буфера порта
  if (!PurgeComm(hCOMPort, PURGE_RXCLEAR))
  {
    wprintf(L"PurgeComm error!!!\n\r");
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    wprintf(L"COM port close successfull\n\rPress any key for
exit...");
    _getch();
    return 0;
  }
  wprintf(L"PurgeComm successfull\n\r");
  //проход по всем скоростям
  unsigned short baudRateIndex(0);
  while (baudRateIndex < baudRateCount)
  {
    if (_kbhit())
    {
      _getch();
      break;
    }
    //проход по всем проверкам бита четности
    unsigned short parityIndex(0);
    while (parityIndex < parityCount)
    {
      if (_kbhit())
      {
        _getch();
        break;
      }
```

```
        //считывание структуры DCB
        DCB dcb;
        dcb.DCBlength = sizeof(DCB);
        if (!GetCommState(hCOMPort, &dcb))
        {
          wprintf(L"DCB read error!!!\n\r");
          break;
        }
        //установка параметров обмена данными
        dcb.DCBlength = sizeof(DCB);
        dcb.fBinary = TRUE;
        dcb.BaudRate = baudRates[baudRateIndex];
        dcb.ByteSize = 8;
        dcb.Parity = parities[parityIndex];
        dcb.StopBits = ONESTOPBIT;
        //установка структуры DCB
        if (!SetCommState(hCOMPort, &dcb))
        {
          printf("DCB write error!!!\n\r");
          break;
        }
        wprintf(L"Baudrate %d bit/sec, parity %d (press any key to go to
next parity or baudrate value)\n\r",
          baudRates[baudRateIndex],
          parities[parityIndex]);
        //проход по всем возможным адресам устройств
        unsigned int success(0);
        for (unsigned char node = 2; node < 63; ++node)
        {
          wprintf(L"\r\t\tCurrent address %d", node);
          if (_kbhit())
          {
            _getch();
            break;
          }
          bool bResult(false);
          //формирование запроса
          std::vector<unsigned char> requestData;
          createRequest(node, requestData);
          //посылка запроса в порт
          DWORD bytesWritten(0);
          BOOL bStatus = WriteFile(hCOMPort,
              requestData.data(),
              requestData.size(),
              &bytesWritten,
              NULL);
          if (bStatus == TRUE && requestData.size() == bytesWritten)
          {
            //ожидание прохождения запроса
            long sleepTime4Request = getSleepTimeout(requestData.size(),
baudRates[baudRateIndex]);
```

```
          double tTime = getTime();
          while (sleepTime4Request > (getTime() - tTime) * 1000)
            Sleep(1);
          long sleepTimeAfterResponse = getSleepTimeout(4,
baudRates[baudRateIndex]) + 50;
          tTime = getTime();
          //ожидание ответа от устройства
          std::vector<unsigned char> responseData;
          while (1)
          {
            //запрос состояния порта
            COMSTAT comstat;
            bStatus = ClearCommError(hCOMPort, NULL, &comstat);
            if (bStatus != TRUE)
              break;
            //проверка наличия байт в буфере приема порта
            if (comstat.cbInQue > 0)
            {
              //чтение байт из буфера приема порта
              tTime = getTime();
              std::vector<unsigned char> data(comstat.cbInQue);
              DWORD bytesRead(0);
              bStatus = ReadFile(hCOMPort, data.data(),
comstat.cbInQue, &bytesRead, NULL);
              if (bStatus != TRUE || comstat.cbInQue != bytesRead)
                break;
              responseData.insert(responseData.end(), data.begin(),
data.end());
              //проверка правильности ответа от устройства
              if (checkResponse(node, responseData))
              {
                //ожидание времени до формирования следующего запроса
                bResult = true;
                Sleep(sleepTimeAfterResponse);
                break;
              }
            }
            else
            {
              Sleep(1);
              //проверка превышения времени ожидания ответа от
устройства
              if ((getTime() - tTime) * 1000 >
sleepTimeAfterResponse)
              {
                break;
              }
            }
          }
        }
        if (bResult)
```

```
            ++success;
        }
        wprintf(L"\r\t\tSuccessful transmissions %d\n\r", success);
        ++parityIndex;
      }
      ++baudRateIndex;
    }
    //закрытие порта
    CloseHandle(hCOMPort);
    hCOMPort = NULL;
    wprintf(L"COM port close successfull\n\rPress any key for
exit...");
    _getch();
    return 0;
}
```

4.3. Пример 3

Конфигурирование модулей ZETSENSOR в большинстве случаев выполняется через штатное программное обеспечение ZETLAB при помощи программы “Диспетчер устройств” вкладки “Сервисные”. Однако, часть прикладных задач подразумевает изменение настроек цифровых датчиков без использования ПО ZETLAB. Для такой работы предусмотрен отдельный алгоритм по работе с ZETSENSOR (см. Рисунок 63)

Рисунок 63. Схема взаимодействия при конфигурировании цифровых датчиков

ПО ZETLAB при работе с цифровыми датчиками для каждого из них создает так называемый “слепок” настроек, который представляет собой файл с расширением .dat и бинарным содержимым. Имя такого файла имеет формат вида ZET 7AAA № 0xBBBBBBBBBBBBBBB_C.dat, где 7AAA - тип (например, 7121), BBBBBBBBBBBBBBB - серийный номер в шестнадцатеричном формате (например, 2b0c575b5a2f0922), C - адрес устройства (например, 4). Содержимое файла полностью соответствует содержимому регистров измерительного модуля. Суть конфигурирования

заключается в изменении содержимого файла настроек в соответствии с таблицей регистров для данного модуля. Таблица адресов регистров генерируется с помощью программы “SensorWork”. После того, как файл настроек изменится, программа, работающая с ZETSENSOR, подхватит изменения и внесет их в устройство автоматически.

Алгоритм конфигурирования сводится к следующей последовательности действий:

1) Поиск нужного файла в директории “..\config\ZET7xxx\” по типу, серийному номеру и адресу устройства. Путь до директории “config” хранится в реестре и может быть получен через компонент ZETPath.ocx.

Пример на C++

```
CString sConfigFile;
sConfigFile.Format(L"0x%I64x", m_serial);
CString sFindFile = m_sConfigPath + L"\\*.*";
WIN32_FIND_DATA win32_find_data;
HANDLE hFind = FindFirstFile(sFindFile, &win32_find_data);
if (hFind != INVALID_HANDLE_VALUE)
{
	do
	{
		CString sFileName(CString(win32_find_data.cFileName));
		if (!(win32_find_data.dwFileAttributes &
FILE_ATTRIBUTE_DIRECTORY))
		{
			if (sFileName.Find(sConfigFile) != -1)
			{
				if (sFileName.Find(L".dat") != -1)
					sFileNameDat = m_sConfigPath + L"\\" +
sFileName;
				if (!sFileNameDat.IsEmpty())
					break;
			}
		} while (FindNextFile(hFind, &win32_find_data));
		FindClose(hFind);
	}
}
```

2) Определение размера файла, открытие файла на чтение и чтение из него содержимого в массив байт.

Пример на C++

```
FILE* pFile(nullptr);
BYTE* pConfig(nullptr);
uint16_t sizeConfig(0);
if ((_wfopen_s(&pFile, sFileNameDat, L"rb") == 0) && (pFile !=
nullptr))
{
	fseek(pFile, 0, SEEK_END);
	sizeConfig = ftell(pFile);
```

```
	fseek(pFile, 0, SEEK_SET);
	pConfig = new BYTE[sizeConfig];
	fread(pConfig, sizeof(BYTE), sizeConfig, pFile);
	fclose(pFile);
	pFile = nullptr;
}
```

3) Изменение массива, считанного из файла. Место, где производить изменения, определяется по отступу и размеру регистра из таблицы адресов регистров.

Пример на C++

```
float value = 123;
memcpy(&pConfig[offset], &value, sizeof(value));
```

4) Запись измененного содержимого в файл настроек.

Пример на C++

```
FILE* pFile(nullptr);
if ((_wfopen_s(&pFile, m_DatFileName, L"wb") == 0) && (pFile != 
nullptr))
{
	fwrite(pConfig, sizeof(BYTE), sizeConfig, pFile);
	fclose(pFile);
	pFile = nullptr;
}
```

4.4. Пример 4

Пример демонстрирует подключение к устройству с адресом 0x4 и чтение с него значения по каналу (регистр 0x14 с типом float). Язык Python.

````
```
#!/usr/bin/env python
import minimalmodbus
import serial
import struct

## Read float in ZETSENSOR format
def read_float_zet(instrument, reg_addr):
    regs = instrument.read_registers(reg_addr, 2)
    bytestring = struct.pack('<HH', *regs)
    value = struct.unpack('<f', bytestring)[0]
    return value

minimalmodbus.BAUDRATE = 19200
minimalmodbus.PARITY = serial.PARITY_ODD
minimalmodbus.BYTESIZE = 8
minimalmodbus.STOPBITS = 1
minimalmodbus.TIMEOUT = 0.05

instrument = minimalmodbus.Instrument('/dev/ttyUSB0', 4) # port name, slave address
(in decimal)

## Read channel value (float, register 0x14)
```
````

```
value = read_float_zet(instrument, 0x14)

print 'Channel value =', value
````