

ОРГАНИЗАЦИЯ СВЯЗИ КПК-КОНТРОЛЛЕР ПО ПОСЛЕДОВАТЕЛЬНОМУ КАНАЛУ

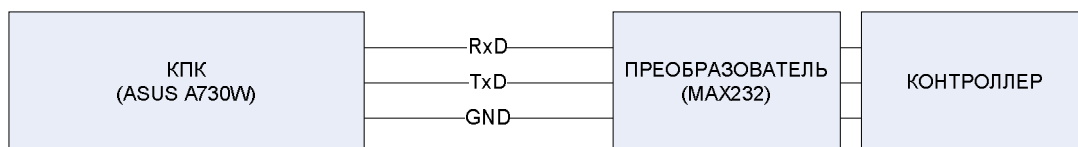
Клебан Виталий, 2007

vkle@mail.ru

<http://quark-bot.blogspot.com>

Некоторое время назад я занялся постройкой робота на основе гусеничного шасси. Естественно возник вопрос о выборе электроники, которая будет установлена на роботе в качестве бортовой. После недолгих оценок в качестве «мозга» был выбран КПК в связке с контроллером Atmel Mega32. Об организации связи КПК-Контроллер я и хочу рассказать.

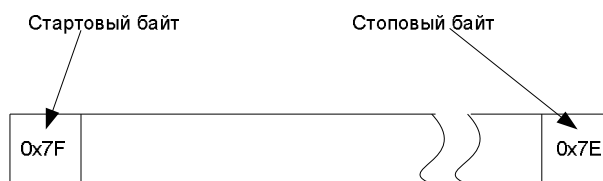
Связь организована с помощью последовательного RS232 канала (УАПП), используются только линии TxD, RxD и земля.



Распиновка разъема КПК, точно также как и необходимость преобразователя зависят от модели КПК, но я рассматриваю свой конкретный случай.

Итак, решение о физическом уровне организации связи принято, но для организации обмена данными естественно необходим транспортный протокол, который позволил бы обмениваться данными в пакетном режиме.

Самой распространенной схемой формирования пакетов является схема с передаваемым количеством байт, т.е. в составе пакета (обычно в заголовке) передается длина пакета. Подобная схема не лишена недостатков, так, например, при возникновении ошибки в длине пакета протокол становится практически полностью неработоспособным. Я предлагаю использовать другой протокол, основанный на маскировании данных. Пакет представляет собой набор данных начинающийся и заканчивающийся спецсимволами:



Если в передаваемых данных встречаются спецсимволы, то к ним применяется процедура маскирования. Для осуществления процедуры маскирования вводится еще один спецсимвол *маска*, со значением $0x7D$. Спецсимволы заменяются на *escape-последовательности* как показано на рисунке:

$0x7F = 0x7D \text{ xor } 0x02$
 $0x7E = 0x7D \text{ xor } 0x03$
 $0x7D = 0x7D \text{ xor } 0x00$

$0x01, 0x02, 0x7F, 0x03 \longrightarrow 0x01, 0x02, 0x7D, 0x02, 0x03$

Для формирования исходных данных применяется процедура демаскирования, которая заменяет *escape-последовательности* в пакете обратно на исходный символ.

Листинг процедуры формирования пакета представлен ниже:

```
public static byte[] Send(byte[] buffer)
{
    List<byte> list = new List<byte>(buffer.Length);
    list.Add(0x7F);
    for (int i = 0; i < buffer.Length; i++)
    {
        byte b = buffer[i];
        if (b == 0x7F || b == 0x7D || b == 0x7E)
        {
            list.Add(0x7D);
            list.Add(Convert.ToByte(b^0x7D));
        }
        else
        {
            list.Add(b);
        }
    }
    list.Add(0x7D);
    return list.ToArray();
}
```

Прием данных бортовым контроллером и КПК происходит побайтно, что делает целесообразным построение управляющего автомата.

Перечислим события (e), на которые реагирует автомат, и их номера.

Внешние воздействия от управления

- E10 – получен байт.

Таймеры

- E30 – время ожидания ответа истекло.

Перечислим названия выходных воздействий (z) и их номера.

Выходные - управляющий объект

- Z10 – уведомление о получении пакета;
- Z11 – занесение принятого байта в буфер ответа;
- Z14 – уведомление об ошибке получения пакета;
- Z15 – отказ.

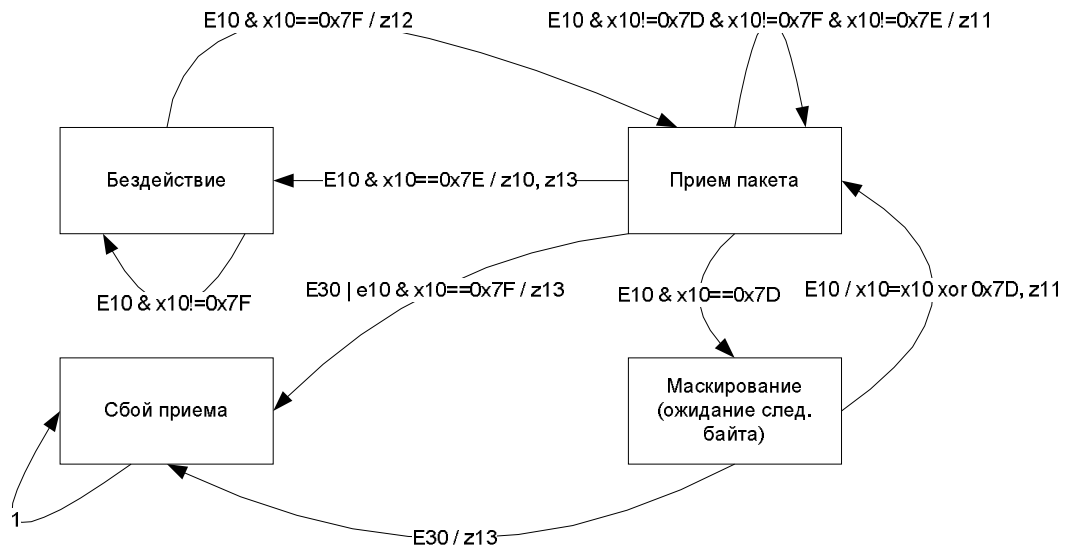
Таймеры

- Z12 – старт таймера времени ожидания;
- Z13 – остановка таймера времени ожидания.

Перечислим входные переменные автомата (x):

- X10 – Принятый байт

На рисунке представлен граф переходов автомата приема данных.



Реализация описанного подхода в моем варианте:

```

namespace BaseLibrary.Protocols.Concrete
{
    internal enum TransportAutomataStates
    {
        ST_STOPPED,
        ST_RECEIVE,
        ST_MASK,
        ST_ERROR,
        ST_FAULT
    }

    internal enum TransportAutomataEvents
    {
        e10, e13
    }

    public class SimpleTransport : IProtocol
    {
        public delegate void RxCompleteDelegate(byte[] data);

        /// <summary>
        /// Событие возбуждается при окончании приема каждого пакета.
        /// </summary>
        public event RxCompleteDelegate RxComplete;

        private TransportAutomataStates state
            = TransportAutomataStates.ST_STOPPED;
        private List<byte> outputBuffer = new List<byte>();
        private const byte CharBegin = 0x7F;
        private const byte CharMask = 0x7D;
        private const byte CharEnd = 0x7E;

        /// <summary>
        /// Конечный автомат для разбора входного потока байт.
        /// </summary>
        /// <param name="ev">event</param>
        /// <param name="b">input byte</param>
        private void RxAutomata(TransportAutomataEvents ev, byte b)
    }
}

```

```

{
    TransportAutomataStates y_old = state;

    switch (state)
    {
        case TransportAutomataStates.ST_STOPPED:
            if (ev == TransportAutomataEvents.e10)
            {
                if (b == CharBegin)
                    state = TransportAutomataStates.ST_RECEIVE;
                else
                    state = TransportAutomataStates.ST_STOPPED;
            }
            else
                state = TransportAutomataStates.ST_FAULT;
            break;
        case TransportAutomataStates.ST_RECEIVE:
            if (ev == TransportAutomataEvents.e10)
            {
                if (b == CharMask)
                    state = TransportAutomataStates.ST_MASK;
                else if (b == CharEnd)
                {
                    state = TransportAutomataStates.ST_STOPPED;
                    if (RxComplete != null)
                        RxComplete(outputBuffer.ToArray());
                }
                else if (b == CharBegin)
                {
                    state = TransportAutomataStates.ST_ERROR;
                }
                else
                {
                    outputBuffer.Add(b);
                }
            }
            else
                state = TransportAutomataStates.ST_FAULT;
            break;
        case TransportAutomataStates.ST_MASK:
            if (ev == TransportAutomataEvents.e10)
            {
                outputBuffer.Add(Convert.ToByte(b ^ CharMask));
                state = TransportAutomataStates.ST_RECEIVE;
            }
            else
                state = TransportAutomataStates.ST_FAULT;
            break;
        case TransportAutomataStates.ST_ERROR:
            if (ev == TransportAutomataEvents.e13)
            {
                state = TransportAutomataStates.ST_STOPPED;
            }
            else
                state = TransportAutomataStates.ST_FAULT;
            break;
        case TransportAutomataStates.ST_FAULT:
            state = TransportAutomataStates.ST_FAULT;
            break;
    }

    if (y_old == state) return;
}

```

```

switch (state)
{
    case TransportAutomataStates.ST_STOPPED:
        break;
    case TransportAutomataStates.ST_RECEIVE:
        break;
    case TransportAutomataStates.ST_MASK:
        break;
    case TransportAutomataStates.ST_ERROR:
        break;
    case TransportAutomataStates.ST_FAULT:
        throw new QFaultException();
}
}

public void SetParameters(object options)
{
    return;
}

public byte[] Encode(byte[] data, object options)
{
    return Encode(data);
}

public byte[] Decode(byte[] data, object options)
{
    return Decode(data);
}

/// <summary>
/// Функция возвращает сформированный в соответствии с правилами
транспортный
/// пакет.
/// </summary>
/// <param name="data">
/// Входные данные
/// </param>
/// <returns>
/// Пакет
/// </returns>
public byte[] Encode(byte[] data)
{
    List<byte> list = new List<byte>(data.Length);
    list.Add(CharBegin);
    for (int i = 0; i < data.Length; i++)
    {
        byte b = data[i];
        if (b == CharBegin || b == CharEnd || b == CharMask)
        {
            list.Add(CharMask);
            list.Add(Convert.ToByte(b ^ CharMask));
        }
        else
        {
            list.Add(b);
        }
    }
    list.Add(CharEnd);
    return list.ToArray();
}

/// <summary>

```

