Практическое задание №4

Вычисление проверочного кода протокола Modbus

1. Напишите программу, рассчитывающую проверочные коды протокола Modbus (в режиме RTU и ASCII).
2. Проверьте работоспособность программы на контрольном примере:

Данные	LRC	CRC
21 03 00 00 00 04	D8	43 69
6A 04 03 7D 00 01	11	A9 4D
4D 10 00 C7 00 02 04 47 DE 60	51	09 48

3. Рассчитайте контрольные суммы передаваемых пакетов данных, в соответствии с вариантом задания (неизвестные символы, соответствующие проверочным кодам обозначены как xx).
4. Интерпретируйте Modbus сообщение и данные.

Вариант	Сообщение
1	01030020000Cxxxx :0C0105CD6BB20E1Bxx↵
2	02032000000Bxxxx :0A0106000F0FAA55AFxx↵
3	03040002000Axxxx :0B0205CD6BE20E2Bxx↵
4	040420000009xxxx :090204CBBE20B1xx↵
5	050100200025xxxx :070206000A0120CDEFxx↵
6	060100020005xxxx :080105A000E3C4F7xx↵
7	07050007FF00xxxx :060207000000000000xx↵
8	08050000FF00xxxx :04010520E1B34567xx↵
9	09040009000Axxxx :050107FFFFFF0000FF01xx↵
10	0A040A0A000Axxxx :030205CD6AA20E11xx↵
11	0B06001A0EDAxxxx :010207000132750EB1F3xx↵
12	0C0600A20000xxxx :0202040AAA5501xx↵

Вычисление проверочного кода протокола Modbus

Протокол Modbus использует два вида проверочных кодов:

- LRC8 (используется в Modbus-ASCII);
- CRC16 (используется в Modbus-RTU).

Генерация LRC

Longitudinal Redundancy Check (LRC) – это один байт. LRC вычисляется передающим устройством и добавляется к концу сообщения. Принимающее устройство также вычисляет LRC в процессе приема и сравнивает вычисленную величину с полем контрольной суммы пришедшего сообщения. Если суммы не совпали, то имеет место ошибка.

LRC вычисляется сложением последовательности байтов сообщения, отбрасывая все переносы, и затем двойным дополнением результата. LRC – это 8-ми битовое поле, где каждое новое прибавление символа, приводящее к результату более чем 255, приводит к простому перескакиванию через 0. Так как это поле не является 9-ти битовым, перенос отбрасывается автоматически.

Алгоритм генерации LRC

1. Сложить все байты сообщения, исключая стартовый символ **<:>** и конечные **<CR><LF>**, складывая их так, чтобы перенос отбрасывался.
2. Отнять получившееся значение от числа FF_{16} – это является первым дополнением.
3. Прибавить к получившемуся значению 1 – это второе дополнение.

ПРИМЕР

```
static unsigned char LRC(unsigned char *auchMsg, /* Сообщение */
                         unsigned char usDataLen /* Длина сообщения */ )
{
      unsigned char uchLRC=0; /* Инициализация LRC */
      while(usDataLen--)
         uchLRC+=*auchMsg++;
      return((unsigned char)(-((char) uchLRC)));
}
```

Функция **LRC** принимает два аргумента:

auchMsg	Указатель на буфер данных
usDataLen	Количество байт в буфере

Функция возвращает LRC как тип **unsigned char**.

Генерация CRC

CRC это 16-ти разрядная величина, то есть состоит из двух байт. CRC вычисляется передающим устройством и добавляется к сообщению. Принимающее устройство также вычисляет CRC в процессе приема и сравнивает вычисленную величину с полем контрольной суммы пришедшего сообщения. Если суммы не совпали, то имеет место ошибка.

16-ти битовый регистр CRC предварительно загружается числом FF_{16}. Процесс начинается с добавления байтов сообщения к текущему содержимому регистра. Для генерации CRC используются только 8 бит данных. Старт и стоп биты, бит паритета, если он используется, не учитываются в CRC.

В процессе генерации CRC, каждый 8-ми битовый символ складывается по ИСКЛЮЧАЮЩЕМУ ИЛИ с содержимым регистра. Результат сдвигается в направлении младшего бита, с заполнением 0 старшего бита. Младший бит извлекается и проверяется. Если младший бит равен 1, то содержимое регистра складывается с определенной ранее, фиксированной величиной, по ИСКЛЮЧАЮЩЕМУ ИЛИ. Если младший бит равен 0, то ИСКЛЮЧАЮЩЕЕ ИЛИ не делается.

Этот процесс повторяется, пока не будет сделано 8 сдвигов. После последнего (восьмого) сдвига, следующий байт складывается с содержимым регистра, и процесс повторяется снова. Финальное содержание регистра, после обработки всех байтов сообщения и есть контрольная сумма CRC.

Алгоритм генерации CRC

1. 16-ти битовый регистр загружается числом $FFFF_{16}$ (все 1), и используется далее как регистр CRC.
2. Первый байт сообщения складывается по ИСКЛЮЧАЮЩЕМУ ИЛИ с содержимым регистра CRC. Результат помещается в регистр CRC.
3. Регистр CRC сдвигается вправо (в направлении младшего бита) на 1 бит, старший бит заполняется 0.
4. (Если младший бит 0): Повторяется шаг 3 (сдвиг)

   (Если младший бит 1): Делается операция ИСКЛЮЧАЮЩЕЕ ИЛИ регистра CRC и полиномиального числа $A001_{16}$.
5. Шаги 3 и 4 повторяются восемь раз.
6. Повторяются шаги со 2 по 5 для следующего сообщения. Это повторяется до тех пор, пока все байты сообщения не будут обработаны.
7. Финальное содержание регистра CRC и есть контрольная сумма.

ПРИМЕР

```
unsigned short CRC16(unsigned char *puchMsg,  /* Сообщение */
                     unsigned short usDataLen /* Длина сообщения */ )
{
   unsigned short crc = 0xFFFF;
   unsigned short uIndex;
   int i;

   for (uIndex = 0; uIndex < usDataLen; uIndex++) {
       crc ^= (unsigned short)*(puchMsg +uIndex);

       for (i = 8; i != 0; i--) {
           if ((crc & 0x0001) != 0) {
               crc >>= 1;
               crc ^= 0xA001;
           }
           else
               crc >>= 1;
       }
   }
   crc = ((crc >> 8) & 0x00FF) | ((crc << 8) & 0xFF00);
   return crc;
}
```

На практике чаще всего используется иной способ вычисления CRC. Все возможные величины CRC загружаются в два массива. Один массив содержит все 256 возможных комбинаций CRC для старшего байта поля CRC, другой – данные для младшего байта.

```
static unsigned char auchCRCHi[] = {
      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,
      0x22,0x00,0xC1,0x81,0x40,0x00,0xC1,0x81,0x40,0x01,0xC0,0x80,0x41,
      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,
      0x40,0x01,0xC0,0x80,0x41,0x00,0xC0,0x80,0x41,0x00,0xC1,0x81,0x40,
      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 char auchCRCLo[] = {
     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};
```

Индексация CRC в этом случая обеспечивает быстрое выполнение вычислений новой величины CRC для каждого нового байта из буфера сообщения.

```
 unsigned short CRC16(unsigned char *puchMsg, /* Сообщение */
                      unsigned short usDataLen /* Длина сообщения */ )
 {
     unsigned char uchCRCHi = 0xFF;
     unsigned char uchCRCLo = 0xFF;
     unsigned int uIndex;
     while (usDataLen--) {
          uIndex = uchCRCHi ^ *puchMsg++;
          uchCRCHi = uchCRCLo ^ auchCRCHi[uIndex];
          uchCRCLo = auchCRCLo[uIndex];
     }
     return (uchCRCHi << 8 | uchCRCLo);
 }
```

Функция **CRC16** принимает два аргумента:

puchMsg	Указатель на буфер данных
usDataLen	Количество байт в буфере

Функция возвращает CRC как тип **unsigned short**.