Быстрое погружение с «черными плавниками». Часть 2. Пишем драйвер последовательного порта

Андрей САВИЧЕВ
avisv@hotbox.ru
Олег РОМАНОВ
oleg.rom@eltech.spb.ru

В этой публикации мы продолжаем осваивать новый мир, открывшийся нам благодаря технологии Blackfin.

В прошлый раз (см. КиТ № 6 `2006, стр. 130–134) мы не только отыскали первые сокровища, скрытые в глубинах технологии Blackfin, но и весьма неплохо освоились в новом окружении. Наша «мини-субмарина» ADSP-BF533 EZ-KIT Lite сыграла в этом выдающуюся роль. Теперь мы готовы не только собирать сокровища, оставленные другими, но и создавать свои. Словом, приступаем к написанию драйвера последовательного порта, работающего по прерываниям. Если вы опытный программист, то справедливо заметите, что задача эта не столь уж сложная. Согласимся, что «дьявол таится в мелочах», и порой неприметные для неофита Blackfin «подводные камни» (ошибки в реализации архитектуры проектировщиком, отображаемые в errata), а также другие помехи в виде почти неуловимых «подводных течений» (особенностей во взаимодействии отдельных подсистем на кристалле) могут растянуть проект на неопределенное время. Для отладки нашего драйвера мы, конечно же, должны использовать дополнительный компьютер с установленной на нем терминальной программой. Ситуация весьма распространенная — чтобы повернуть Землю, требуется точка опоры. Ни на одну программу нельзя полагаться априори, какими бы гениальными программистами она не была создана. Убедиться в работоспособности выбранной терминальной программы можно с помощью «петли» — соединения выхода последовательного COM-порта с его входом. Как это сделать физически — не суть важно. В простейшем случае достаточно в 9-контактном разъеме DB9M соединить между собой контакты 2 и 3. Результат тестирования использованной нами программы Terminal v1.9b представлен на рис. 1.

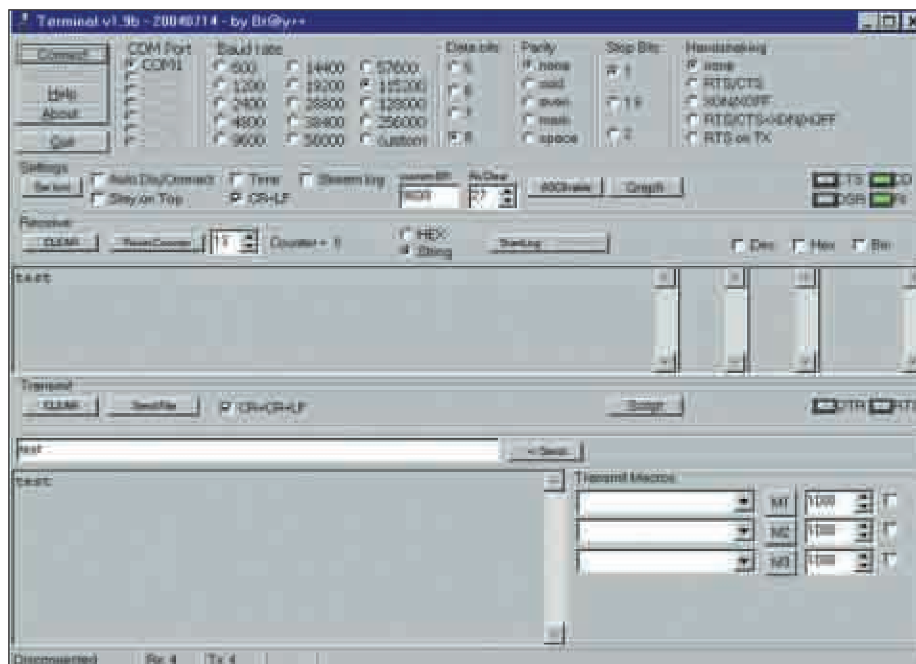

Рис. 1. Результат тестирования терминальной программы Terminal v1.9b с помощью LoopBack

Поясним то, что изображено на рис. 1, в нескольких словах, поскольку и далее мы будем пользоваться для иллюстрации результатов работы именно этой программой. Серое окно в средней части с четырьмя полосами прокрутки является консолью приемника, нижнее — передатчика последовательного порта. Однострочное окно с белым фоном отображает набираемые на клавиатуре данные для передачи. Чтобы произвести ввод и редактирование в этой строке, необходимо установить курсор в нужное место. Подробное описание возможностей данной программы можно получить, нажав кнопку Help, а скачать ее можно с сайта ht tp:/ /bray.velenje.cx/avr/terminal.

Вспомогательный компьютер, на котором запущена терминальная программа, будем называть терминальным. На основном же компьютере, к которому присоединена плата ADSP-BF533 EZ-KIT Lite, в это время выполняется программа VisualDSP++. На следующем шаге нам понадобится соединительный кабель между терминальным компьютером и ADSP-BF533 EZ-KIT Lite. В комплект поставки он обычно не входит, и его придется изготовить самостоятельно.

В приложении B Schematics Sheet 9 [1] в правой нижней части под надписью UART изображено соединение цепей платы ADSP-BF533 EZ-KIT Lite со стандартным 9-контактным разъемом DB9M. Только будьте внимательны: не следует соединять выводы «перекрестно», как в схеме «нуль-модема»! Из-за этого может произойти путаница, и собранный кабель окажется непригодным для работы. К счастью, изготовители нашей «мини-субмарины» и здесь предусмотрели «аварийный выход». В поставке программного обеспечения имеются заведомо рабочие примеры для работы с COM-портом, с помощью которых можно протестировать собранный кабель. Авторы обнаружили два демонстрационных проекта, пригодных для этой цели. Первый реализован на ассемблере Blackfin. Он как раз работает по прерываниям и содержит много

ценных подсказок. Но чтобы ими воспользоваться, нужно прежде овладеть ассемблером. Так как предмет наших статей — быстрое введение в проблематику, этот путь, очевидно, не для нас. Хотя, если возникнут непреодолимые препятствия, держим этот вариант в уме «про запас».

Располагается этот проект в директории:

```
<Путь установки>\Blackfin\EZ-KITs\ADSP-BF533\Examples\UART RS-232 HyperTerminal session.
```

Вы еще помните, как открывать проекты в IDE Visual DSP++? Об этом подробно рассказывалось в первой части нашей статьи. На всякий случай напомним: в меню **File** -> **Open** -> **Project** выбираем нужный проект (файл с расширением **.dpj**). К счастью для нас, имеется и проект, пригодный для тестирования собранного кабеля, на языке Си. Только он, в полном согласии с законами Мерфи, не работает по прерываниям. С другой стороны, именно это обстоятельство дает нам возможность внести свою скромную лепту в копилку кода **Examples**.

Проект STDIO UART можно найти в директории:

```
<Путь установки>\Blackfin\EZ-KITs\ADSP-BF533\Examples\STDIO UART.
```

Первое, что мы обнаруживаем в комментариях к **main.c**:

```
// Special Connections:
//       - EIA-232 Serial Cable (1:1)
//       - Terminal Program (i.e. Hyperterminal for Windows)
//       - 2400 bits per second (default)
//       - 8-bit, no parity, 1 stop bit
//       - no handshake
//       - echo off
```

Отсюда понятно, из-за чего не следует применять «нуль-модемный» перекрестный кабель. Видны также необходимые параметры соединения для терминальной программы. Впрочем, вы можете изменять их по своему усмотрению, но только одновременно — и в данном проекте, и в вашей терминальной программе. Код этого примера выполнен в хорошем стиле (в отличие от рассмотренных нами примеров в первой части статьи).

Если ваш кабель работоспособен хотя бы на вывод из ADSP-BF533 EZ-KIT Lite, на экране монитора появится:

“Type in something, press <Enter>, and I'll repeat it backwards!” («Напечатайте что-нибудь, нажмите <Ввод>, и я повторю напечатанное!»).

И программа действительно делает это! Конечно, при условии, что вы правильно спаяли кабель и еще не успели вывести из строя свои COM-порты. Словом, наступает момент истины, и мы пожелаем вам удачи!

Надеемся, что у вас, как и у нас, испытание прошло благополучно. Продолжаем в таком случае с благодарностью разбирать **main.c** из проекта STDIO UART:

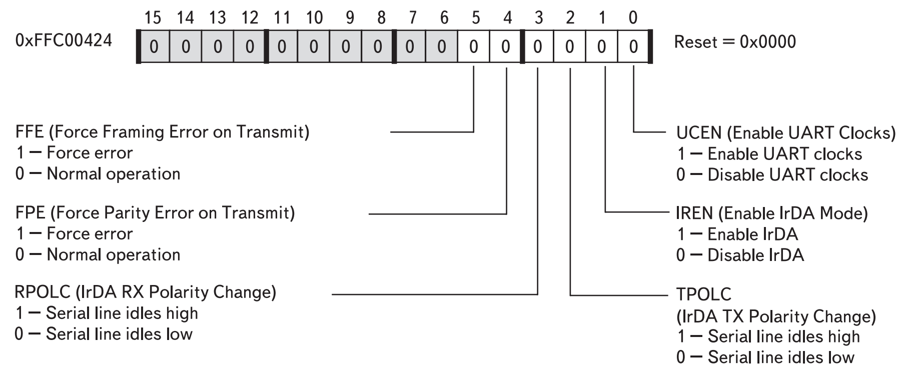

Рис. 2. Регистр UART Global Control Register (UART_GCTL)

```
unsigned short CLKIN = 27;
// CLKIN frequency is 27MHz on the ADSP-BF533 EZ-Kits.
// If using a different BF board, then change this variable to
// your new board's CLKIN frequency.

#define BAUDRATE 115200
// Default baudrate = 2400.
// You can change to any baudrate that matches your Terminal Program baudrate.
// Else set this #define to 0 for auto baud rate detection. Start auto baud
// rate detection by entering «@» inside Terminal Program.
```

Убеждаемся, что на нашей плате ADSP-BF533 EZ-KIT Lite действительно установлена указанная в комментарии тактовая частота. В противном случае программа может оказаться неработоспособной. Смотрим далее файл STDIO_UART.c и обнаруживаем в нем настоящую россыпь драгоценностей. В самом деле, берем себе на заметку используемые заголовочные файлы:

```
#include <device.h>
#include <ccblkfn.h>
#include <cdefBF533.h>
#include <sys/exception.h>
#include <stdio.h>
```

и собственно процедуру инициализации UART:

```
//
// Configures UART in 8 data bits, no parity, 1 stop bit mode.
//
void UART_inititialize(int divisor)
```

Далее привычным по предыдущей статье методом поиска в **ADSP-BF533 Blackfin Processor Hardware Reference** находим необходимые для понимания кода инициализации регистры:

```
// enable UART clock.
   *pUART_GCTL = UCEN;
// Read period value and apply formula: divisor = period/16*8
// Write result to the two 8-bit DL registers (DLH:DLL).
   *pUART_LCR = DLAB;
   *pUART_DLL = divisor;
   *pUART_DLH = divisor>>8;

// Clear DLAB again and set UART frame to 8 bits, no parity, 1 stop bit.
   *pUART_LCR = WLS(8);
```

В этом регистре (рис. 2) нас, прежде всего, интересует бит UCEN (Enable UART Clocks). Впрочем, автор программы под UCEN мог определить все, что ему заблагорассудится. Но строка из **defBF532.h**:

```
#define UCEN 0x01
```

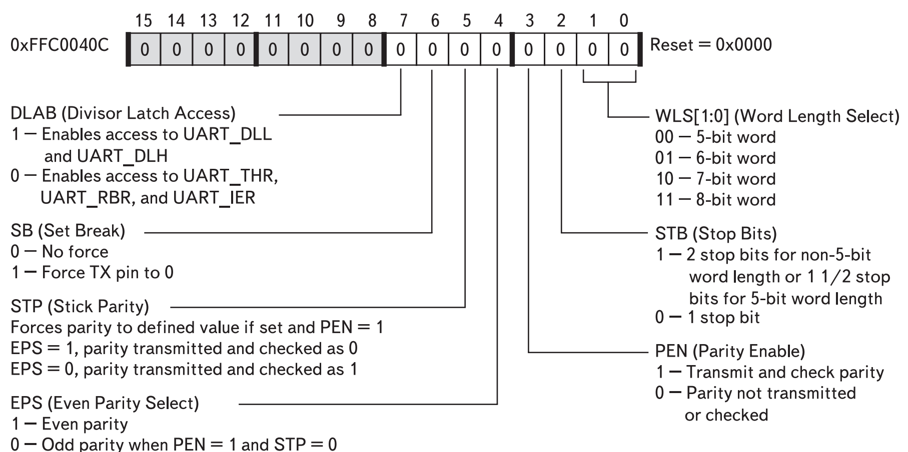

Рис. 3. Регистр UART Line Control Register

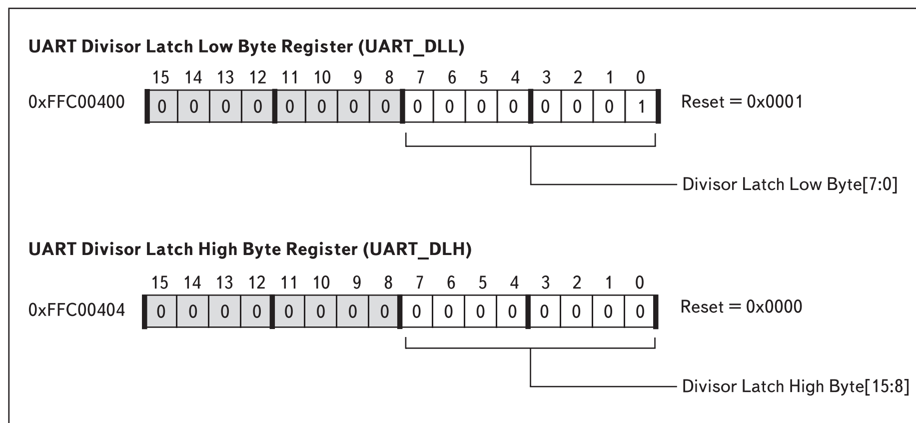

Рис. 4. Регистр UART Divisor Latch Registers

убеждает нас в обратном. А теперь, внимание — «подводный камушек номер один». Этот управляющий бит введен для поддержки режима энергосбережения POWER SAVE для отключения тактирования UART. Во всех остальных случаях он должен быть установлен! Другие биты этого регистра станут нам интересны для написания более сложных программ.

Поверим автору программы и своим глазам: DLAB, очевидно, всего лишь устанавливает бит 7 регистра UART_LCR, Divisor Latch Access (DLAB) (рис. 3).

Это необходимо, чтобы обратиться к регистрам UART_DLL и UART_DLH, а не к регистрам UART_THR, UART_RBR и UART_IER. Дело в том, что регистры UART_THR, UART_RBR и UART_DLL находятся по физическому адресу 0xffc0 0400, а UART_DLH и UART_IER — по физическому адресу 0xffc0 0404. Логика функционирования UART такова, что никогда не возникает необходимости использовать эти конкурентные по адресам регистры одновременно. Как видим, дно на нашей подводной стезе просто усеяно камнями. Со следующими тремя строками нам будет проще. Действительно, очень логично, что каков бы ни был по величине divisor, но получить его значение для инициализации UART_DLH можно именно сдвигом вправо. Старшие биты просто уничтожатся при выводе в регистр UART_DLH (рис. 4). Однако не удивляйтесь, если разные значения divisor приведут к одинаковым скоростям обмена UART. Просто только 16 бит этой переменной значащие!

И, наконец, WLS(8) — это не что иное, как макрос с параметром, определенным в **defBF532.h** следующим образом:

```
#define WLS(x) ((x-3) &0x03)
```

Таким образом, в нашем случае имеем 5&0x03 (8-битное слово).

Теперь, когда мы в общих чертах разобрались с инициализацией UART, приступим к созданию собственного проекта драйвера. Самое сложное для начала — определить, из чего должен состоять проект, и как обозначить данные. Опытные программисты серьезно обдумывают этот этап, чтобы сэкономить усилия и время в дальнейшем. Мы не имеем в виду определенной операционной системы, поэтому правильнее говорить не о драйвере, а скорее лишь о его «заготовке». Но чтобы наблюдать результаты работы, нам все равно придется «погрузить» ее в некую среду. Расположим эту «примитивную среду» в главном программном модуле **main.c**. Процедуры инициализации устройств определим, как и в рассмотренном в первой части статьи демонстрационном примере, в модуле **Initialization.c**. Прочие обработчики прерывания (например, от таймера и кнопок) будем держать в файле **ISRs.c**, и, наконец, все относящееся к UART — в файле **Uart.c**. А зачем нам кнопки (светодиоды) и таймеры, работающие по прерываниям? Не забывайте, что основная цель применения DSP Blackfin — цифровая обработка сигналов в реальном времени, поэтому все вспомогательное и несущественное, включая обмен по последовательному каналу UART с терминальным компьютером, должно работать по прерываниям, «в фоне».

Если вывод на терминал не производится вследствие ошибок в программе, то должна оставаться возможность их как-то визуализировать. Используем для этого светодиоды, как было описано в первой части статьи. Кнопки же помогут в том случае, если по какой-то причине не обрабатываются команды ввода с терминального компьютера. Для простоты и надежности будем использовать стандартные заголовочные файлы. Итак, вот что у нас содержится в **main.c**:

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
  (C) Copyright 2005 — Ltd. Sankt-Petersburg, Russia

  Developer: Andrey Savichev, e-mail: avisv@hotbox.ru
  Project Name: COM- port Test
  Date Last Modified: 08/02/07
  Version: 1.0
   Compiled with: VisualDSP++ 4.0
              Product Version: 4.0.0.0
              IDDE Version: 6.5.0.1
  Hardware: ADSP-BF533 EZ-KIT Lite, CLKIN 27 MHz
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

«Шапка» главного модуля в проектах, выполненных на Си, исключительно важный атрибут «правильной программы». Сразу видно, к кому обращаться за разъяснениями, в какой среде написана и отлажена программа, на каком «железе» должна работать.

```
#define UART
```

Это ключ для условной компиляции. В таком простом примере он кажется избыточным, но при отладке бывает очень удобно комментированием одной строки отключать огромные куски кода, заключенные в скобки #ifdef …#endif. Основная идея драйвера достаточно проста. Поскольку обмен с терминальным компьютером асинхронный, мы хотим застраховаться от потери символов при приеме и передаче. Будем использовать циклические буферы **RcvBuf** и **TrBuf** соответственно на прием и передачу. Пусть они имеют одинаковую длину в 256 байт. В реальной ситуации для эффективной работы их длины могут быть различны. Новичкам в программировании сообщим, что циклический буфер организуется за счет указателя ***pTr_Buf** текущей позиции записи для **TrBuf** или чтения ***pRcv_Buf** для **RcvBuf**. Указатель перемещается от начала буфера к его «хвосту». Поэтому при вводе (выводе) очередного символа их значение инкрементируется. Обратите внимание, программа среды исполнения драйвера, размещенная в модуле main.c, использует две различные процедуры:

- ***int putStr (unsigned char* s, unsigned s_size)*** — для записи символов в буфер передатчика;
- ***int getStr (unsigned char* s, unsigned* s_size)*** — для считывания символов из буфера приемника.

Сам драйвер состоит из части, обслуживающей циклический буфер передатчика через указатель ***pTr_Buf**, и другой, поддерживающей буфер приемника посредством ***pRcv_Buf**. Когда указатель оказывается на последней позиции буфера и переходить ему дальше некуда, он благополучно перемещается обратно на свое начало («лиса, схватившая себя за хвост»). Символ в этой первой позиции, конечно, стирается, и на его место записывается новый (в случае **TrBuf**), или считывается повторно, если он не был обновлен (в случае **RcvBuf**). Какой в этом смысл? Драйвер в части, связанной с приемом, является «поставщиком данных». Если бы он не буферизировал данные, они бы пропадали для потребителя уже в регистре **UART_RBR**! А так он сохраняет их в зависимости от скорости приема (вернее, от реального темпа поступления входной информации) и размера циклического буфера. Потребителем данных у нас является программа в **main.c**, вызывающая функцию **getStr**.

```
while(1)
  {
    if ( DataTr_flag )
      { putStr(s2,l2);
        DataTr_flag=0;
      }
    if ( DontConnect_flag )
      {
         if ( !getStr(sr,&lsr) )
           {
             if (RcvBufOverflow_flag)
               {putStr(s3,l3);
                 RcvBufOverflow_flag=0;
               }
             else if (lsr != 0)
                 putStr(sr,lsr);
             *pFlashA_PortB_Data = 0x0; // гасим светодиод
             CntDown = 10;
         } // end if ( !getStr(sr,&lsr) )
      } // end if ( !getStr(sr,&lsr) )

    *pRTC_ICTL = SIE | MIE | HIE | SWIE;// | AIE | DIE | DAIE | WCIE
     while ((*pRTC_ISTAT&0x8000)==0);
  }
```

Не пытайтесь понять сразу весь приведенный фрагмент кода, мы разберем его детально шаг за шагом. Сейчас мы хотели лишь обратить ваше внимание на то, что функция **getStr** вызывается лишь в одном месте и в бесконечном цикле, следовательно, с некоторой вполне определенной периодичностью.

Если за время между обращениями к **getStr** буфер приемника будет заполнен, то драйверу в его части, обслуживающей прием, ничего не останется, как записать новую информацию на место предыдущей, еще не считанной. Правда, при этом он может установить флажок **RcvBufOverflow_flag**. Да, но каким образом обнаружить переполнение буфера? Хороший вопрос. Чтобы обеспечить такую возможность, мы должны сделать глобальной переменной указатель на тот же самый **unsigned char RcvBuf**[256] циклический буфер программы-потребителя информации **unsigned char *pcur_Rcv**. Теперь давайте мысленно представим себе модель того, что у нас произойдет на примере из другой области. По кольцевой дороге в одном направлении движутся два автомобиля. На одном эмблема «поставщик», на другом «потребитель». Причем «поставщик» не обращает никакого внимания на скорость движения «потребителя», а последний сбрасывает свою до скорости первого в том случае, если его нагоняет. При этом условии выбросить флаг «обгон» (то есть **RcvBufOverflow_flag**) со стороны «поставщика» не столь сложно. Если он был впереди (первым пересек некую мысленную черту, например, на половине пути от места их старта), а затем вновь настиг своего соперника, значит, произошел «обгон», то есть в нашем случае — «переполнение». А если он обгоняет второй, и третий, и сотый раз — это уже не принципиально, так как несчитанная информация все равно уже пропала. Чтобы еще более приблизить эту модель к реальности, добавим, что автомобиль «поставщика» едет не непрерывно, не своим ходом. Его как бы подталкивают с определенным усилием, и далее он движется по инерции, затем останавливается. Автомобиль же «потребителя», напротив, едет сам, но только в том случае, если рядом не оказывается соперника по бесконечной кольцевой гонке. Если «поставщик» оказался рядом и без флага «обгон», «потребитель» выбрасывает флажок «делать мне нечего» и никуда не едет. Теперь вам должно быть понятно, как это все работает. Для кольцевого буфера передатчика модель с кольцевыми гонками разработайте по аналогии сами. Только в этом случае происходит смена ролей — часть драйвера является уже «потребителем», а процедура **putStr** из основной программы — «поставщиком». Давайте вернемся к исходным текстам. Для архитектуры **Blackfin** события «освобождение регистра передатчика UART_THR» и «новый символ в регистре приемника UART_RBR» вызывают одно и то же прерывание, и, следовательно, запускается одна и та же программа, обслуживающая это прерывание. Различить, от какого именно события прерывание возникает, возможно, анализируя бит «статус» в регистре pUART_IIR. Для хранения статуса вводим переменную **unsigned stat** в модуле **UART.c**:

```
EX_INTERRUPT_HANDLER(UART_ISR)
 { unsigned stat;
    unsigned t;

    stat= *pUART_IIR;
    stat= stat>>1;
    if ( stat == 0x0 )
       {
         ; //это сбой
       }
       // эта часть драйвера обслуживает кольц. буфер передатчика
         else if ( (stat&0x1) == 0x1 )       // UART_THR empty.
           { if ( pTr_Buf == pcur_Tr )    // если выводить нечего
             { Data4TrAbsent_flag = 1;
             }
             else
           { Data4TrAbsent_flag=0;
             *pUART_THR = *pTr_Buf; // вывод очередн. байта
из кольц. буфера
               if ( pTr_Buf == (TrBuf+buf_size-1)) // закольцовы-
ваем буфер передачи
                 pTr_Buf =TrBuf;
               else
             pTr_Buf++;
           }
         }
       // а эта часть драйвера обслуживает кольц. буфер приемника
       else if ( (stat&0x2) == 0x2 )          // Receive data ready.
         { t= *pUART_RBR;
            if ( t != 0 )                      // не обрабатываем NULL!
              { *pRcv_Buf=t;
                 if ( pRcv_Buf == pcur_Rcv ) //
                   { if ( CyclEnd_flag )
                     { RcvBufOverflow_flag=1;
                       CyclEnd_flag = 0;
                     }
                   //else первый символ
              }
           if ( pRcv_Buf == (RcvBuf+buf_size-1)) // закольцовы-
ваем буфер
              { pRcv_Buf = RcvBuf;
                 CyclEnd_flag = 0x1;
              }
                else
                  pRcv_Buf++;
           } end if ( t != 0 )
       } end if ( (stat&0x2) == 0x2 )
 } //end EX_INTERRUPT_HANDLER
```

На что нужно обратить внимание в этом фрагменте кода? Прежде всего, на флаги. **Data4TrAbsent_flag** устанавливается тогда, когда освобождается регистр передатчика ***pUART_THR**, а все символы для передачи уже выведены. В этом случае, чтобы возобновить процесс передачи по прерываниям, первый готовый для передачи символ требуется вывести без прерывания! Взгляните на **putStr**:

```
int putStr(unsigned char* s, unsigned s_size)
  { unsigned i;
     unsigned char st_char;
       tested=0;
       st_char=*s;
       for ( i=0; i< s_size; i++ )
       { *pcur_Tr= *s;
          s++;
          if (pcur_Tr == (TrBuf+buf_size-1))
            {pcur_Tr = TrBuf;
            }
            else
          pcur_Tr++;

       }
     if(Data4TrAbsent_flag ){
       *pUART_THR =st_char;
       if (pTr_Buf==(TrBuf+buf_size-1))
              pTr_Buf=TrBuf;
              else
              pTr_Buf++;
     Data4TrAbsent_flag=0;}
     return 0;
  }
```

Сначала мы заносим все символы для вывода в кольцевой буфер передатчика, затем проверяем, успел ли за это время (или, быть может, даже до первоначального момента записи в буфер) прекратиться вывод. Если да, мы должны выводить символ без прерывания. Но какой именно? Тот, что мы предусмотрительно сохранили в **st_char**. Теперь (внимание!) в процедуру «поставщика» мы вставляем кусок кода из «потребителя» (то есть из драйвера). Почему мы вынуждены это сделать? Потому, что в драйвер *EX_INTERRUPT_HANDLER(UART_ISR)* мы сможем попасть только по прерыванию, а оно не возникнет, раз мы были в обработчике прерывания и не записали очередной символ в **pUART_THR**, поскольку нам нечего было выводить. А что же делать в самом начале работы, чтобы процесс вывода символов стартовал? Просто установить флаг **Data4TrAbsent_flag**, и при первом же обращении к **putStr** все произойдет, как надо.

**pUART_THR = NULL* в начале работы при этом делается совершенно не нужным.

Посмотрим теперь, как работает **getStr**:

```
int getStr ( unsigned char* s, unsigned *ps_size )
 { unsigned char* temp;
    temp=pRcv_Buf;
    *ps_size=0;
    if ( pcur_Rcv < temp ){ *ps_size=1;
                 while (pcur_Rcv < temp)
                 {*s=*pcur_Rcv; s++; pcur_Rcv++;}
                 }
       else if ( temp < pcur_Rcv )
         { if ( CyclEnd_flag )
           { while ( pcur_Rcv != temp )
             {*s = *pcur_Rcv; s++; *ps_size++;
             if ( pcur_Rcv == (RcvBuf+ buf_size-1) )
               { pcur_Rcv= RcvBuf; CyclEnd_flag=0;
               }
               else
               pcur_Rcv++;
             }
           }
         }
       else if ( temp == pcur_Rcv ) return 1;

    return 0;
    }
```

Зачем введена переменная temp? Для защелкивания указателя **pRcv_Buf**. Ведь основ-

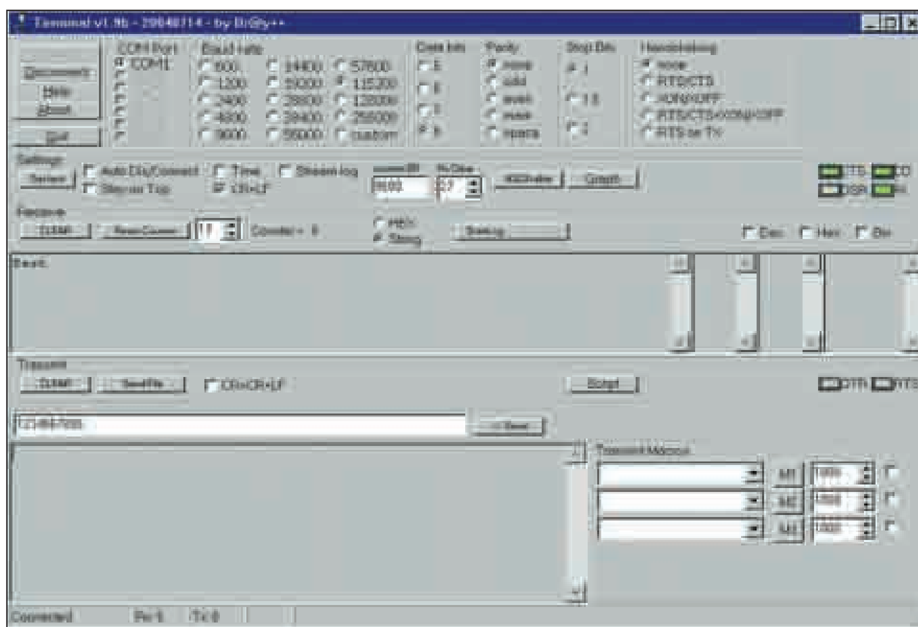

Рис. 5. Вид терминальной программы при запуске основной программы

ная программа у нас выполняется в «фоне». При очередном обращении к **getStr** мы пытаемся считать все символы, которые появились в буфере от драйвера за время, прошедшее с момента предыдущего вызова **getStr**. Но процесс ввода по прерываниям может продолжаться и в то время, пока мы находимся в **getStr**.

pRcv_Buf при этом будет перемещаться по кольцевому буферу приемника, а мы его используем в условных циклах и сравнениях! Да, когда имеешь дело с прерываниями, надо быть весьма собранным и осторожным. Еще вопрос: для чего нужен **CyclEnd_flag**? Чтобы различить две схожих ситуации, когда **pRcv_Buf == pcur_Rcv**. В одном случае это переполнение кольцевого буфера приемника, когда **CyclEnd_flag** установлен. Когда же он сброшен, это означает, что с момента предыдущего обращения к **getStr** изменений не было, так как от драйвера никаких новых символов в буфер не поступало.

Если в буфере есть новые символы, getStr возвращает нам 0. Поэтому:

```
if ( !getStr(sr,&lsr) )
   { // наши действия, при наличии новых символов
      // в кольцевом буфере приемника
      if (RcvBufOverflow_flag)
         { // в буфере приемника были потеряны данные
            RcvBufOverflow_flag=0;
         }
      else
      { // все данные в буфере приемника достоверны
      }
   }
```

Что же делать, если данные все же были потеряны, или возникали ошибки при приеме данных в *EX_INTERRUPT_HANDLER(UART_ERR_ISR)*? Они не фатальны, если программа со стороны терминального компьютера поддерживает протокол более высокого уровня, чем рассматриваемый нами сейчас. Многоуровневые протоколы обмена по последовательному каналу позволяют повысить надежность доставки информации к потребителям. А могут ли возникнуть проблемы при работе с кольцевым буфером передатчика? Да, конечно. Поскольку производительность процессора BlackFin огромная, то весьма вероятно, что при интенсивной передаче по каналу столь маленький кольцевой буфер будет не в состоянии обеспечить передачу данных без потерь. Мы оставляем читателю модернизацию нашего кода, хотя в следующей части статьи и улучшим его. Необходимо добавить флаг TrBufOverflow_flag и написать соответствующие секции кода для драйвера EX_INTERRUPT_HANDLER(UART_ISR) и функции **putStr**. Можно, конечно, поддаться соблазну максимального увеличения размеров кольцевых буферов, но это далеко не лучшее решение. В конкретной задаче для минимальных издержек на ввод-вывод необходимо выбрать оптимальную величину. Тонкая настройка систем, работающих по прерываниям, это тоже отдельная тема, которую мы не будем затрагивать. Если задача сложная, и вы ограничены во времени, лучше использовать какую-нибудь подходящую систему реального времени. Хотя, чтобы понять, насколько она вам подходит, тоже придется потратить время. Но вернемся к нашему простому примеру. Построим проект в директории OurProject, как это описано в первой части статьи. Если ошибки отсутствуют, вы увидите в нижней части экрана визуализацию процесса загрузки в ADSP-BF533 EZ-KIT Lite. Перед запуском программы на основном компьютере убедитесь, что терминальная программа на вспомогательном компьютере запущена и готова к обмену. После запуска программы **Debug**->**Run** мы наблюдаем на экране терминального компьютера следующее окно (рис. 5).

Вы видите, что в окне консоли приемника появилось Test. Это произошло в результате выполнения кода из **Main.c**:

```
unsigned char* s1=»Test\n»;
unsigned l1= 5;
…
putStr(s1,l1);
```

В строке редактирования сообщений для передачи терминальной программы Terminal v1.9b видим «1234567890» Добавим к этой строке символ «_» для удобства контроля выводимых строк.

Если теперь нажать на виртуальную клавишу «-> **Send**», эта строка пересылается в нашу основную программу, и выполняются операторы в бесконечном цикле:

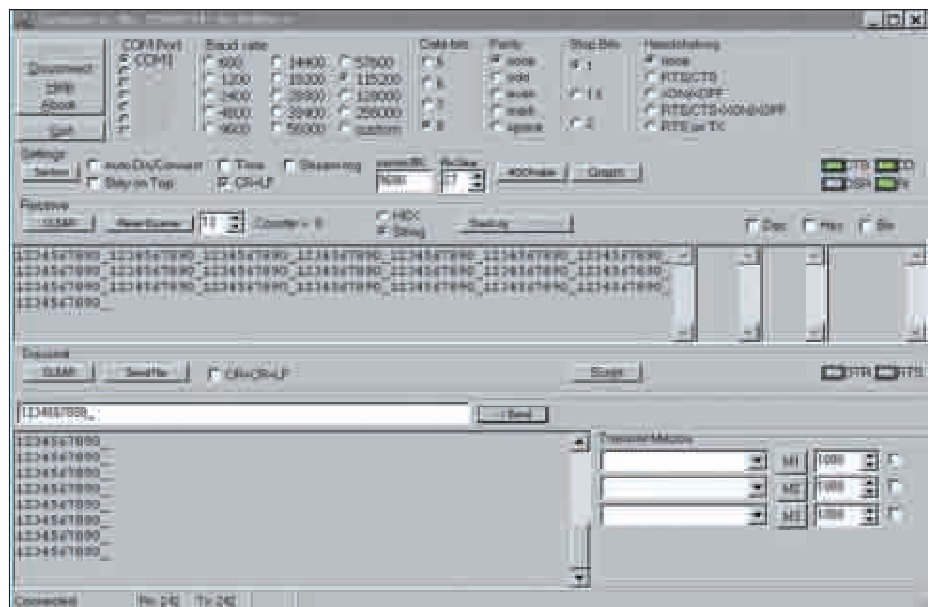

Рис. 6. Вид терминальной программы при многократном нажатии на виртуальную клавишу ADSP-BF533 EZ-KIT Lite

```
if ( !getStr(sr,&lsr) )
    if (RcvBufOverflow_flag)
        …
    else if (lsr != 0)
                    putStr(sr,lsr);
            *pFlashA_PortB_Data = 0x0; //гасим светодиод
            CntDown = 10;
        …
    }
```

Повторим нажатие на клавишу «-> **Send**» многократно.

Фактически мы снова реализовали LoopBack, но уже не с помощью короткого соединения ввода и вывода на COM-порте терминального компьютера, а через плату ADSP-BF533 EZ-KIT Lite (рис. 6). Если вы не будете нажимать на клавишу «-> **Send**» в течение 10 с, то загорится светодиод на плате ADSP-BF533 EZ-KIT Lite. Но об этом поговорим в другой раз. Собственно, пора переходить к созданию приложения, обрабатывающего сигналы. Однако прежде мы разработаем полноценный протокол связи, чтобы можно было обмениваться информацией между терминальным компьютером и платой ADSP-BF533 EZ-KIT Lite. Мы разработаем также систему, позволяющую синхронизировать время, чтобы отладить на ней наш протокол обмена. Одновременно это позволит нам разобраться с производительностью процессора и рассмотреть вопросы, связанные с ее изменением и потреблением мощности, что чрезвычайно важно для определенных приложений. Если у вас возникли вопросы, замечания, уточнения или пожелания, направляйте их авторам по адресу электронной почты, указанному в начале статьи. Все исходные тексты программы доступны по адресу **ww w.eltech.spb.r u/addons/bfcom.zip**.

Литература

1. ADSP-BF533 EZ-KIT Lite Evaluation System Manual **h t-t p://w ww.analog.c om/UploadedFiles/Associated_Docs/68863602ADSP_BF533_EZ_KIT_Lite_Manual_Rev_3.0.pdf**
2. ADSP-BF533 Blackfin Processor Hardware Reference. **ht tp://w ww.analog.c om/UploadedFiles/Associated_Docs/892485982bf533_hwr.pdf**