Getting video stream from USB web-camera on Arduino Due - Part 2: UART

Mar 6, 2015 6 min read words

C Dev Beginner Intermediate

by Sergiy Bogdancev
Contributor

Introduction

Part 1 is here.

Two more things are needed to start dealing with USB, a serial communication with a computer and some kind of monitor to see stream video. In this article I'll show how to initialize serial communication component, create functions to print strings, hex, decimal and binary numbers. Then in the next article I'll briefly discuss TFT monitor and watchdog timer.

Serial communication

I will use UART (Universal Asynchronous Receiver Transmitter) **[1, p.755]** for serial communication with my laptop. Computers used to have COM ports (RS-232) for serial communication, UART is perfectly compatible with such ports except signal voltage levels. But don't worry, in Arduino Due it all happens via USB cable and virtual COM port that is created by Arduino software (see previous article).

UART initialization

of UART.c

CODE

```
#include "UART.h"
```

All function declarations will go to UART.h file. I will call UART initialization function as UART_Init and add it to UART.h file:

CODE

```
#ifndef UART_H_
#define UART_H_

void UART_Init(void);

#endif
```

Implementation will go into UART.c file. RX and TX pins belong to PIOA as pins number 8 and 9. First I'll create the pin mask:

CODE

```
uint32_t ul_UART_pins_mask = PIO_PA8A_URXD | PIO_PA9A_UTXD;
```

In ARM processors pins can play a role of normal inputs / outputs or be used by peripherals. For example, pin 8 of PIOA can be normal input / output or can be input pin (RX) for UART. Maximum two peripherals can be assigned to one pin but only 1 of them can be selected at a time. Selection is controlled by PIO_ABSR register:

CODE

```
PIOA->PIO_ABSR &= ~ul_UART_pins_mask;       //Peripheral A selected for both
```


switch control of those pins over from PIO controller to UART:

CODE

```
PIOA->PIO_PDR = ul_UART_pins_mask;          //Moves pins control from PIO
controller to Peripheral
```

From this moment RX and TX pins are controlled by UART. Next I will specify UART parameters, speed will be 115200 b/s:

CODE

```
UART->UART_BRGR =84000000/16/115200;                    //Clock Divisor value
UART->UART_MR = UART_MR_CHMODE_NORMAL | UART_MR_PAR_NO;  //Normal mode, no
parity
UART->UART_CR = UART_CR_TXEN;                           //Enable transmitter
UART->UART_IDR = 0xFFFFu;                               //Disable all
interrupts
```

Last step is to enable clock to UART otherwise it won't work:

CODE

```
PMC->PMC_PCER0 |= (1<< ID_UART);
```

This the end of UART initialization and from that moment UART is able to transmit data to a connected computer.

Sending string via UART

I will write a sending string function that accepts pointer to null-terminated string. UART sends data byte-by-byte, thus sending a string requires iteration through each byte of the string until **null** is reached. During iteration each byte must be copied to Transmit Holding Register (THR) when it is empty. Let's call this function **PrintStr**, put its declaration into

```
void PrintStr(char *Text)
{
    int i = 0;
    do
    {
        while(0 == (UART->UART_SR & UART_SR_TXRDY)) {}   //wait until previous
character is processed
        UART->UART_THR = Text[i];                        //send next character
        i++;
    }
    while (Text[i] != 0);                                //loop until string
termination symbol (null)
}
```

To test it I'll add a call to this function to the same place where it toggles LEDs (see previous article). Thus every time LEDs toggle message will be sent to a computer. Main file is looking like this now:

CODE

```
#include "sam.h"
#include "UART.h"

uint32_t SysTickCounter;

void SysTick_Handler(void)
{
    uint32_t Pin_LEDRX;
    uint32_t Pin_LEDTX;

    SysTickCounter ++;
    if(1000 == SysTickCounter)
    {
        SysTickCounter = 0;
```

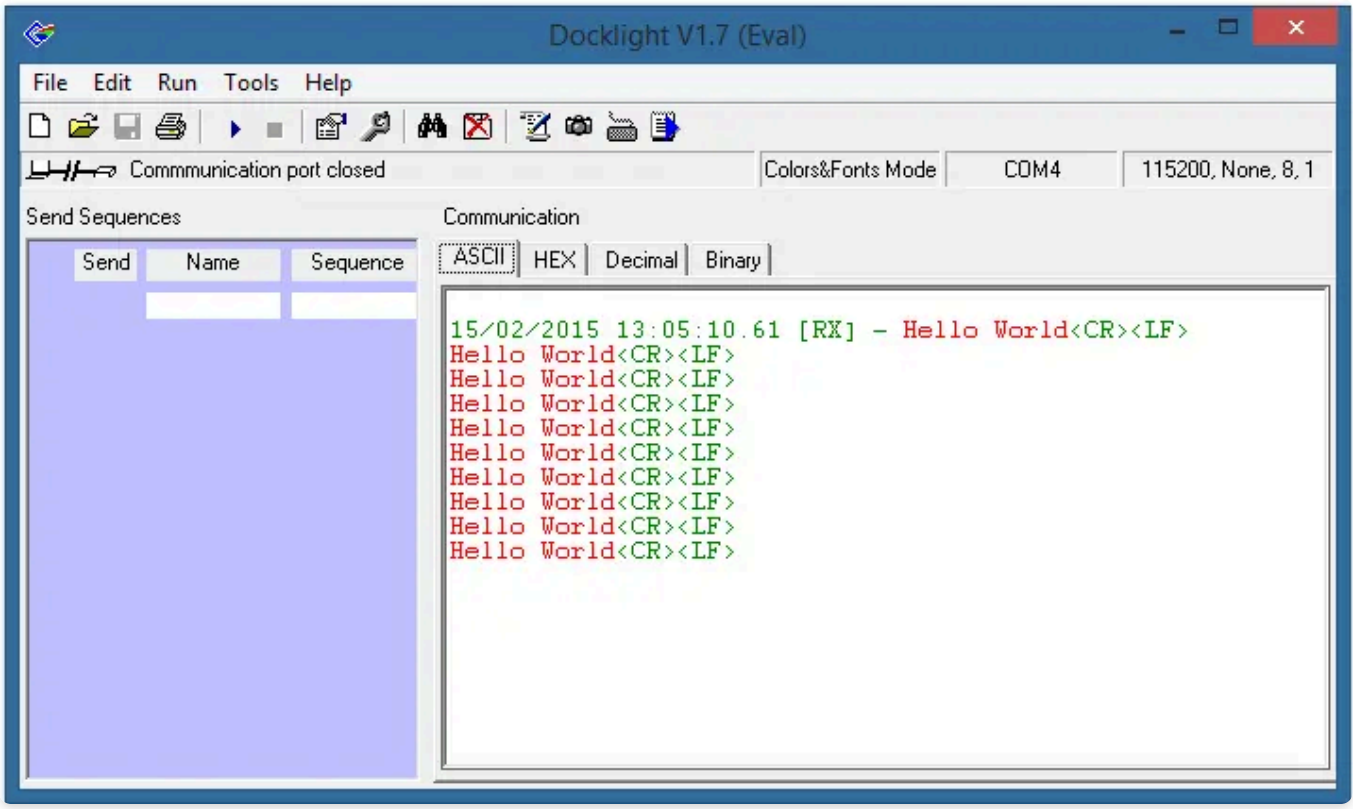
```
                PIOA->PIO_CODR = Pin_LEDTX;
            }
            else
            {
                PIOC->PIO_SODR = Pin_LEDRX;    //Lights off
                PIOA->PIO_SODR = Pin_LEDTX;
            }

            PrintStr("Hello World\r\n");       //Testing call is here!!!
        }
    }

    int main(void)
    {
        SystemInit();
        UART_Init();
        while(1);
    }
```

Please note that I added **#include UART.h** to top of the main file to link our UART code and a call to **UART_Init()** in the main function.

Build it and upload to Arduino Due board then launch Docklight, the output should be following:



Now you can see that sending data via UART is really simple. In future we are going to print USB device descriptors, for that more than just printing strings is needed. We also need to be able to print number in hex, bin and decimal format. So let's write such functions.

Printing in HEX format

Printing in HEX format is easy. As you know (don't you?) a byte is represented by two hexadecimal digits, 4 bits per each digit. For example 200 is 0xC8. Prefix 0x means that number is represented in HEX. Thus the algorithm of printing in HEX is following: first 0x is printed, then half byte is extracted, then that half byte will be an index in the following array:

CODE

```
const char arHex[16] =
{'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
```

with the index we get readable representation (char) that can be directly fed into UART.

CODE

```
void PrintHEX(uint8_t* ptrBuffer, uint32_t Length);
```

then in UART.c file:

CODE

```
void PrintHEX(uint8_t* ptrBuffer, uint32_t Length)
{
    uint8_t ByteToSend;
    for (uint32_t i = 0; i < Length; i ++)
    {
        PrintStr("0x");                                 //Printing prefix

        ByteToSend = ptrBuffer[i] >> 4;                 //Extracting higher
half-byte
        ByteToSend = arHex[ByteToSend];                 //Getting readable char
        while(0 == (UART->UART_SR & UART_SR_TXRDY)) {} //Waiting until UART is
ready
        UART->UART_THR = ByteToSend;                    //Sending

        ByteToSend = ptrBuffer[i] & 0x0F;               //Extraction lower
half-byte
        ByteToSend = arHex[ByteToSend];
        while(0 == (UART->UART_SR & UART_SR_TXRDY)) {}
        UART->UART_THR = ByteToSend;

        if(i != (Length - 1))
        {
            PrintStr(" ");                              //Separating bytes for
readability
        }
    }
}
```

with 0 in the higher half which makes higher half zero and I don't need to shift as bits are already in correct position to use them as array index.

For the rest HEX printing functions please see source code at the end of this article.

Printing in BIN format

To print in BIN format, each bit must be tested if it is 0 or 1. I'll use shift-left operation, once a bit is tested and printed, I shift it out which places next bit into position. Quantity of shift operations depends on the size of printed number, if it is one byte number - 8 times, two bytes number - 16 times, etc.

For better readability all half-bytes are separated by one white space, all bytes are separated by two white spaces.

Add declaration to UART.h file:

CPP

```
void PrintBIN8(uint8_t Number);
```

Then in UART.c file:

CPP

```
void PrintBIN8(uint8_t Number)
{
    uint32_t Result;                                    //Working variable
    for(int i = 0; i < 8; i ++)                         //There are 8 bits in
a byte
    {
        Result = (Number << i);                         //Getting needed bit
into position
        Result = (Result & (1u << 7));                  //Extracting it

        while(0 == (UART->UART_SR & UART_SR_TXRDY)) {}  //Waiting till UART is
        else
```

```
            UART->UART_THR = '1';                           //Sending '1'

        if((i + 1) % 4 == 0)                                //Every half-byte is
separated
        {                                                   //by white space for
better readability
            while(0 == (UART->UART_SR & UART_SR_TXRDY)) {}
            UART->UART_THR = ' ';
        }
    }
}
```

For the rest BIN printing functions please see source code at the end of this article, they are very similar to this one.

Printing decimals

This one is a bit more complicated than others. Algorithm is following: a number is divided by 10, 100, etc. depending of its order of magnitude. For example: to show 243 it needs to be divided by 100, the result will be 2, then 2*100 is subtracted from 243 which results in 43. Next, 43 needs to be divided by 10, the result is 4, then 4*10 is subtracted which results in 3. That is how all three individual numbers are extracted. Lets' do this in C.

Add declaration to UART.h file:

CODE

```
void PrintDEC(uint32_t Number);
```

Then in UART.h file:

CODE

```
void PrintDEC(uint32_t Number)
```

```
    while(Temp / Divider)                       //Finding the divider
        Divider = Divider * 10;

    Divider = Divider / 10;                     //Don't need last one as
division by it makes zero
    while(Divider)
    {
        Result = Temp / Divider;                //Extracting a digit
(decimals are discarded)
        Temp = Temp - Result * Divider;         //Decreasing by 1 order of
magnitude
        Divider = Divider / 10;                 //Readjusting divider

        while(0 == (UART->UART_SR & UART_SR_TXRDY)) {}
        UART->UART_THR = Result + 48;           //Sending out the digit
    }
}
```

Take a note how individual digits are sent. 48 is added. In ANSII code 48 is '0', and for example adding 48 to 1 gives 49 which is '1'.

Testing all printing functions

After we wrote all necessary printing functions it's better to test them. This process includes specifying a parameter (number) to a function in the same format as we want it to be printed. For example, to test PrintDEC I'll specify number 240 and I should see "240" printed in RS232 monitor program. Following code tests PrintDEC function:

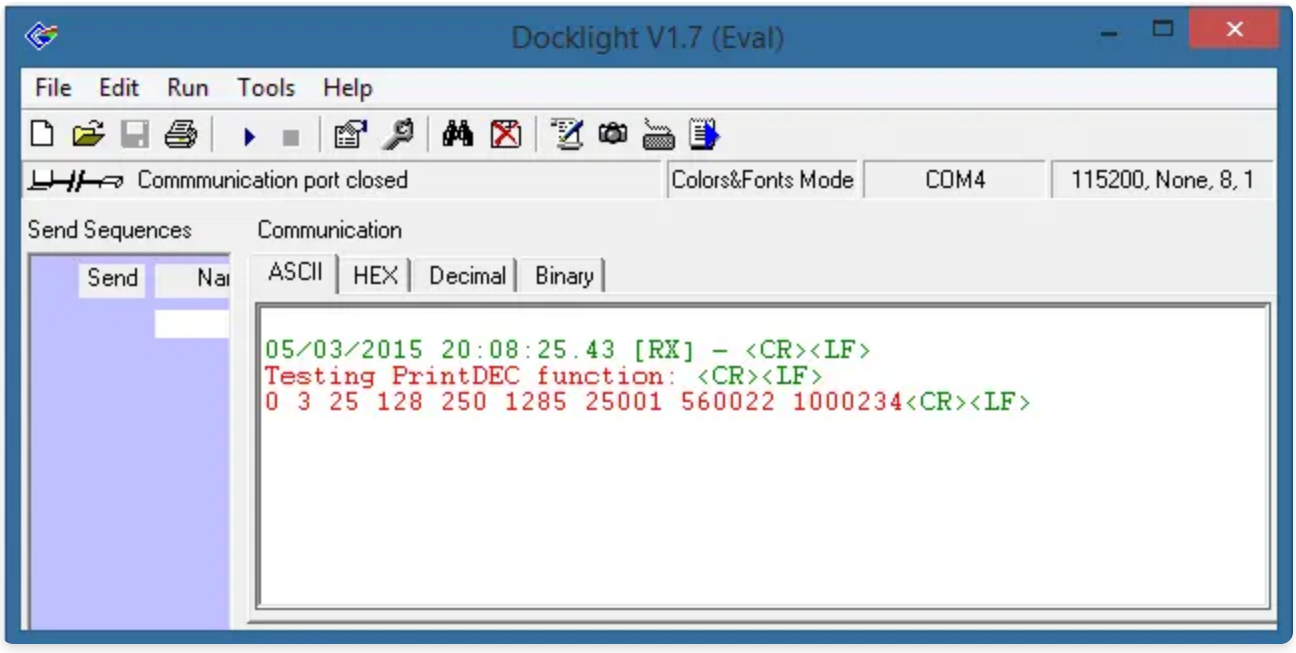
CODE

```
int main(void)
{
    SystemInit();
    UART_Init();
```

```
        PrintStr(" ");
        PrintDEC(25);
        PrintStr(" ");
        PrintDEC(128);
        PrintStr(" ");
        PrintDEC(250);
        PrintStr(" ");
        PrintDEC(1285);
        PrintStr(" ");
        PrintDEC(25001);
        PrintStr(" ");
        PrintDEC(560022);
        PrintStr(" ");
        PrintDEC(1000234);
        PrintStr("\r\n");

        while(1);
    }
```

Build it and upload to Arduino Due. Output will be:



As you can see the output is exactly what input is. Please see source code for more testing

In this article I created necessary printing methods for debugging and reading USB descriptors. There will be one more "preparation" article before I start dealing with USB - about watchdog timer and TFT monitor.

Source code is here.

Part 3 is here.

UPDATE 10-07-2015: Reloaded source code without Debug folder.

License

by Sergiy Bogdancev
Contributor

Comments And Discussions (0)

Sort: Newest

Share your thoughts

You need to be signed in to participate in the discussion. Sign in

No comments yet.