

AN10689

Full-duplex software UART for LPC2000

Rev. 01 — 17 January 2008

Application note

Document information

Info	Content
Keywords	LPC2000, UART, software
Abstract	This application note illustrates how a simple software UART can be added to a LPC2000 microcontroller in case more of these serial interfaces are needed in the application.

Revision history

Rev	Date	Description
01	20080117	The initial release of the LPC2000 software UART application note

Contact information

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

LPC2000 family of microcontrollers offers a wide range of communication interfaces. Each microcontroller contains at least two Universal Asynchronous Receiver/Transmitters (UARTs). However, some applications require more of such peripherals to be available. While attaching an external stand-alone UART could help in most of these situations ever increasing price pressure and miniaturization requirements result in more demands for software UARTs. This application note describes one possible implementation of a full-duplex software UART.

2. Implementation basics

This solution relies on a single LPC2000 general purpose 32-bit timer and its ability to toggle a dedicated pin when needed while monitoring another pin at the same time.

Suggested software UART supports transfer with format of 1 start bit, 8 data and 1 stop bit with no parity bits. Baudrate and transmitter's and receiver's FIFO depth are selected during the compiling process and once set can not be changed later. These two FIFOs are easily accessible with help of several simple routines that allow individual character and/or string transfers.

The demo has been developed with Keil's uVision3 environment and tested on a MCB2140 board. Standard baudrate of 9600 bits/s in systems running at 12 MHz or 60 MHz was easily achieved. Port pins P0.29 and P0.30 were selected for the software UART transmit and receive lines. While hardware UART0 is not used in the demo, P0.29 and P0.30 are connected to P0.00 and P0.01 of the MCB2140 board. This is because UART0 pins are equipped with an external interface components that enable microcontroller to communicate over a serial link with another piece of equipment. Simply put software UART pins are piggybacked to UART0 pins.

Main code as well as oscilloscope screenshots acquired during the validation process are part of this document. Complete project can be downloaded from NXP microcontroller's support page on the Internet.

3. Transmission model

[Figure 1 "Software UART transmission model"](#) show details of the transmission model that is implemented in this software UART.

An assumption was made that once a waveform is started to be generated (i.e. the transmitter (Tx) line goes low because of the "start" bit) the CPU should have minimal involvement all the way till the end of the "stop" bit. Reasons for this are twofold. Firstly, the LPC2000 core has so much processing capabilities that it would be a waste of resources to have it more involved in toggling a single digital pin output. Secondly, real life examples easily prove that no matter how extensive timing analysis is performed on system's interrupt service routines (ISRs) and their execution time, there will always be a scenario that will take more time than anticipated. While UART timing is a pretty relaxed one where the transmitter and the receiver can operate with up to $\pm 2.5\%$ relative error from the nominal baudrate, achieving higher transfer rates is not all that simple.

In our transmission model two phases can be clearly identified: a data processing phase and a waveform generation phase.

A single character output, as per the UART format specified earlier, is broken into a sequence of pin toggling events on the software UART Tx pin. A LPC2000 peripheral that easily deals with time and toggling pins when needed is a general-purpose 32-bit timer (in particular TIMER0 was selected for this demo).

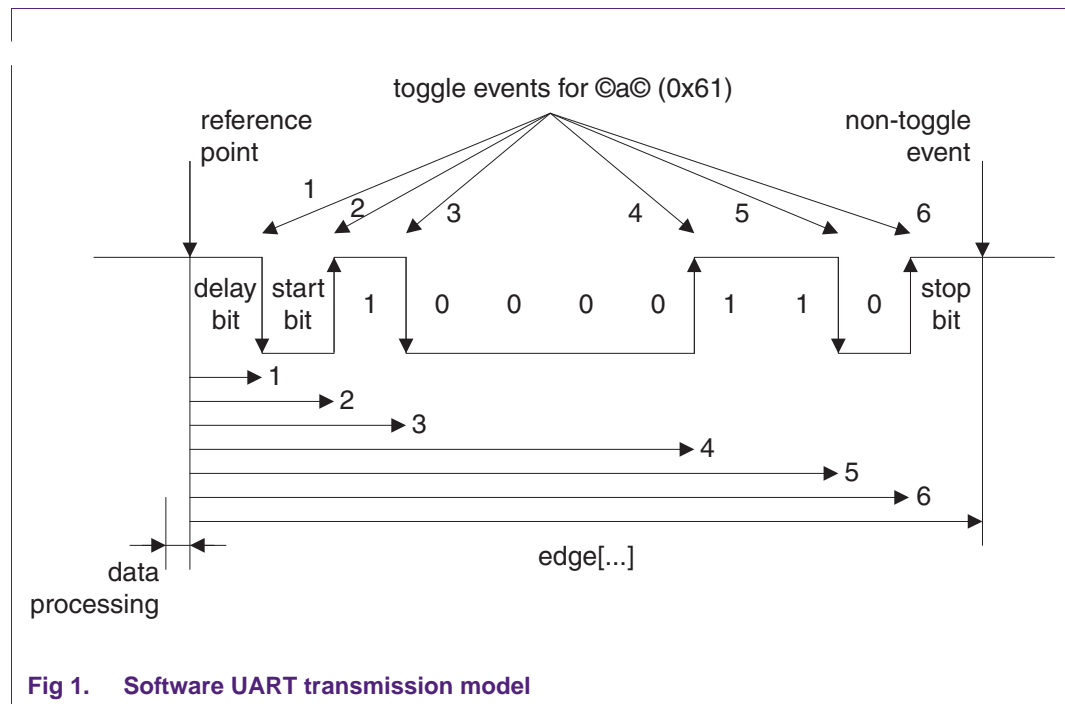


Fig 1. Software UART transmission model

Given character is analyzed during the data processing phase and points in time (based on the baudrate) when either a falling or a rising edge has to be generated on the Tx line are stored in an array (the edge[...] array). Each UART transmitted character begins with a falling edge (a “start” bit) and must have at least one rising edge (so that a “stop” bit can be generated). In between, rising and falling edges will alternate in presence. Once all edges are counted, the exact points in time when they will be generated are recalculated based on a single TIMER0 counter reading. This timer reading is what is marked as a reference point in [Figure 1](#). The whole waveform timing will be based on this point in time.

Although this software UART is equipped with a Tx FIFO, it is not capable of producing back-to-back transmissions. This is usually a price one has to pay when a communication peripheral is implemented in software. A gap of a single bit when the Tx line is high (“idle line”) will always be present in between two transmitted characters for several reasons. The key one is that a transmission can be triggered in an environment with a lot of ISRs firing off on their will. Having those ISRs interfering with timer setup code can result in offsets that could completely void timing of the sw UART output.

Updating the edge[...] array does not take long and immediately after it is finished the TIMER0 MAT3 channel is ready to perform first requested toggle on its physical pin and generate a TIMER0 interrupt. Once this happens, a dedicated ISR simply updates MR3 register with data on the next toggle point. When all toggling is finally done, the last remaining task for TIMER0 is to generate a “stop” bit.

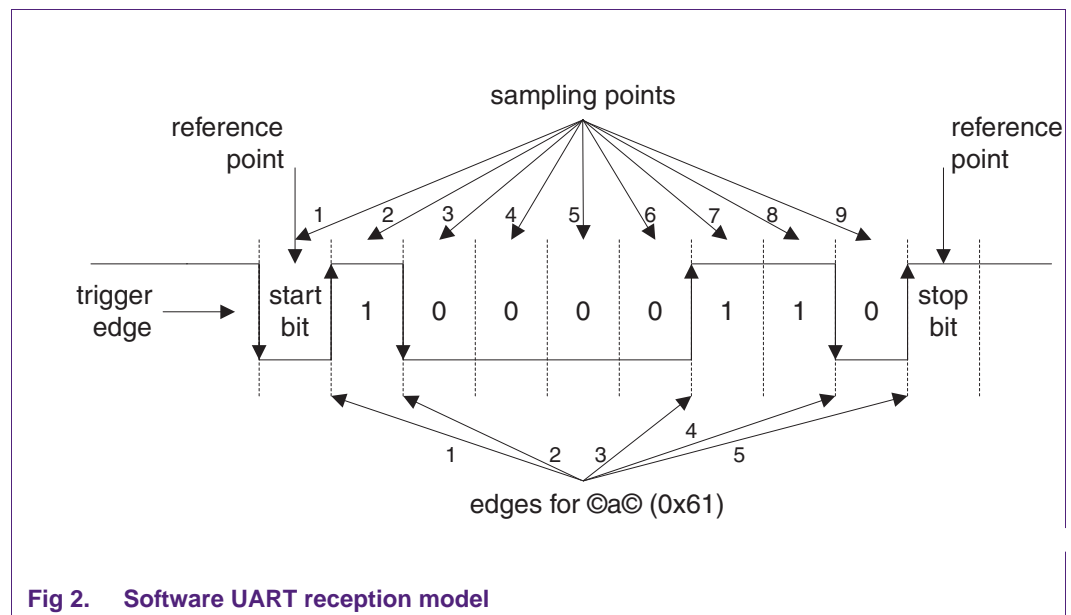
In case of the example shown in [Figure 1 “Software UART transmission model”](#), the first event occurs one bit after the reference point. This is when a falling edge is generated. Following this a TIMER0 ISR is executed and the TIMER0 MR3 register is loaded with edge[2] content. After another bit of delay, a toggle on MAT3 output will result in a rising edge on this pin. This is how the “start” bit will be generated. This procedure goes on as long as there are data in the edge[] array.

Having the requested waveform finished, the sw UART Tx checks for more data in the Tx FIFO. If there is none, it turns itself off. Otherwise, the next available character is fetched from the Tx FIFO and the cycle is repeated.

There are two routines provided with the demo code one can use to write into the Tx FIFO. They are `swu_tx_wr_chr(unsigned char)` and `swu_tx_wr(unsigned char*)`. The first one updates the Tx FIFO content with a new character if the Tx FIFO is not full. Otherwise, this routine will sit and wait for the software UART Tx to send at least one character and make more space available. This routine is also responsible for triggering a transmission in case the transmitter is not already active. The second routine basically writes a character after a character to the Tx FIFO using the services of the first routine.

4. Reception model

The reception model is more complex than the transmission model simply because in this case the microcontroller has to follow someone else lead -- it can never say for sure when (or even if) the receiver's (Rx) input will toggle or not. [Figure 2 “Software UART reception model”](#) illustrates fundamentals of the receiving process implemented here.



A single falling edge on the Rx line triggers a reception of the character. TIMER0 with its ability to capture timestamp of an edge event on the dedicated input is the perfect candidate to provide hardware support for the software Rx UART.

Similarly to the way the software UART Tx functionality is implemented, it is the edges in the received waveform that are used by the software UART Rx to estimate the actual data sent. In this process the software UART keeps track of all input line level changes using

the TIMER0 CAP0 pin. At the same time, sampling points located in the middle of each of the data bits in the waveform are compared against the timestamps the TIMER0 capture pin provides. If an edge is recorded between two sampling points, the next data bit will be different from the previous one. It is the change in the input line level that drives the whole reception process. Once the second reference point is reached (the center of the “stop” bit), the received content is tested for framing error. After this the data is stored in the Rx FIFO. If the Rx FIFO is full, the latest data gets dumped and an overflow indicator is set in the software UART status register.

Let's see some details of the receiving process shown in [Figure 2 “Software UART reception model”](#).

After a trigger/falling edge is detected by the TIMER0 CAP0 pin, the first reference point is calculated. This is the center of the start bit, half a bit delayed from the first falling edge. At the same time, the second reference point (the center of the “stop” bit) is calculated and loaded into one of TIMER0 match channels. It is used to trigger an interrupt in case no more edges are detected. Having no more edges detected will prevent the TIMER0 to identify the “stop” bit and will keep it looping forever.

Since a falling edge has been detected, TIMER0 CAP0 pin is reconfigured to detect rising edges as of that point on. In our example, the first edge following the trigger one is a rising edge. The Rx software will evaluate the following parameters: timestamp of the previous edge (the trigger one), timestamp of the current edge and all sampling points that lay in between these two.

Note that there is only one sampling point between the trigger edge and edge₁. This means that only one bit was received and that was a “start” bit.

Now it's time for CAP0 to start looking for a falling edge. Edge₂ matches this description. Sample point 2 is the only one between edge₁ and edge₂ and since edge₁ was a rising edge, bit “1” was received. CAP0 gets reconfigured again and starts looking for rising edges on the Rx line.

As this example shows, edge₃ follows after four bit intervals. Here is what happens now.

First, the software sets the last edge to the edge₂ timestamp and the current one to what is marked as edge₃. The next step is to find out if any sampling points can be squeezed in between. As we can see the first sampling point is sampling point 3. This results in a data bit ‘0’ being shifted into the receiver. After this the last edge gets moved one bit closer to edge₃ and the same process is repeated. It is now sampling point 4 that is in between the current edge and the last one. Again, a data bit ‘0’ gets shifted into the receiver and the process continues as long as the last edge can be moved to the right yet not getting beyond the current one and at least a single sampling point is in between them. If these conditions are not met the CAP0 pin is reconfigured and a new edge is sought.

Functionality similar to the receiver trigger level interrupt (an interrupt after so many characters get stored in the Rx FIFO) available in hardware UARTs is implemented in this software, too. When the specified trigger level is reached (or exceeded) a user's routine is called. The demo code illustrates this with a simple routine.

Only one subroutine in charge of a single byte transfer out of the Rx FIFO is provided in this application. It is left to the end user to build more complex code using this one.

5. Real-life results

This demo code was tested under two extreme conditions: in a system running an external 12 MHz crystal only and in a system with the PLL on generating a system clock of 60 MHz. In both cases peripheral and system clock were of the same rate. In addition to regular UART Tx and Rx lines, several debugging digital lines were added. Having a low level on any of those lines indicate matching activity in progress. These lines are labelled "CALL", "TX_PRO", "INT", "INT_TX", and "INT_RX" in the following screenshots.

The CALL line is low when a character based Tx FIFO loading routine is active. The TX_PRO line is low while the CPU is preparing toggle point parameters for the next character to be sent out. A low output on the INT line means that a TIMER0 interrupt was requested. While software UART Tx interrupt is serviced the INT_TX line is low and while the UART Rx interrupt is processed by the CPU the INT_RX line is low.

It is only when all these five diagnostics lines are high that the CPU is actually doing something different than handling the software UART code. Amount of time made available for the CPU will depend on two basic parameters of the demo code: the system clock and the software UART buffer sizes. The real life experiment results will illustrate impact each of these parameters has on the system performances.

5.1 Demo application output

Regardless of the selected system clock and Tx and Rx FIFO size, as long as it is a human being that is interacting with the demo using a PC's keyboard and a serial cable connected to a LPC2140 UART0, the application is likely to produce an output similar to the one in [Figure 3 "Demo code output"](#).

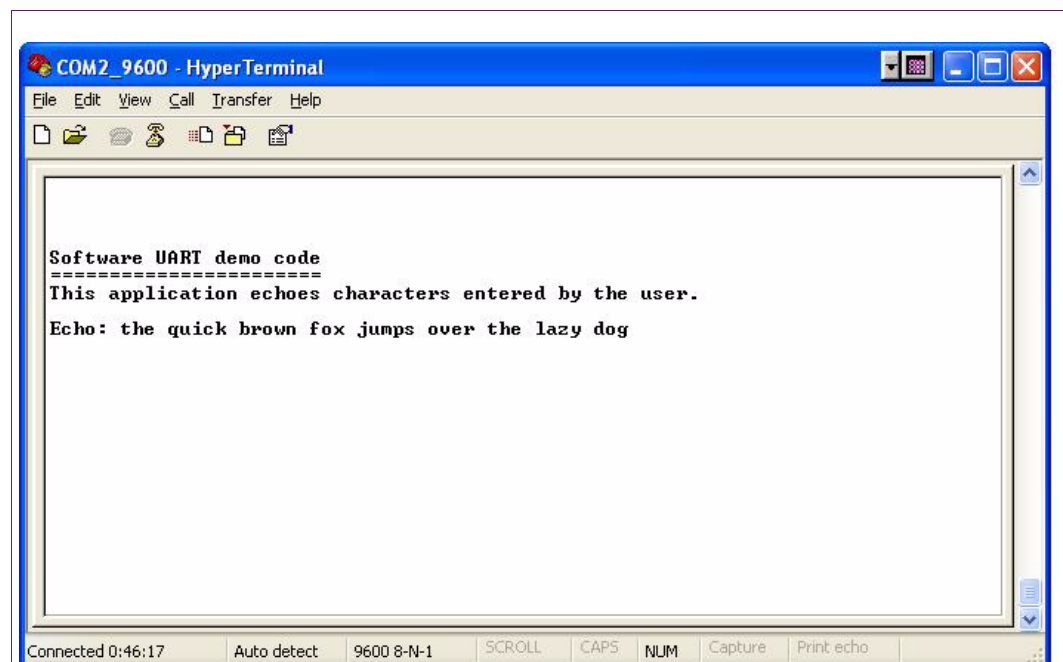


Fig 3. Demo code output

After a greeting message is displayed (a nice example on using a string copy and a character copy Tx routine in the demo code), the application ends in an infinite loop. Received character interrupts will drive further character display. Once a character is received on the software UART Rx line, it is echoed back via the software UART Tx line. In this echo mode single character read and write FIFO routines are fully utilized.

The demo code is written so that the LPC2000 on-chip PLL is not used at all. However, nothing prevents a user to turn it on and run the demo using a higher rate system clock. One has to remember that in case the system clock is changed from 12 MHz to any other value the serial data bit length specified as the "BIT_LENGTH" parameter must be adjusted accordingly. A single bit length is calculated based on the system clock and the desired baudrate as their ratio.

For example, in a 12 MHz system transferring serial data at 9600 bit/s, the BIT_LENGTH is equal to $12000000/96000 = 1250$.

5.2 Case 1: system clock 12 MHz, transmit FIFO size 16 bytes

This system runs with the MAM disabled (MAMCR = 0) and MAMTIM = 1. A single bit length in terms of system clocks is BIT_LENGTH = 1250.

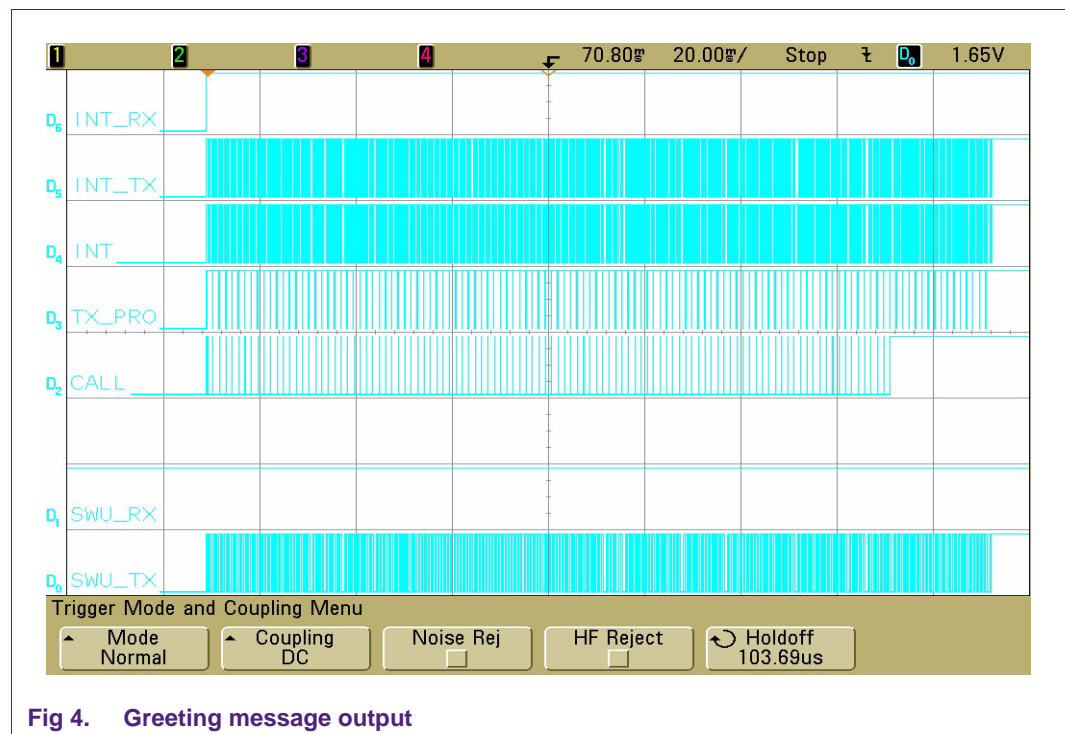
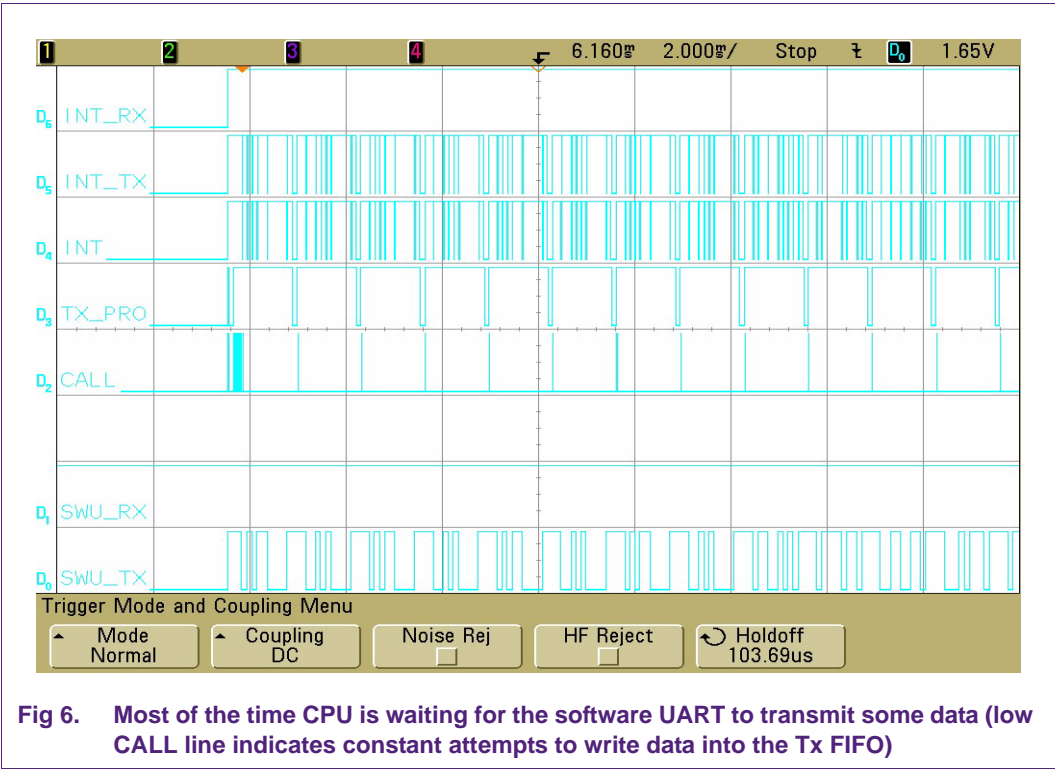
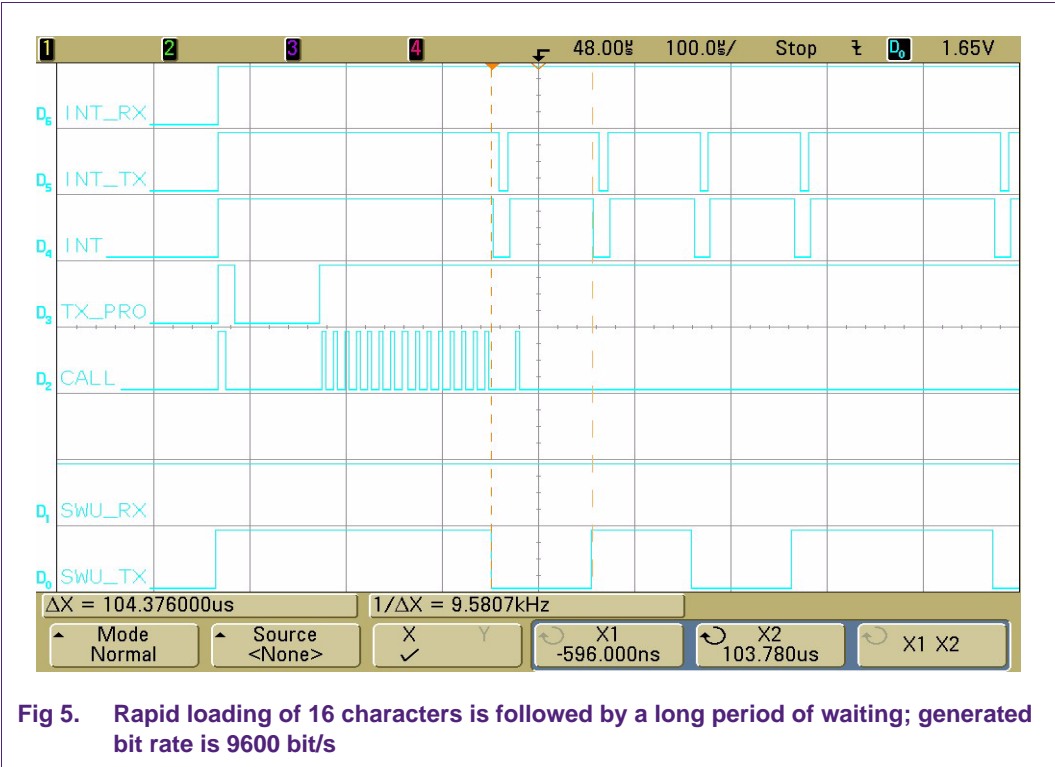


Fig 4. Greeting message output



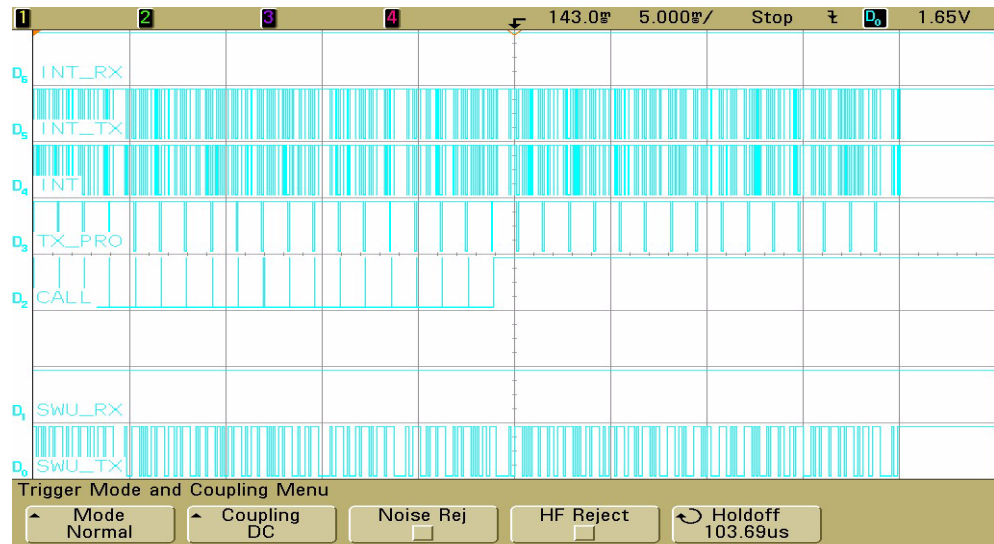
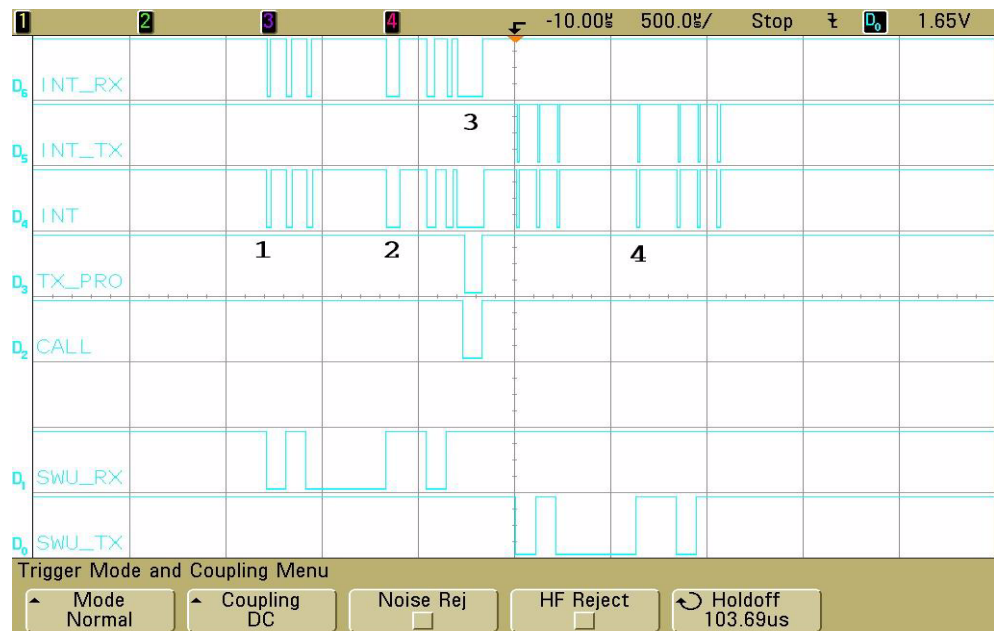


Fig 7. At the end of the message the CPU can fill the Tx FIFO to its maximum (16 bytes) and continue doing other things (the CALL line becomes high)



- (1) Rx line edge processing interrupt (start bit): 26 μ s
- (2) Rx line edge processing interrupt (4 bits): 70 μ s
- (3) Sending received character to output: 140 μ s
- (4) Tx line toggle interrupt (constant): 18 μ s

Fig 8. Receiving a character and sending it to output triggers numerous interrupts of various lengths; software UART interrupts should be assigned the highest priority since any distortion in the waveform can impact actual data content

5.3 Case 2: system clock 12 MHz, transmit FIFO size 128 bytes

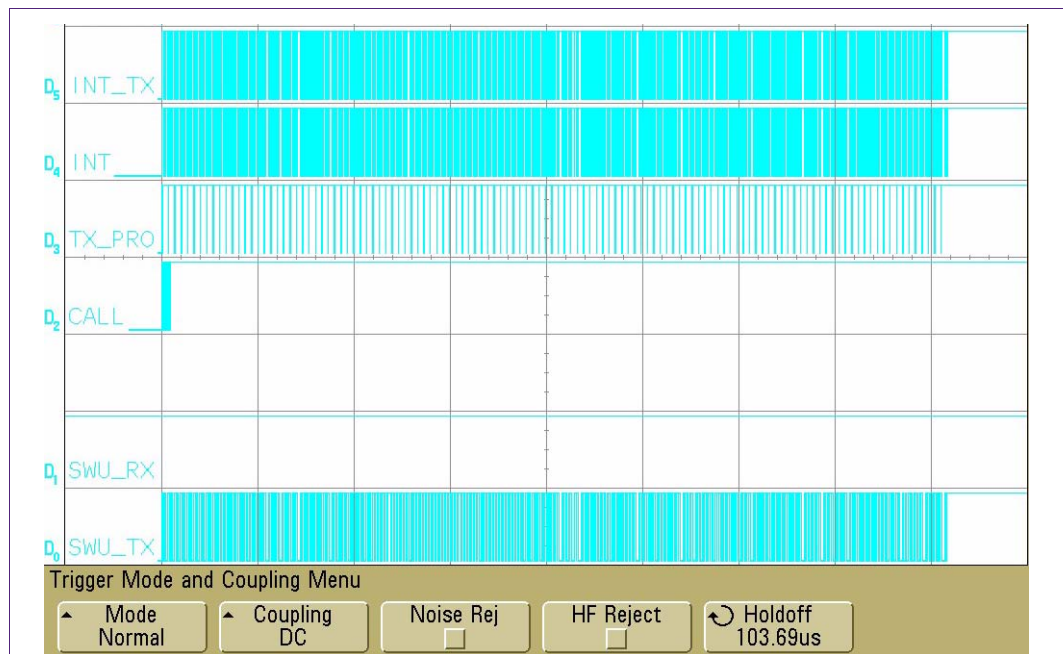


Fig 9. Increasing the Tx FIFO depth from 16 bytes to 128 bytes significantly improves system performance; the CPU spends less the time waiting for the software UART to actually do something

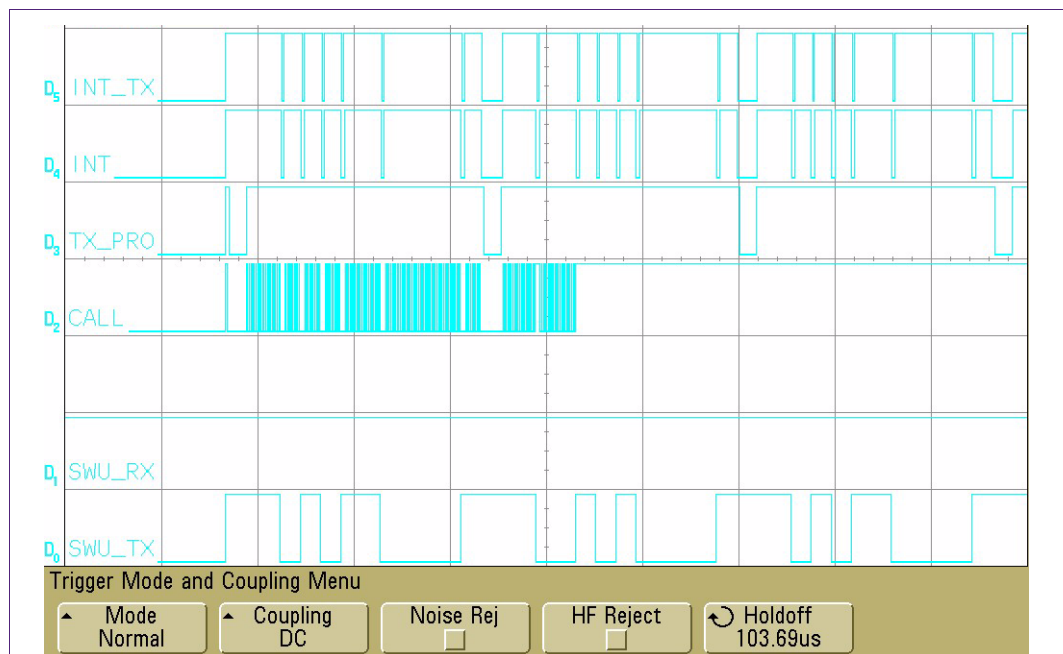
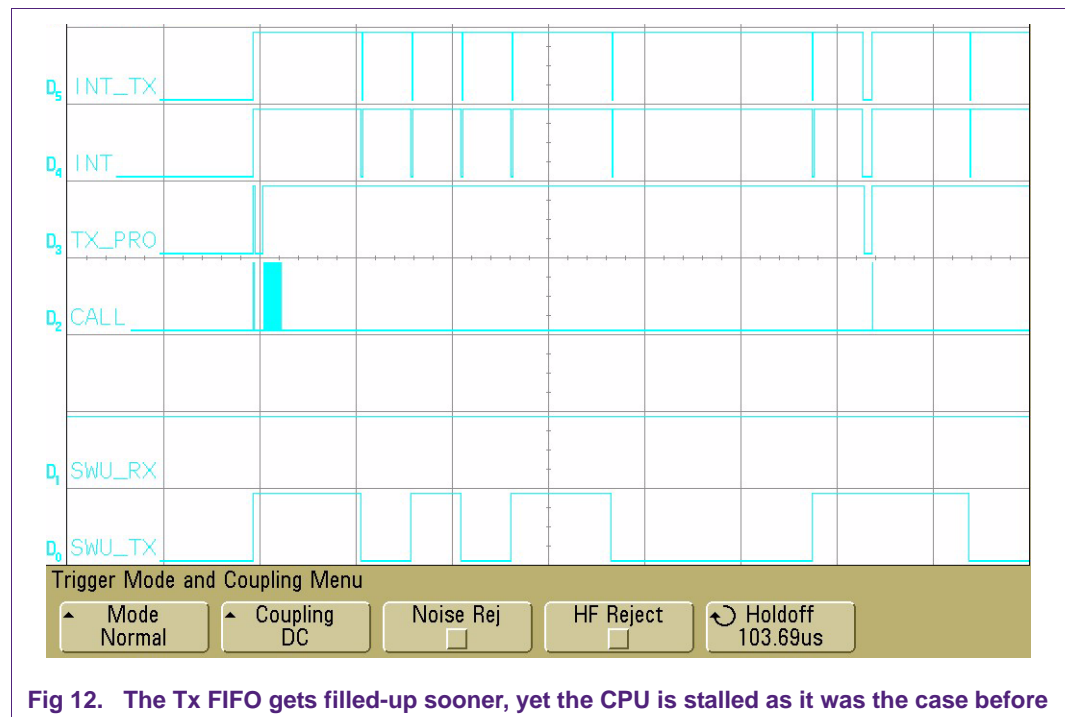
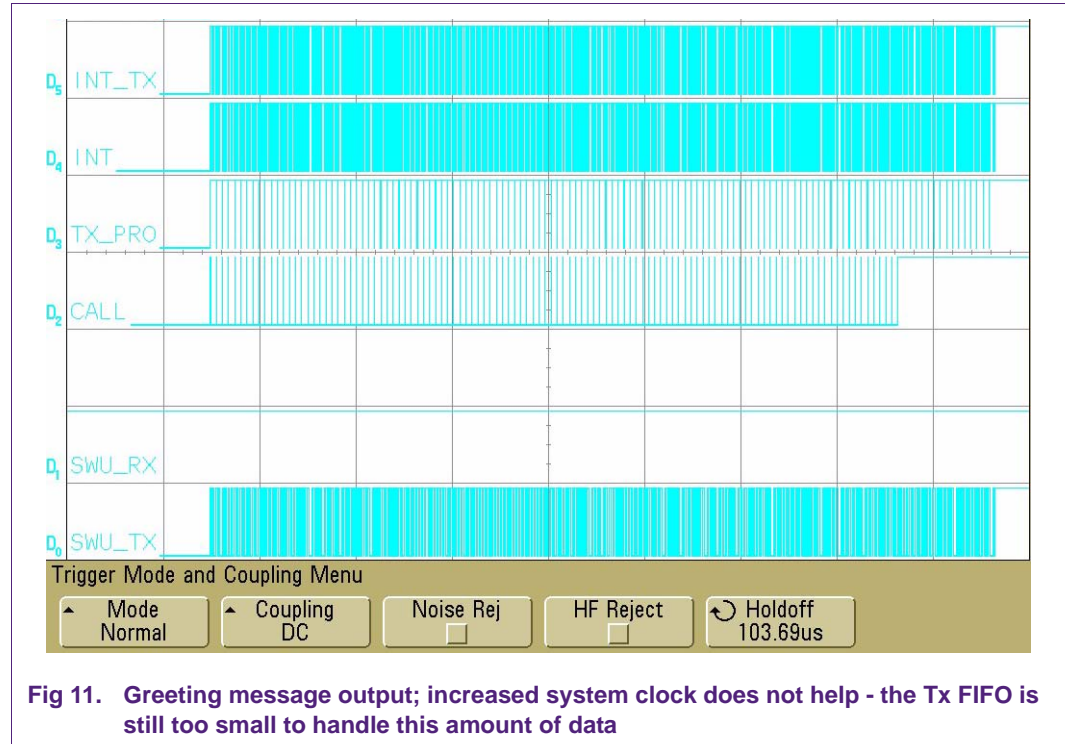
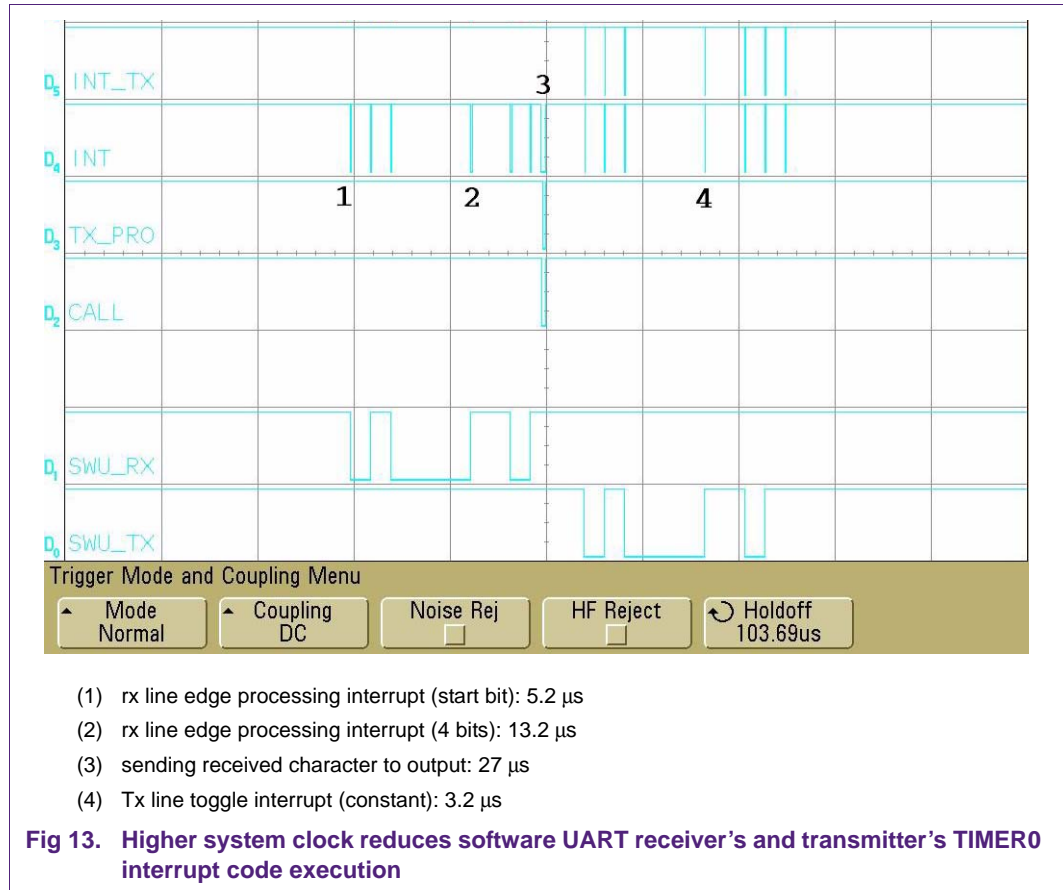


Fig 10. The Tx FIFO is larger than the whole greeting message (128 bytes vs. 122 bytes); Tx FIFO loading is interrupted by the software UART transmitter interrupt; individual interrupt servicing does not depend on the FIFO size

5.4 Case 3: system clock 60 MHz, transmit FIFO size 16 bytes

This system runs with the MAM fully enabled (MAMCR = 2) and MAMTIM = 3. The PLL is configured in the start-up file. A single bit length is BIT_LENGTH = 6250 system clocks.





6. Discussion

This implementation of a communication module tried to use as few LPC2000 peripherals as possible, yet providing a useful full-duplex software UART functionality. Having to combine time-tracking, generation of a controlled output within a tightly specified spec and monitoring an input pin at the same time led us to believe that a general-purpose 32-bit timer could be the right solution.

Real-life system examples were oriented toward interaction between a machine and a human. In such setup response time is usually not critical. Yet, in systems where two machines are communicating this might not be the case.

Two basic parameters have been varied in the real-life examples and impacts of these changes have been tracked on several oscilloscope screenshots. One can easily see that it is the size of the Tx FIFO that made the most significant impact to the overall system performance. In case of a shallow Tx FIFO, the CPU spends a lot of time waiting for the character transfer routine to successfully put one byte into the Tx FIFO before the CPU can do anything else. Until there is a place in this FIFO for more data, the byte transfer routine will sit and wait and wait and wait. At low baudrates this can have a significant impact and effectively render systems' ability to do anything useful while simple text output is performed.

Increasing the system frequency helped as expected in shortening the ISR execution. It is of utmost importance to carefully select the system clock in a system with communication peripherals implemented in software.

The demo code we are running here has no other interrupts active. Yet, in case of the system frequency of only 12 MHz, having an IRQ requesting CPU's attention while the CPU is transferring data from the software UART Rx to the Tx can result in all kinds of unpleasant situations. Interrupt latency and the worst case software UART imposed delays must be thoroughly investigated and analyzed before other interrupt sources are allowed into the embedded system.

The need for more interrupt sources and handling their ISRs could potentially impact the software UART significantly. In a 12 MHz system, a non-software UART ISR occupying the CPU for more than 100 μ s will completely disrupt the software UART Rx functionality. Implemented "stop" bit timeout will kick-in after a while and the software UART is likely to get back on track. However, as a result of this event, at least one invalid character will be sitting in the Rx FIFO and will be picked up by a higher level application later on.

7. AppendixDemo code

```

/*****
/*
/* LPC2148 Software UART (Tx&Rx)
/*
/*
/* NXP Semiconductors
/* 2007/12/20 20:15
/*
/* This software acts as a standard UART with 1 start, 8 data and 1
/* stop bit (no parity) setup. TIMER0 MAT0.3 on P0.29 (pin 14) generates
/* UART output while TIMER0 CAP0.0 on P0.30 (pin 15) receives serial
/* data.
/*
/*
/* While P0.10 is low, the routine that calculates future transmission
/* parameters is active. P0.12 is low while the TIMER0 interrupt service
/* routine (ISR) is running. P0.15 is low while the software UART
/* transmission portion of the TIMER0 ISR is executed. P0.17 indicates
/* when the software UART receiving related TIMER0 ISR is active. P0.18
/* activity matches write access to the tx FIFO.
/*
/* For this demo to run successfully pin P0.00 must be connected to
/* P0.29 and pin P0.01 must be connected to P0.30. A PC or similar
/* equipment capable of handling ASCII data at 9600 bit/s must be
/* connected to the UART0 of the microcontroller.
/*
/* UART0 of the LPC2148 is not used in this demo. Regular U0Tx and U0Rx
/* functionality is replaced by the software uart pins. UART0 pins are
/* selected because most of the development boards already have needed
/* hardware for voltage translation between the microcontroller and an
/* external device.
/*
/* fosc = 12 MHz; PLL is not used; PCLK=CCLK
/*
/* pins used:
/*
/* Pin P0.00 must be connected to P0.29 and P0.01 must be tied to P0.30
/* when running this demo!
/* =====
/*
/* P0.00:      [GPI] - needed because of a standard interface hw
/* P0.01:      [GPI] - needed because of a standard interface hw
/* P0.10: TX_PRO [GPO] - next tx parameter processing (active low)
/* P0.12: INT    [GPO] - TIMER0 interrupt indicator (active low)
/* P0.15: INT_TX [GPO] - swu tx interrupt indicator (active low)
/* P0.17: INT_RX [GPO] - swu rx interrupt indicator (active low)
/* P0.18: CALL   [GPO] - tx UART routine call ind. (active low)
/* P0.29: SWU_Tx [MAT0.3]- software UART Tx pin
/* P0.30: SWU_RX [CAP0.0]- software UART Rx pin
/*
*****/

```

```

#include "LPC214x.h"
#define pin_txpro    (1<<10)
#define pin_int      (1<<12)
#define pin_inttx    (1<<15)
#define pin_intrx    (1<<17)
#define pin_call     (1<<18)
#define pin_swrx     (1<<30)

void T0_Handler(void) __irq;
void IRQ_default(void) __irq;

volatile unsigned char cnt_edges;           //no of char edges
volatile unsigned char edge_index;         //edge index for output
volatile unsigned char swu_tx_st;          //sw UART status
volatile unsigned long int edge[11];       //array of edges
volatile unsigned char last_edge_index, char_end_index; //last two events index

void swu_tx(void);                          //tx param processing

//software UART configurable parameters begin
#define TXBUFF_LEN  16
#define RXBUFF_LEN  16
//12000000/9600 = 1250 PCLKs
//PCLK=12MHz:
#define BIT_LENGTH  1250
//60000000/9600 = 6250 PCLKs
//PCLK=60MHz:
//#define BIT_LENGTH 6250
#define STOP_BIT_SAMPLE (9*BIT_LENGTH)
//software UART configurable parameters end

//definitions common for transmitting and receiveign data begin
volatile unsigned long int swu_status;
#define RX_OVERFLOW 4
#define RX_ACTIVE  2
#define TX_ACTIVE  1
#define ADJUST     (1<<30)
#define ALL1       0x000000FF
//definitions common for transmitting and receiveign data end

//software UART Tx related definitions begin
volatile unsigned long int tx_fifo_wr_ind,tx_fifo_rd_ind;
volatile signed long int swu_tx_cnt, swu_tx_trigger;
volatile unsigned short int swu_tx_fifo[TXBUFF_LEN];
void swu_tx_wr(unsigned char *);
void swu_tx_wr_chr(unsigned char);
//software UART Tx related definitions end

//software UART Rx related definitions begin
volatile unsigned long int rx_fifo_wr_ind,rx_fifo_rd_ind;
volatile signed long int swu_rx_cnt, swu_rx_trigger;

```



```

volatile unsigned char swu_bit, cnt, cnt_bits, swu_rx_chr_fe;
volatile unsigned long int swu_rbr, swu_rbr_mask;
volatile signed long int edge_last, edge_sample, edge_current, edge_stop;
volatile unsigned short int swu_rx_fifo[RXBUFF_LEN];
unsigned char swu_rx_rd_chr(void);
void swu_rx_isr(void);
//software UART Rx related definitions end

int main(void){

    PCONP = 0x00000000;           //turn off all peripherals
    VPBDIV = 0x01;                //PCLK = CCLK

    //setup the software uart
    swu_tx_cnt = 0;                //no data in the swu tx FIFO
    tx_fifo_wr_ind = 0;            //last char written was on 0
    tx_fifo_rd_ind = 0;            //last char updated was on 0
    swu_rx_trigger = 1;            //>=1 char gnrt's a rx interrupt
    swu_status = 0;                //neither tx nor rx active

    //TIMER0 setup
    PCONP |= (1<<1);              //enable TIMER0 clock
    TOTCR = 0x00;                  //stop TIMER0
    TOTCR = 0x02;                  //reset counters
    TOTCR = 0x00;                  //release the reset
    TOIR = 0x0FF;                  //clear all TIMER0 flags
    TOPR = 0x00000000;             //no prescaler
    TOMR0 = 0x3FFFFFFF;            //TIMER0 counts 0 - 0x3FFFFFFF
    TOMCR = 2;                     //reset TIMER0 on MR0
    TOEMR = 0x0008;                //drive MAT0.3 high
    TOTCR = 0x01;                  //let TIMER0 run

    //VIC setup
    VICIntSelect = 0x00000000;     //all interrupts are IRQs
    VICIntEnClr = 0xFFFFFFFF;      //disable all interrupts
    VICVectCntl0 = 0x20 | 4;        //use slot 0 for TIMER0
    VICVectAddr0 = (unsigned long)T0_Handler; //set the TIMER0 handler
    VICDefVectAddr = (unsigned long)IRQ_default; //set the default handler
    VICIntEnable = 1<<4;           //enable TIMER0 interrupts

    //port pin control: P0.29
    PINSEL0 = 0x00000000;           //P0 pins are GPIOs except...
    PINSEL1 = (PINSEL1 & 0xC3FFFFFF) | 0x3C000000; //... P0.30:CAP0.0, P0.29:MAT0.3
    IODIR0 = pin_txpro | pin_int | //set indicator pins to high...
    IOSET0 = pin_txpro | pin_int | //...
              pin_inttx | pin_intrx | pin_call; //...
    IOSET0 = pin_txpro | pin_int | //...
              pin_inttx | pin_intrx | pin_call; //...

    while((IOPIN0&pin_swrx)==0);    //wait for 1 on sw UART rx line
    TOIR = 0x10;                    //clear CAP0.0 flag

```

```

TOCCR    = 0x0006;                                //int on CAP0.0 falling edge
cnt_bits = 0;                                       //reset the rx bit count

//main demo code outputs a greeting message and
//waits for user's input to echo it
swu_tx_wr("\x0D\x0A\x0D\x0A\x0D\x0A\x0D\x0A");
swu_tx_wr("Software UART demo code\x0D\x0A");
swu_tx_wr("=====\x0D\x0A");
swu_tx_wr("This application echoes characters entered by the user.\x0D\x0A");
swu_tx_wr_chr('\x0D');
swu_tx_wr_chr('\x0A');
swu_tx_wr("Echo: ");

while(1);                                          //idle loop
}

//this routine prepares an array off toggle points that will be used to generate
//a waveform for the currently selected character in the software UART transmission
//FIFO; at the end this routine starts the transmission intself
void swu_tx(void){
    unsigned char bit,i;
    unsigned long int ext_data, delta_edges, mask, reference;

    IOCLR0 = pin_txpro;                            //indicate routine begin
    if(tx_fifo_wr_ind!=tx_fifo_rd_ind){             //data to send, proceed
        swu_status |= TX_ACTIVE;                   //sw uart tx is active
        tx_fifo_rd_ind++;                           //update the tx fifo ...
        if(tx_fifo_rd_ind==TXBUFF_LEN)              //read index...
            tx_fifo_rd_ind = 0;                      //...
        ext_data = (unsigned long int) swu_tx_fifo[tx_fifo_rd_ind]; //read the data
        ext_data = 0xFFFFFE00 | ( ext_data<<1);    //prepare the pattern
        edge[0] = BIT_LENGTH;                       //at least 1 falling edge...
        cnt_edges = 1;                               //... because of the START bit
        bit = 1;                                     //set the bit counter
        reference = 0x00000000;                       //init ref is 0 (start bit)
        mask = 1<<1;                                 //prepare the mask
        delta_edges = BIT_LENGTH;                     //next edge at least 1 bit away
        while(bit!=10){                              //until all bits are examined
            if((ext_data&mask)==(reference&mask)){    //bit equal to the reference?
                delta_edges += BIT_LENGTH;            //bits identical=>update length
            }
            else{                                     //bits are not the same:
                edge[cnt_edges] =                     //store new...
                    edge[cnt_edges-1] + delta_edges; //... edge data
                reference = ~reference;               //update the reference
                delta_edges = BIT_LENGTH;             //reset delta_ to 1 bit only
                cnt_edges++;                           //update the edges counter
            }
            mask = mask<<1;                           //update the mask
            bit++;                                     //move on to the next bit
        }
    }
}

```

```

        edge[cnt_edges]=                                //add the stop bit end...
            edge[cnt_edges-1]+delta_edges;                //... to the list
        cnt_edges++;                                     //update the number of edges
        last_edge_index = cnt_edges-2;                   //calculate the last edge index
        char_end_index = cnt_edges-1;                   //calc. the character end index

        edge_index = 0;                                 //reset the edge index
        reference = T0TC + BIT_LENGTH;                  //get the reference from TIMER0
        for(i=0;i!=cnt_edges;i++)                        //recalculate toggle points...
            edge[i] = (edge[i] + reference)              //... an adjust for the...
                & 0x3FFFFFFF;                           //... timer range
        T0MR3 = edge[0];                                 //load MR3
        T0MCR = T0MCR | (1<<9);                         //enable interrupt on MR3 match
        T0EMR = T0EMR | (3<<10);                        //enable toggle on MR3 match
    }
    IOSET0 = pin_txpro;                                  //indicate routine exit
    return;                                              //return from the routine
}

//this is the TIMER0 interrupt service routine
//key software UART transmit and receive code resides here
void T0_Handler(void)__irq
{
    signed long int edge_temp;

    IOCLR0 = pin_int;                                   //indicate TIMER0 isr activity

    //sw uart receive isr code begin
    if((T0IR&0x10)!=0x00){                               //capture interrupt occurred:
        IOCLR0 = pin_intrx;                             //rx interrupt activity begin
        T0IR = 0x10;                                    //edge dtcted=>clear CAP0 flag
        T0CCR = 0x0004 | (0x0003 - (T0CCR&0x0003));    //change the targeted edge
        if((swu_status&RX_ACTIVE)==0){                  //sw UART not active (start):
            edge_last = (signed long int) T0CR0;         //initialize the last edge
            edge_sample = edge_last+(BIT_LENGTH>>1);    //initialize the sample edge
            if(edge_sample<edge_last)                    //adjust the sample edge...
                edge_sample |= ADJUST;                  //... if needed
            swu_bit = 0;                                 //rx bit is 0 (a start bit)
            T0IR = 0x02;                                 //clear MAT1 int flag
            edge_stop = edge_sample+STOP_BIT_SAMPLE;    //estimate the end of the byte
            if (edge_stop<edge_last)                     //adjust the end of byte...
                edge_stop |= ADJUST;                    //... if needed
            T0MR1 = edge_stop;                           //set MR1 (stop bit center)
            T0MCR = T0MCR | (1<<3);                     //int on MR1
            cnt = 9;                                      //initialize the bit counter
            swu_status |= RX_ACTIVE;                     //update the swu status
            swu_rbr = 0x0000;                           //reset the sw rbr
            swu_rbr_mask = 0x0001;                      //initialize the mask
        }
        else{                                           //reception in progress:
            edge_current = (signed long int) T0CR0;      //initialize the current edge

```

```

        if (edge_current<edge_last)           //adjust the current edge...
            edge_current |= ADJUST;           //... if needed
        while(edge_current>edge_sample){       //while sampling edge is within
            if(cnt_bits!=0){
                if(swu_bit!=0)                 //update data...
                    swu_rbr |= swu_rbr_mask;   //...
                swu_rbr_mask = swu_rbr_mask<<1; //update mask
            }
            cnt_bits++;                         //update the bit count
            edge_temp = edge_last + BIT_LENGTH; //estimate the last edge
            if(edge_temp<edge_last)             //adjust...
                edge_last = edge_temp | ADJUST; //... the last edge...
            else                                //... if...
                edge_last = edge_temp;          //... needed
            edge_temp = edge_sample+BIT_LENGTH; //estimate the sample edge
            if(edge_temp<edge_sample)           //adjust...
                edge_sample = edge_temp|ADJUST; //... the sample edge...
            else                                //... if...
                edge_sample = edge_temp;        //... needed
            cnt--;                             //update the no of rcved bits
        }
        swu_bit = 1 - swu_bit;                 //change the received bit
    }
    IOSET0 = pin_intrx;                       //rx interrupt activity end
}
if((T0IR&0x02)!=0x00){                       //stop bit timing matched:
    IOCLR0 = pin_intrx;                      //rx interrupt activity begin
    T0IR    = 0x02;                          //clear MR1 flag
    if(cnt!=0){                               //not all data bits received...
        swu_rbr = swu_rbr<<cnt;              //... make space for the rest...
        if(swu_bit!=0) swu_rbr += ALL1<<(8-cnt); //... add needed 1(s)...
    }
    swu_rbr &= 0x00FF;                        //extract data bits only
    if(swu_bit==0)                             //if the stop bit was 0 =>
        swu_rbr |= 0x00000100;               //... framing error!
    swu_status &= ~RX_ACTIVE;                 //sw UART not active any more
    cnt_bits = 0;                             //reset the rx bit count
    if(swu_rx_cnt!=RXBUFF_LEN){               //store the rcved character...
        swu_rx_cnt++;                         //... into the sw UART...
        rx_fifo_wr_ind++;                     //... rx FIFO
        if(rx_fifo_wr_ind==RXBUFF_LEN) rx_fifo_wr_ind = 0; //...
        swu_rx_fifo[rx_fifo_wr_ind] = swu_rbr; //...
        if(swu_rx_cnt>=swu_rx_trigger) swu_rx_isr(); //rx 'isr' trig excded
    }
    else{
        swu_status |= RX_OVERFLOW;            //rx FIFO full => overflow
    }
    TOMCR    &= ~(7<<3);                     //MR0 impacts TIMER0 no more
    IOSET0 = pin_intrx;                       //rx interrupt activity end
}
//sw uart receive isr code end

```

```

//sw uart transmit isr code begin
if((T0IR&0x08)!=0){
    IOCLR0 = pin_inttx;
    T0IR = 0x08;
    if(edge_index==char_end_index){
        TOMCR &= ~(7<<9);
        swu_tx_cnt--;
        if(tx_fifo_wr_ind!=tx_fifo_rd_ind)
            swu_tx();
        else
            swu_status &= ~TX_ACTIVE;
    }
    else{
        if(edge_index==last_edge_index)
            TOEMR = 0x000003FF;
        edge_index++;
        TOMR3 = edge[edge_index];
    }
    IOSET0 = pin_inttx;
}
//sw uart transmit isr code end

VICVectAddr = 0xFF;
IOSET0 = pin_int;
}

void IRQ_default(void)__irq
{
    VICVectAddr=0xFF;
}

//this routine transfers a string of characters one by one into
//the software UART tx FIFO
void swu_tx_wr(unsigned char * ptr_out){
    while(*ptr_out!=0x00){
        swu_tx_wr_chr(*ptr_out);
        ptr_out++;
    }
    return;
}

//this routine puts a single character into the software UART tx FIFO
void swu_tx_wr_chr(unsigned char out_char){
    IOCLR0 = pin_call;
    while(swu_tx_cnt==TXBUFF_LEN);
    tx_fifo_wr_ind++;
    if(tx_fifo_wr_ind==TXBUFF_LEN)
        tx_fifo_wr_ind = 0;
    swu_tx_fifo[tx_fifo_wr_ind] = out_char;
    swu_tx_cnt++;
    if((swu_status&TX_ACTIVE)==0) swu_tx();
}

```

```
        IOSET0 = pin_call;                //write access to tx FIFO end
        return;                          //return from the routine
    }

//this routine reads a single character from the software UART rx FIFO
//if no new data is available, it returns the last one read; framing error
//indicator is updated, too
unsigned char swu_rx_rd_chr(void){
    if(swu_rx_cnt!=0){                    //update the rx indicator...
        rx_fifo_rd_ind++;                 //... if data are present...
        if(rx_fifo_rd_ind==RXBUFF_LEN) rx_fifo_rd_ind = 0; //...
        swu_rx_cnt--;                     //...
    }
    if((swu_rx_fifo[rx_fifo_rd_ind]&0x0100)==0) //update...
        swu_rx_chr_fe = 0;                 //... the framing error...
    else                                     //... indicator...
        swu_rx_chr_fe = 1;                 //...
    swu_status &= ~RX_OVERFLOW;            //clear the overfloer flag
    return((unsigned char)(swu_rx_fifo[rx_fifo_rd_ind]&0x00FF)); //return data
}

//this code acts as a standard uart rx interrupt routine for the specified
//received count character trigger; this routine is called at the end
//of the received byte that increased overall number of characters in the
//rx FIFO to or beyond the specified trigger
void swu_rx_isr(void){
    swu_tx_wr_chr(swu_rx_rd_chr());        //transmit the last rcvd char
    return;                                //return from the routine
}
```

8. Abbreviations

Table 1. Abbreviations

Acronym	Description
UART	Universal Asynchronous Receiver/Transmitter
ISR	Interrupt Service Routine
Tx	Transmitter
Rx	Receiver

9. Legal information

9.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

9.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are the property of their respective owners.

10. Contents

1 Introduction 3

2 Implementation basics 3

3 Transmission model 3

4 Reception model 5

5 Real-life results 7

5.1 Demo application output 7

5.2 Case 1: system clock 12 MHz, transmit FIFO size
16 bytes 8

5.3 Case 2: system clock 12 MHz, transmit FIFO size
128 bytes 11

5.4 Case 3: system clock 60 MHz, transmit FIFO size
16 bytes 12

6 Discussion 13

7 AppendixDemo code..... 15

8 Abbreviations..... 23

9 Legal information..... 24

9.1 Definitions..... 24

9.2 Disclaimers..... 24

9.3 Trademarks..... 24

10 Contents 25



Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.