```
1 //*****************************************************************************
2 //
3 // usb_dev_serial.c - Main routines for the USB CDC serial example.
4 //*****************************************************************************
5
6 #include <stdbool.h>
7 #include <stdint.h>
8 #include "inc/hw_ints.h"
9 #include "inc/hw_memmap.h"
10 #include "inc/hw_types.h"
11 #include "inc/hw_gpio.h"
12 #include "inc/hw_uart.h"
13 #include "inc/hw_sysctl.h"
14 #include "driverlib/debug.h"
15 #include "driverlib/fpu.h"
16 #include "driverlib/gpio.h"
17 #include "driverlib/pin_map.h"
18 #include "driverlib/interrupt.h"
19 #include "driverlib/sysctl.h"
20 #include "driverlib/systick.h"
21 #include "driverlib/timer.h"
22 #include "driverlib/uart.h"
23 #include "driverlib/usb.h"
24 #include "driverlib/rom.h"
25 #include "usblib/usblib.h"
26 #include "usblib/usbcdc.h"
27 #include "usblib/usb-ids.h"
28 #include "usblib/device/usbdevice.h"
29 #include "usblib/device/usbdcdc.h"
30 #include "utils/ustdlib.h"
31 #include "usb_serial_structs.h"
32 #include "utils/uartstdio.h"
33
34 //*****************************************************************************
35 //
36 //! \addtogroup example_list
37 //! <h1>USB Serial Device (usb_dev_serial)</h1>
38 //!
39 //! This example application turns the evaluation kit into a virtual serial
40 //! port when connected to the USB host system.  The application supports the
41 //! USB Communication Device Class, Abstract Control Model to redirect UART0
42 //! traffic to and from the USB host system.
43 //!
44 //! Assuming you installed TivaWare C Series in the default directory, a
45 //! driver information (INF) file for use with Windows XP, Windows Vista and
46 //! Windows7 can be found in C:/ti/TivaWare-for-C-Series/windows_drivers.  For
47 //! Windows 2000, the required INF file is in
48 //! C:/ti/TivaWare-for-C-Series/windows_drivers/win2K.
49 //
50 //*****************************************************************************
51
52 //*****************************************************************************
53 //
54 // Configuration and tuning parameters.
55 //
56 //*****************************************************************************
57
```

```
58 //*****************************************************************************
59 //
60 // The system tick rate expressed both as ticks per second and a millisecond
61 // period.
62 //
63 //*****************************************************************************
64 #define SYSTICKS_PER_SECOND 100
65 #define SYSTICK_PERIOD_MS (1000 / SYSTICKS_PER_SECOND)
66
67 //*****************************************************************************
68 //
69 // Variables tracking transmit and receive counts.
70 //
71 //*****************************************************************************
72 unsigned char* puc_initialUSBRxAdd = 0;
73 unsigned char* puc_initialUSBTxAdd = 0;
74 volatile unsigned char* puc_lastUSBRxAdd = 0;
75 volatile unsigned char* puc_lastUSBTxAdd = 0;
76 volatile uint32_t g_ui32USBTxCount = 0;
77 volatile uint32_t g_ui32USBRxCount = 0;
78 volatile uint32_t g_ui32USBTxByteCount = 0;
79 volatile uint32_t g_ui32USBRxByteCount = 0;
80 volatile uint32_t g_ui32USBRxSilenceTime = 0;
81 volatile uint8_t g_ui8MBUSBSlaveID = 1;
82 #ifdef DEBUG
83 uint32_t g_ui32UARTRxErrors = 0;
84 #endif
85
86 //*****************************************************************************
87 //
88 // The base address, peripheral ID and interrupt ID of the UART that is to
89 // be redirected.
90 //
91 //*****************************************************************************
92 //
93 // Default line coding settings for the redirected UART.
94 //
95 //*****************************************************************************
96 #define DEFAULT_BIT_RATE        115200
97 #define DEFAULT_UART_CONFIG     (UART_CONFIG_WLEN_8 | UART_CONFIG_PAR_NONE | \
98                                  UART_CONFIG_STOP_ONE)
99
100 //*****************************************************************************
101 //
102 // Global system tick counter
103 //
104 //*****************************************************************************
105 volatile uint32_t g_ui32SysTickCount = 0;
106
107 //*****************************************************************************
108 //
109 // Flags used to pass commands from interrupt context to the main loop.
110 //
111 //*****************************************************************************
112 #define COMMAND_PACKET_RECEIVED 0x00000001
113 #define COMMAND_STATUS_UPDATE   0x00000002
114
```

```
115 volatile uint32_t g_ui32Flags = 0;
116 char *g_pcStatus;
117
118 //*****************************************************************************
119 //
120 // Global flag indicating that a USB configuration has been set.
121 //
122 //*****************************************************************************
123 static volatile bool g_bUSBConfigured = false;
124
125 //*****************************************************************************
126 //
127 // Internal function prototypes.
128 //
129 //*****************************************************************************
130 static void CheckForSerialStateChange(const tUSBDCDCDevice *psDevice,
131                                       int32_t i32Errors);
132 static void SetControlLineState(uint16_t ui16State);
133 static bool SetLineCoding(tLineCoding *psLineCoding);
134 static void GetLineCoding(tLineCoding *psLineCoding);
135 extern unsigned int MB_RTUFrameProcessing(uint8_t *uc_BufferRec, uint8_t *uc_BufferSend,
    uint8_t slaveID);
136
137 //*****************************************************************************
138 //
139 // The error routine that is called if the driver library encounters an error.
140 //
141 //*****************************************************************************
142 #ifdef DEBUG
144 __error__(char *pcFilename, uint32_t ui32Line)
150 #endif
151
152 //*****************************************************************************
153 //
154 // This function is called whenever serial data is received from the UART.
155 // It is passed the accumulated error flags from each character received in
156 // this interrupt and determines from them whether or not an interrupt
157 // notification to the host is required.
158 //
159 // If a notification is required and the control interrupt endpoint is idle,
160 // we send the notification immediately.  If the endpoint is not idle, we
161 // accumulate the errors in a global variable which will be checked on
162 // completion of the previous notification and used to send a second one
163 // if necessary.
164 //
165 //*****************************************************************************
166 static void
167 CheckForSerialStateChange(const tUSBDCDCDevice *psDevice, int32_t i32Errors)
168 {
169     uint16_t ui16SerialState;
170
171     //
172     // Clear our USB serial state.  Since we are faking the handshakes, always
173     // set the TXCARRIER (DSR) and RXCARRIER (DCD) bits.
174     //
175     ui16SerialState = USB_CDC_SERIAL_STATE_TXCARRIER |
176                       USB_CDC_SERIAL_STATE_RXCARRIER;
```

```
177
178     //
179     // Are any error bits set?
180     //
181     if(i32Errors)
182     {
183         //
184         // At least one error is being notified so translate from our hardware
185         // error bits into the correct state markers for the USB notification.
186         //
187         if(i32Errors & UART_DR_OE)
188         {
189             ui16SerialState |= USB_CDC_SERIAL_STATE_OVERRUN;
190         }
191
192         if(i32Errors & UART_DR_PE)
193         {
194             ui16SerialState |= USB_CDC_SERIAL_STATE_PARITY;
195         }
196
197         if(i32Errors & UART_DR_FE)
198         {
199             ui16SerialState |= USB_CDC_SERIAL_STATE_FRAMING;
200         }
201
202         if(i32Errors & UART_DR_BE)
203         {
204             ui16SerialState |= USB_CDC_SERIAL_STATE_BREAK;
205         }
206
207         // Call the CDC driver to notify the state change.
208         USBDCDCSerialStateChange((void *)psDevice, ui16SerialState);
209     }
210 }
211
212 //*****************************************************************************
213 //
214 // Interrupt handler for the system tick counter.
215 //
216 //*****************************************************************************
217 unsigned int responseLength;
218 void
219 SysTickIntHandler(void)
220 {
221     //
222     // Update our system time.
223     //
224     g_ui32SysTickCount++;
225     ++g_ui32USBRxSilenceTime;
226 }
227
228 //*****************************************************************************
229 //
230 // Set the state of the RS232 RTS and DTR signals.
231 //
232 //*****************************************************************************
233 static void
```

```
234 SetControlLineState(uint16_t ui16State)
235 {
236     //
237     // TODO: If configured with GPIOs controlling the handshake lines,
238     // set them appropriately depending upon the flags passed in the wValue
239     // field of the request structure passed.
240     //
241 }
242
243 //*****************************************************************************
244 //
245 // Set the communication parameters to use on the UART.
246 //
247 //*****************************************************************************
248 static bool
249 SetLineCoding(tLineCoding *psLineCoding)
250 {
251     uint32_t ui32Config;
252     bool bRetcode;
253
254     //
255     // Assume everything is OK until we detect any problem.
256     //
257     bRetcode = true;
258
259     //
260     // Word length.  For invalid values, the default is to set 8 bits per
261     // character and return an error.
262     //
263     switch(psLineCoding->ui8Databits)
264     {
265         case 5:
266         {
267             ui32Config = UART_CONFIG_WLEN_5;
268             break;
269         }
270
271         case 6:
272         {
273             ui32Config = UART_CONFIG_WLEN_6;
274             break;
275         }
276
277         case 7:
278         {
279             ui32Config = UART_CONFIG_WLEN_7;
280             break;
281         }
282
283         case 8:
284         {
285             ui32Config = UART_CONFIG_WLEN_8;
286             break;
287         }
288
289         default:
290         {
```

```
291             ui32Config = UART_CONFIG_WLEN_8;
292             bRetcode = false;
293             break;
294         }
295     }
296
297     //
298     // Parity.  For any invalid values, we set no parity and return an error.
299     //
300     switch(psLineCoding->ui8Parity)
301     {
302         case USB_CDC_PARITY_NONE:
303         {
304             ui32Config |= UART_CONFIG_PAR_NONE;
305             break;
306         }
307
308         case USB_CDC_PARITY_ODD:
309         {
310             ui32Config |= UART_CONFIG_PAR_ODD;
311             break;
312         }
313
314         case USB_CDC_PARITY_EVEN:
315         {
316             ui32Config |= UART_CONFIG_PAR_EVEN;
317             break;
318         }
319
320         case USB_CDC_PARITY_MARK:
321         {
322             ui32Config |= UART_CONFIG_PAR_ONE;
323             break;
324         }
325
326         case USB_CDC_PARITY_SPACE:
327         {
328             ui32Config |= UART_CONFIG_PAR_ZERO;
329             break;
330         }
331
332         default:
333         {
334             ui32Config |= UART_CONFIG_PAR_NONE;
335             bRetcode = false;
336             break;
337         }
338     }
339
340     //
341     // Stop bits.  Our hardware only supports 1 or 2 stop bits whereas CDC
342     // allows the host to select 1.5 stop bits.  If passed 1.5 (or any other
343     // invalid or unsupported value of ui8Stop, we set up for 1 stop bit but
344     // return an error in case the caller needs to Stall or otherwise report
345     // this back to the host.
346     //
347     switch(psLineCoding->ui8Stop)
```

```
348     {
349         //
350         // One stop bit requested.
351         //
352         case USB_CDC_STOP_BITS_1:
353         {
354             ui32Config |= UART_CONFIG_STOP_ONE;
355             break;
356         }
357 
358         //
359         // Two stop bits requested.
360         //
361         case USB_CDC_STOP_BITS_2:
362         {
363             ui32Config |= UART_CONFIG_STOP_TWO;
364             break;
365         }
366 
367         //
368         // Other cases are either invalid values of ui8Stop or values that we
369         // cannot support so set 1 stop bit but return an error.
370         //
371         default:
372         {
373             ui32Config |= UART_CONFIG_STOP_ONE;
374             bRetcode = false;
375             break;
376         }
377     }
378 
379     //
380     // Let the caller know if we had a problem or not.
381     //
382     return(bRetcode);
383 }
384 
385 //*****************************************************************************
386 //
387 // Get the communication parameters in use on the UART.
388 //
389 //*****************************************************************************
390 static void
391 GetLineCoding(tLineCoding *psLineCoding)
392 {
393     uint32_t ui32Config;
394     uint32_t ui32Rate;
395 
396     //
397     // Get the current line coding set in the UART.
398     //
399     psLineCoding->ui32Rate = ui32Rate;
400 
401     //
402     // Translate the configuration word length field into the format expected
403     // by the host.
404     //
```

```
405     switch(ui32Config & UART_CONFIG_WLEN_MASK)
406     {
407         case UART_CONFIG_WLEN_8:
408         {
409             psLineCoding->ui8Databits = 8;
410             break;
411         }
412
413         case UART_CONFIG_WLEN_7:
414         {
415             psLineCoding->ui8Databits = 7;
416             break;
417         }
418
419         case UART_CONFIG_WLEN_6:
420         {
421             psLineCoding->ui8Databits = 6;
422             break;
423         }
424
425         case UART_CONFIG_WLEN_5:
426         {
427             psLineCoding->ui8Databits = 5;
428             break;
429         }
430     }
431
432     //
433     // Translate the configuration parity field into the format expected
434     // by the host.
435     //
436     switch(ui32Config & UART_CONFIG_PAR_MASK)
437     {
438         case UART_CONFIG_PAR_NONE:
439         {
440             psLineCoding->ui8Parity = USB_CDC_PARITY_NONE;
441             break;
442         }
443
444         case UART_CONFIG_PAR_ODD:
445         {
446             psLineCoding->ui8Parity = USB_CDC_PARITY_ODD;
447             break;
448         }
449
450         case UART_CONFIG_PAR_EVEN:
451         {
452             psLineCoding->ui8Parity = USB_CDC_PARITY_EVEN;
453             break;
454         }
455
456         case UART_CONFIG_PAR_ONE:
457         {
458             psLineCoding->ui8Parity = USB_CDC_PARITY_MARK;
459             break;
460         }
461
```

```
462         case UART_CONFIG_PAR_ZERO:
463         {
464             psLineCoding->ui8Parity = USB_CDC_PARITY_SPACE;
465             break;
466         }
467     }
468
469     //
470     // Translate the configuration stop bits field into the format expected
471     // by the host.
472     //
473     switch(ui32Config & UART_CONFIG_STOP_MASK)
474     {
475         case UART_CONFIG_STOP_ONE:
476         {
477             psLineCoding->ui8Stop = USB_CDC_STOP_BITS_1;
478             break;
479         }
480
481         case UART_CONFIG_STOP_TWO:
482         {
483             psLineCoding->ui8Stop = USB_CDC_STOP_BITS_2;
484             break;
485         }
486     }
487 }
488
489 //*****************************************************************************
490 //
491 // Handles CDC driver notifications related to control and setup of the device.
492 //
493 // \param pvCBData is the client-supplied callback pointer for this channel.
494 // \param ui32Event identifies the event we are being notified about.
495 // \param ui32MsgValue is an event-specific value.
496 // \param pvMsgData is an event-specific pointer.
497 //
498 // This function is called by the CDC driver to perform control-related
499 // operations on behalf of the USB host.  These functions include setting
500 // and querying the serial communication parameters, setting handshake line
501 // states and sending break conditions.
502 //
503 // \return The return value is event-specific.
504 //
505 //*****************************************************************************
506 uint32_t
507 ControlHandler(void *pvCBData, uint32_t ui32Event,
508                uint32_t ui32MsgValue, void *pvMsgData)
509 {
510     uint32_t ui32IntsOff;
511
512     //
513     // Which event are we being asked to process?
514     //
515     switch(ui32Event)
516     {
517         //
518         // We are connected to a host and communication is now possible.
```

```
519         //
520         case USB_EVENT_CONNECTED:
521             g_bUSBConfigured = true;
522 
523             //
524             // Flush our buffers.
525             //
526             USBBufferFlush(&g_sTxBuffer);
527             USBBufferFlush(&g_sRxBuffer);
528 
529             //
530             // Tell the main loop to update the display.
531             //
532             ui32IntsOff = ROM_IntMasterDisable();
533             g_pcStatus = "Connected";
534             g_ui32Flags |= COMMAND_STATUS_UPDATE;
535             if(!ui32IntsOff)
536             {
537                 ROM_IntMasterEnable();
538             }
539             break;
540 
541         //
542         // The host has disconnected.
543         //
544         case USB_EVENT_DISCONNECTED:
545             g_bUSBConfigured = false;
546             ui32IntsOff = ROM_IntMasterDisable();
547             g_pcStatus = "Disconnected";
548             g_ui32Flags |= COMMAND_STATUS_UPDATE;
549             if(!ui32IntsOff)
550             {
551                 ROM_IntMasterEnable();
552             }
553             break;
554 
555         //
556         // Return the current serial communication parameters.
557         //
558         case USBD_CDC_EVENT_GET_LINE_CODING:
559             GetLineCoding(pvMsgData);
560             break;
561 
562         //
563         // Set the current serial communication parameters.
564         //
565         case USBD_CDC_EVENT_SET_LINE_CODING:
566             SetLineCoding(pvMsgData);
567             break;
568 
569         //
570         // Set the current serial communication parameters.
571         //
572         case USBD_CDC_EVENT_SET_CONTROL_LINE_STATE:
573             SetControlLineState((uint16_t)ui32MsgValue);
574             break;
575 
```

```
576         //
577         // Send a break condition on the serial line.
578         //
579         case USBD_CDC_EVENT_SEND_BREAK:
580             //SendBreak(true);
581             break;
582
583         //
584         // Clear the break condition on the serial line.
585         //
586         case USBD_CDC_EVENT_CLEAR_BREAK:
587             //SendBreak(false);
588             break;
589
590         //
591         // Ignore SUSPEND and RESUME for now.
592         //
593         case USB_EVENT_SUSPEND:
594         case USB_EVENT_RESUME:
595             break;
596
597         //
598         // We don't expect to receive any other events.  Ignore any that show
599         // up in a release build or hang in a debug build.
600         //
601         default:
602 #ifdef DEBUG
603             while(1);
604 #else
605             break;
606 #endif
607
608     }
609
610     return(0);
611 }
612
613 //*****************************************************************************
614 //
615 // Handles CDC driver notifications related to the transmit channel (data to
616 // the USB host).
617 //
618 // \param ui32CBData is the client-supplied callback pointer for this channel.
619 // \param ui32Event identifies the event we are being notified about.
620 // \param ui32MsgValue is an event-specific value.
621 // \param pvMsgData is an event-specific pointer.
622 //
623 // This function is called by the CDC driver to notify us of any events
624 // related to operation of the transmit data channel (the IN channel carrying
625 // data to the USB host).
626 //
627 // \return The return value is event-specific.
628 //
629 //*****************************************************************************
630 uint32_t
631 TxHandler(void *pvCBData, uint32_t ui32Event, uint32_t ui32MsgValue,
632           void *pvMsgData)
```

```
633 {
634     //
635     // Which event have we been sent?
636     //
637     switch(ui32Event)
638     {
639         case USB_EVENT_TX_COMPLETE:
640             //
641             // Since we are using the USBBuffer, we don't need to do anything
642             // here.
643             //
644             break;
645 
646         //
647         // We don't expect to receive any other events.  Ignore any that show
648         // up in a release build or hang in a debug build.
649         //
650         default:
651 #ifdef DEBUG
652             while(1);
653 #else
654             break;
655 #endif
656 
657     }
658     return(0);
659 }
660 
661 //*****************************************************************************
662 //
663 // Handles CDC driver notifications related to the receive channel (data from
664 // the USB host).
665 //
666 // \param ui32CBData is the client-supplied callback data value for this channel.
667 // \param ui32Event identifies the event we are being notified about.
668 // \param ui32MsgValue is an event-specific value.
669 // \param pvMsgData is an event-specific pointer.
670 //
671 // This function is called by the CDC driver to notify us of any events
672 // related to operation of the receive data channel (the OUT channel carrying
673 // data from the USB host).
674 //
675 // \return The return value is event-specific.
676 //
677 //*****************************************************************************
678 uint32_t
679 RxHandler(void *pvCBData, uint32_t ui32Event, uint32_t ui32MsgValue,
680           void *pvMsgData)
681 {
682     uint32_t ui32Read;
683     uint8_t ui8Char;
684 
685     //
686     // Which event are we being sent?
687     //
688     switch (ui32Event)
689     {
```

```
690     //
691     // A new packet has been received.
692     //
693     case USB_EVENT_RX_AVAILABLE:
694     {
695         //
696         // Feed some characters into the UART TX FIFO and enable the
697         // interrupt so we are told when there is more space.
698         //
699         ui32Read = USBBufferRead((tUSBBuffer *) &g_sRxBuffer, &ui8Char, 1);
700         *puc_lastUSBRxAdd = ui8Char;
701         g_ui32USBRxSilenceTime = 0;
702         ++g_ui32USBRxByteCount;
703         break;
704     }
705
706         //
707         // We are being asked how much unprocessed data we have still to
708         // process. We return 0 if the UART is currently idle or 1 if it is
709         // in the process of transmitting something. The actual number of
710         // bytes in the UART FIFO is not important here, merely whether or
711         // not everything previously sent to us has been transmitted.
712         //
713     case USB_EVENT_DATA_REMAINING:
714     {
715
716         return (0);
717     }
718
719         //
720         // We are being asked to provide a buffer into which the next packet
721         // can be read. We do not support this mode of receiving data so let
722         // the driver know by returning 0. The CDC driver should not be sending
723         // this message but this is included just for illustration and
724         // completeness.
725         //
726     case USB_EVENT_REQUEST_BUFFER:
727     {
728         return (0);
729     }
730
731         //
732         // We don't expect to receive any other events.  Ignore any that show
733         // up in a release build or hang in a debug build.
734         //
735     default:
736 #ifdef DEBUG
737         while(1);
738 #else
739         break;
740 #endif
741     }
742
743     return (0);
744 }
745
746 //*****************************************************************************
```

```
747 //
748 // This is the main application entry function.
749 //
750 //*****************************************************************************
751 int
752 main(void)
753 {
754     uint32_t ui32TxCount;
755     uint32_t ui32RxCount;
756 
757     //
758     // Enable lazy stacking for interrupt handlers.  This allows floating-point
759     // instructions to be used within interrupt handlers, but at the expense of
760     // extra stack usage.
761     //
762     ROM_FPULazyStackingEnable();
763 
764     //
765     // Set the clocking to run from the PLL at 50MHz
766     //
767     ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
768                        SYSCTL_XTAL_16MHZ);
769 
770     //
771     // Configure the required pins for USB operation.
772     //
773     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
774     ROM_GPIOPinTypeUSBAnalog(GPIO_PORTD_BASE, GPIO_PIN_5 | GPIO_PIN_4);
775 
776     //
777     // Enable the GPIO port that is used for the on-board LED.
778     //
779     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
780 
781     //
782     // Enable the GPIO pins for the LED (PF2 & PF3).
783     //
784     ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_3|GPIO_PIN_2);
785 
786     //
787     // Not configured initially.
788     //
789     g_bUSBConfigured = false;
790     //
791     // Enable the system tick.
792     //
793     ROM_SysTickPeriodSet(ROM_SysCtlClockGet() / 100);
794     ROM_SysTickIntEnable();
795     ROM_SysTickEnable();
796 
797     //
798     // Initialize the transmit and receive buffers.
799     //
800     USBBufferInit(&g_sTxBuffer);
801     USBBufferInit(&g_sRxBuffer);
802 
803     puc_initialUSBRxAdd = g_sRxBuffer.pui8Buffer;
```

```
804     puc_initialUSBTxAdd = g_sTxBuffer.pui8Buffer;
805
806     //
807     // Set the USB stack mode to Device mode with VBUS monitoring.
808     //
809     USBStackModeSet(0, eUSBModeForceDevice, 0);
810
811     //
812     // Pass our device information to the USB library and place the device
813     // on the bus.
814     //
815     USBDCDCInit(0, &g_sCDCDevice);
816
817     //
818     // Clear our local byte counters.
819     //
820     ui32RxCount = 0;
821     ui32TxCount = 0;
822
823     //
824     // Main application loop.
825     //
826     while(1)
827     {
828         //
829         // Have we been asked to update the status display?
830         //
831         if(g_ui32Flags & COMMAND_STATUS_UPDATE)
832         {
833             //
834             // Clear the command flag
835             //
836             ROM_IntMasterDisable();
837             g_ui32Flags &= ~COMMAND_STATUS_UPDATE;
838             ROM_IntMasterEnable();
839         }
840
841         //
842         // Has there been any transmit traffic since we last checked?
843         //
844         if(ui32TxCount != g_ui32USBTxByteCount)
845         {
846             //
847             // Turn on the Green LED.
848             //
849             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);
850
851             //
852             // Delay for a bit.
853             //
854             SysCtlDelay(ROM_SysCtlClockGet() / 3 / 20);
855
856             //
857             // Turn off the Green LED.
858             //
859             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0);
860
```

```
861             //
862             // Take a snapshot of the latest transmit count.
863             //
864             ui32TxCount = g_ui32USBTxByteCount;
865         }
866 
867         //
868         // Has there been any receive traffic since we last checked?
869         //
870         if(ui32RxCount != g_ui32USBRxByteCount)
871         {
872             //
873             // Turn on the Blue LED.
874             //
875             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
876 
877             //
878             // Delay for a bit.
879             //
880             SysCtlDelay(ROM_SysCtlClockGet() / 3 / 20);
881 
882             //
883             // Turn off the Blue LED.
884             //
885             GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
886 
887             //
888             // Take a snapshot of the latest receive count.
889             //
890             ui32RxCount = g_ui32USBRxByteCount;
891 
892         }
893 
894 
895         if(g_ui32USBRxSilenceTime >= 100){
896             g_ui32USBRxSilenceTime = 0;
897             responseLength = 0;
898             responseLength = MB_RTUFrameProcessing(puc_initialUSBRxAdd, puc_initialUSBTxAdd,
    g_ui8MBUSBSlaveID);
899 
900             if(responseLength <= 256){
901                 //uint8_t ui8Char;
902                 uint8_t i;
903                 for(i = 0; i < responseLength; i++){
904                     USBBufferWrite((tUSBBuffer *)&g_sTxBuffer, (puc_initialUSBTxAdd + i), 1);
905                 }
906                 USBBufferFlush(&g_sTxBuffer);
907             }
908 
909             USBBufferInit(&g_sTxBuffer);
910             USBBufferInit(&g_sRxBuffer);
911 
912             puc_initialUSBRxAdd = g_sRxBuffer.pui8Buffer;
913             puc_initialUSBTxAdd = g_sTxBuffer.pui8Buffer;
914     }
915     }
916 }
```

917