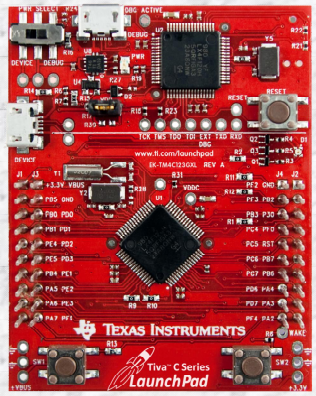
Introduction

This chapter will introduce you to the capabilities of the Universal Asynchronous Receiver/Transmitter (UART). The lab uses the LaunchPad board and the Stellaris Virtual Serial Port running over the debug USB port.

Agenda

Introduction to ARM® Cortex™-M4F and Peripherals
Code Composer Studio
Introduction to TivaWare™, Initialization and GPIO
Interrupts and the Timers
ADC12
Hibernation Module
USB
Memory and Security
Floating-Point
BoosterPacks and grLib
Synchronous Serial Interface
UART
µDMA
Sensor Hub
PWM



Features...

Chapter Topics

UART 12-1

UART Features and Block Diagram 12-3

Basic Operation 12-4

UART Interrupts and FIFOs 12-5

UART "stdio" Functions and Other Features 12-6

Lab 12 12-7

Objective 12-7

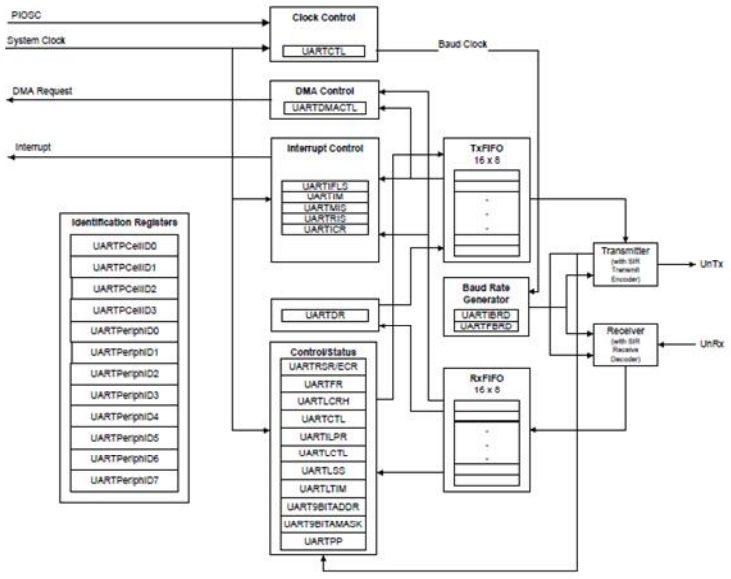
UART Features and Block Diagram

UART Features

- Separate 16x8 bit transmit and receive FIFOs
- Programmable baud rate generator
- Auto generation and stripping of start, stop, and parity bits
- Line break generation and detection
- Programmable serial interface
  - 5, 6, 7, or 8 data bits
  - even, odd, stick, or no parity bits
  - 1 or 2 stop bits
  - baud rate generation, from DC to processor clock/16
- Modem flow control on UART1 (RTS/CTS)
- IrDA and EIA-495 9-bit protocols
- µDMA support

Block Diagram...

Block Diagram



Basic Operation...

Basic Operation

Basic Operation

- Initialize the UART
  - Enable the UART peripheral, e.g.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

  - Set the Rx/Tx pins as UART pins

```
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```

  - Configure the UART baud rate, data configuration

```
ROM_UARTConfigSetExpClk(UART0_BASE, ROM_SysCtlClockGet(), 115200,
                        UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                        UART_CONFIG_PAR_NONE));
```

  - Configure other UART features (e.g. interrupts, FIFO)
- Send/receive a character
  - Single register used for transmit/receive
  - Blocking/non-blocking functions in driverlib:

```
UARTCharPut(UART0_BASE, 'a');
newchar = UARTCharGet(UART0_BASE);
UARTCharPutNonBlocking(UART0_BASE, 'a');
newchar = UARTCharGetNonBlocking(UART0_BASE);
```

Interrupts...

UART Interrupts and FIFOs

UART Interrupts

Single interrupt per module, cleared automatically

Interrupt conditions:

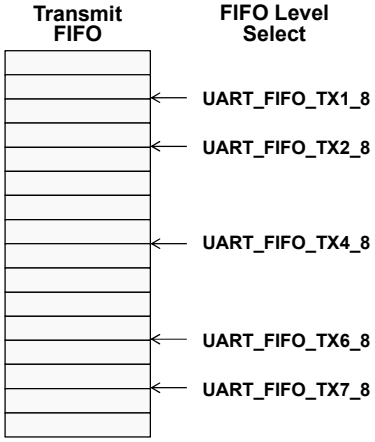
- Overrun error
- Break error
- Parity error
- Framing error
- Receive timeout – when FIFO is not empty and no further data is received over a 32-bit period
- Transmit – generated when no data present (if FIFO enabled, see next slide)
- Receive – generated when character is received (if FIFO enabled, see next slide)

Interrupts on these conditions can be enabled individually

Your handler code must check to determine the source of the UART interrupt and clear the flag(s)

FIFOs...

Using the UART FIFOs



- **Both FIFOs are accessed via the UART Data register (UARTDR)**
- **After reset, the FIFOs are enabled*, you can disable by resetting the FEN bit in UARTLCRH, e.g.**

```
UARTFIFODisable(UART0_BASE);
```

- **Trigger points for FIFO interrupts can be set at 1/8, 1/4, 1/2,3/4, 7/8 full, e.g.**

```
UARTFIFOLevelSet(UART0_BASE,
  UART_FIFO_TX4_8,
  UART_FIFO_RX4_8);
```

*** Note: the datasheet says FIFOs are disabled at reset**

stdio Functions...

UART “stdio” Functions and Other Features

UART “stdio” Functions

- **TivaWare “utils” folder contains functions for C stdio console functions:**

```
c:\TivaWare\utils\uartstdio.h
c:\TivaWare\utils\uartstdio.c
```

- **Usage example:**

```
UARTStdioInit(0); //use UART0, 115200
UARTprintf("Enter text: ");
```

- **See `uartstdio.h` for other functions**
- **Notes:**
  - **Use the provided interrupt handler `UARTStdioIntHandler()` code in `uartstdio.c`**
  - **Buffering is provided if you define UART_BUFFERED symbol**
    - Receive buffer is 128 bytes
    - Transmit buffer is 1024 bytes

Other UART Features...

Other UART Features

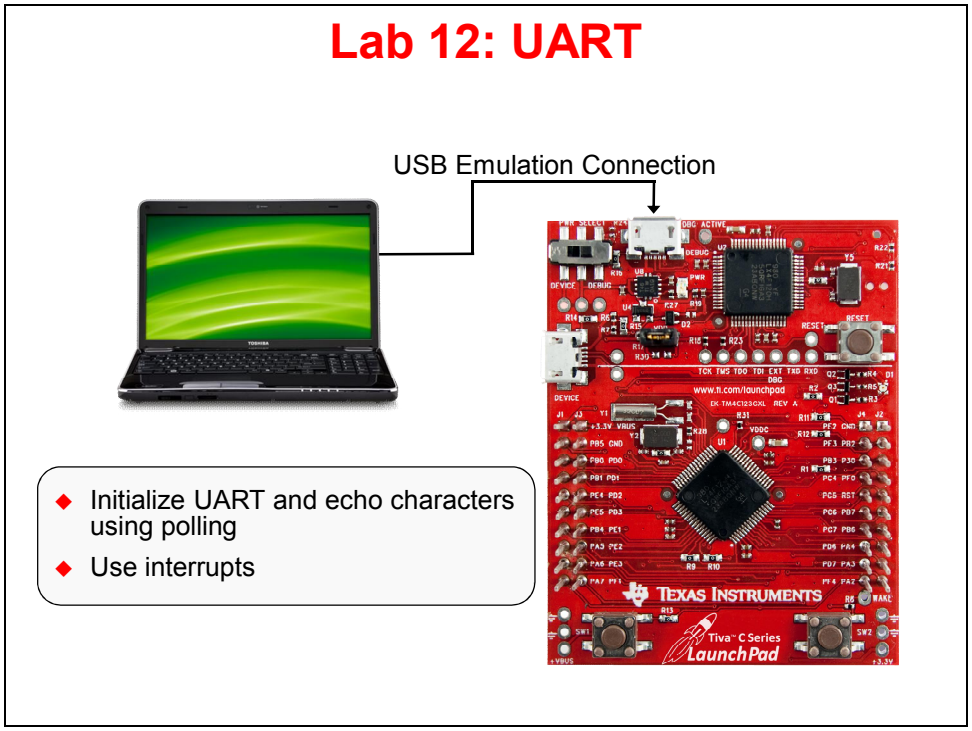
- **Modem flow control on UART1 (RTS/CTS)**
- **IrDA serial IR (SIR) encoder/decoder**
  - External infrared transceiver required
  - Supports half-duplex serial SIR interface
  - Minimum of 10-ms delay required between transmit/receive, provided by software
- **ISA 7816 smartcard support**
  - UnTX signal used as a bit clock
  - UnRx signal is half-duplex communication line
  - GPIO pin used for smartcard reset, other signals provided by your system design
- **LIN (Local Interconnect Network) support: master or slave**
- **µDMA support**
  - Single or burst transfers support
  - UART interrupt handler handles DMA completion interrupt
- **EIA-495 9-bit operation**
  - Multi-drop configuration: one master, multiple slaves
  - Provides “address” bit (in place of parity bit)
  - Slaves only respond to their address

Lab...

Lab 12

Objective

In this lab you will send data through the UART. The UART is connected to the emulator's virtual serial port that runs over the debug USB cable.



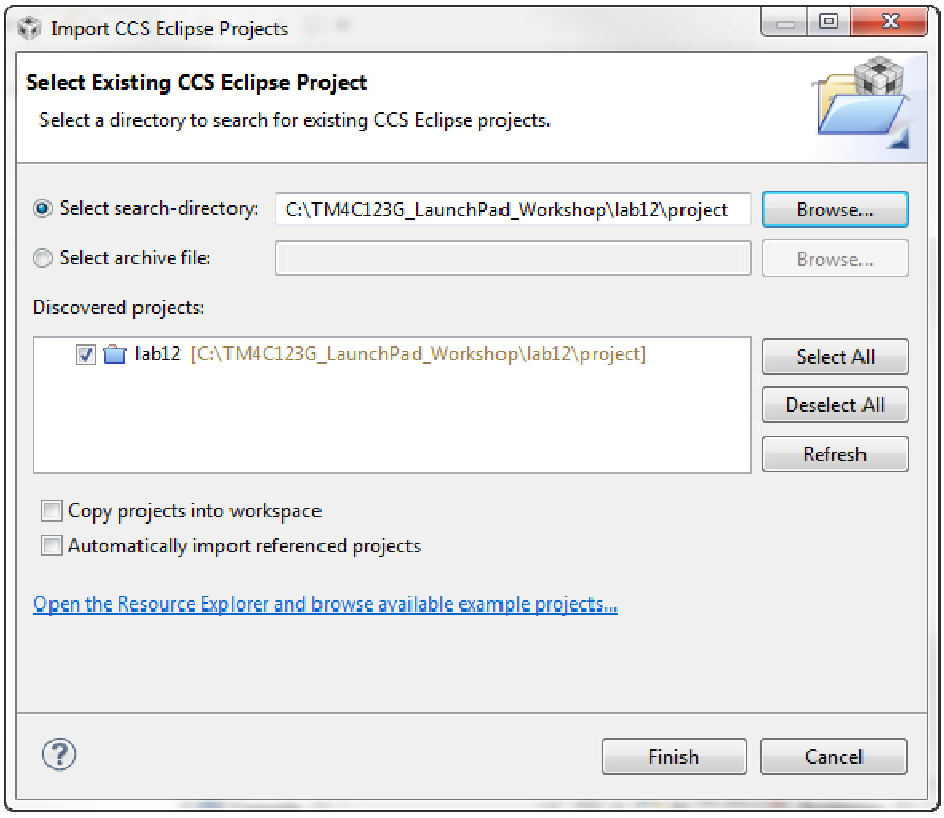
Procedure

Import Lab12

1. We have already created the lab12 project for you with a **main.c** file, a startup file, and all the necessary project and build options set.

   ► Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish.

   Make sure that the "Copy projects into workspace" checkbox is **unchecked**.



2. ► Expand the project by clicking on the + or ▷ next to lab12 in the Project Explorer pane. Double-click on **main.c** to open it for review. The code looks like the next page:

```
#include<stdint.h>
#include<stdbool.h>
#include"inc/hw_memmap.h"
#include"inc/hw_types.h"
#include"driverlib/gpio.h"
#include"driverlib/pin_map.h"
#include"driverlib/sysctl.h"
#include"driverlib/uart.h"

intmain(void) {

SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

UARTCharPut(UART0_BASE, 'E');
UARTCharPut(UART0_BASE, 'n');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'r');
UARTCharPut(UART0_BASE, ' ');
UARTCharPut(UART0_BASE, 'T');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'x');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, ':');
UARTCharPut(UART0_BASE, ' ');

while (1)
    {
        if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
    }

}
```

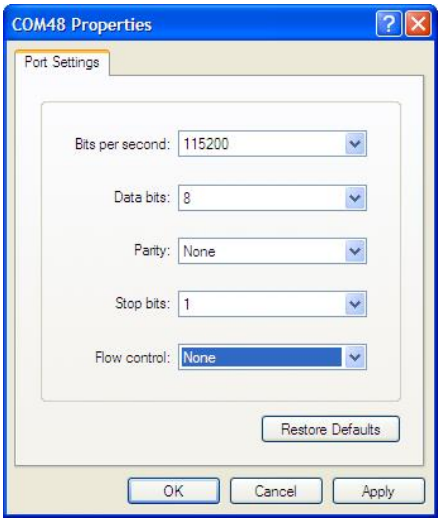
3. In **main()**, notice the initialization sequence for using the UART:
   - Set up the system clock
   - Enable the UART0 and GPIOA peripherals (the UART pins are on GPIO Port A)
   - Configure the pins for the receiver and transmitter using GPIOPinConfigure
   - Initialize the parameters for the UART: 115200, 8-1-N
   - Use simple "**UARTCharPut()**" calls to create a prompt.
   - An infinite loop. In this loop, if there is a character in the receiver, it is read, and then written to the transmitter. This echos what you type in the terminal window.

Build, Download, and Run the UART Example Code

4. ► Click the Debug button to build and download your program to the TM4C123GH6PM flash memory.

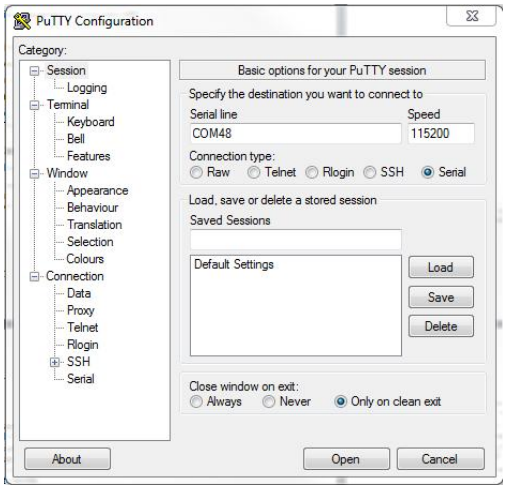
   We can communicate with the board through the UART, which is connected as a virtual serial port through the emulator USB connection. You can find the COM port number for this serial port back in chapter one of this workbook on page 18 or 19.

   In WinXP, ► open HyperTerminal by clicking Start → Run…, then type hypertrm in the Open: box and click OK. Pick any name you like for your connection and click OK. In the next dialog box, change the Connect using: selection to COM##, where ## is the COM port number you noted earlier from Device Manager. Click OK. Make the selections shown below and click OK.

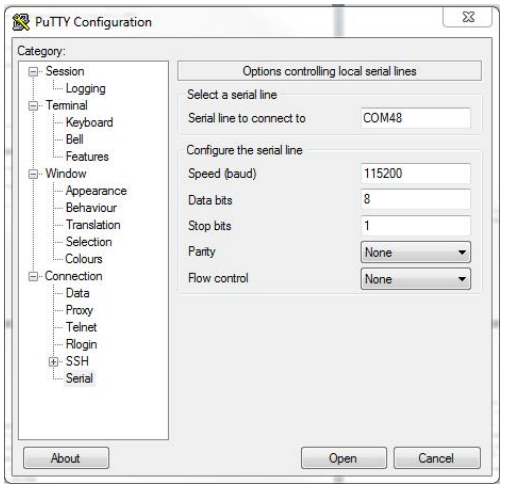


   When the terminal window opens click the Resume button in CCS, then type some characters and you should see the characters echoed into the terminal window.
   Skip to step 8.

5. In **Win7**, ► double-click on **putty.exe**. Make the settings shown below and then click Open. Your COM port number will be the one you noted earlierin chapter one.



   When the terminal window opens► click the Resume button in CCS, thentype some characters and you should see the characters echoed into the terminal window.



Using UART Interrupts

Instead of continually polling for characters, we'll make some modifications to our code to allow the use of interrupts to receive and transmit characters. In the first part of this lab, the only indication we had that our code was running was to open the terminal window to type characters and see them echoed back. In this part of the lab, we'll add a visual indicator to show that we received and transmitted a character. So we'll need to add code similar to previous labs to blink the LED inside the interrupt handler.

6. First, let's add the code in **main()** to enable the UART interrupts we want to handle. ► Click on the Terminate button to return to the CCS Edit perspective. We need to add two additional header files at the top of the file:

```
#include"inc/hw_ints.h"
#include"driverlib/interrupt.h"
```

7. Now we need to add the code to enable processor interrupts, then enable the UART interrupt, and then select which individual UART interrupts to enable. We will select receiver interrupts (RX) and receiver timeout interrupts (RT). The receiver interrupt is generated when a single character has been received (when FIFO is disabled) or when the specified FIFO level has been reached (when FIFO is enabled). The receiver timeout interrupt is generated when a character has been received, and a second character has not been received within a 32-bit period. ► Add the following code just below the **UARTConfigSetExpClk()** function call:

```
IntMasterEnable();
IntEnable(INT_UART0);
UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);
```

8. We also need to initialize the GPIO peripheral and pin for the LED. ► Just before the function **UARTConfigSetExpClk()** is called, add these two lines:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);
```

9. ► Finally, we can create an empty **while(1)** loop at the end of main by commenting out the line of code that's already there:

```
 while (1)
{
//if (UARTCharsAvail(UART0_BASE))UARTCharPut(UART0_BASE,UARTCharGet(UART0_BASE));
}
```

10. ► Save the changes you made to main.c (but leave it open for making additional edits).

11. Now we need to write the UART interrupt handler. The interrupt handler needs to read the UART interrupt status register to know which specific interrupt event(s) just occurred. This value is then used to clear the interrupt status bits (we only enabled RX and RT interrupts, so those are the only possible sources for the interrupt). The next step is to receive and transmit all the characters that have been received. After each character is "echoed" to the terminal, the LED is blinked for about 1 millisecond. ► Insert this code below the include statements and above **main()**:

```
void UARTIntHandler(void)
{
uint32_t ui32Status;

ui32Status = UARTIntStatus(UART0_BASE, true); //get interrupt status

    UARTIntClear(UART0_BASE, ui32Status); //clear the asserted interrupts

    while(UARTCharsAvail(UART0_BASE)) //loop while there are chars
    {
        UARTCharPutNonBlocking(UART0_BASE, UARTCharGetNonBlocking(UART0_BASE));
                //echo character
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2); //blink LED
        SysCtlDelay(SysCtlClockGet() / (1000 * 3)); //delay ~1 msec
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0); //turn off LED
    }
}
```

12. We're almost done. We've added all the code we need. The final step is to insert the address of the UART interrupt handler into the interrupt vector table. ► Open the **tm4c123gh6pm_startup_ccs.c**file.Just below the prototype for _c_int00(void), add the UART interrupt handler prototype:

```
extern void UARTIntHandler(void);
```

13. On about line 68, you'll find the interrupt vector table entry for "UART0 Rx and Tx". It's just below the entry for "GPIO Port E". The default interrupt handler is named IntDefaultHandler. ► Replace this name with UARTIntHandlerso the line looks like:

```
UARTIntHandler,                         // UART0 Rx and Tx
```

14. Save your work. Your **main.c**code should look like this.

```
#include<stdint.h>
#include<stdbool.h>
#include"inc/hw_ints.h"
#include"inc/hw_memmap.h"
#include"inc/hw_types.h"
#include"driverlib/gpio.h"
#include"driverlib/interrupt.h"
#include"driverlib/pin_map.h"
#include"driverlib/sysctl.h"
#include"driverlib/uart.h"

voidUARTIntHandler(void)
{
uint32_t ui32Status;
    ui32Status = UARTIntStatus(UART0_BASE, true); //get interrupt status
UARTIntClear(UART0_BASE, ui32Status); //clear the asserted interrupts

while(UARTCharsAvail(UART0_BASE)) //loop while there are chars
    {
UARTCharPutNonBlocking(UART0_BASE, UARTCharGetNonBlocking(UART0_BASE)); //echo character
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2); //blink LED
SysCtlDelay(SysCtlClockGet() / (1000 * 3)); //delay ~1 msec
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0); //turn off LED
    }
}

intmain(void) {

SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); //enable GPIO port for LED
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2); //enable pin for LED PF2

UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

IntMasterEnable(); //enable processor interrupts
IntEnable(INT_UART0); //enable the UART interrupt
UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT); //only enable RX and TX interrupts

UARTCharPut(UART0_BASE, 'E');
UARTCharPut(UART0_BASE, 'n');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'r');
UARTCharPut(UART0_BASE, ' ');
UARTCharPut(UART0_BASE, 'T');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'x');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, ':');
UARTCharPut(UART0_BASE, ' ');

while (1) //let interrupt handler do the UART echo function
    {
//      if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
    }

}
```

15. ► Click the Debug button to build and download your program to the TM4C123GH6PM memory.
16. ► If you've closed it, open Hyperterminal or puTTY, and configure it as before.
17. ► Click the Resume button. Type some characters and you should see the characters echoed into the terminal window. Note the LED.
18. ► Close puTTY or HyperTerminal. Click the Terminate button to return to the CCS Edit perspective.
    ► Close the Lab12 project and minimize Code Composer Studio.

You're done.