SimpliciTI-compatible UART Driver

By Jim Noxon and Kristoffer Flores

Keywords

- *SimpliciTI*
- *UART driver*
- *RS-232*
- *SmartRF®04EB*
- *SmartRF®05EB*
- *CCMSP-EM430F2618*
- *MSP430FG4618 Experimenter Board*
- *CC1110Fx*
- *CC1111Fx*
- *CC2510Fx*
- *CC2511Fx*
- *CC2430*
- *CC2431*
- *CC2530*

1 Introduction

This design note introduces a UART driver that is compatible with the SimpliciTI^{TM} low-power wireless networking protocol. This note describes the driver implementation, the process of adding the driver into an existing SimpliciTI project, and the basic driver API function calls.

The supplied UART driver can facilitate interfacing a SimpliciTI device with other UART-capable devices, for example, a PC via RS-232. Using a UART with an RS-232 transceiver allows for live in-system data transfer from a SimpliciTI device to a PC or PC control of the SimpliciTI device for end applications or for development and debugging purposes. The driver package includes an example project based on the 'Simple Peer-to-Peer' SimpliciTI example to demonstrate the UART operation. The example program sends messages to a PC via the RS-232 port on the SmartRF®04, SmartRF®05, and MSP430FG4618/F2013 Experimenter Boards. The sample code in this design note can be downloaded from http//:www.ti.com/lit/zip/SWRA306.

Table of Contents

2 Abbreviations

UART	Universal Asynchronous Receiver/Transmitter
DMA	Direct Memory Access
ISR	Interrupt Service Routine
SoC	System On Chip
EM	Evaluation Module
TX	Transmit
RX	Receive

3 UART Driver Description

3.1 Compatibility

The driver has been tested on the following SimpliciTI development platforms:

- SmartRF®04 board with CC1110EM, CC2510EM, and CC2430EM
- SmartRF®05 board with CCMSP-EM430F2618 daughter board and CC2520EM
- SmartRF®05 board with CC2530EM
- MSP430FG4618/F2013 Experimenter Board with CC1101EM

Code modifications may be necessary to ensure proper compilation and UART operation when using MSP430s, microcontrollers, or development platforms other than those listed above. The driver can be used in both IAR and Code Composer Essentials (CCE).

3.2 Driver Implementation

The supplied UART driver uses first-in first-out (FIFO) ring buffers for transmit (TX) and receive (RX) data. To send data, the user calls an API function that writes the data to the end of the TX FIFO. To receive data, the user calls a function that reads data from the top of the RX FIFO.

The actual transmission and reception of data bytes over the UART is driven by interrupts. If there is data in the TX FIFO, the UART TX interrupt service routine (ISR) moves the data byte by byte to the UART TX byte register for transmission until the FIFO is empty. The UART RX ISR moves each received byte from the RX byte register to the RX FIFO as long as there is room in the buffer.

The alternative to an interrupt-based UART is one that is supported by the Direct Memory Access (DMA) controller to move data to and from the UART TX/RX byte registers. An interrupt-based solution was chosen for the following reasons:

1. A UART with DMA support would require two DMA channels, one for transmit and one for receive. An ISR-driven solution keeps DMA resources free to use with other peripherals.

2. While UART using DMA could operate as a background process with very little overhead, some of the functions provided with the supplied driver, for example, the function to check the amount of unread data in the RX FIFO, have more practical implementations in an ISR-based UART.

3.2.1 Flow control on MSP430s

Flow control prevents overflow conditions via signalling on the RTS and CTS UART pins. If the RX FIFO is full, the RTS pin will output a high logic level as a signal that the device is not ready to receive data. On the transmit side, the UART will wait for the CTS pin (driven by the target device’s RTS pin) to go to a low logic level before transmitting data.

The 8051-based SoCs have a built-in flow control mechanism managed by hardware while the MSP430s do not. To provide flow control capability to MSP430-based systems, a flow control mechanism managed by software is enabled when using an MSP430 and flow control is turned on. The implementation maps two I/O pins as the RTS/CTS signals and toggles the lines manually in software. Since this handshaking mechanism is software-based, the latency involved in toggling the RTS and CTS lines can lead to lost data when using high baud rates.

3.3 User Options and Interface

The UART driver is designed for minimal complexity for the user. Using `#define` statements in a preinclude file, the user can specify which UART on the CPU to use, the baud rate, flow control mode, parity mode, and number of stop bits. The user can also specify the size of the transmit and receive FIFO buffers. However, this driver does not allow the above settings to be changed during runtime. Section 4 explains the driver installation and project setup procedures.

The driver API provides seven function calls for the UART. These are high-level functions for sending and receiving messages to and from the buffers, checking for free space in the transmit buffer, and checking how much unread data is in the receive buffer. Section 6 describes the API functions in detail.

4 Installing the Driver into an Existing SimpliciTI Project

Installing the UART driver into a project consists of three main steps: extracting the files into the project directories, configuring the UART, and setting up the project to use the UART.

4.1 Extracting the Files into the Project Directories

The first step in installing the UART driver is to extract the zip file contents into the SimpliciTI directory. If the extraction process preserves the directory structure, the file contents should extract into the recommended folders; however, the table below lists the files and their recommended destinations:

Extract into $SimpliciTI_Installation_Folder$\Components\bsp	
Filename	**Description**
`pp_utils.h`	Macro definitions used by UART driver.
Extract into $SimpliciTI_Installatio _Folder$\Components\bsp\drivers	
Filename	**Description**
`uart.h`	UART definitions and low-level function declarations
`uart_intfc.h`	UART driver include file with API function prototypes
Extract into $SimpliciTI_Installation_Folder$\Components\bsp\drivers\code	
Filename	**Description**
`uart.c`	Low-level UART function implementations and ISRs
`uart_intfc.c`	UART API function implementations
Extract into $SimpliciTI_Installation_Folder$\Projects\Examples\Applications	
Filename	**Description**
`main_LinkToWithUART.c`	Example program (see Section 5)
`options.h`	Project preinclude file with UART configuration definitions

4.2 Configuring the UART

The `options.h` preinclude file contains `#define` statements that determine the following UART settings:

- Transmit and receive FIFO buffers size
- For 8051-based SoCs, the number and location of the USART module to use
- For MSP430s, the letter and module number of the USCI module to use
- Baud rate
- Flow control mode
- Number of stop bits
- Parity mode

The 8051-based SoCs use a consistent pin mapping such that a given USART (number and location) maps to the same I/O pins regardless of the SoC. The driver has built-in pin mappings for the different USART configurations. For the MSP430s, it is necessary to define the pins used by the chosen USCI module because the pin mapping for the same module may differ among different MSP430s. Since the MSP430 does not have hardware-handshaking, the pins to use for CTS and RTS must also be defined in `options.h`.

When using an 8051-based SoC EM on a SmartRF®04 (CC1110/1, CC2510/1, CC2430/1) or SmartRF®05 board (CC2530/1), specifying USART port number 0 at location 1 will map the UART to the on-board RS-232 transceiver.

The comments in `options.h` identify the pin mappings for USCI_A0 for the CCMSP-EM430F2618 SmartRF®05 board and for the MSP430FG4618 Experimenter Board. Using the mapping will route the UART to the on-board RS-232 transceivers. On the CCMSP-EM430F2618 SmartRF®05 board, it is possible to enable hardware-handshaking because pins 2.6 and 2.7 on the MSP430 connect to the RTS and CTS pins of the RS-232 transceiver. On the Experimenter Board, hardware-handshaking cannot be enabled because the RTS and CTS pins on the RS-232 connector are open.

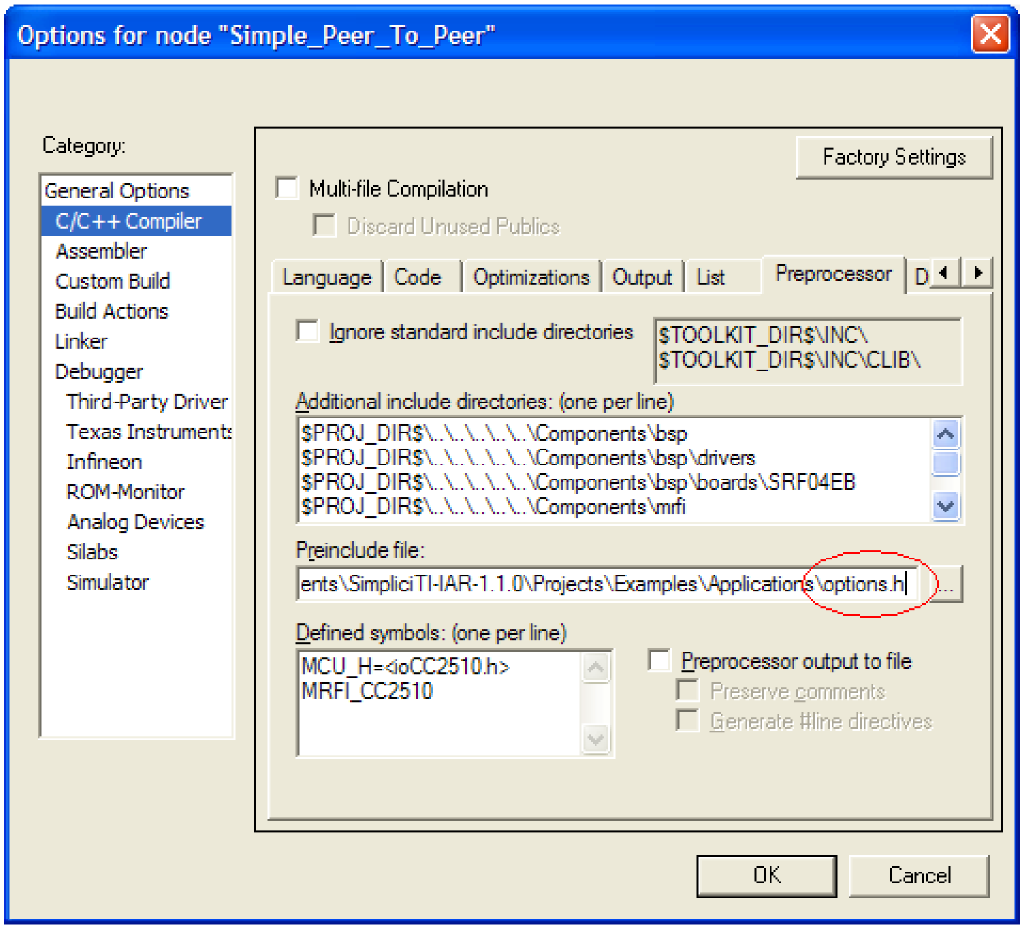
4.3 Setting Up the Project

4.3.1 In IAR (for 8051-based SoC or MSP430)

The following steps set up an existing SimpliciTI project to use the UART driver:

1. With the workspace and project open in IAR, add `uart.c` and `uart_intfc.c` to the project:
    a. Navigate to <Project➔Add Files…>.
    b. Browse for `uart.c` and `uart_intfc.c` in the file browser that appears, highlight the files, and press Open.
    c. Both `uart.c` and `uart_intfc.c` should now appear in the project files list. Drag and drop the files to the desired folder in the project or leave them in the default location.

2. Set `options.h` as a preinclude file for the project:
    a. Navigate to <Project➔Options>
    b. Under Category, select C/C++ Compiler, then choose the Preprocessor tab.
    c. Under Preinclude File, enter the location or browse for the `options.h` file.



Multi-file Compilation: unchecked
Discard Unused Publics: unchecked

Ignore standard include directories: unchecked

```
$TOOLKIT_DIR$\INC\
$TOOLKIT_DIR$\INC\CLIB\
```

Additional include directories: (one per line)

```
$PROJ_DIR$\..\..\..\..\..\Components\bsp
$PROJ_DIR$\..\..\..\..\..\Components\bsp\drivers
$PROJ_DIR$\..\..\..\..\..\Components\bsp\boards\SRF04EB
$PROJ_DIR$\..\..\..\..\..\Components\mrfi
```

Preinclude file:

```
ents\SimpliciTI-IAR-1.1.0\Projects\Examples\Applications\options.h
```

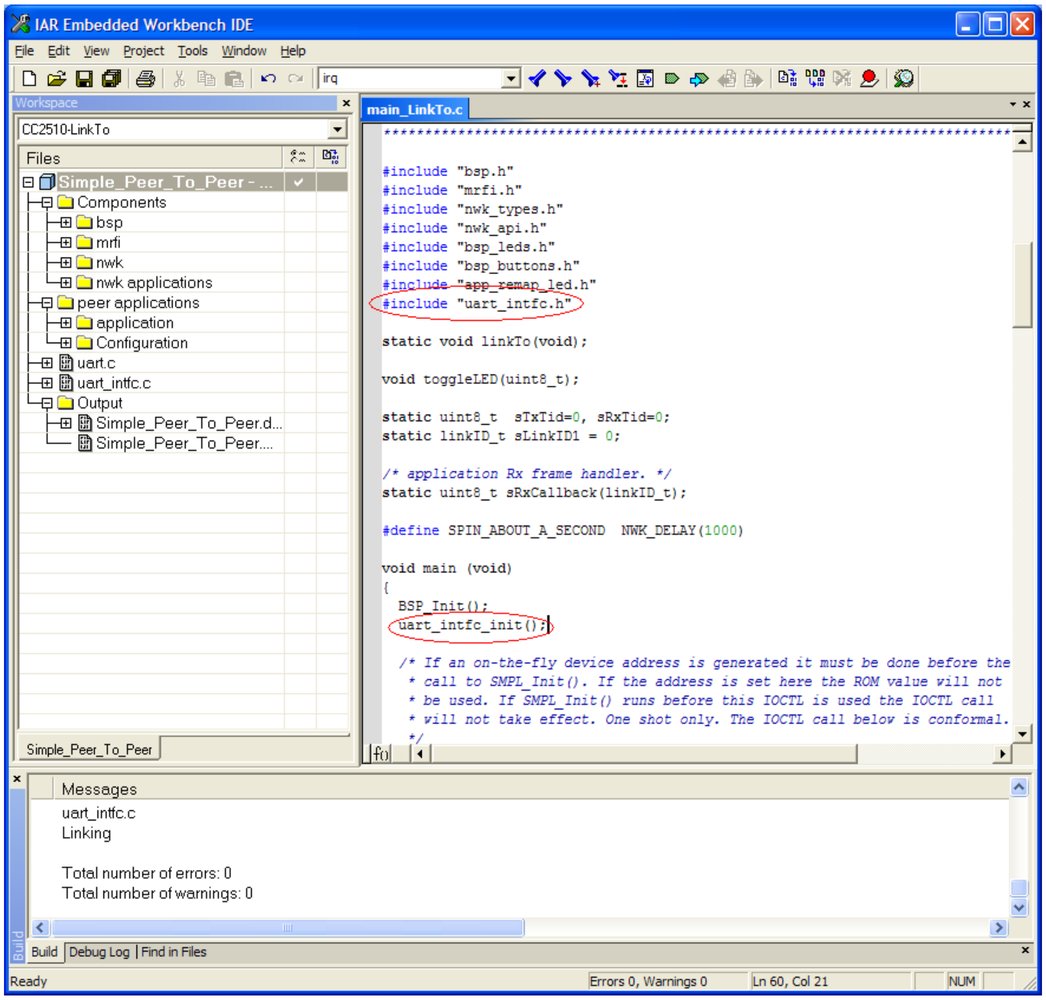
Defined symbols: (one per line)

```
MCU_H=<ioCC2510.h>
MRFI_CC2510
```

Preprocessor output to file: unchecked
Preserve comments: unchecked
Generate #line directives: unchecked

Figure 1. Example preprocessor options tab for CC2510 project

3. In the main C file of the project, include `uart_intfc.h` and add a line of code to call `uart_intfc_init()` after the call to `BSP_Init()` (as shown in Figure 2). Rebuild the project. The project should compile with no errors. The other UART API functions can be called after `SMPL_Init()` has been called.



```
*****************************************************************************

#include "bsp.h"
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"
#include "app_remap_led.h"
#include "uart_intfc.h"

static void linkTo(void);

void toggleLED(uint8_t);

static uint8_t  sTxTid=0, sRxTid=0;
static linkID_t sLinkID1 = 0;

/* application Rx frame handler. */
static uint8_t sRxCallback(linkID_t);

#define SPIN_ABOUT_A_SECOND  NWK_DELAY(1000)

void main (void)
{
  BSP_Init();
  uart_intfc_init();

  /* If an on-the-fly device address is generated it must be done before the
   * call to SMPL_Init(). If the address is set here the ROM value will not
   * be used. If SMPL_Init() runs before this IOCTL is used the IOCTL call
   * will not take effect. One shot only. The IOCTL call below is conformal.
   */
```

```
Messages
uart_intfc.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

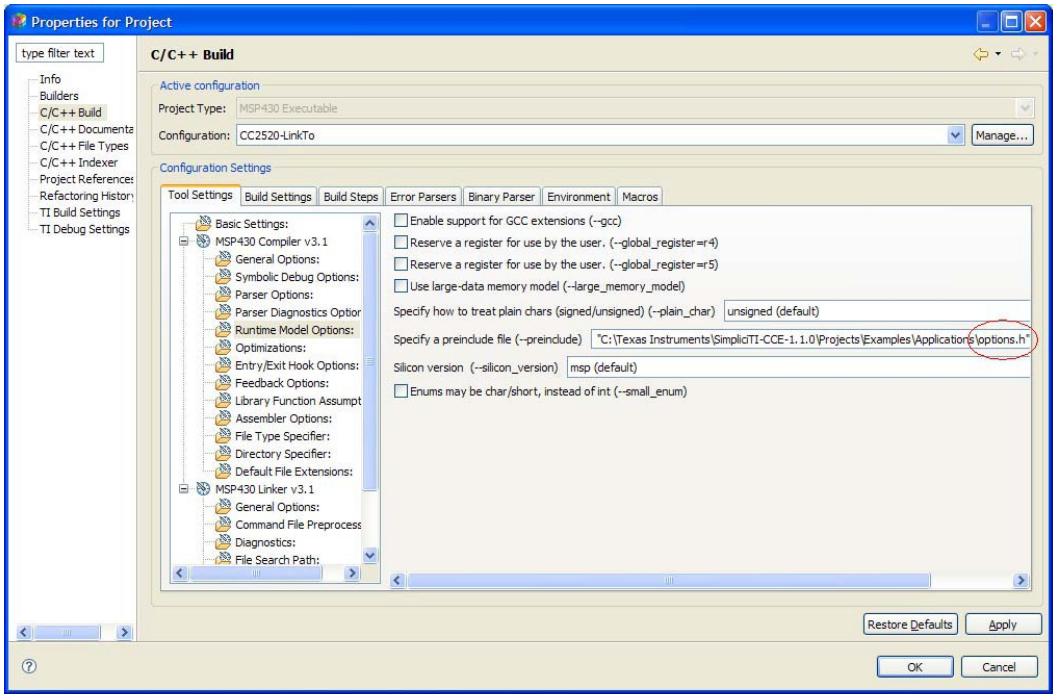
Figure 2. No build errors after successful driver install and initialization in IAR

4.3.2 *In Code Composer Essentials (for MSP430)*

The following steps set up an existing SimpliciTI project to use the UART driver:

1. With a workspace and project open in CCE, link `uart.c` and `uart_intfc.c` to the project:
   a. Navigate to <Project➔Link Files to Active Project...>
   b. Browse for `uart.c` and `uart_intfc.c` in the file browser that appears, highlight the files, and press Open.
   c. Both `uart.c` and `uart_intf.c` should now appear in the project files list. Drag and drop the files to the desired folder in the project or leave them in the default location.

2. Configure `options.h` as a preinclude file
   a. Navigate to <Project➔Properties>
   b. On the left menu in the window that appears, select C/C++ Build.
   c. Under Configuration Settings➔Tool Settings, select the Runtime Model Options.
   d. Specify `options.h` as a preinclude file by entering the file location or browsing for the file (Figure 3).



Active configuration
Project Type: MSP430 Executable
Configuration: CC2520-LinkTo

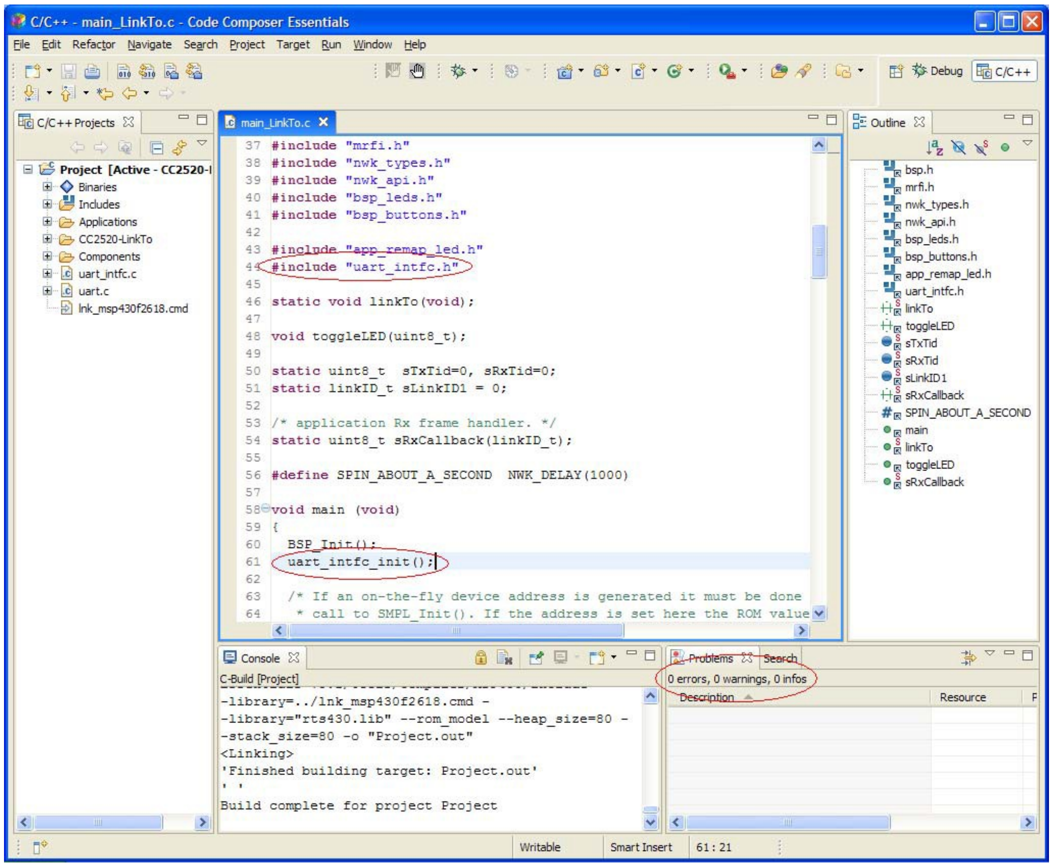
Configuration Settings
- [ ] Enable support for GCC extensions (--gcc)
- [ ] Reserve a register for use by the user. (--global_register=r4)
- [ ] Reserve a register for use by the user. (--global_register=r5)
- [ ] Use large-data memory model (--large_memory_model)

Specify how to treat plain chars (signed/unsigned) (--plain_char): unsigned (default)
Specify a preinclude file (--preinclude): "C:\Texas Instruments\SimpliciTI-CCE-1.1.0\Projects\Examples\Applications\options.h"
Silicon version (--silicon_version): msp (default)

- [ ] Enums may be char/short, instead of int (--small_enum)

Figure 3. Specifying a preinclude file in CCE.

3. In the main project C file, include `uart_intfc.h` and add a line of code to call `uart_intfc_init()` after the call to `BSP_Init()`(see Figure 4). Rebuild the project. The project should compile with no errors. The other UART API functions can be called after `SMPL_Init()` has been called.



```
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"

#include "app_remap_led.h"
#include "uart_intfc.h"

static void linkTo(void);

void toggleLED(uint8_t);

static uint8_t  sTxTid=0, sRxTid=0;
static linkID_t sLinkID1 = 0;

/* application Rx frame handler. */
static uint8_t sRxCallback(linkID_t);

#define SPIN_ABOUT_A_SECOND  NWK_DELAY(1000)

void main (void)
{
  BSP_Init();
  uart_intfc_init();

  /* If an on-the-fly device address is generated it must be done
   * call to SMPL_Init(). If the address is set here the ROM value
```

```
-library=../lnk_msp430f2618.cmd -
-library="rts430.lib" --rom_model --heap_size=80 -
-stack_size=80 -o "Project.out"
<Linking>
'Finished building target: Project.out'
' '
Build complete for project Project
```

0 errors, 0 warnings, 0 infos

Figure 4. No build errors after successful driver install and initialization in CCE

5 Example – Simple Peer-to-Peer with HyperTerminal

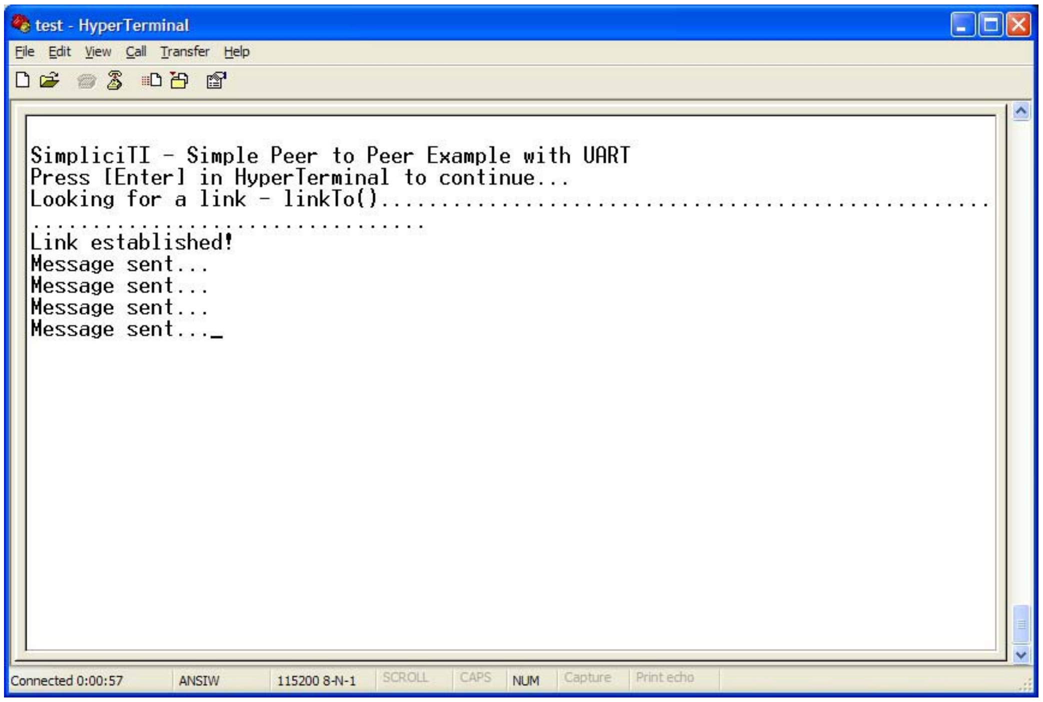
The file `main_LinkToWithUART.c` is a modified version of the Simple Peer-to-Peer example provided with SimpliciTI. The code provides a simple demonstration of the UART interface and is intended to be used with one of development platforms with an RS-232 interface (i.e. SmartRF®04 board, SmartRF®05 board, or MSP430FG4618 Experimenter Board with their compatible SoCs or radios). The example program communicates to a PC HyperTerminal via RS-232 to deliver status messages and to receive input from the PC.

The original LinkTo program waits for a button press on the development platform to initiate a link; the modified version waits for a carriage return in the HyperTerminal window. Pressing keys other than the carriage return key during this time will cause LED 1 to toggle. The rest of the program operates as the original, with the modification that the device sends status messages to the PC HyperTerminal as shown in Figure 5.

To run the example, follow the instructions below:

1. Open the Simple Peer-to-Peer LinkTo project for the device and platform to be used.
2. Per the instructions in Section 4, extract the UART driver files, configure the UART in `options.h`; and set up the project. The default Simple Peer-to-Peer project should compile with the minor changes to `main_LinkTo.c` as shown in Figure 2 and Figure 4.
3. Replace the code in `main_LinkTo.c` with the code in `main_LinkToWithUART.c` OR add (link in CCE) `main_LinkToWithUART.c` to the project and exclude `main_LinkTo.c` from the build. Rebuild the project.

4. Open a HyperTerminal and configure the terminal to match the UART settings specified in `options.h`
5. With a serial cable connected to the development board, run the example program.



```
SimpliciTI - Simple Peer to Peer Example with UART
Press [Enter] in HyperTerminal to continue...
Looking for a link - linkTo()..................................................
................................
Link established!
Message sent...
Message sent...
Message sent...
Message sent..._
```

Figure 5. HyperTerminal Window of Example Program

6 Basic API Function Calls

This section gives a description of each of the 7 UART API functions. For more details, see the code and comments in `uart_intfc.h` and `uart_intfc.c`.

6.1 uart_intfc_init()

6.1.1 Description

The `uart_intfc_init()` function prepares the UART by configuring the CPU registers (per the definitions in `options.h`) and initializing the UART buffers. `uart_intfc_init()` must be called before attempting to use any other API functions.

6.1.2 Prototype

```
void uart_intfc_init(void)
```

6.2 tx_peek()

6.2.1 Description

The `tx_peek()` function returns the number of unused bytes in the transmit FIFO.

6.2.2 Prototype

```
int tx_peek( void )
```

6.2.3 Return

Data Type	Description
integer	Number of bytes of free space in UART transmit FIFO

6.3 tx_send()

6.3.1 Description

The `tx_send()` function pushes a message to the end of the transmit FIFO if there is enough room for the entire message. If there is inadequate free space in the FIFO, nothing is appended to the FIFO. The transmit FIFO is set to 50 bytes by default, but can be resized in the `options.h` file.

6.3.2 Prototype

```
bool tx_send(const void* data, size_t len)
```

6.3.3 Parameter Details

Parameter	Description
`(void *) data`	Pointer to first byte of data to be sent
`len`	Length in bytes of data to be sent (max 50)

6.3.4 Return

Boolean Value	Description
`true` (1)	Data successfully pushed onto buffer
`false` (0)	Not enough room in transmit buffer; No data pushed onto buffer

6.4 tx_send_wait()

6.4.1 Description

The `tx_send_wait()` function pushes a message to the end of the transmit FIFO, but unlike `tx_send()`, the message length can exceed the size of the FIFO. For messages longer than the available space in the FIFO, `tx_send_wait()` pushes the message into the FIFO in pieces, filling the buffer with the data as space becomes available due to completed transmissions. The function is a blocking task in that it does not terminate until the entire message has been delivered to the FIFO.

6.4.2 Prototype

```
bool tx_send_wait(const void* data, size_t len)
```

6.4.3 Parameter Details

Parameter	Description
`(void *) data`	Pointer to first byte of data to be sent
`len`	Length in bytes of data to be sent

6.4.4 Return

Boolean Value	Description
`true` (1)	Data successfully pushed onto buffer
`false` (0)	Null pointer or `len = 0`

6.5 rx_peek()

6.5.1 Description

The `rx_peek()` function returns the number of bytes of unread received data in the receive FIFO. It is recommended to call `rx_peek()` to determine if there is unread data in the receive FIFO before calling `rx_receive()`.

6.5.2 Prototype

```
int rx_peek()
```

6.5.3 Return

Data Type	Description
integer	Number of unread bytes available in the UART receive FIFO

6.6 rx_receive()

6.6.1 Description

The `rx_receive()` function pulls unread data out of the UART receive FIFO to a specified location until the specified maximum number of bytes has been read or the receive FIFO has been emptied. The function returns the actual number of bytes read.

6.6.2 Prototype

```
int rx_receive(void *data, int max_len)
```

6.6.3 Parameter Details

Parameter	Description
`(void *) data`	Location to which buffer data should be read
`max_len`	Maximum number of bytes to be read from buffer

6.6.4 Return

Data Type	Description
integer	Number of bytes actually read from buffer

6.7 uart_busy()

6.7.1 Description

The uart_busy() function indicates whether there is any impending actions required by the UART, either there is still data in the transmit FIFO or there is unread data in the receive FIFO.

6.7.2 Prototype

```
bool uart_busy( void )
```

6.7.3 Return

Boolean Value	Description
`true` (1)	UART is in use; UART transmit and/or receive buffers have data
`false` (0)	UART idle; both UART buffers are empty

7 General Information

7.1 Document History

Revision	Date	Description/Changes
SWRA306	2009.09.22	Initial release.