ECE 362 Lab Verification / Evaluation Form

Experiment 3

Evaluation:

IMPORTANT! You must complete this experiment during your scheduled lab period. All work for this experiment must be demonstrated to and verified by your lab instructor *before the end* of your scheduled lab period.

STEP	DESCRIPTION	MAX	SCORE
1	`findc`	5	
2	`popcnt`	5	
3	`correl`	5	
4	`minmax`	5	
5	`ASCII table generator`	5	
	TOTAL	25	

Signature of Evaluator: _____

Academic Honesty Statement:

IMPORTANT! Please carefully read and sign the Academic Honesty Statement, below. *You will not receive credit for this lab experiment unless this statement is signed in the presence of your lab instructor.*

"In signing this statement, I hereby certify that the work on this homework and experiment is my own and that I have not copied the work of any other student (past or present) while completing them. I understand that if I fail to honor this agreement, I will receive a score of ZERO and be subject to possible disciplinary action."

`Printed Name: _____ Class No. __ __ __ __ - __`

`Signature: _____ Date: _____`

Experiment 3: Assembly Language Programming Techniques – Part 2

Instructional Objectives:

- To learn to write more complex assembly language programs which include various subroutine parameter passing techniques.
- To learn about macros and their appropriate use in assembly language programs.

Reference:

- ***S12CPUV2 Reference Manual***, Sections 2, 3, 5, and Appendix A (on course website)
- ***CPU12 Reference Guide*** (on course web site)

Prelab Preparation:

- **Read this document and attempt writing the code for *all* steps prior to coming to lab**

Parameter Passing Techniques

Writing modular code will be heavily emphasized in this course. Requisite to creation of modular code is a thorough understanding of how parameters may be passed from one procedure to another. You were introduced to this concept in Part 1, and will further explore it in Part 2. There are four basic ways parameters may be passed: (a) via registers ("call by value"), (b) via pointers ("call by address") to a global parameter area, (c) via a private parameter area (here, following the "call" instruction), and (d) via the stack.

(a) Via Registers

Any time a small number of parameters is involved, passing them via the CPU's registers represents the most effective technique. Conversion routines and so-called "I/O device driver" routines are good examples of applications where passing parameters via registers is highly desirable. There are two character I/O routines that utilize this technique which we will use considerably in the remaining of the course: `inchar` and `outchar`. The subroutine `outchar` transmits the ASCII character passed to it in the A register to the terminal screen using the on-chip HCS12 Serial Communications Interface (SCI). The subroutine `inchar` waits to receive a character from the keyboard (also using the SCI), and when it does, returns its ASCII value in the A register.

(b) Via Pointers

Often times a *structure* must be passed to a subroutine. Because structures (e.g., arrays or linked lists) are often large and of variable size, it is not feasible to pass them via registers. Therefore, a *pointer* to the *starting address* is often passed to the subroutine – typically, via an *index register*. Since the actual values are not passed (rather, they reside in *globally accessible* locations in memory), we refer to this technique as "call by address" or "call by name".

(c) Via Private Parameter Area

In some instances it is appropriate to associate a *private parameter area* with each call to a given subroutine. A good example is a message printing routine, where a different string is to be printed each time the routine is called. One way this can be accomplished in assembly language programming is to *place the parameters following the "call" (i.e.,* `jsr` *or* `bsr` *instruction).*

 © 2015 by D. G. Meyer

Use of a private parameter area is illustrated in the following example:

```
      jsr     pmsg
      fcb     'Print this string'
      fcb     NULL    ; ASCII null (string termination character)
      {next instruction}
```

Note the data (here, an *ASCII string of characters*) is "sandwiched" between the "JSR" instruction and the instruction that immediately follows. As indicated in the listing of `pmsg`, below, two things must be accomplished: (1) a pointer to the start of the string must be determined, and (2) the return address (on top of the stack) must be *corrected* so it points to the "next instruction." Also note the `pmsg` subroutine continues to output characters until it reaches the `NULL` character. It is therefore important to remember to terminate your string with the `NULL` character or you might be printing some interesting and unwanted characters.

```
NULL    equ     0       ;ASCII null (string termination character)
pmsg    pulx            ;return address (top stack item)
                        ;points to start of string
ploop   ldaa    1,x+    ;access next character of string
        cmpa    #NULL   ;test for ASCII null (termination)
        beq     pexit   ;exit if encounter ASCII null
        jsr     outchar ;print character on terminal using
                        ; outchar routine
        bra     ploop
pexit   pshx            ;place corrected return address back
        rts             ;on top of stack and return
```

(d) Via the Stack

The most "generic" way to pass parameters is via the stack. Most modern high-level language compilers (e.g., "C") utilize this technique every time a procedure call is generated – use of the stack to pass parameters facilitates *recursion* as well as *arbitrary nesting* of subroutine calls.

Writing More Complex Assembly Language Programs

In the following exercises, you will have the opportunity to utilize the various parameter-passing techniques while learning to write more complex programs. They will also introduce you to the character I/O routines that we will be using the rest of the semester (and eventually writing ourselves!) as well as the use of macros. These programs can all be written and tested using the Full Chip Simulation mode available in Code Warrior, so you are highly encouraged to download your own copy of this software development tool (follow link on course website).

Steps 1-4:

Download and unzip the Code Warrior project folders **`Lab3-Step1`** through **`Lab3-Step4`** from the course website and proceed to write subroutines that solve each programming question (*a given solution should require no more than **15** lines of assembly code*). Follow the directions provided to test each of your solutions. Grading will be as follows: 5 points if pass all ten test cases with no "stack creep", 2 points if pass all test cases but with "stack creep" error flagged, 0 points otherwise (no partial credit awarded for non-working code).

 © 2015 by D. G. Meyer

Step 5:
Write a program that outputs selected portions of the ASCII table (ASCII is a standard way of encoding characters, which can be found on page 30 of the *CPU12 Reference Guide*). This program should prompt the user for the *starting* character followed by the (single-digit) number (range: 1-9) of *subsequent* characters to output. If the number is "out of range" an error message should be displayed. A sample session follows:

```
Enter starting character: N
Enter number: 8
NOPQRSTUV
Enter starting character: a
Enter number: 1
ab
Enter starting character: A
Enter number: 0
*** ERROR *** Invalid number
Enter starting character: A
Enter number: B
*** ERROR *** Invalid number
Enter starting character: ;
Enter number: 6
;<=>?@A
```

Download and unzip the project **Lab3-Step5** from the course website. Open **main.asm** in Code Warrior and enter the code that implements your solution. Follow the steps outlined below:

a) Write a subroutine **nchars** that will print N consecutive ASCII characters following the specified character. The A register will contain the ASCII value of the first character to be output, and the B register will contain the number of consecutive characters that follow it to be output. For instance, if (A) = $41 (ASCII code of "A") and (B) = $09, a string of *ten* consecutive letters will be printed, producing the following output:

```
ABCDEFGHIJ
```

The character I/O routines `inchar` (to input ASCII characters from an emulated terminal) and `outchar` (to output characters to an emulated terminal) have already been written and are included in the file. A routine to initialize the serial interface (`sinit`) is also provided. As done in Steps 1-4, use the **Terminal** component in the Code Warrior debugger to emulate serial I/O.

b) Complete the **main** program, which should do the following: prompt for a character, store it in the A register, prompt for a number, store it in the B register, then call the subroutine `nchars` (written for Step a) to print N+1 consecutive ASCII characters, beginning with the specified character. The program should keep running in a loop until the process is stopped. Use the macro `print` to format the prompt strings for the code (which calls the `pmsg` routine, also provided).

 © 2015 by D. G. Meyer

PURDUE UNIVERSITY

Debugging SCI using Simulation Mode

Microprocessor System Design and Interfacing
ECE 362

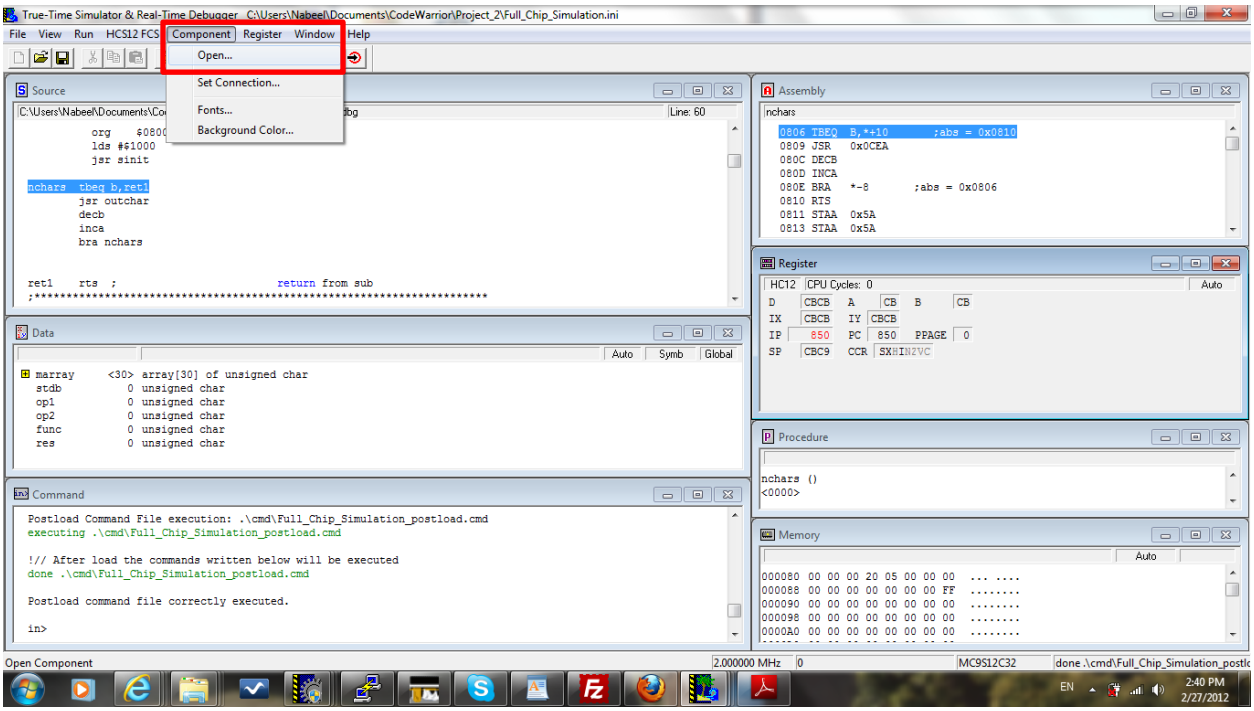
Course Staff

2/27/2012

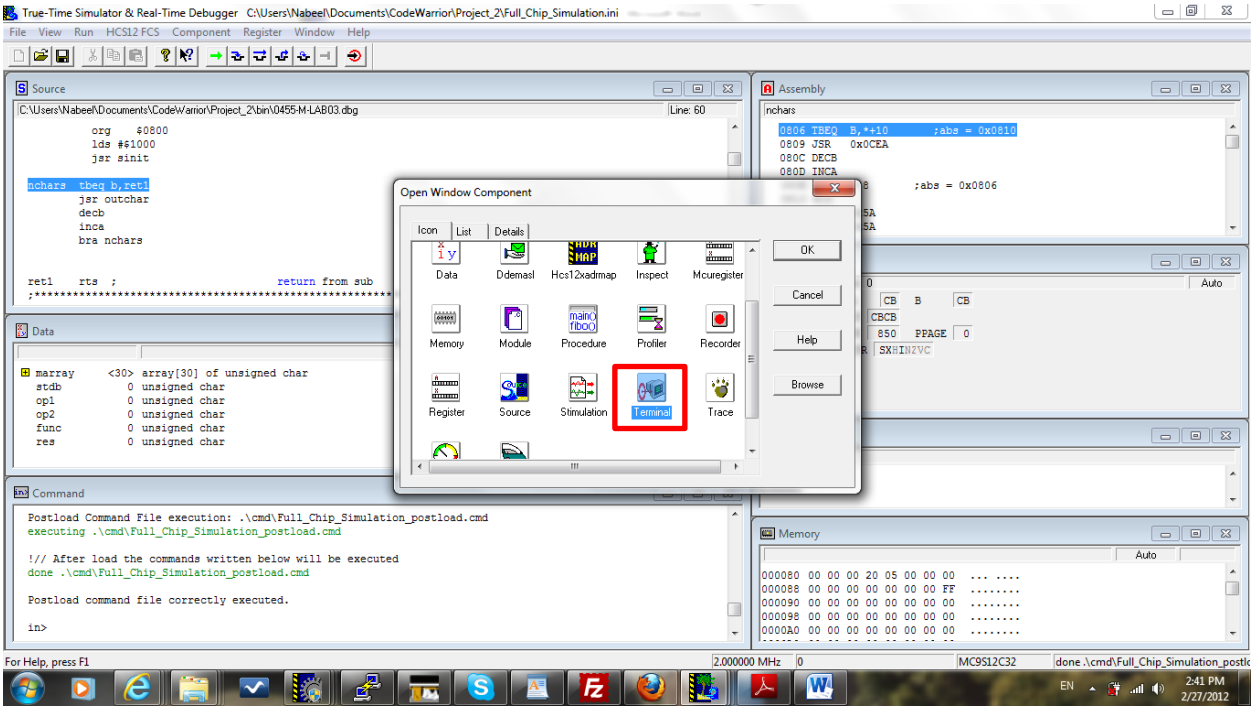
This tutorial shows how to output text to SCI (Serial Communication Interface) using the simulation mode (without the microcontroller). This is very helpful if you need to debug your code on your personal laptop without connecting it to the microcontroller.

Create a new project using the "**Full Chip Simulation**" Mode then follow the steps below to setup the SCI Terminal:

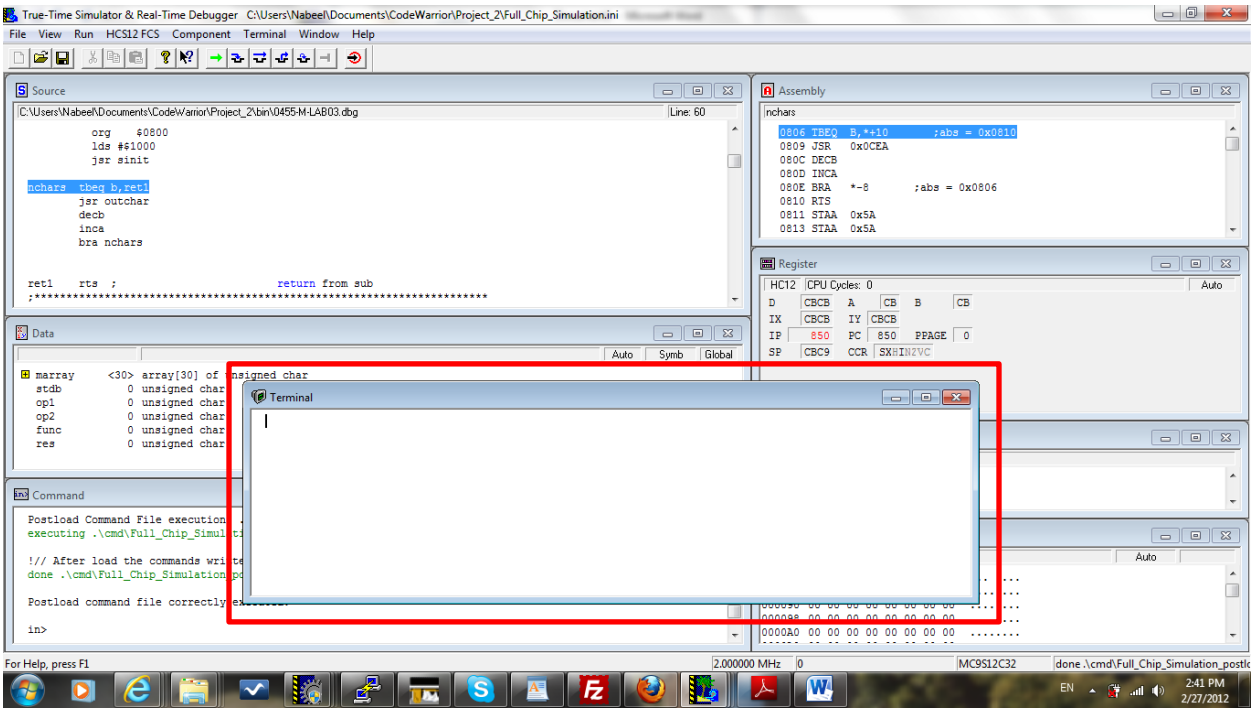
1. Run the code using CodeWarrior
2. From the top menu (in the debug window) select: Component -> Open



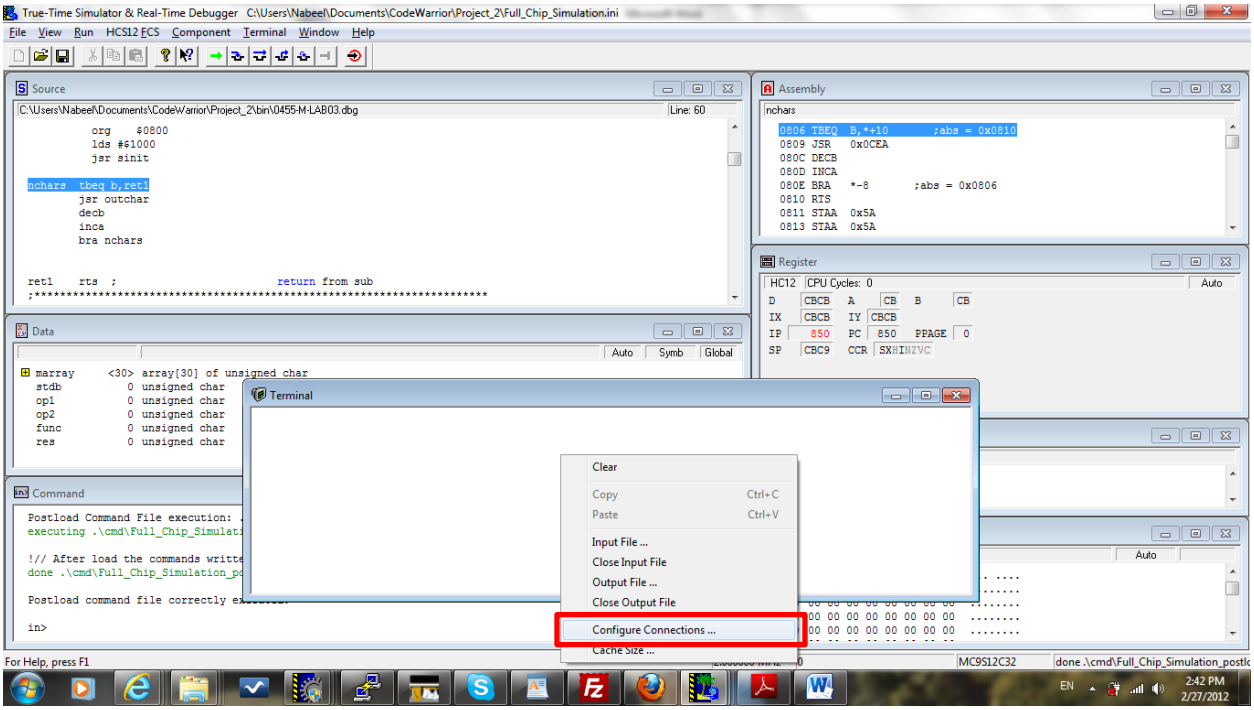
3. Scroll down and select **Terminal**



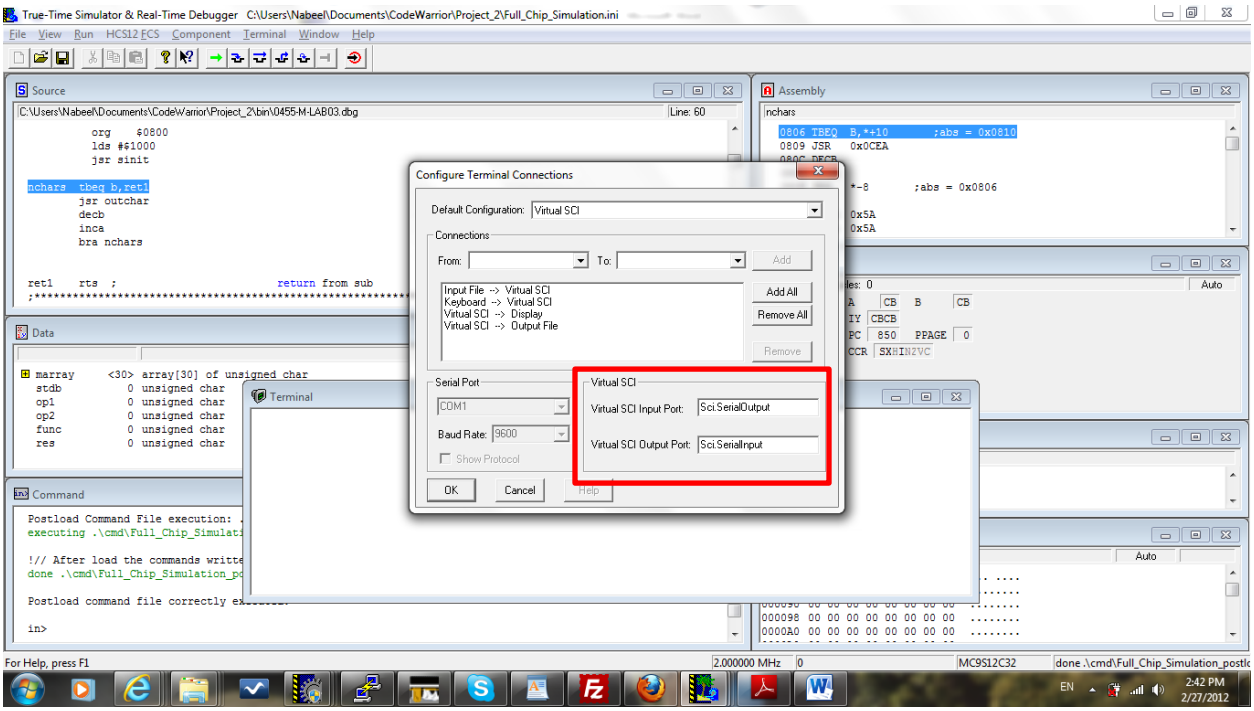
4. The terminal window will open

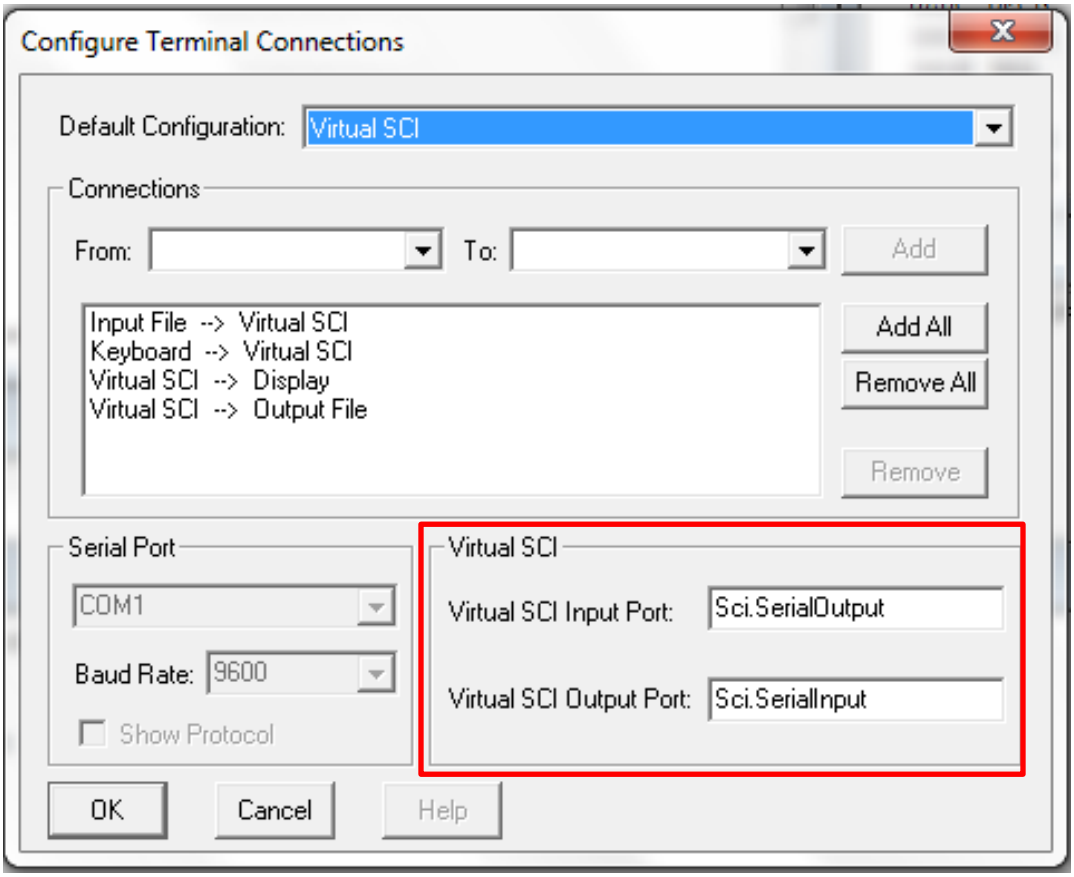


5. Right click on the Terminal window and select "**Configure Connection**"



6. Rename the Virtual SCI ports to
   a. Virtual SCI Output port to **Sci.SerialOutput** (Remove the 0)
   b. Virtual SCI Input port to **Sci.SerialInput** (Remove the 0)





7. Start the program, you should see the output on the terminal screen.