ECE381 – Lab 4 – Universal Asynchronous Receiver/Transmitter (UART)

Introduction:

A Universal Asynchronous Receiver/Transmitter (UART) is one of the more basic ways to send data between two devices. Using unidirectional transmit (TX) and receive (RX) lines, with predetermined communication parameters (baud, data format, control bits, etc.), UARTs can send data with very little configuration and hardware overhead. They are sometimes used as the interface between a microcontroller and an external device, especially if that external device is a communication bridge to a computer using character data. This lab will use UARTs as an opportunity to also learn about stepper motors, a type of motor designed to take discrete steps of known angular resolution. The UART will be used to set the speed (in rpm) and position of a basic, unipolar stepper motor.

Background:

Hardware UART

UARTs are one of the oldest ways to send data from one device to another. They are point-to-point communication, with two unidirectional primary wires. Transmit (TX) is an output wire, Receive (RX) is an input wire. To communicate directly with another device, just swap the TX and RX wires on devices (making sure there is a shared ground). There is no shared clock, so UARTs must be pre-configured with all data/synchronization parameters (baud rate, start/stop bits, data bits per word, msb/lsb order, parity, etc) before communication will be successful. Older UARTs (such as those in the RS-232 standard) also had extra control wires (Clear-to-Send, Request-to-Send, Ring, etc.), while some used in-band, software flow control (special bytes to indicate status).

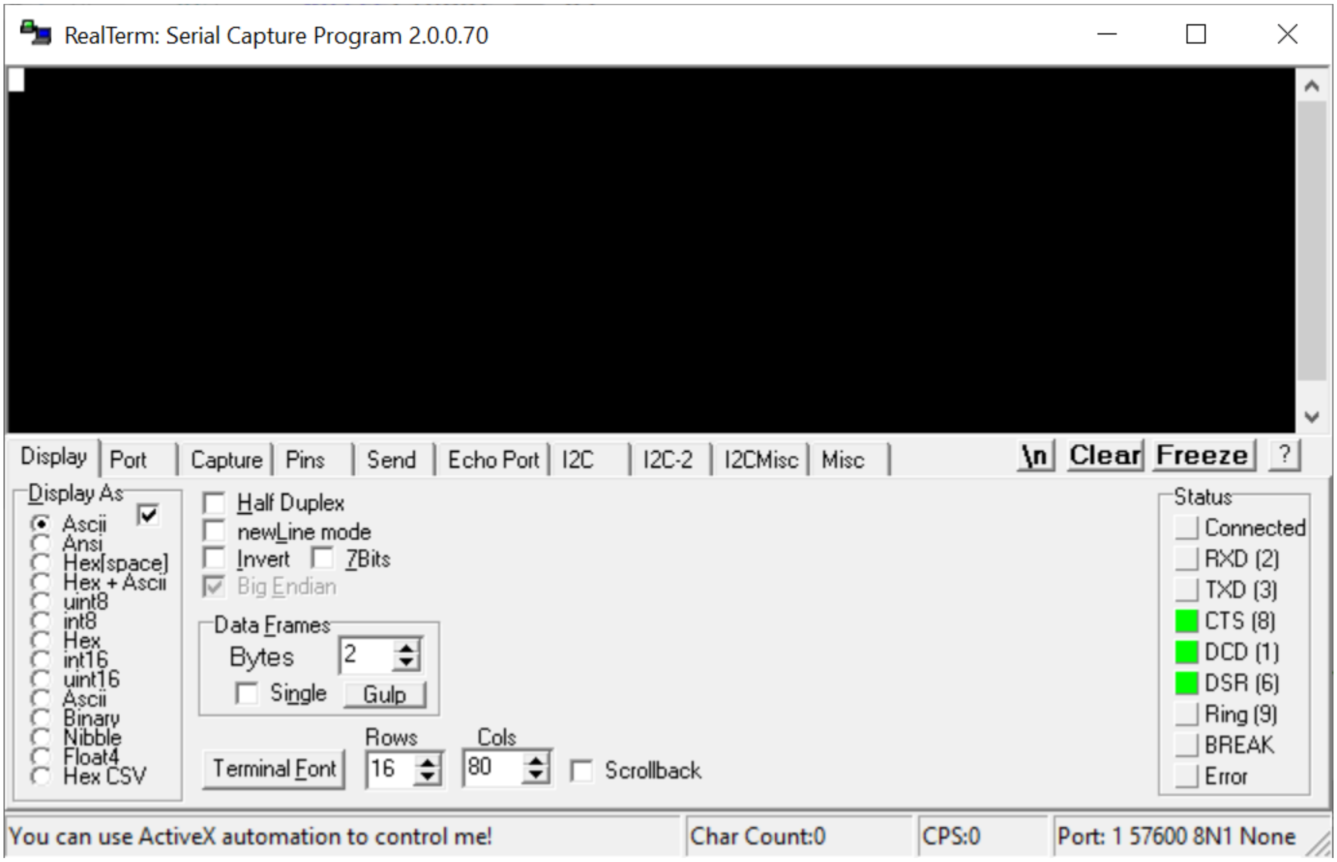
Newer PCs and devices rarely come with RS-232 (or any other) serial ports. Instead, most serial ports now are implemented over USB. Full discussion of USB is beyond the scope of this lab (maybe a future lab…), but USB-UARTs are typically implemented using the USB Communication Device Class (CDC). USB-UART bridges (such as the common FTDI-231x based ones) are preconfigured as CDCs, and are designed to appear on PCs as “Virtual COM Ports”, if the PC has the correct driver. They take the raw UART data, packetize it for USB, and send it over USB to the PC so it appears as character data over the COM/tty port. Sending/receiving such data requires some type of terminal emulator program (RealTerm or HyperTerminal on Windows, minicom/screen/tmux on Linux/Mac terminals) to open the COM/tty port and read/write data. Some microcontroller programmer firmware (like the Arduino or KitProg PSoC5 like we have) have a USB-UART bridge built-in. Others (like the STM32) require alternate hardware (like an FTDI-231x) or USB device firmware to achieve this PC-to-Microcontroller interfacing.

Our CY8CKIT-044 has a full speed (12 Mbps) USB-to-UART bridge built-in to the KitProg firmware. This means that our PSoC should appear on the connected computer as a virtual COM port. To communicate between the actual PSoC 4200M running our code and the KitProg, pins 7[0] and 7[1] on the 4200M are dedicated UART pins to the KitProg PSoC5. The KitProg firmware handles all of the packetization and formatting needed to have this data appear as serial data from a COM port on the host PC side.

Terminal Emulation

If the USB-UART bridge is functional, the host OS will see the connected device as a COM (Windows) or TTY (Linux/MacOS) device. These devices are often referred to as “serial” devices, and are accessible through drivers on the host OS. In Windows they show up as COM*X* and in Linux usually as /dev/ttyACM*X* or /dev/ttyUSB*X*. Programmatically, there are libraries you can use to read and write to them (The Win32 API has one for C, C++ has Boost/QT with serial libraries, Python has pyserial, Rust has serialport, etc.), but there are also typically “Terminal Emulator” programs already written with a GUI or other interface for sending/receiving UART data.

Windows through version XP used to come with a terminal emulator called “HyperTerm” built-in, but as this is no longer guaranteed, an alternate, RealTerm, will be used for this lab. RealTerm (https://sourceforge.net/projects/realterm/) is a fully open-source, Windows terminal emulator that works very well and has lots of features, including the ability to display non-printables and automatic hex/char conversion. The main window looks like this:



RealTerm: Serial Capture Program 2.0.0.70

Display As: ● Ascii ☑

□ Half Duplex
□ newLine mode
□ Invert □ 7Bits
☑ Big Endian

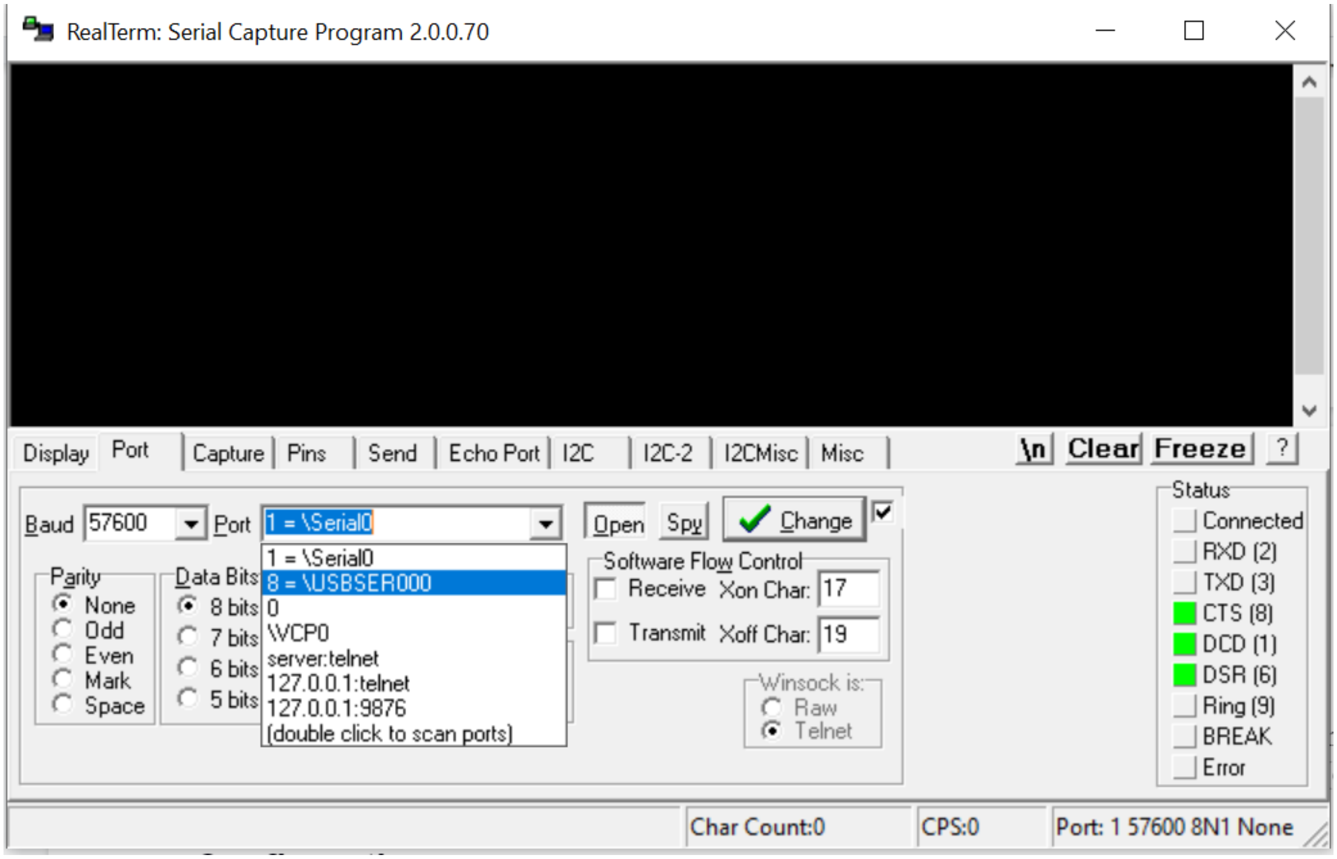
Data Frames: Bytes 2, □ Single

Rows 16, Cols 80, □ Scrollback

Status: □ Connected, □ RXD (2), □ TXD (3), 🟩 CTS (8), 🟩 DCD (1), 🟩 DSR (6), □ Ring (9), □ BREAK, □ Error

You can use ActiveX automation to control me! | Char Count:0 | CPS:0 | Port: 1 57600 8N1 None

You select the COM port using the drop-down here:


RealTerm: Serial Capture Program 2.0.0.70

Baud 57600 Port 1 = \Serial0

1 = \Serial0
8 = \USBSER000
0
\VCP0
server:telnet
127.0.0.1:telnet
127.0.0.1:9876
(double click to scan ports)

Parity: ● None ○ Odd ○ Even ○ Mark ○ Space

Data Bits: ● 8 bits ○ 7 bits ○ 6 bits ○ 5 bits

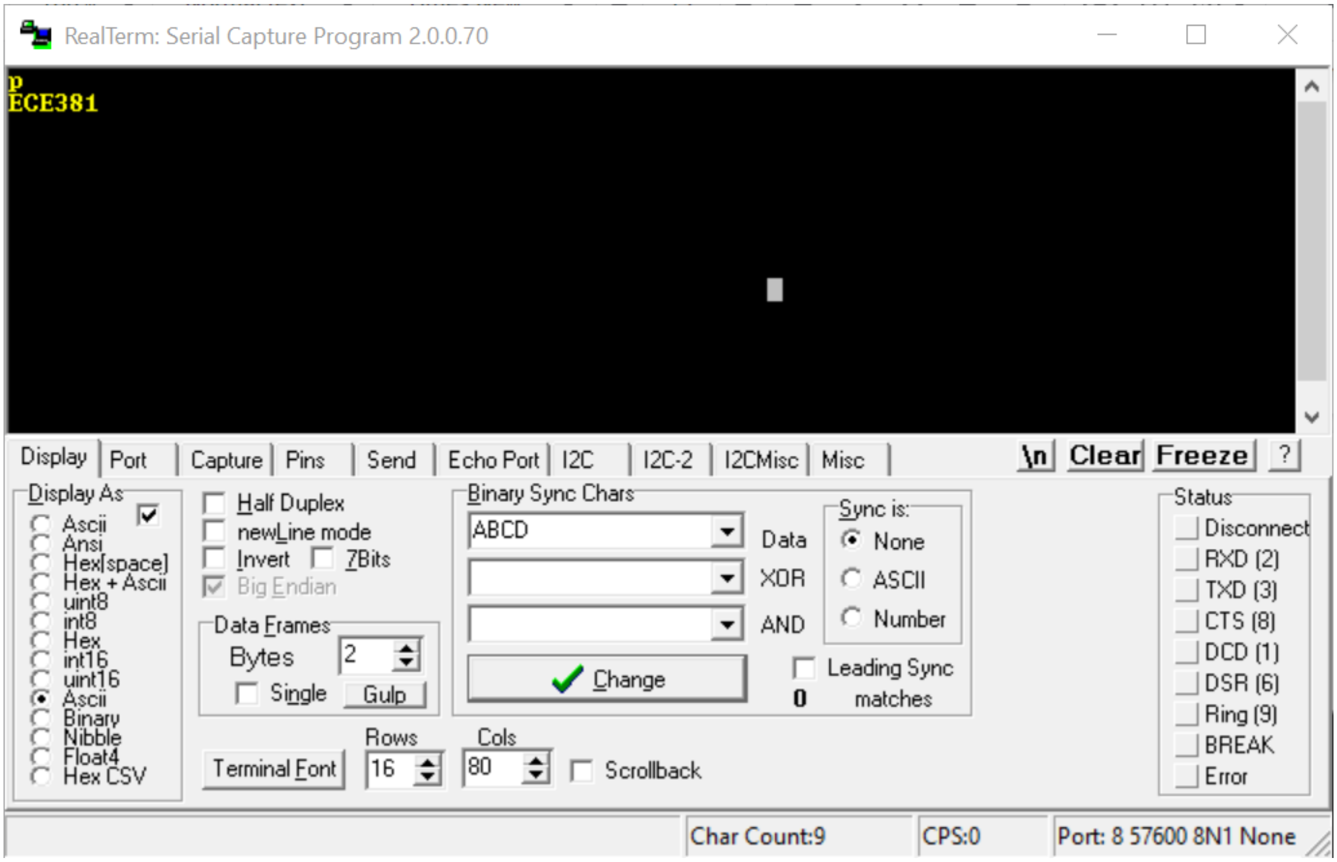
Software Flow Control: □ Receive Xon Char: 17 □ Transmit Xoff Char: 19

Winsock is: ○ Raw ● Telnet

Status: □ Connected □ RXD (2) □ TXD (3) 🟩 CTS (8) 🟩 DCD (1) 🟩 DSR (6) □ Ring (9) □ BREAK □ Error

Char Count:0 CPS:0 Port: 1 57600 8N1 None

Don’t forget to click “Change” after selecting your COM port. Anything typed in the window can be read on the microcontroller through the UART, and anything printed (or raw data sent) from the microcontroller will be displayed here. Sending chars (or arrays of chars as strings) is all that is necessary for data transfer. The example shown below has the user type the letter ‘p’ and prints out “ECE381”. RealTerm will not automatically echo any inputs, as shown here. That’s a job left up to the microcontroller programmer.



RealTerm: Serial Capture Program 2.0.0.70

```
p
ECE381
```

Display As: ○ Ascii ○ Ansi ○ Hex[space] ○ Hex + Ascii ○ uint8 ○ int8 ○ Hex ○ int16 ○ uint16 ● Ascii ○ Binary ○ Nibble ○ Float4 ○ Hex CSV ☑

□ Half Duplex
□ newLine mode
□ Invert □ 7Bits
☑ Big Endian

Binary Sync Chars: ABCD Data; XOR; AND

Sync is: ● None ○ ASCII ○ Number

□ Leading Sync
0 matches

Data Frames: Bytes 2 □ Single

Rows 16 Cols 80 □ Scrollback

Char Count:9 CPS:0 Port: 8 57600 8N1 None

The Stepper Motor

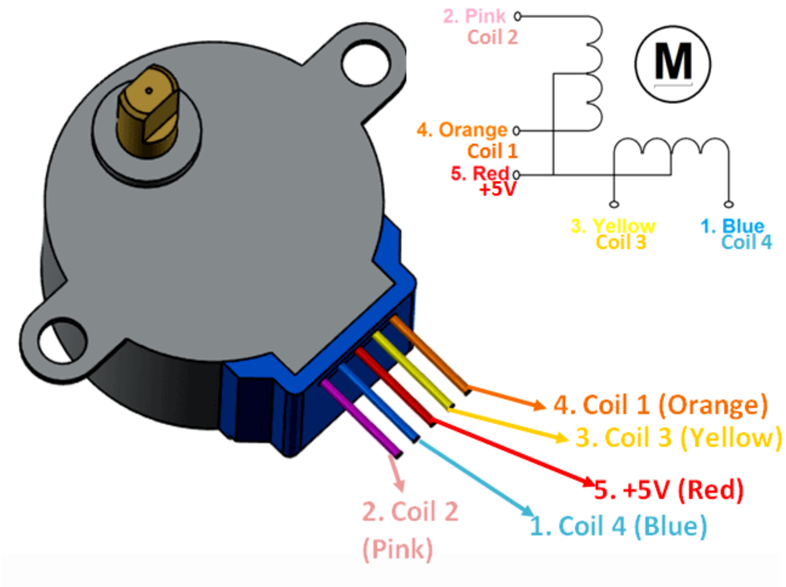


Figure 2a: Stepper Motor Wiring Diagram (28BYJ-48)

Stepper motors are DC driven motors where the rotation takes place in discrete angular steps. If the stepper motor has sufficient torque to drive the mechanical load on the shaft, this allows for open-loop operation of the motor since it does not require positional feedback. Stepper motors function by energizing the coils in a specific order to spin in a clockwise or counterclockwise direction. Doing a complete revolution requires taking a known number of steps in either direction. The speed of the motor is determined by the number of pulses per second. The stepper motors we have in our kit are the 28BYJ-48, which are geared motors. This means they trade-off rotation speed for mechanical torque. The ungeared 28BYJ-48 takes 32 steps per revolution (or 11.25° per step). Ours have a 1/64 gear reduction, so the torque is increased by a factor of 64, but it takes 64 times as many steps to perform a revolution in Full Step mode (32 x 64 = 2048 steps/rev) or (64 x 64 = 4096 steps/rev) in Half Step. Torque (and speed) also depend on Voltage. Since $V=L\frac{di}{dt}$ in an inductor, the change in current per time is *V/L*. Therefore, a higher voltage applied to the same inductance will produce a higher current per unit time. Since the torque is based on the magnetic attraction between coil and stator, and magnetic flux is proportional to current, the higher the current, the stronger the attraction, and the more torque. Therefore, a motor driven by a higher voltage can usually move a larger mechanical load, or can move a smaller mechanical load more quickly (more pulses-per-second).

The Darlington-Pair Motor Driver (ULN2003A)

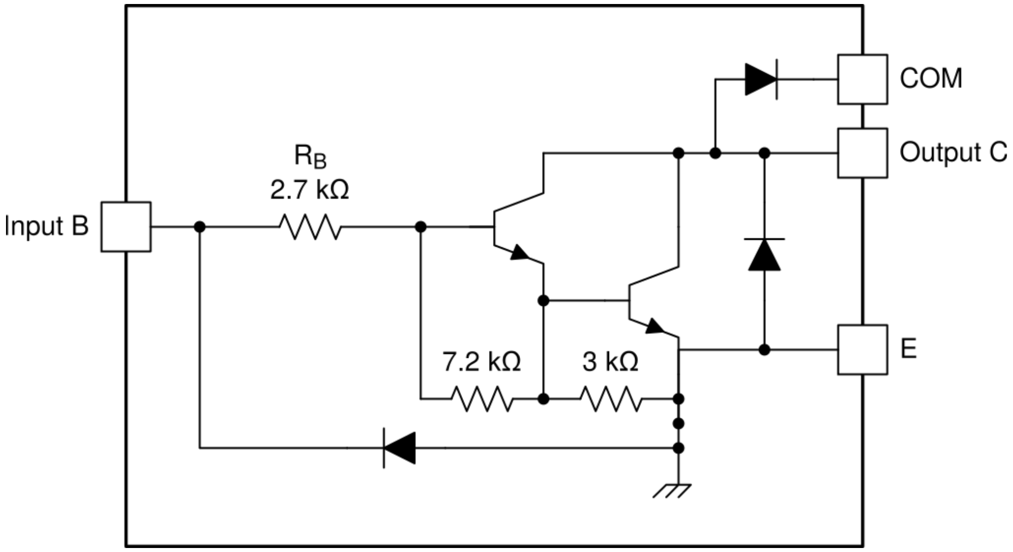


Figure 3: ULN2003A Driver Circuit

There are multiple problems with attempting to drive a stepper motor directly from a microcontroller pin. The coils are typically low-resistance, and thus most microcontroller pins cannot provide the necessary current to energize the coil and turn the motor. Secondly, in many steppers, the back EMF generated in the coils could potentially damage the microcontroller port. Further, many steppers require a higher voltage than the standard +5/3.3 V typical of most microcontrollers, and thus require a way to be driven by a separate, higher voltage supply. Thus, motors typically require some external driver circuit where a microcontroller determines the timing. For this lab, we will use a ULN2003a Darlington Pair driver.

A Darlington pair is a cascaded topology of NPN (or PNP) transistors. Using two transistors, and connecting the emitter of the first to the base of the second, results in an overall current gain of $\beta_{TOTAL} = \beta_1\beta_2$ flowing into the common-collector output (labeled Output C). Thus, GPIO pins, which have very small drive currents, can be amplified by a large amount (1000s-10000s) to provide enough current to energize a motor coil or other inductive load. Additionally, the built-in diode between the *Output* and *E* pin provides a current path for the back EMF of the coil. On top of that, the other diode between the *Output* and *COM* node provides a path for the negative field, stored on the coil, to dissipate when the coil is switched off.

The stepper must be energized in the proper sequence to take steps. There are three options for our stepper [SLVA767A]. Wave Drive, which only energizes one phase in sequence, is the simplest way. However, due to the lack of torque and precision, it is not typically preferred. Full Step Drive is the most common, as it has relatively high, constant torque. This is the one we will use for the lab. Half Step drive provides increased angular resolution due to more steps, but reduces the torque.

Here is the sequence Table for Full Steps:

	Step Sequence			
Coil Lead	`0`	`1`	`2`	`3`
$A(Blue)$	`1`	`0`	`0`	`1`
$B(Pink)$	`1`	`1`	`0`	`0`
$\underline{A}(Yellow)$	`0`	`1`	`1`	`0`
$\underline{B}(Orange)$	`0`	`0`	`1`	`1`

Clockwise rotation goes in order (0 → 1 → 2 → 3 → 0 → 1 → …), Counter-Clockwise goes in the opposite order (0 → 3 → 2 → 1 → 0 → 3 → …). Note also that the logic in the table above matches that of the Darlington in the ULN2003a. When the input to the pair is 1, that corresponds to the logic level high output of the GPIO (typically 3.3V or 5V), which causes base current, and hence collector current in the coil. When the input is 0V, the base current is 0, and the transistor is cutoff, so no current flows.

Configuration:

- Create a new PSoC Creator project titled “Lab4_UART” or something similar
- Place a “Communications/UART (SCB mode)” user module into the schematic window.
- Right-click the module and go to “Configure” or double-click it.
- Click the “UART Basic” tab if it isn’t up, and configure it to be “Full UART (TX+RX)”, “Baud rate (bps)” to “57600”, “Data bits” to “8”, “Parity type” to “None”, “Stop bits” to 1. You can leave the other parameters to their defaults. Rename the module to “UART”, and click OK.
- Go to the Pins configuration window and assign RX to P7[0] and TX to P7[1]. **DO NOT ASSIGN THESE TO ANY OTHER PINS, AS THE UART-to-USB ON THE KITPROG ONLY WORKS WITH RX ON P7[0] AND TX ON P7[1]!!!**
- Windows will assign a COM port to the PSoC somewhat arbitrarily (I think it keeps a permanent, sequential count of all COM ports it has seen and assigns new ones as the next sequential number). Make sure you select the correct one in Realterm (It will say \ USBSER000).
- For the motor, place a “Digital/Control Register” module and set the size to 4 bits. You can name it Motor, and modify its value in code with the `Motor_Write()` function. This is something of a hack to allow us to use non-consecutive pins for the motor.
- Place 4 System/Ports and Pins/Digital Output Pin and connect each to one of the outputs of the control register. According to the datasheet for motor driver, IN1 maps to Phase1_Pos, IN2 maps to Phase2_Pos, IN3 maps to Phase1_Neg and IN4 maps to Phase2_Neg. Map

these to P3[1:0] and P0[3:2]. If they share the same pins as the LEDs, that’s fine, you might even appreciate the multi-color feedback during debugging! (Though the driver board also has LEDs for each of the coil leads too)

Software

- The UART should prompt for input of a string
  - The user is free to type anything for the string
  - It should properly backspace with the rubout, and removal of the deleted character in the string
  - The user shouldn’t be able to backspace beyond the start of a line, play the BEL (0x07) if they try
  - The maximum line should be 16 characters. If a user has typed this long of a string, also play the BEL (0x07)
  - String input should terminate when the user presses enter
- There are 3 valid starting characters for the string: S, F, B
  - If S (or s, it should be case insensitive) is the first character of the string, then the user wishes to change the speed of the motor. The format of the rest of the string would be:

    S *rpm*

    where *rpm* is a floating point rpm value. You should convert the rpm into the target pulse rate for the motor after converting the value as a string into the actual floating-point representation (hint: see the atof() or atod() functions in the C standard library).
    - You should also echo back to the user the speed of the motor in both RPM and the time between steps.
  - If F (or f) is entered, then the user wishes the motor to take steps in the forward/CW direction. The format for the rest of the string would be:

    F *Nsteps*

    where *Nsteps* is an integer number of steps. You convert *Nsteps* into an actual integer, then you should take the required number of steps at this point. After the final step, de-energize the motor by setting all coils to GND. You should remember the value of the last step so that future movements can resume from that position with as little “jump” as possible after re-energizing. If you leave the motor in the last position without de-energizing it, it will get hot!
    - You should echo back to the user the number of steps and direction after they have been taken
  - If B (or b) is entered, then the user wishes the motor to take steps in the backward/CCW direction. The format for the rest of the string would be:

    B *Nsteps*

where *Nsteps* is an integer number of steps. You convert *Nsteps* into an actual integer, then you should take the required number of steps at this point. After the final step, de-energize the motor by setting all coils to GND. You should remember the value of the last step so that future movements can resume from that position with as little "jump" as possible after re-energizing. If you leave the motor in the last position without de-energizing it, it will get hot!

- You should echo back to the user the number of steps and direction after they have been taken

Debugging Tips

- Build this program in steps. Get basic character reading/writing working, then work on storing the characters in one string, then add the 16 character limit, then add the backspace/rubout capability, etc.
- Remember that everything transmitted between the computer and the PSoC is in ASCII. Among other things this means that the number 1 is transmitted as ASCII '1' which has a hex value of 0x31. Take this into consideration when the user enters the string number to be operated upon.
- In the C language, characters can be compared by using apostrophes (`val == 'W'`) NOT quotes (quotes means a NULL terminated string, so "Hello" in memory is 'H','e','l','l','o','\0').
- Remember, the '\0' (NULL, 0x00) is what defines the end of a string, NOT THE ARRAY SIZE!
- The step timing should be done with `CyDelayUs()` for greater precision in matching the target rpm value
- Don't forget to read the Errata document sections on RealTerm being unresponsive and Floating-Point Formating

ASCII Characters in Hexadecimal

```
00 NUL   01 SOH   02 STX   03 ETX   04 EOT   05 ENQ   06 ACK   07 BEL
08 BS    09 HT    0A NL    0B VT    0C NP    0D CR    0E SO    0F SI
10 DLE   11 DC1   12 DC2   13 DC3   14 DC4   15 NAK   16 SYN   17 ETB
18 CAN   19 EM    1A SUB   1B ESC   1C FS    1D GS    1E RS    1F US
20 SP    21  !    22  "    23  #    24  $    25  %    26  &    27  '
28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f DEL
```