****

Fourth of the Forth Contest Entry.

Author: Loopy Byteloose
Forth version : PropForth v5.03 for the 64K EEPROM space of a Propeller Proto Board.
Date: December 4th, 2012

****
Description - This is a supplement to the Asynchronous Serial Tutorial 3.2 for PropForth v5.03 in order to allow users to learn more and verify real hardware connections are correct. It also is presumes the reader needs help in planning what to do, attempts to suggest some creative uses, and actually assigns optional homework.

Not only were the examples written, but all were tested for their results.

What was attempted -

> Verify the PropForth's asynchronous serial interface with the real world via connection to another Propeller device that emulates a VT100 dumb terminal. There are at least two VT100 emulation applications available the Propeller chip. Two were tested and the more reliable was selected.

> Determine the possibility of using an asynchronous serial interface with one pin shared as input and output in a bi-directional mode to a half-duplex device in connection with Parallax's Serial Inkjet Printer Board.

(Note: If the reader does not quite understand Full-duplex, Half-duplex, and Simplex communication modes, some reading up on them is advised. There are differences in their behavior that require different solutions in software, and sometimes hardware.

A Full-duplex device generally communicates over separate Rx and Tx lines quite nicely with another Full-duplex device in both directions without interruption. One does not have to always mate Half-duplex with Half-duplex, but a Full-duplex to Half-duplex connection will result in the Full-duplex having to wait for a prompt to assure the Half-duplex is ready to receive. And a Simplex device can easily communicate in either direction with a Full-duplex device. It is just that there is no means to confirm that the listening device is active or present from the transmitting device. Simplex to Half-Duplex is most likely prone to failure as Half-Duplex is not always ready to listen. With hardware hand-shaking imposed, all combinations work much better.)

What is included -

>Preliminary material.

        - Some review of the original tutorial with additional observations about the PropForth's asynchronous serial support.

        - An explanation of how to acquire, install, and set up a VT100 dumb terminal emulation program for use in the exercises. The same dumb terminal may later be used in an actual PropForth application to make a cost effective permanent installation - the end user would have a keyboard, a VGA, and a Propeller Proto Board instead of a full computer with hard disk, OS, and other costly features. This is appealing to industrial, bench work, and lab installations.

>Exercise One

        - Set up, configure, and demonstrate using PropForth inserted between the default Console_Serial and another Serial port that reaches the outside and onward to a dumb terminal. The other Serial port is connected to the VT100 Terminal and simultaneous full-duplex is explored to verify full function.

>Exercise Two

        - Set up, configure, and demonstrate using PropForth with an independent Serial port linked to an independent cog while PropForth operates through its Console_Serial with another entirely independent cog.

        -After the configuration of both is achieved, the Console_Serial will be used to download software routines available to the 'independent cog' which will reach the outside dumb terminal.

        -Show how specialized words may be created in the shared dictionary in hub ram and executed on the 'independent cog' in a remote fashion by the Console_Serial using the cogx word.

What is incomplete -

> There is NO solution provided for the now notorious bi-directional half-duplex that I was previously trying to resolve in Forth on the Propeller with my chopstick printer project. Nonetheless, I suspect I have a safe and sane solution that may later be demonstrated.

Having one-wire provide communications in both directions may best be resolved by inserting a logic chip - a 74LVC175, known as a Quad 3-state buffer to control direction. This will require more code and the use of two more i/o pins, but seems right to me. I have not yet verified this. I have to purchase components, build, and test. Only two sections of this quad device would be used.

Inserting of the LVC generation of logic is also an opportunity to easily resolve changes from a +3.3v logic to +5.0 logic in both directions. These chips are an important improvement is logic chips as the Propeller 2 will also be 3.3v logic and we all want to use our legacy +5v hardware as long as we can.

This chip - the low 3.3 voltage operation, 5 volt tolerant logic i/o version - will accept 3.3v power while taking 5.0v TTL level inputs and outputting a safe 3.3v logic to the Propeller i/o. And it seems that in the other direction to output a high enough logical 1 to be accept by downstream +5.0 logic. If you need an all 3.3v solution, it will also accept 3.3v logic inputs, apply appropriate control, and output 3.3v. But, be aware that it seems only available in surface mount devices, not DIP packages. These tend not to be available via over-the-counter retail electronics shops that are still clearing DIP inventory.

> Having followed the style and sequence of the original PropForth tutorials for the sake of continuity, I see now that another revision might offer this information in a manner that could be a bit easier for the reader. Both style and content may be revised to accommodate Sal Sanci's preference or whomever he delegates such matters to.

                In other words..... This contest version may not be the final version.

> The original tutorial is rather unclear that the internal communications link from cog to cog is in a byte-parallel transmission mode, while the connection to the outside world is asynchronous serial. There also seems to be another useful internal serial mode, MCS or Multi-Cog Serial and that may be synchronous serial in a Master/Slave configuration, but I have not yet explored that tutorial and its uses. There is a Tutorial 3.3 MCS Loop back, that presents it.

TRIVIA -
        - Only Minicom on Ubuntu Linux has been use verified, no Teraterm in a Windows OS. The author used Minicom on Ubuntu Linux as Teraterm (referred to in the below text) is a Windows application and not generally available on Linux. Since Teraterm is consistently referred to in PropForth tutorials, that convention is followed without verifying on an actual Teraterm installation.

        - The style becomes narrative to keep the reader interested and to help the reader notice and recall what is being presented.

!!!!!!!!!!!!!!!!!!!! Below this line is the complete document that should be read as a finished tutorial !!!!!!!!!!!!!!!!!

```
********************************************************************
***** PREAMBLE *****  NOTICE - ADDED Tutorial to 3.2 Serial Loop back
********************************************************************
            HAVE YOU READ THE ORIGINAL TUTORIAL First?
********************************************************************
```

Author: G. Herzog, aka Loopy Byteloose
Date: Dec 4, 2012
Version: 0.91
~~~~~~

This is added material that demonstrates real world asynchronous full duplex serial connections with PropForth v5.03 in actual use. This text assumes you have studied in detail the original tutorial
~~~~~~

first. And this material provides useful practical configuration skills and helps the user clearly recognize additional useful features of asynchronous port use on the PropForth v5.03.

To study the material below requires.....
the use of an additional Dumb Terminal including another keyboard and another video screen which are connected differently from the original tutorial in two exercises toward the end of this text (For Exercise One, refer especially to sections 7,8, and 9 and for Exercise Two, sections 10 and 11).

The original PropForth tutorial presents an all internal solution convenient for the user who only has one Propeller available and that does not directly demonstrate to the user that connections to an external device are working; so these exercises are designed to confirm external hardwire configurations for the sake of the ubiquitous need to troubleshoot them.

This tutorial will eventually enable the user to use PropForth with a dumb term. Some finished applications may require a video screen and a keyboard, but not a complete suite of computer support at all times. PropForth v3.4 has a JupiterAce project which is a possible alternative solution, if it were to resolve communications with a second and possibly ROMless Propeller.

If the reader does not have an actual Dumb Terminal, but has more than one appropriate Propeller device, he/she can create one with a second Propeller board --- such as with a Parallax Proto Board, plus a keyboard and a VGA. Software is easily available.

There are

PockeTerm software from Briel Computers
http://www.brielcomputers.com/wordpress/?cat=25

Or, OddBitCollector's VT100 Terminal emulator from Jeff Ledger
http://n8vem-sbc.pbworks.com/w/page/4200926/Propeller-VT100-Compatible-Terminal

The author has decided to use and recommend...OddBitCollector's VT100 Terminal emulator at this time.

PockeTerm v0.905 was tested and seemed to have difficulties in booting that are unrelated to PropForth. Because it requires both of its provided serial ports to remain active for it to boot properly and to perform well, it was abandoned as a possible solution. The second serial port made configurations much more complex and contributed to some unexplained crashes.

Many Propeller boards other than the Propeller Proto Board can just as easily provide this VT100 emulator service IF they have keyboard and VGA interfaces provided or the user has the choice of buying the PockeTerm board from Briel Computers which will work with its own or other software.

The communications in this tutorial are mostly 57600 baud 8N1 for the sake of simplicity, but toward the end of the tutorial, the Dumb Terminal and its serial link in the Propeller are changed to 9600 baud 8N1.

Other baud rates are supported by the VT100 Terminal software and by PropForth. But it seems at this time that PropForth only supports the 8N1 configuration - no 7 bits, no parity checking, and only 1 stop bit. Hardware handshaking can still be made available in PropForth and possibly software handshaking - but software handshaking has never been easy to work with and may best be avoided. Many authors have presented demonstrations that software handshaking is unreliable. 7 bit code, parity checking, and additional stop bits may all be considered as relics of older asynchronous hardware with slower baud rates, but when you need these you will have to find another solution or wait for PropForth to provide.

In addition to PropForth v5.03 EEPROM .spin image loaded into a first Propeller Proto board, for these tests the author is using a set of .spin files software from Odd Bit Collector's VT100 Terminal emulator loaded into the second Propeller Proto Board as a Dumb Terminal. When changing the Dumb Terminal baud rate, the VT100 Terminal requires an edit, compile, and reload. Baud rates changes apparently cannot be done from the active Dumb Terminal's keyboard.

While the author's board has been adapted to true RS232 i/o signal levels via an added MAX232 level shifter, the 3.3v logic i/o is also directly available and the available MAX232 level shifter will not be used and is quite unnecessary. Using the MAX232 on one board requires the additional use of another MAX232 level shifter on the second board.

The text below starts out with duplication of introductory tasks as presented in the original tutorials, attempts to faithfully follow the style of those tutorials, and then changes direction to present new added information from a familiar point of departure.

Just read through once for a preview before you try to do things and it should be easy to follow.

~~~~~~
0. Check
README.txt ,
~~~~~~

```
GettingStarted.txt ,
doc\MultiPropTestSystemHardware.txt
for any background and configuration information.

1. Run tutorial-1.1 Standard Development System.txt

2. Start Teraterm.

+++
Reboot
---
reboot

###
reboot

CON:Prop0 Cog0 RESET - last status: 0 ok

CON:Prop0 Cog1 RESET - last status: 0 ok

CON:Prop0 Cog2 RESET - last status: 0 ok

CON:Prop0 Cog3 RESET - last status: 0 ok

CON:Prop0 Cog4 RESET - last status: 0 ok

CON:Prop0 Cog5 RESET - last status: 0 ok

CON:Prop0 Cog6 RESET - last status: 0 ok

Prop0 Cog6 RESET - last status: 0 ok
Prop0 Cog6 ok
%%%



+++
Define_term
---
: term over cognchan min ." Hit CTL-F to exit comterm" cr >r >r cogid 0 r> r> (iolink)
begin key dup h6 = if drop 1 else emit 0 then until cogid iounlink ;

###
: term over cognchan min ." Hit CTL-F to exit comterm" cr >r >r cogid 0 r> r> (iolink)
begin key dup h6 = if drop 1 else emit 0 then until cogid iounlink ;
Prop0 Cog6 ok
%%%

AUTHOR'S COMMENT ON term  -- An optional homework assignment - HOMEWORK #1

'term' remains entirely the same.  Usage requires the cogid number (n1) and cog channel number (n2) be
provided to target the connection --> term (n1 n2 ---   ).

If you haven't already done so, it might be wise to study 'term' word by word to see exactly how it
operates to (a) divert the input from Cog 6 to Cog 2 via (iolink) and (b) how the duplication of input
is set up in a loop on Cog 6 to trap a <CNTL-F> that causes it to evoke iounlink.

It is as advisable to determine why the word 'term' is sending to and getting configuration data from
the RS stack and not just the data stack, why it uses (iolink) rather than iolink, and why the iocomm
or (iocomm) words were NOT chosen for this configuration. And please note that PropForth does not use
the parenthesis for comments that are ignored by the interpreter.  Instead, it seems that the
parenthesis indicate versions of words that are considered internal and mainly called by the words
that are without parenthesis, though it seems in some cases that it is handy to directly use them.

h6 is equal to <CNTL-F> in hexadecimal, but since 6 is the same in hexadecimal, decimal, and even
octal, it is merely a style choice of the programmer to clarify hexadecimal.


We start with the conventional default configuration in Run_cog?_1



+++
Run_cog?_1
```

```
---
cog?

###
cog?
Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)
Cog:7  #io chan:1                           SERIAL  7(0)->6(0)
Prop0 Cog6 ok
%%%
```

3. Explanation:

In the original tutorial, this section presented us details about the report generated by cog? queries -- the status of each cog, and where it's interior byte-parallel io is connected, and so on. That information is in the original tutorial, so there is no need to repeat it if you have done the original tutorial first. If not, go back and do it. The SERIAL cog actually converts the byte-by-byte parallel data into serial data. This may in some cases be a source of delays, but none were observed.

It is expected that you have learned the significance of each item in the cog? report.

There are additional items that should be pointed out -- what the cog? report does not include. The cog? report does not indicate the dira status of each external i/o pin in each cog or the actual assigned baud rate of SERIAL cogs. There are times when the user might feel these may require verification, but it seems that there is no way to verify. When in doubt, just reset the cog and repeat the serial configuration with fresh data. Trying to reach a SERIAL cog with >con will freeze the Console_Serial and cogs running independent applications may run without the Console_Serial being able to reset or interrupt. The only solution is a hardware reset.

**************** BELOW the Tutorial begins to explore new configurations *********************

4. Start a serial driver, pin 0 Rx, pin 1 Tx, 57600 baud, use cog 1

```
+++
Run_startserialcog1
---
c" 0 1 57600 serial" 1 cogx    <==== This REMAINS THE SAME as before, cog 0 and cog 1 will configure
the same as before.

###
c" 0 1 57600 serial" 1 cogx
Prop0 Cog6 ok
%%%
```

NOTE --- We have started with what you already know and evolve toward further new and unfamiliar information.

5. Start a second serial driver, pin 2 Rx, pin 3 Tx, 57600 baud, use cog 2

```
+++
Run_startserialcog2
---
c" 2 3 57600 serial" 2 cogx   <==== Please note New Pin numbers,the previous cross-connected i/o pins
ARE NO LONGER

###
c" 2 3 57600 serial" 2 cogx
Prop0 Cog6 ok
%%%
```

```
+++
Set_flags
---
1 1 sersetflags 1 2 sersetflags  <==== Remains the same as before, each serial port needs to set
active flags

###
1 1 sersetflags 1 2 sersetflags
Prop0 Cog6 ok
%%%
```

The user can turn off serial port activity with sersetflags by using 0 for n1 and the cogid for n2. Also there is a verification word that can be used, serflag? and it will return the actual status. The 'serflag?' query only requires a cogid for n1 on the top of the stack. Though the word sersetflags is plural, it seems only the first bit is used.

```
+++
Run_cog?_2
---
cog?

###
cog?
Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:1  #io chan:1                           SERIAL         <==== Referred to as Serial_A, but not
linked at this time
Cog:2  #io chan:1                           SERIAL         <==== Referred to as Serial_B, but not
linked at this time
Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)
Cog:7  #io chan:1                           SERIAL  7(0)->6(0)  <==== Referred to as Console_Serial
and linked to Cog 6
Prop0 Cog6 ok
%%%
```

6. We can now see that cogs 1 & 2 are running serial drivers. BUT we also know that the Rx/Tx pins from cog 1
are NO LONGER cross-connected to the Rx/Tx pins of cog 2 as they were in the original tutorial.

We will now link cog 0 to cog 1 as before using iocomm, but it cannot be reached in the same manner as done in the previous tutorial text. Exercise Two will explain how to reach cog 0 for Forth program control.

Pin configurations are as follows for TWO independent Serial ports which require external connections:

        Pin 0 is Rx and Pin 1 is Tx for Serial_A on cog 1 and
        Pin 2 is Rx and Pin 3 is Tx for Serial_B on cog 2

These pin specifications are important as tests in the exercises below will connect directly to the Dumb Terminal's 3.3v i/o serial. Be sure to include a third connection for ground. As usual, the 3.3v serial is inverted output and by adding a MAX232 or transistor interface, the signal is converted to non-inverted standard RS232.

In any event, rather than provide two unnecessary MAX232 level shifters with related inversions, it is easiest for bench work to just connect 3.3v i/o from one Propeller to 3.3v i/o in the other and avoid a lot of unnecessary construction.

We are referring to the connections as 3.3v i/o because true TTL is 5.0v i/o logic. (There is the term LVTTL for 3.3v, but the Propeller does not really use this reference term as such alphabet-soup descriptions get rather obscure and confusing.)

Never connect a Propeller directly to 5.0v i/o without knowing which direction the data is being transfered and be aware that by-directional 5V connections will require additional interface hardware. Direct output to 5.0v i/o is okay, but input from 5.0v i/o requires a 3K ohm or higher resistor in line to protect the Propeller's i/o pin from damage.

The main points are that

(a) we are using 3.3v logic directly connected and
(b) the inverted output canceled itself out when the RS232 level shifters are excluded.

```
+++
Run_ioconn
---
0 1 ioconn  <==== Stays the same as before - Cog 0 and Serial_A on Cog 1 are now linked to each other.

###
0 1 ioconn
Prop0 Cog6 ok
%%%
```

```
+++
Run_cog?_3
---
cog?

###
cog?
Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  0(0)->1(0)
Cog:1  #io chan:1                           SERIAL  1(0)->0(0)   <==== This is Serial_A in a loop with
Cog 0
Cog:2  #io chan:1                           SERIAL               <==== This is Serial_B without
connections
Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)
Cog:7  #io chan:1                           SERIAL  7(0)->6(0)   <==== This is Console_Serial
%%%
```

We now see cog 0 is connected to the serial driver, referred to as Serial_A, running on cog 1 which will not be used until Exercise Two in Section 10. If for some reason you crash the PropForth or desire to reboot the PropForth, you will have to re-enter this desired configuration to begin to do Exercise Two.

The Serial_A driver is assigned to Rx on Pin 0 and Tx on Pin 1 on cog1 which is connected internally to cog 0 and CANNOT be reached from Serial_B without hardwire jumpers.

For now, that should be noticed as a significant difference from the original tutorial, that is all.

Another optional homework assignment - HOMEWORK #2
===
If you wish to later try to reach cog 0 to have it loop-back,
Serial_A Rx on Pin 0 requires a jumper to Serial_B Tx on Pin 4 and
Serial_A Tx on Pin 1 requires a jumper to Serial_B Rx on Pin 2.
===

If you are following this tutorial from start to finish, do not connect these jumpers as part of the study. Just consider this a suggested optional homework exercise after you have completed the tutorial exercises. We are trying to connect to other things for practical configuration and trouble shooting purposes, not to observe circles running within the Propeller itself. PropForth provides the internal byte-by-byte channels for such communications and only really needs the asynchronous serial for reaching outside devices.

At this point, you should notice that cog? provides the same information as provided in the first tutorial. But we clearly know that cog 1 and cog 2 are no longer cross-connected. As mentioned above, that information is nowhere to be seen. And we cannot confirm the baud rate.

The only possible way to investigate the differences in pin configuration might be to read directly from the Pin direction register -- called dira -- in cog 1 and cog 2 to determine the actual differences.

But since SERIAL is running on cogs 1 & 2 and not PropForth there is no way to do so. Attempts to reach the dira registry will crash the Propeller's PropForth, require a complete reboot and reconstruction of the desired configuration. So it is critically important that the user is sure that the 'serial' word is supported with the right i/o pins and the right baud rate.

Aside from those items, the above cog? report shows everything is ready for use. And if you followed instructions exactly, the i/o pins and baud rate are right. If you are unsure, you can reset the cogs and reconfigure.

**************** The NEW Exercises **

7. Exercise One - Your computer to Console Serial to Serial_B to a 57600 baud Dumb Terminal

You are ready to connect the external Dumb Terminal to Serial_B via Pin 2 and Pin 3 in an appropriate fashion via hardwire. It is best to have the Propeller board off to do so. Hot swapping may cause damage to the Propeller's i/o pin. This is somewhat of a dilemma. If you turn off both the Dumb Terminal board and the PropForth board you will be much safer, but the PropForth configurations will be lost and you will have redo portions of sections 1 through 6. If you don't turn anything off, you might get away with it. But one day, this bad habit may really damage something.

This should be a cross-over connection of the PropForth Propeller Pin 2 Rx to the Dumb Terminal Propeller Tx pin (which you determine in the top .spin file along with the Dumb Terminal's baud rate) and the PropForth Propeller Pin 3 Tx to the Dumb Terminal Propeller Rx pin (again determined by you). The Dumb Terminal also needs to be configured to 57600 baud 8N1 in its top .spin file, compiled, and loaded.

To avoid issues with different terminal voltages, the easiest construction is to use another Propeller that also has 3.3v i/o. Other VT100 emulators with +5 logic require a few additional items.

But there are ways to adapt to +5 i/o logic to and from 3.3 i/o logic or to include a MAX232 level shift to connect to a Dumb Terminal that only accepts true RS232 voltages in proper standard form. The method of connecting to +5 i/o logic was mentioned above. Building a level shifter interface is not discussed herein for the sake of brevity.

If you are sure you have SERIAL_B ready in cog 2 and Console_Serial on Cog 7 is cross-linked to Cog 6, you are ready to run the phrase... 2 0 term.

After running 2 0 term it is necessary to hit enter a couple of times to bring up the connection.

```
+++
Run_term
---
2 0 term

###
2 0 term
Hit CTL-F to exit comterm

(Display may provides nothing without actual keyboard activity from the Dumb Terminal , you will not
see Prop0 Cog0 ok and do not expect a prompt.)

%%%
```

8. We now have the Console_Serial internally connected through the Serial_B driver on cog 2 via the Rx/Tx i/o pins, which are Pin 2 and Pin 3 respectively. One can no longer run cog? to gain a cog? report. Data received from the Dumb Terminal to to Serial_B Rx Pin 2 and is then passed on to the Console_Serial on cog 7. Data transmitted from your interface to Console_Serial is still passed to Cog 6 where term is actively awaiting a CNTL-F to disconnect, but as long as there is no disconnect, term passes data on to Serial_B and to the Tx pin 3.

```
+++
Run_cog?_4
---
cog?

###

( Nothing!!! This merely sends cog? to the Dumb Terminal that does not't understand the word. After
all, it is dumb and is not a PropForth cog. And similarly, keyboard entry of cog? on the Dumb Terminal
will arrive at your computer screen without any report being generated.)
```

```
%%%



One can attempt to check the dira register in a SERIAL cog, but it just makes a big mess.



+++
Checking the dira or Cog 1 -- WARNING, this will crash everything. You can try it or skip it
---
1 >con dira COG@ .
###

(The PropForth freezes, no way out but to reboot and completely reconfigure!)

%%%
```

9. As we know, the term program running on cog 6 has connected its output to the input of the serial driver called Serial_B running on cog 2, and the output of the Serial_B driver running on cog 2 is going to the input of the Console_Serial driver on cog 7, through the Propclip and arrives at the Teraterm display in your computer.

In other words, everything is ready to test for good communications from your computer to the Dumb Terminal.

The main reason to do this is to see if what you type on your computer outputs correctly on the Dumb Terminal's screen; and that what you type on the Dumb Terminal's keyboard outputs correctly on your computer screen. Cursor control is as important as typing letters and numbers, please verify. Also you need to see if Delete and Backspace are working right.

AT this point, if you are having trouble with the display of your Dumb Terminal or its keyboard output to your computer, check your wiring. It may be loose or in someway defective.

Just try using your Dumb Terminal and see where the information arrives. And then try your usual input to the PropForth and see where the information arrives.

Because, the keyboard output from Teraterm goes through the Propclip, then through the Console_Serial driver on cog 7, which is connected to the input of cog 6; the term program running on cog6 routes these bytes to the input of cog 2 until a CNTL-F is hit.

After you have verified that the communications link is working, you can disconnect it by using CNTL-F. And if you wish to reconnect, you can simply again use the phrase, 2 0 term and you will reconnect. You may have noticed that the 'term' word actually finds out for you that you are in Cog 6 and uses that information to help you use (iolink) since (iolink) requires four parameters on the stack and you provided only two. And you may have also figured out that iolink and (iolink) do not disconnect anything, while iocomm and (iocomm) disconnect a previous configuration as well as setting up a new one. In some cases, it is useful to have iocomm and (iocomm) do housekeeping for you; in other cases, it is not.

```
+++
Hit_CNTL-F
---
(CNTL-F)
###

(.... Whatever was serial reception from the Dumb Terminal remains, but we also see a new item...)

Prop0 Cog6 ok   <====  term is finished an the connection reverts to Prop0 Cog6 ok as its prompt

%%%



+++
Run_cog?_5
---
cog?
###
```

```
Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  0(0)->1(0)
Cog:1  #io chan:1                           SERIAL  1(0)->0(0)
Cog:2  #io chan:1                           SERIAL
Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1
Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)
Cog:7  #io chan:1                           SERIAL  7(0)->6(0)
Prop0 Cog6 ok
%%%
```

At this point, we believe your Dumb Terminal's keyboard has proven able to send serial to your computer screen at 57600 baud; and your computer was proven able to send 57600 baud serial to the Dumb Terminal's screen when you evoke the 'term' word, and you were able revert to Prop0 Cog6 by Control-F.

You have confirmed a real world communications connection with the two devices in full duplex.

While this verifies you have reached another device and everything is working right, the exercise requires human input on each end. And that is not always wanted, needed or required with micro-controllers. If communications are not right, but they are present - you may have to reconfigure the Dumb Terminal and its related Serial_B to a lower baud rate where problems don't show up. This is exactly why we are doing a hardwire setup -- to determine that the baud rate is not too high for the construction you have created.

Often one needs a micro-controller to provide an automated communications with another device without human interaction. The true RS232 serial link can easily accommodate 20 feet of wire when used with MAX232 level shifters. You may even go longer distances but at some point you may have to use slower baud rates to maintain accuracy. Or you can use RS485 line drivers instead of MAX232 level shifters and go kilometers over a twisted pair of wires. Or if you don't like wires, you can use a wireless full-duplex setup; such as XBee or Bluetooth.

In the next section, we will explore an automated PropForth cog by reconfiguring the Dumb Terming to reach Cog 0 via Serial_A on Cog1 and through Pin 1 and Pin 2 and then, we will be installing a simple software application that automates the PropForth's Cog 0 communications.

We will still require a human operator at the Dumb Terminal to confirm operations. The Dumb terminal will be able to reach the Fourth dictionary in hub RAM and any words put there for it can be easily called to run on any cog. So it is also feasible to eventually remove the Dumb Terminal and provide another automated device that is programed to handshake and a exchange data with the PropForth Propeller without any direct human control or input.

An actual setup without any human input will not be demonstrated in this text. People have many different devices with different requirements.

No human input really is just a matter of setting up the PropForth Propeller in the way you personally want to and the examples included in this tutorial are quite adequate help you to do so. More over, it is possible to use many serial devices; such as serial LCD displays, serial GPS, text-to-speech and so on. All can be exploited with the Propeller in PropForth once you have loaded into the Forth dictionary in hub RAM the right routines as words to configure and activate. These attempts to provide just enough examples that will get you into actually deploying PropForth on a stand-alone Propeller yourself that is not operator dependent.

10. Second Example -- Configuring, Testing, and Programming for an independent cog 0 Forth machine connected to Serial_A (or Serial_B) to an outside Dumb Terminal. Changing baud rate to and from the Dumb Terminal.

Using >con to put words onto the cog 0 stack destroys the established link from cog 0 to Serial_A on cog 1.

You can try it, but run a cog? report after you do so and see what you have messed up and need to rebuild. Above, it was mentioned that both ioconn and (ioconn) disconnect the established link when they try to create another; >con is another Forth word that does this kind of reconfiguration. All these words will certainly not work for setting up routines in an established communications configuration as they first disconnect existing links before connecting new ones. Another method that is non-destructive is required. You need to maintain the links and insert your words to run it.

You might try iolink or (iolink), but this can more easily be done by writing a new word to the Forth dictionary and then
(a) having the Dumb Terminal operator type it in, or

(b) by use of the cogx word to reach cog 0 without breaking the link.

When cogx is used, words can be inserted by using the example phrase below. This method, rather than trying to create an interactive link is preferable as you can write your code in a .txt file that is saved on your computer and these words can be put into the PropForth dictionary with a fast load.

(PropForth uses .txt files, but usually changes the extensions to the .f extension so that you know they are PropForth and not something else.).

In this way, you don't have to have an active keyboard to input a Forth dictionary word. You also can use cut and paste from your .f files to insert your bits of code as you wish. You don't always have to fast load an entire file. A lot of development is achieved in using bits and pieces from various .f files.

An interactive keyboard link might seem attractive in some situations, but you eventually have to disconnect that link as well and keyboard errors might crash everything. The process becomes more complicated than is really necessary. This way is less likely to upset your communications configurations. And, you have saved .f files of preferred words and communications configurations to save time at the keyboard.

We will use the following typical phrase to reach cog 0 from the Console_Serial:

```
                c" . ..... a one or more Forth words ...... ." 0 cogx
```

The cogx word is a very important one because allows us to send a few words to a cog without interrupting the communications links that are established by the internal byte-parallel connections. And, we need those links in good shape to eventually reach a SERIAL cog and the outside world.

After creating a reliable routine as a Forth word, you are almost ready to test the communications routine is working properly from the Dumb Terminal that has been cross-connected to Rx at Pin 0 and Tx at Pin 1. The Dumb Terminal is used to mimic your target serial device. But you have to type the messages that the PropForth Cog 0 is expecting to see. What the device expects to receive will be shown on the Dumb Terminal screen - except for the hidden-ASCII codes.

But first, some reconfiguration is required as we are still have our Dumb Terminal wired to Serial_B via Pins 2 and 3. We need to move our Dumb Terminal connects to Pin 0 and Pin 1 as we are using Serial_A, not Serial_B. We won't be using term, but we will need the Console_Serial to be actively connected to Cog 6. Cog 6 will be sending the cogx instructions.

By the way, are you 'hot swamping' your i/o pins again?. The Propeller is a MOSfet device and you really should NOT get in the habit of swamping wires around while the Propeller is on. In fact, it is ideal that you use anti-static procedures to make sure you don't zap something.

(OR, if you are really smart, you could just keep the wiring the way it is and move the i/o Pins 2 and 3 to Serial_A as the wires are already connected there. Also remove Serial_B with a reset of cog 2 as we want to remove its duplicate links to i/o Pins 2 and 3.)

Here is the code for the alternative solution of moving serial port Tx and Rx and not rewiring the i/o pins. This is a lot easier than moving the wiring and a lot safer than hot swapping. It really is a rather special interactive ability of Forth on a Propeller.

```
===================================

c" reset " 2 cogx
c" 2 3 57600 serial" 1 cogx
1 1 sersetflags
0 1 iocomm

===================================
```

And try a cog? report to see that everything is quite right.

After you have completed either moving wires to pins 0 and 1 or switching to Pins 2 and 3 with Serial_A to the Dumb Terminal's Tx pin (determined by you) and the Dumb Terminal's Rx pin (also determined by you) verify that the Dumb Terminal is working, hit return and you should get Prop0 Cog0 as a prompt on your Dumb Terminal.

This is something new --- Prop0 Cog0 ok. You actually have two independent consoles operating at the same time on one Propeller. And both share the same Forth Dictionary.

If this is working, see if you can also request a cog? report from the Dumb Terminal as all the active cogs use the same dictionary.

Do you see something strange? If the answer is 'no', you can just continue using 57600 baud.

But if your answer is 'yes', welcome to the world of real trouble shooting. We have a problem and you are going to have to get cog? reports from the Serial_Console until this is solved.

In some cases, there seems to be a problem with the display of multi-lined reports with columns. But you may be able to get one line input or output clearly.

What is the solution?

Reduce the baud rate until the serial is working properly. You can do this in steps, but one big jump will make this tutorial a lot shorter. Let's reconfigure to 9600 baud. The code below is setup for pins 2 and 3 as Rx and Tx respectively.

```
===================================

c" 2 3 9600 serial" 1 cogx
1 1 sersetflags
0 1 iocomm

===================================
```

Can you see anything, or did you forget that you have to reconfigure the Dumb Terminal to 9600 baud 8N1 as well? That means opening the .spin file, editing the software rate, doing a complete recompile, and EEPROM reload.

But you don't have to change your Teraterm baud rate or your PropForth software, that can stay at 57600 baud because the USB port wiring is more stable. And the Console-Serial can stay at 57600 baud serial. You will just have the Serial_A running at a slower baud rate along with the Dumb Terminal.

Do you have a good cog? report now? Can you also get a good words report on the Dumb Terminal? If not, use an even lower baud rate. High speed serial requires excellent wiring and your bench wiring may not be good enough for it.

11. Creating a resident routine on Cog 0 to communicate with external serial boards, including handshaking via prompts.

This is the final portion of Exercise Two and the end of this rather long tutorial.

Its purpose is to satisfy yourself that you can develop any needed software to interface with an outside asynchronous serial device as follows:

(a) The connected device does not need to be reprogrammed to evoke Forth words (More often than not it is impossible to reprogram such devices. You usually have to follow their protocol.)

(b) Capture a simple prompt character to indicate a ready to receive condition, and

(c) Show that the device connected will receive a well-formed character string that is supposed to be processed by the external device.

We need a typical example and Parallax has an EMIC2 Text-to-Speech device that requires a simple communication exchange to work. So we will use its method as an example.

Here is what actually needs to be done.

1. Emit a <CR> in order to get a > as the Propeller will have not been ready. Usually a CR or a <SPACE> are safe characters to use. You can always ask for another if the Propeller has missed the prompt.

2. The EMIC2 device sends a > as a prompt to tell the PropForth it is ready and please send a string of characters, then it waits to receive. We need the PropForth cog 0 to verify the > prompt has arrived before it sends anything.

3. The PropForth will respond with a packet of text framed by a <STX> and <ETX> So let's just use

<STX> Hello <ETX> and see what the Dumb Terminal does with those special hidden ASCII characters that are verify important.

We can write a couple of simple words and have the Forth dictionary hold them, let's name the first word GOTit and the second one SENDit.

```
        GOTit will first send a <CR>, then receive the serial byte and confirm it is equal to the >
and then evoke a SENDit just once,
        SENDit will just send the phrase <STX> Hello <ETX>
```

We are using a short phrase, just 9 ASCII characters. Why? Well, PropForth has a limited data stack. If you want to send long text messages, you might have to load and reload the PropForth data stack more than once. Do you know how many cells the PropForth data stack has? If not, look it up.

It is easiest to first make sure our output is right and then use it to develop the trigger from input. So we will create SENDit first. We can then clearly test its received behavior on the Dumb Terminal.

```
: SEND it   ......... test test code here ......... ;
```

We obviously need either the decimal or hexadecimal for >, for <STX>, and <ETX>. We have a PropForth word for <SPACE> but well will just use the ASCII code and emit.

```
           >     is 62 in decimal, or h3E
        <STX>    is  2 in decimal, or h02
        <ETX>    is  3 in decimal, or h03
        <SPACE>  is 32 in decimal, or h20
```

You have to look these up in an ASCII chart or use a different PropForth method. Sometimes emit is very handy, mostly with hidden ASCII codes that are not shown on a display but need to be included in the message text.

We need still more ASCII codes, the letters for Hello. If you use the hexadecimal, only capital letters A through F will work, lower case a through f will be ignored.

```
               H  is  73 in decimal, or h48
               e  is 101 in decimal, or h65
               l  is 108 in decimal, or h6C
               o  is 111 in decimal, or h6F
```

And for the output, we can use an Emit in a do... loop with all ASCII codes rather than get fancy. There are other ways to do this, and they may be better. But we stay with this one for now. The reason for adding a <SPACE> after the <STX> and before the <ETX> is that your computer and the Dumb Terminal might over-write Hello with these hidden codes. Newer computers tend to show the hidden characters as a tile; older terminals, such as a VT100 would ignore these characters and show nothing.

After testing, you could make the actual code shorter without the <SPACE> and you may have to for it to work with your EMIC2 for another device to work properly.

So for a first SENDit we use this ASCII code in hexadecimal to represent our message ====> h02 h20 h48 h65 h6C h6C h6F h20 h03

```
: SENDit h02 h20 h48 h65 h6C h6C h6F h20 h03 9 0 do emit loop ;  \ This is a first try
```

```
+++
Try SENDit
---
SENDit cr

###
␃ olleH ␂       <==== Output is backwards and shows the tiles that represent hidden ASCII codes
Prop0 Cog6 ok
%%%
```

First try SENDit in your computer. Does that work?

Also try it on the Dumb Terminal. Does it look a little different than what your computer does? Different terminals may handle hidden characters differently. Some just ignore, and other output a key code tile.

I guess you have noticed the we have olleH, not Hello. This is because we have forgotten that the stack actually reverses the order when data comes off of it. We just have to rewrite SENDit to work properly.

Here is a revised version. Please notice that <ETX> code is now first and <STX> is now last. We are putting the data on the stack backwards and taking it off forwards.

```
: SENDit h03 h20 h6F h6C h6C h65 h48 h20 h02 9 0 do emit loop ;  \ This is the right output
```

```
+++
Try the revised SENDit
---
SENDit cr

###
␂ Hello ␃       <==== Output is correct
Prop0 Cog6 ok
%%%
```

That should be everything for the word SENDit. You might want to save it with saveforth so that it will be available after a reboot or you can save this in a text file and wait until everything is right. Using saveforth too soon may just waste time.

It is now time to create GOTit. For this well will send a <CR> and then trap one ASCII character, and if > is received, the message in SENDit is sent.

```
: GOTit cr key 62 = -1 = if SENDit else cr then  ;  \ You have to trigger with > from keyboard to
test, other keys will ignore.
```

Your finished routine is a word that loads GOTit to Cog 0, let's call it TRIGGER1. It does not loop and only outputs once.

```
: TRIGGER1 c" GOTit " 0 cogx ;
```

TRIGGER1 makes both GOTit and SENDit receive and send only to the serial port connected to Cog 0 and is evoked from the Console_Serial or within other word definitions.

Anytime you need to reload your message to Cog 0, you just use the word TRIGGER1. Try this and then try your > from the Dumb Terminal. If it is all working, you are finished with this tutorial.

Sure this may not work the exact way you want to use your EMIC2 Text-to-speech, you will want have better control over when the text is sent. But it is working and a good beginning. It is up to you to create the messages exactly as you want them. At least you know you can send Hello and observe if that works.

If you have questions ask at the Parallax Forums. You have learned a lot and may just not realize how easy PropForth can be.

Here is nearly all the needed code for review.

```
======================================

c" 2 3 9600 serial" 1 cogx
```

```
c" reset " 0 cogx
0 1 ioconn
1 1 sersetflags

: SENDit h03 h20 h6F h6C h6C h65 h48 h20 h02 9 0 do emit loop cr ;
: GOTit cr key 62  =  -1 = if SENDit else cr then  ;

\ You have to trigger GOTit with > from keyboard to test. If another byte is received, you have to
evoke GOTit again.

: TRIGGER1 c" GOTit " 0 cogx ;

{
TRIGGER1 makes both GOTit and SENDit receive and send only to the serial port connected to Cog 0 and
it is evoked from the Console_Serial or within other word definitions.

This TRIGGER1 is only one such trigger instance, other words for TRIGGER2 and so on can be used to do
other outputs - text packets, active motors, or active software. And defined together in one word you
can have several cogs start nearly simultaneously.

}
```

======================================

12. Conclusion

For a last review, here is a list of the PropForth words that are most important for using the asynchronous serial port.

```
serial
_serial

term  -- Note that this requires cog and channel at as n1 and n2 with n2 on TOS

emit
key
c" .... "

cogx
cogid
cognchan

ioconn
(ioconn)
iodis
(iodis)
iolink
(iolink)
iounlink
(iounlink)

sersetflags
serflags?
```

Don't forget, the parenthesis are used more powerful internal versions of words.