***************DRAFT***************

UART No Flow Ctrl
10/26/20
iverlog Simulation
and
RPi with Catboard
running minicom

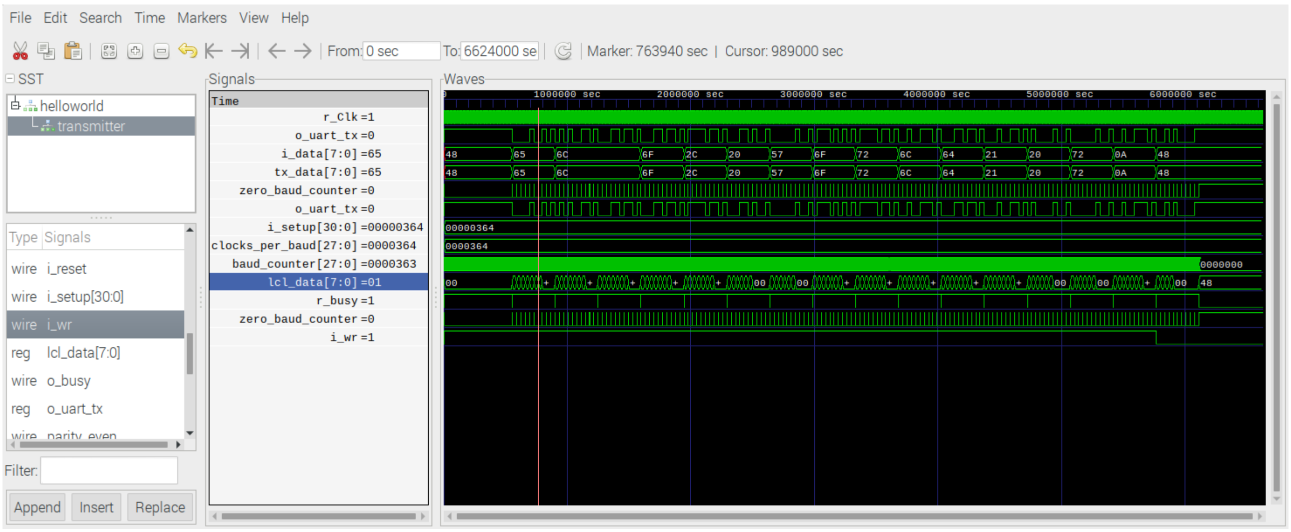
***************DRAFT***************

The files used to create the module uart are found at Appendix A. File tb_txuart.v & Appendix B File txuart.v

iverilog -o uart tb_txuart.v txuart.v

To create the dump.vcd "vvp uart" and run the simulation.

gtkwave “dump.vcd”



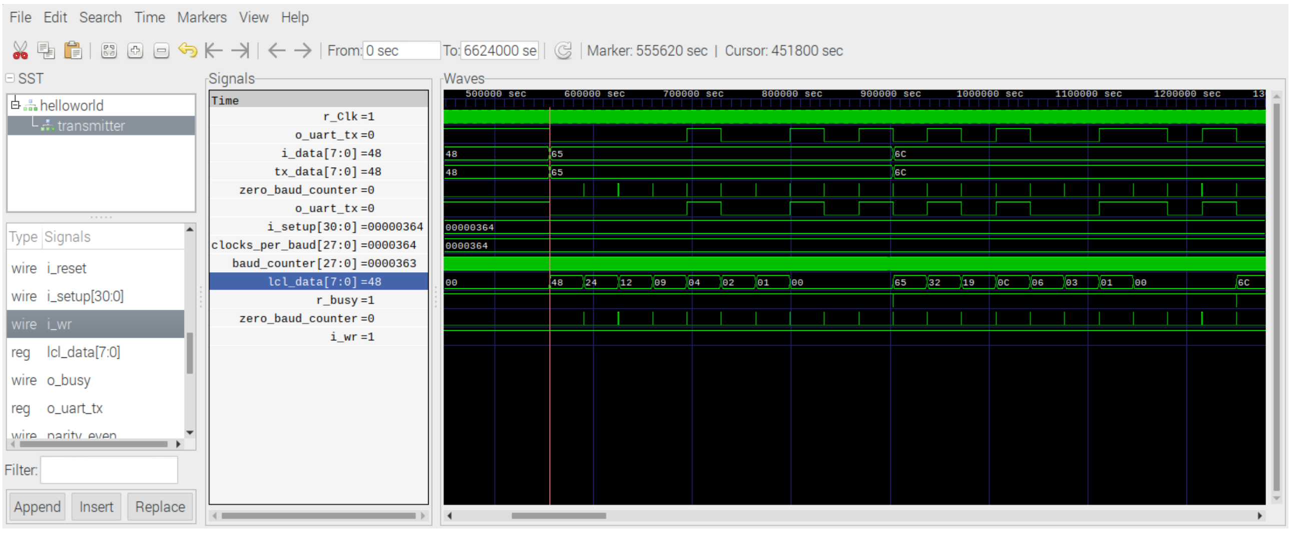
```
Type Signals
wire i_reset
wire i_setup[30:0]
wire i_wr
reg  lcl_data[7:0]
wire o_busy
reg  o_uart_tx
wire parity_even
```

```
Time
                r_Clk =1
             o_uart_tx =0
           i_data[7:0] =65
          tx_data[7:0] =65
     zero_baud_counter =0
             o_uart_tx =0
        i_setup[30:0] =00000364
clocks_per_baud[27:0] =0000364
    baud_counter[27:0] =0000363
         lcl_data[7:0] =01
                r_busy =1
     zero_baud_counter =0
                  i_wr =1
```

```
i_data:  48 65 6C 6F 2C 20 57 6F 72 6C 64 21 20 72 0A 48
tx_data: 48 65 6C 6F 2C 20 57 6F 72 6C 64 21 20 72 0A 48
i_setup: 00000364
clocks_per_baud: 0000364
baud_counter: 0000000
lcl_data: 00 ... 48
```

48.png

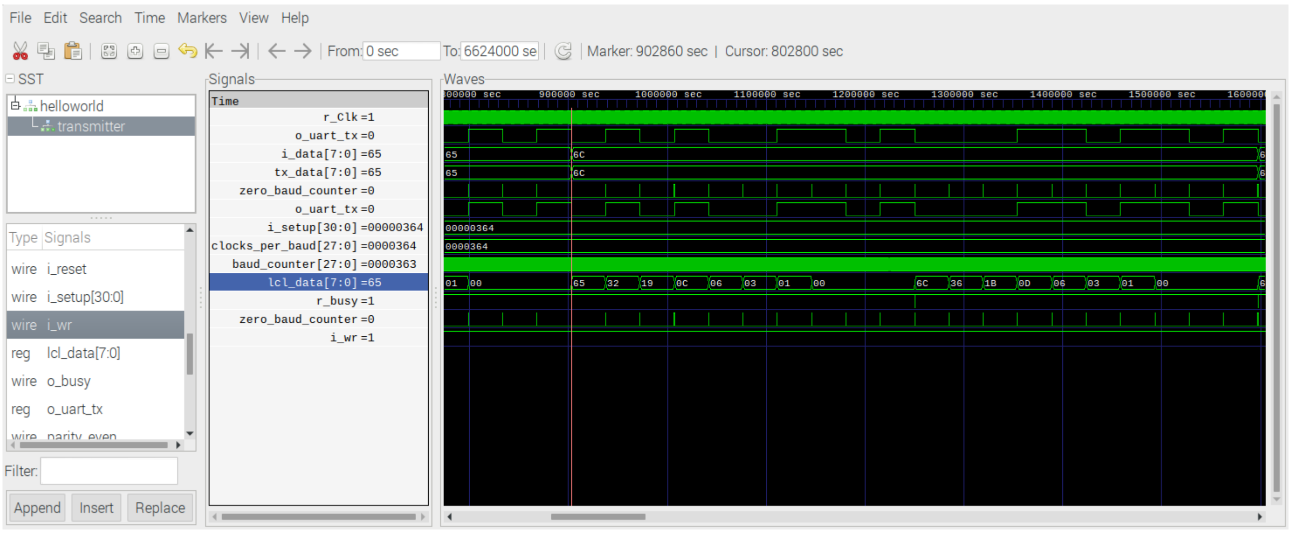
4 8
0 1 0 0 1 0 0 0
 0 1 2 3 4 5 6 7
0 0 0 0 1 0 0 1 0 1



```
Time
r_Clk =1
o_uart_tx =0
i_data[7:0] =48
tx_data[7:0] =48
zero_baud_counter =0
o_uart_tx =0
i_setup[30:0] =00000364
clocks_per_baud[27:0] =0000364
baud_counter[27:0] =0000363
lcl_data[7:0] =48
r_busy =1
zero_baud_counter =0
i_wr =1
```

65.png

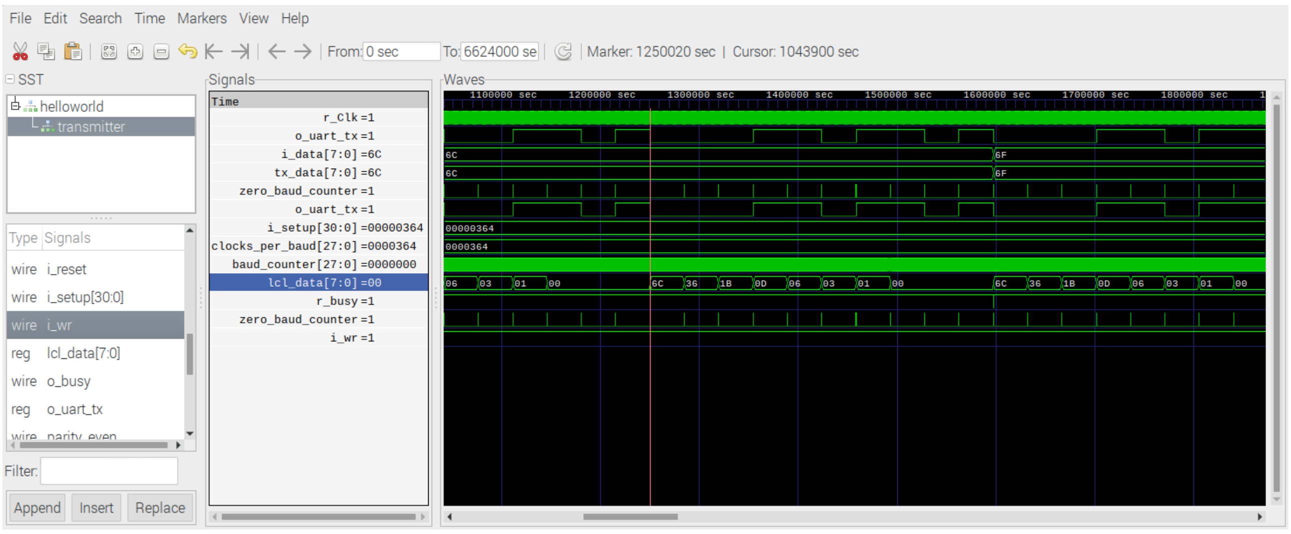
```
6      5
0 1 1 0 0 1 0 1
  0 1 2 3 4 5 6 7
0 1 0 1 0 0 1 1 0 1
```



```
Time
r_Clk =1
o_uart_tx =0
i_data[7:0] =65
tx_data[7:0] =65
zero_baud_counter =0
o_uart_tx =0
i_setup[30:0] =00000364
clocks_per_baud[27:0] =0000364
baud_counter[27:0] =0000363
lcl_data[7:0] =65
r_busy =1
zero_baud_counter =0
i_wr =1
```

6C.png

```
6      C
0 1 1 0 1 1 0 0
  0 1 2 3 4 5 6 7
0 0 0 0 1 1 0 1 1  1
```



From: 0 sec | To: 6624000 se | Marker: 1250020 sec | Cursor: 1043900 sec

SST:

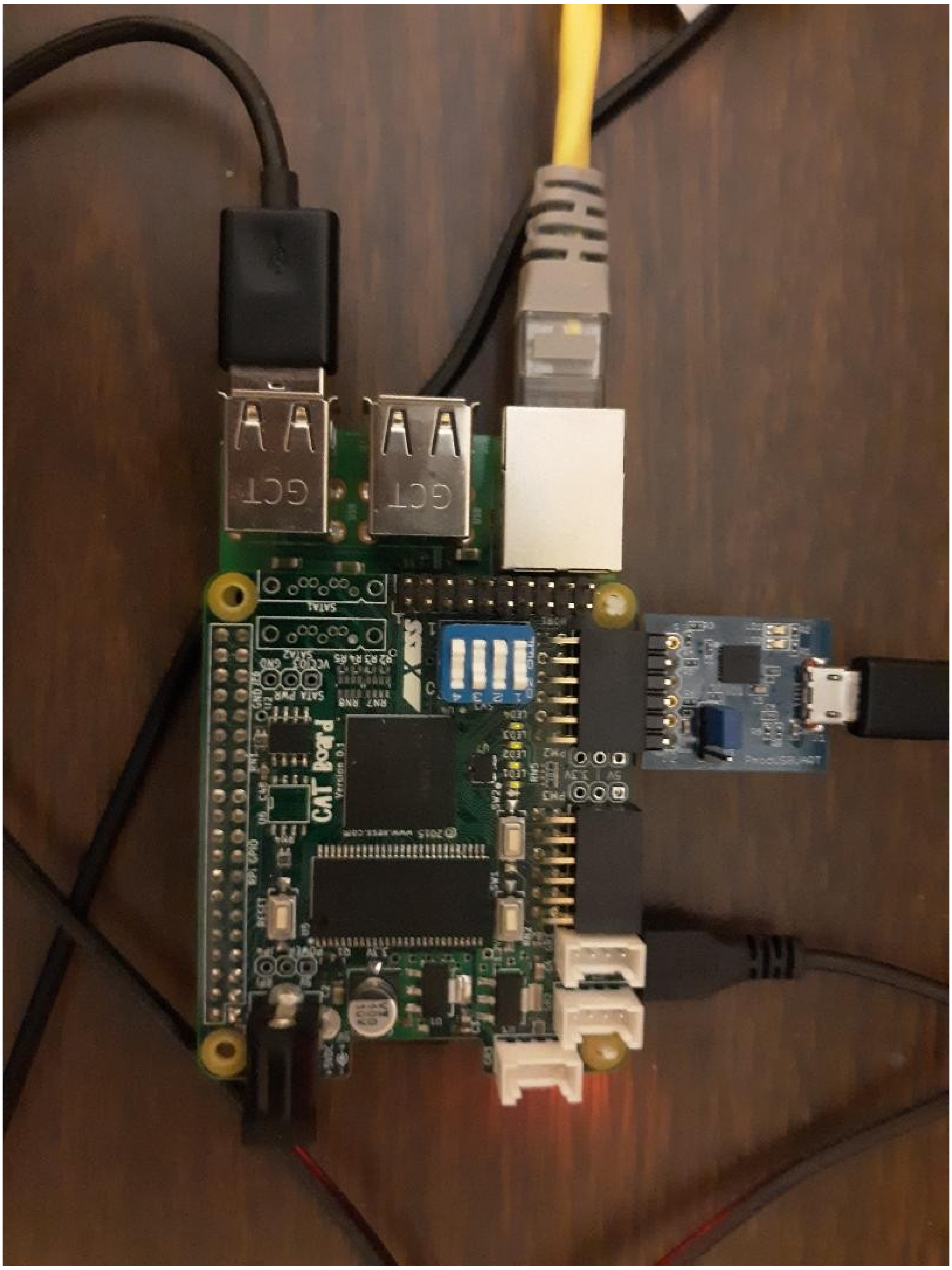
```
helloworld
  transmitter
```

Type	Signals
wire	i_reset
wire	i_setup[30:0]
wire	i_wr
reg	lcl_data[7:0]
wire	o_busy
reg	o_uart_tx
wire	parity_even

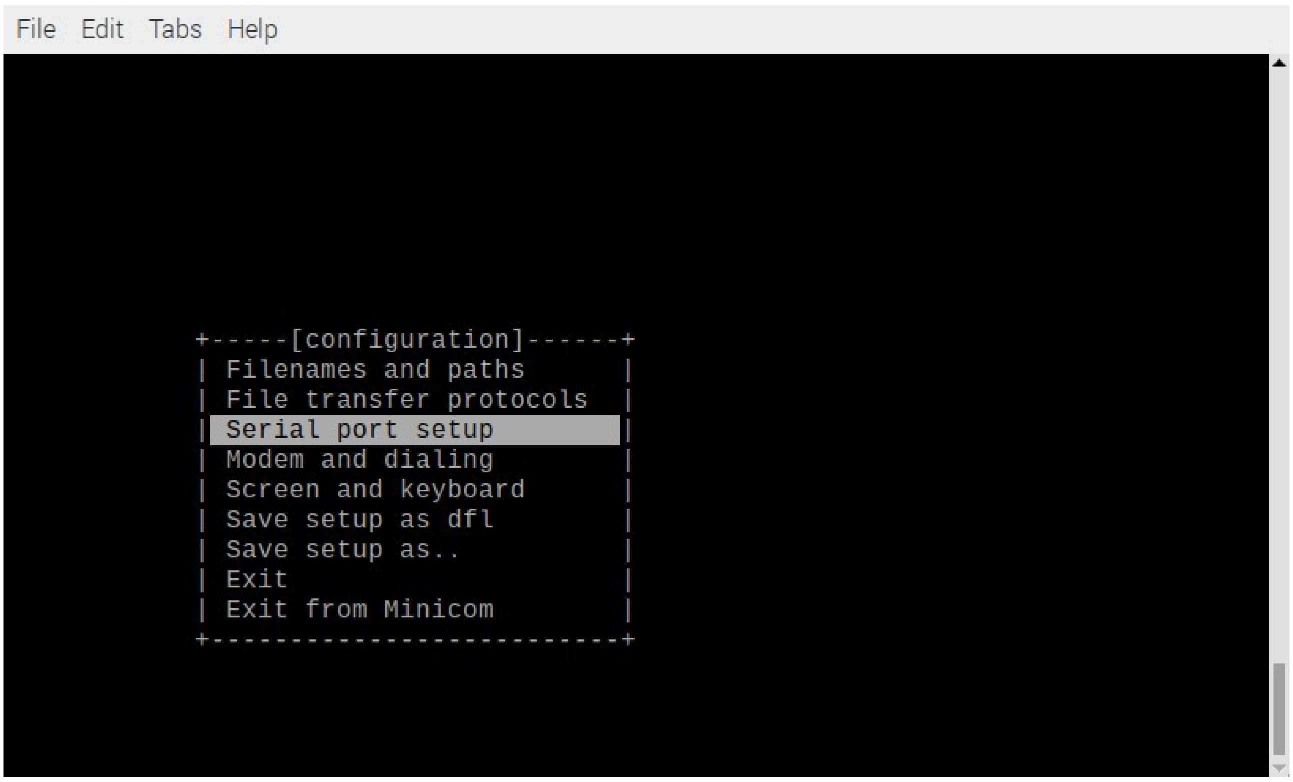
```
Time
                r_Clk =1
             o_uart_tx =1
           i_data[7:0] =6C
          tx_data[7:0] =6C
     zero_baud_counter =1
             o_uart_tx =1
         i_setup[30:0] =00000364
clocks_per_baud[27:0] =0000364
    baud_counter[27:0] =0000000
         lcl_data[7:0] =00
                r_busy =1
     zero_baud_counter =1
                  i_wr =1
```

```
1100000 sec  1200000 sec  1300000 sec  1400000 sec  1500000 sec  1600000 sec  1700000 sec  1800000 sec
i_data:   6C                                                              6F
tx_data:  6C                                                              6F
i_setup:  00000364
clocks_per_baud: 0000364
lcl_data: 06 03 01 00   6C 36 1B 0D 06 03 01 00   6C 36 1B 0D 06 03 01 00
```

RPi

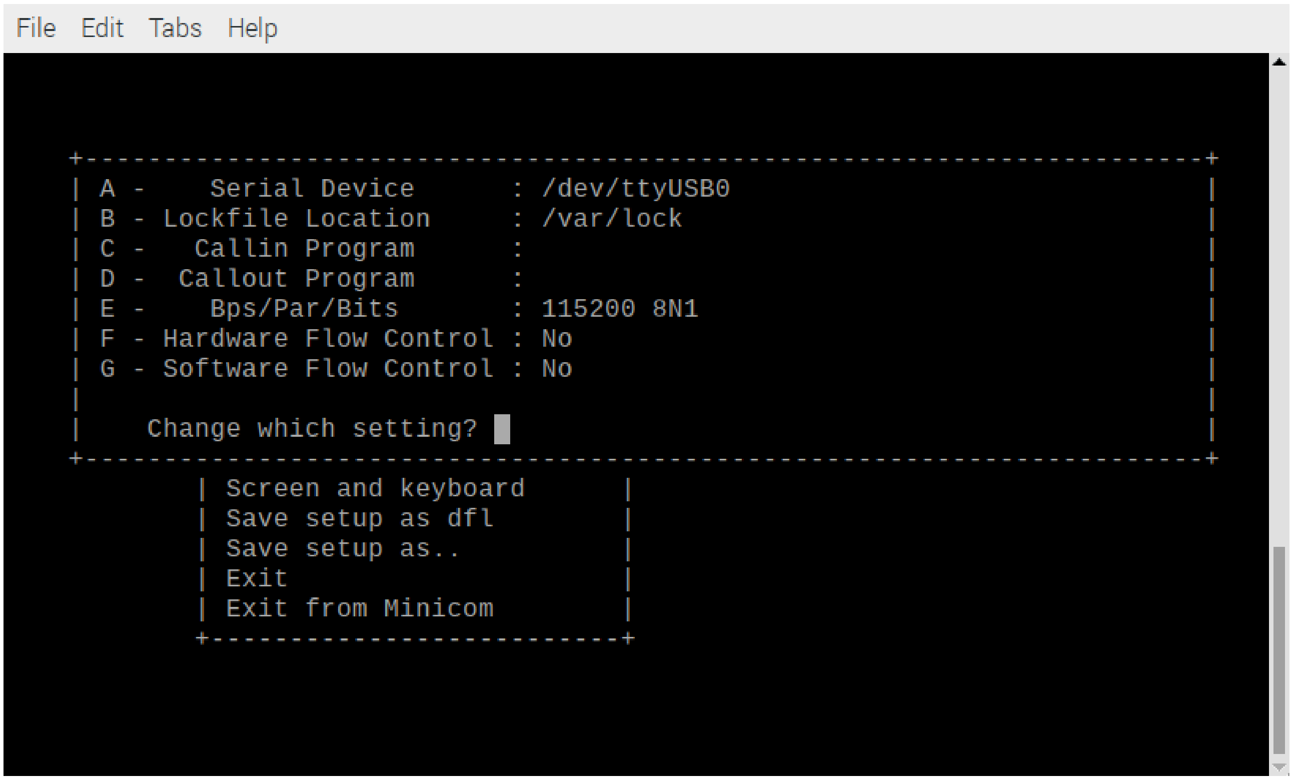


devel@mypi3-21:~/*home*/devl/uart/noflowcntl/
“sudo minicom -s”


File Edit Tabs Help

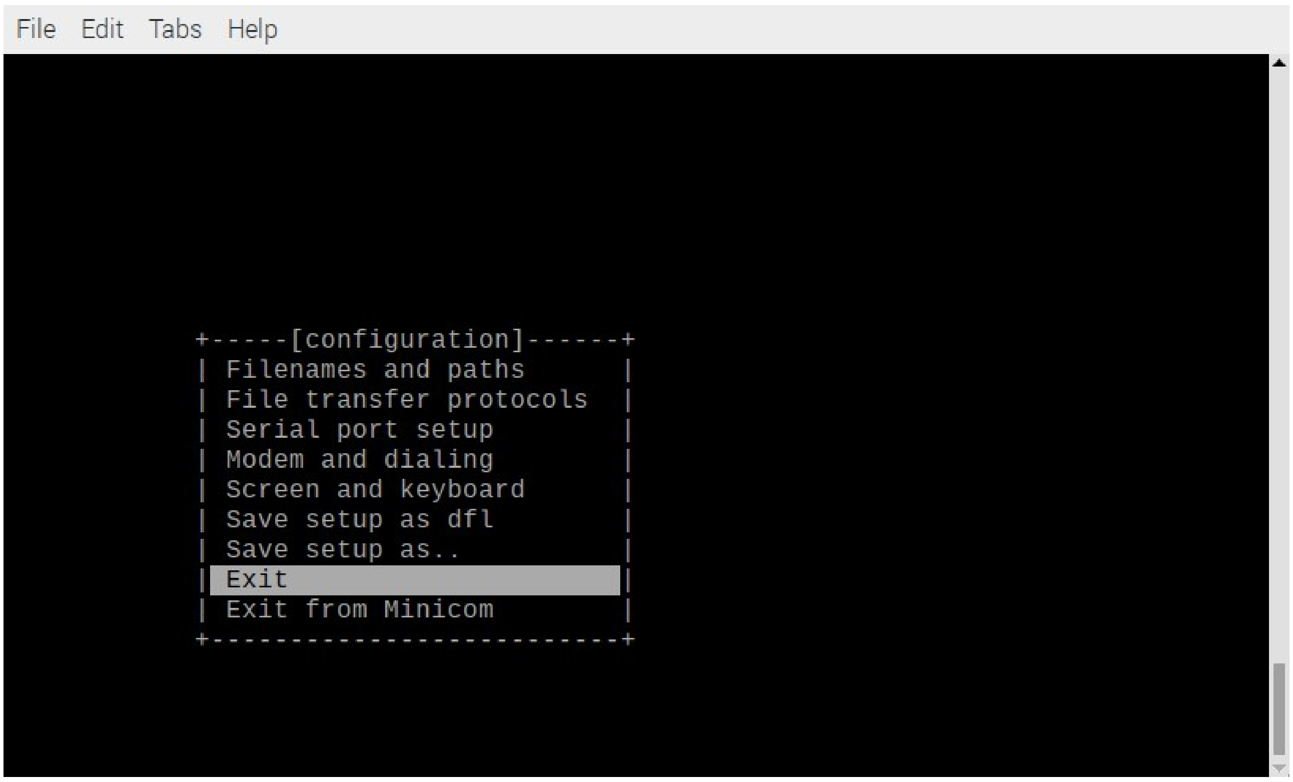
```
+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

Scroll down to "Serial port setup" "Depress enter"


File Edit Tabs Help

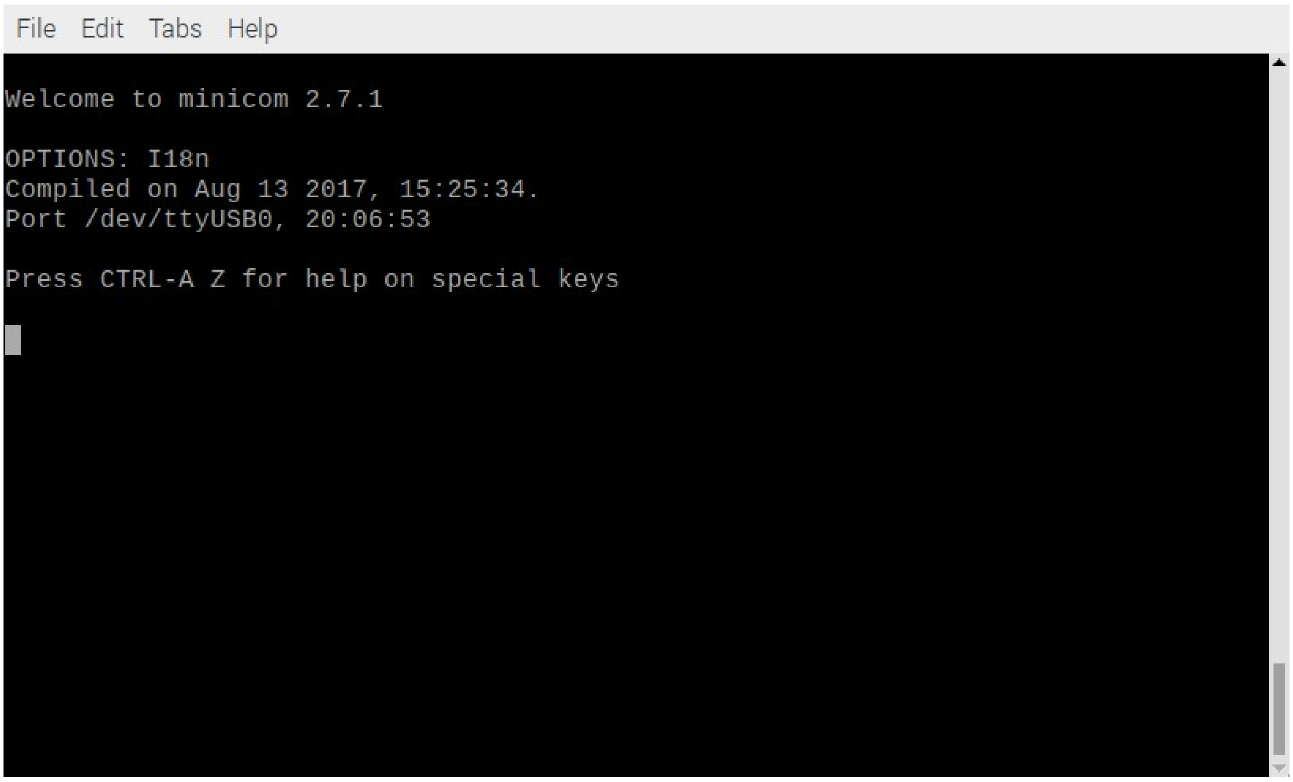
```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0                              |
| B - Lockfile Location     : /var/lock                                 |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : No                                        |
| G - Software Flow Control : No                                        |
|                                                                       |
|   Change which setting?                                               |
+-----------------------------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

"Depress enter"



```
+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

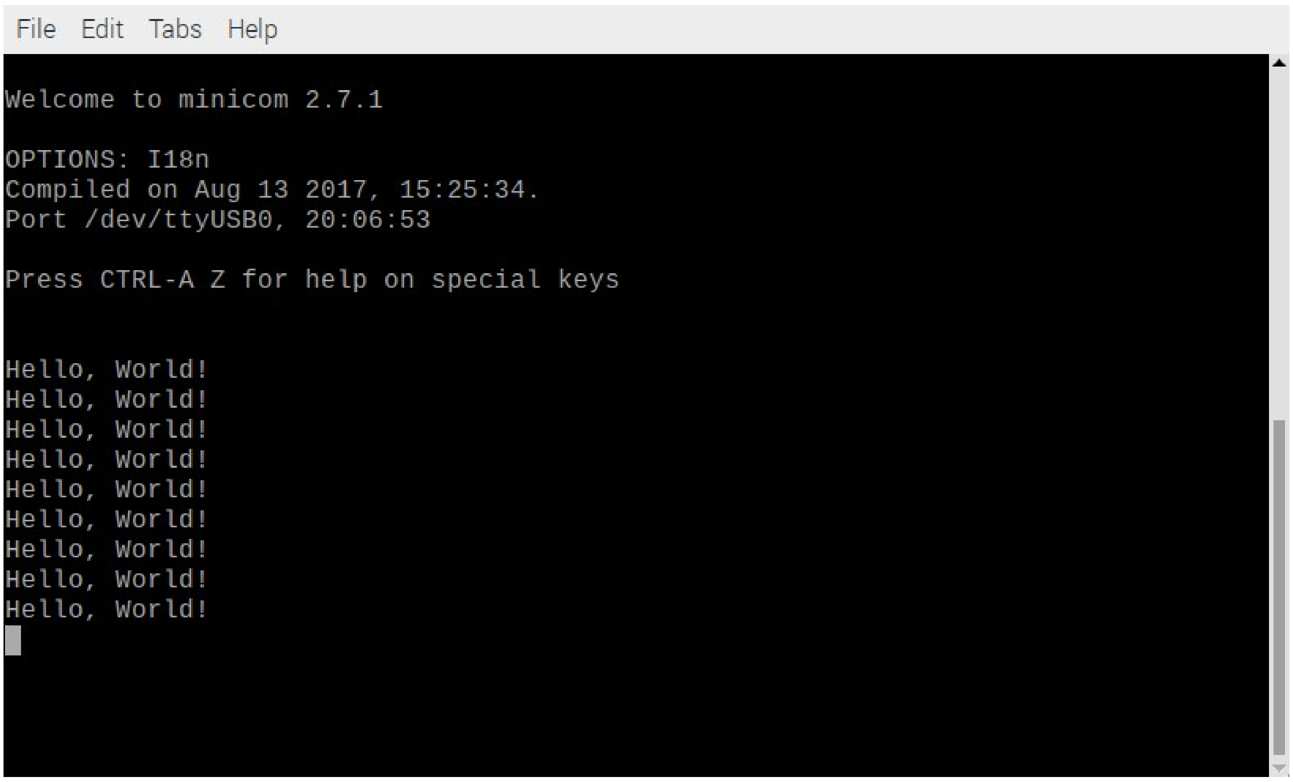
Scroll Down “Exit” “Depress enter”



```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 20:06:53

Press CTRL-A Z for help on special keys
```

Program the Catboard FPGA to send “Hello, World!”

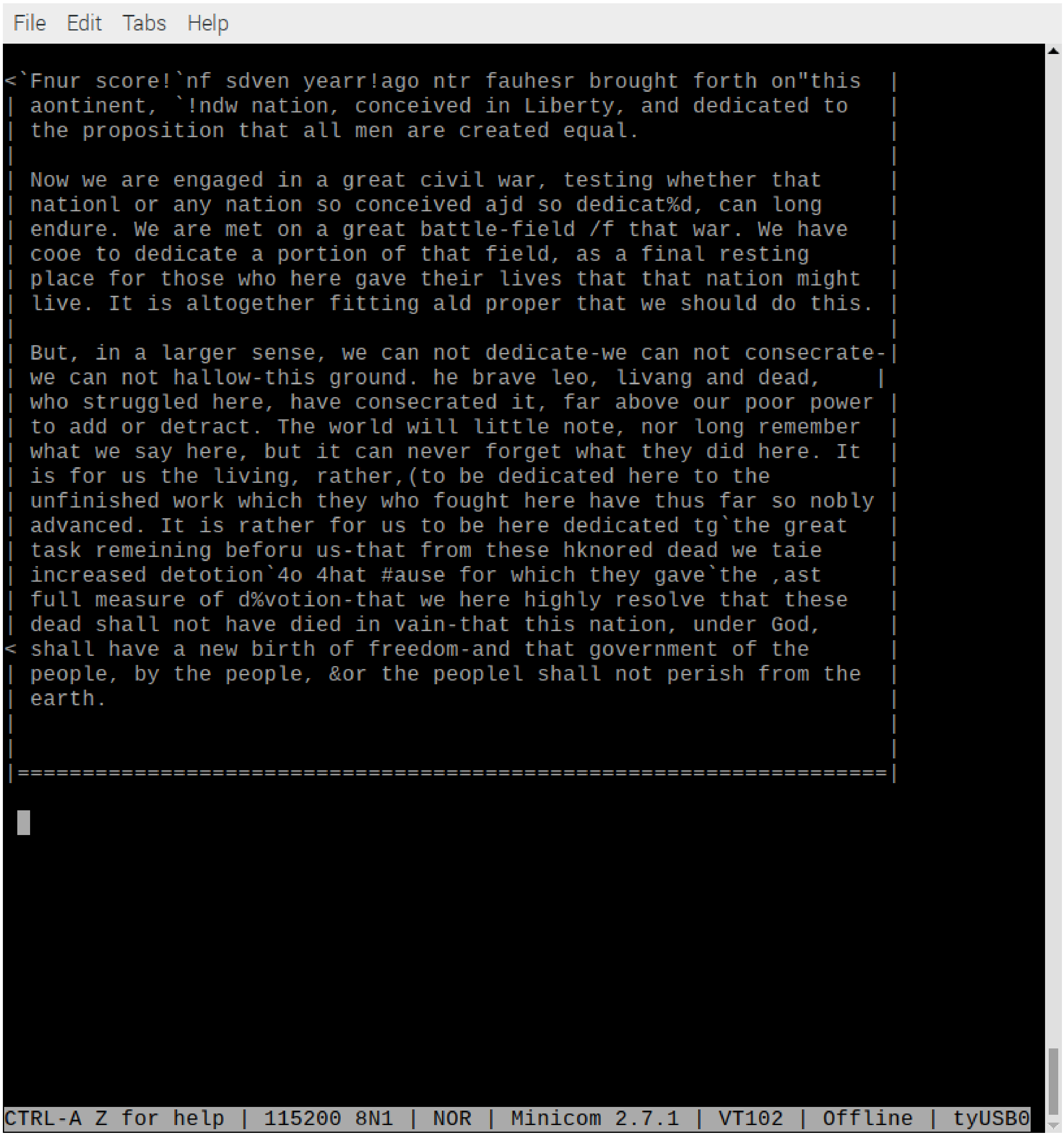
devel@mypi3-21:~/*home*/devl/uart/noflowcntl/
“sudo ~/catboard_yosys/config_cat helloworld.bin”



```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 20:06:53

Press CTRL-A Z for help on special keys


Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Program the Catboard FPGA to send “speech.hex”

devel@mypi3-21:~/*home*/devl/uart/noflowcntl/
“sudo ~/catboard_yosys/config_cat speechfifo.bin”



```
<`Fnur score!`nf sdven yearr!ago ntr fauhesr brought forth on"this  |
| aontinent, `!ndw nation, conceived in Liberty, and dedicated to   |
| the proposition that all men are created equal.                   |
|                                                                    |
| Now we are engaged in a great civil war, testing whether that     |
| nationl or any nation so conceived ajd so dedicat%d, can long     |
| endure. We are met on a great battle-field /f that war. We have   |
| cooe to dedicate a portion of that field, as a final resting      |
| place for those who here gave their lives that that nation might  |
| live. It is altogether fitting ald proper that we should do this. |
|                                                                    |
| But, in a larger sense, we can not dedicate-we can not consecrate-|
| we can not hallow-this ground. he brave leo, livang and dead,    |
| who struggled here, have consecrated it, far above our poor power |
| to add or detract. The world will little note, nor long remember  |
| what we say here, but it can never forget what they did here. It  |
| is for us the living, rather,(to be dedicated here to the         |
| unfinished work which they who fought here have thus far so nobly |
| advanced. It is rather for us to be here dedicated tg`the great   |
| task remeining beforu us-that from these hknored dead we taie     |
| increased detotion`4o 4hat #ause for which they gave`the ,ast     |
| full measure of d%votion-that we here highly resolve that these   |
| dead shall not have died in vain-that this nation, under God,     |
< shall have a new birth of freedom-and that government of the      |
| people, by the people, &or the peoplel shall not perish from the  |
| earth.                                                             |
|                                                                    |
|                                                                    |
|===================================================================|

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | tyUSB0
```

Appendix A. File tb_txuart.v

```
////////////////////////////////////////////////////////////////////////////////
//
// Filename:	helloworld.v
//
// Project:	wbuart32, a full featured UART with simulator
//
// Purpose:	To create a *very* simple UART test program, which can be used
//		as the top level design file of any FPGA program.
//
//	With some modifications (discussed below), this RTL should be able to
//	run as a top-level testing file, requiring only the UART and clock pin
//	to work.
```

```
//
// Creator:	Dan Gisselquist, Ph.D.
//		Gisselquist Technology, LLC
//
////////////////////////////////////////////////////////////////////////////////
//
// Copyright (C) 2015-2016, Gisselquist Technology, LLC
//
// This program is free software (firmware): you can redistribute it and/or
// modify it under the terms of  the GNU General Public License as published
// by the Free Software Foundation, either version 3 of the License, or (at
// your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT
// ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or
// FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along
// with this program.  (It's in the $(ROOT)/doc directory, run make with no
// target there if the PDF file isn't present.)  If not, see
// <http://www.gnu.org/licenses/> for a copy.
//
// License:	GPL, v3, as defined and found on www.gnu.org,
//		http://www.gnu.org/licenses/gpl.html
//
//
////////////////////////////////////////////////////////////////////////////////
//
//
// One issue with the design is how to set the values of the setup register.
// (*This is a comment, not a verilator attribute ... )  Verilator needs to
// know/set those values in order to work.  However, this design can also be
// used as a stand-alone top level configuration file.  In this latter case,
// the setup register needs to be set internal to the file.  Here, we use
// OPT_STANDALONE to distinguish between the two.  If set, the file runs under
// (* Another comment still ...) Verilator and we need to get i_setup from the
// external environment.  If not, it must be set internally.
//
`ifndef VERILATOR
`define OPT_STANDALONE
`endif
//
//
// Two versions of the UART can be found in the rtl directory: a full featured
// UART, and a LITE UART that only handles 8N1 -- no break sending, break
// detection, parity error detection, etc.  If we set USE_LITE_UART here, those
// simplified UART modules will be used.
//
// `define	USE_LITE_UART
//
//
```

```
module		helloworld(i_clk,
`ifndef	OPT_STANDALONE
			i_setup,
`endif
			o_uart_tx);
	input		i_clk;
	output	wire	o_uart_tx;

	// Here we set i_setup to something appropriate to create a 115200 Baud
	// UART system from a 100MHz clock.  This also sets us to an 8-bit data
	// word, 1-stop bit, and no parity.  This will be overwritten by
	// i_setup, but at least it gives us something to start with/from.
	//parameter	INITIAL_UART_SETUP = 31'd50;
	parameter	INITIAL_UART_SETUP = 31'd868;

	// The i_setup wires are input when run under Verilator, but need to
	// be set internally if this is going to run as a standalone top level
	// test configuration.
`ifdef	OPT_STANDALONE
	wire	[30:0]	i_setup;
	assign	i_setup = INITIAL_UART_SETUP;
`else
	input	[30:0]	i_setup;
`endif
reg r_Clk = 1'b0;



always #20 r_Clk <= ~r_Clk;
	reg	pwr_reset;
	initial	pwr_reset = 1'b1;
	always @(posedge r_Clk)
		pwr_reset <= 1'b0;

	reg	[7:0]	message		[0:15];



	reg	[27:0]	counter;
	initial	counter = 28'hffffff0;
	always @(posedge r_Clk)
		counter <= counter + 1'b1;

	wire		tx_break, tx_busy;
	reg		tx_stb;
	reg	[3:0]	tx_index;
	reg	[7:0]	tx_data;

	assign	tx_break = 1'b0;

	initial	tx_index = 4'h0;
	always @(posedge r_Clk)
```

```
			if ((tx_stb)&&(!tx_busy))
				tx_index <= tx_index + 1'b1;
		always @(posedge r_Clk)
			tx_data <= message[tx_index];

		initial	tx_stb = 1'b0;
		always @(posedge r_Clk)
			if (&counter)
				tx_stb <= 1'b1;
			else if ((tx_stb)&&(!tx_busy)&&(tx_index==4'hf))
				tx_stb <= 1'b0;

		// Bypass any hardware flow control
		wire	cts_n;
		assign cts_n = 1'b0;

`ifdef	USE_LITE_UART
		txuartlite
			#(24'd868)
			transmitter(r_Clk, tx_stb, tx_data, o_uart_tx, tx_busy);
`else
		txuart transmitter(r_Clk, pwr_reset, i_setup, tx_break,
				tx_stb, tx_data, cts_n, o_uart_tx, tx_busy);
`endif
  initial
  begin
			message[ 0] = "H";
			message[ 1] = "e";
			message[ 2] = "l";
			message[ 3] = "l";
			message[ 4] = "o";
			message[ 5] = ",";
			message[ 6] = " ";
			message[ 7] = "W";
			message[ 8] = "o";
			message[ 9] = "r";
			message[10] = "l";
			message[11] = "d";
			message[12] = "!";
			message[13] = " ";
			message[14] = "\r";
			message[15] = "\n";

    #6624000;
    $finish();
  end

  initial
  begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
```

```
endmodule
```

Appendix B File txuart.v

```
////////////////////////////////////////////////////////////////////////////////
//
// Filename:	txuart.v
//
// Project:	wbuart32, a full featured UART with simulator
//
// Purpose:	Transmit outputs over a single UART line.
//
//	To interface with this module, connect it to your system clock,
//	pass it the 32 bit setup register (defined below) and the byte
//	of data you wish to transmit.  Strobe the i_wr line high for one
//	clock cycle, and your data will be off.  Wait until the 'o_busy'
//	line is low before strobing the i_wr line again--this implementation
//	has NO BUFFER, so strobing i_wr while the core is busy will just
//	cause your data to be lost.  The output will be placed on the o_txuart
//	output line.  If you wish to set/send a break condition, assert the
//	i_break line otherwise leave it low.
//
//	There is a synchronous reset line, logic high.
//
//	Now for the setup register.  The register is 32 bits, so that this
//	UART may be set up over a 32-bit bus.
//
//	i_setup[30]	Set this to zero to use hardware flow control, and to
//		one to ignore hardware flow control.  Only works if the hardware
//		flow control has been properly wired.
//
//		If you don't want hardware flow control, fix the i_rts bit to
//		1'b1, and let the synthesys tools optimize out the logic.
//
//	i_setup[29:28]		Indicates the number of data bits per word.  This will
//		either be 2'b00 for an 8-bit word, 2'b01 for a 7-bit word, 2'b10
//		for a six bit word, or 2'b11 for a five bit word.
//
//	i_setup[27]	Indicates whether or not to use one or two stop bits.
//		Set this to one to expect two stop bits, zero for one.
//
//	i_setup[26]	Indicates whether or not a parity bit exists.  Set this
//		to 1'b1 to include parity.
//
//	i_setup[25]	Indicates whether or not the parity bit is fixed.  Set
//		to 1'b1 to include a fixed bit of parity, 1'b0 to allow the
//		parity to be set based upon data.  (Both assume the parity
//		enable value is set.)
//
//	i_setup[24]	This bit is ignored if parity is not used.  Otherwise,
//		in the case of a fixed parity bit, this bit indicates whether
```

```
//              mark (1'b1) or space (1'b0) parity is used.  Likewise if the
//              parity is not fixed, a 1'b1 selects even parity, and 1'b0
//              selects odd.
//
//      i_setup[23:0]  Indicates the speed of the UART in terms of clocks.
//              So, for example, if you have a 200 MHz clock and wish to
//              run your UART at 9600 baud, you would take 200 MHz and divide
//              by 9600 to set this value to 24'd20834.  Likewise if you wished
//              to run this serial port at 115200 baud from a 200 MHz clock,
//              you would set the value to 24'd1736
//
//      Thus, to set the UART for the common setting of an 8-bit word,
//      one stop bit, no parity, and 115200 baud over a 200 MHz clock, you
//      would want to set the setup value to:
//
//      32'h0006c8              // For 115,200 baud, 8 bit, no parity
//      32'h005161              // For 9600 baud, 8 bit, no parity
//
//
// Creator:     Dan Gisselquist, Ph.D.
//              Gisselquist Technology, LLC
//
////////////////////////////////////////////////////////////////////////////////
//
// Copyright (C) 2015-2017, Gisselquist Technology, LLC
//
// This program is free software (firmware): you can redistribute it and/or
// modify it under the terms of  the GNU General Public License as published
// by the Free Software Foundation, either version 3 of the License, or (at
// your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT
// ANY WARRANTY; without even the implied warranty of MERCHANTIBILITY or
// FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along
// with this program.  (It's in the $(ROOT)/doc directory.  Run make with no
// target there if the PDF file isn't present.)  If not, see
// <http://www.gnu.org/licenses/> for a copy.
//
// License:     GPL, v3, as defined and found on www.gnu.org,
//              http://www.gnu.org/licenses/gpl.html
//
//
////////////////////////////////////////////////////////////////////////////////
//
//
`default_nettype        none
//
`defineTXU_BIT_ZERO     4'h0
`defineTXU_BIT_ONE      4'h1
```

```
`defineTXU_BIT_TWO      4'h2
`defineTXU_BIT_THREE  4'h3
`defineTXU_BIT_FOUR   4'h4
`defineTXU_BIT_FIVE     4'h5
`defineTXU_BIT_SIX       4'h6
`defineTXU_BIT_SEVEN  4'h7
`defineTXU_PARITY        4'h8    // Constant 1
`defineTXU_STOP   4'h9    // Constant 1
`defineTXU_SECOND_STOP       4'ha
// 4'hb // Unused
// 4'hc  // Unused
// `define        TXU_START4'hd    // An unused state
`defineTXU_BREAK         4'he
`defineTXU_IDLE   4'hf
//
//
module txuart(i_clk, i_reset, i_setup, i_break, i_wr, i_data,
                i_cts_n, o_uart_tx, o_busy);
        parameter     [30:0]  INITIAL_SETUP = 31'd868;
        input   wire            i_clk, i_reset;
        input   wire    [30:0]  i_setup;
        input   wire            i_break;
        input   wire            i_wr;
        input   wire    [7:0]   i_data;
        // Hardware flow control Ready-To-Send bit.  Set this to one to use
        // the core without flow control.  (A more appropriate name would be
        // the Ready-To-Receive bit ...)
        input   wire            i_cts_n;
        // And the UART input line itself
        outputreg               o_uart_tx;
        // A line to tell others when we are ready to accept data.  If
        // (i_wr)&&(!o_busy) is ever true, then the core has accepted a byte
        // for transmission.
        outputwire              o_busy;

        wire    [27:0]  clocks_per_baud, break_condition;
        wire    [1:0]   data_bits;
        wire            use_parity, parity_even, dblstop, fixd_parity,
                        fixdp_value, hw_flow_control;
        reg     [30:0]  r_setup;
        assign clocks_per_baud = { 4'h0, r_setup[23:0] };
        assign break_condition = { r_setup[23:0], 4'h0 };
        assign hw_flow_control = !r_setup[30];
        assign data_bits      =  r_setup[29:28];
        assign dblstop        =  r_setup[27];
        assign use_parity     =  r_setup[26];
        assign fixd_parity    =  r_setup[25];
        assign parity_even    =  r_setup[24];
        assign fixdp_value    =  r_setup[24];

        reg     [27:0]  baud_counter;
        reg     [3:0]   state;
```

```
reg	[7:0]	lcl_data;
reg		calc_parity, r_busy, zero_baud_counter;


// First step ... handle any hardware flow control, if so enabled.
//
// Clock in the flow control data, two clocks to avoid metastability
// Default to using hardware flow control (uart_setup[30]==0 to use it).
// Set this high order bit off if you do not wish to use it.
reg	q_cts_n, qq_cts_n, ck_cts;
// While we might wish to give initial values to q_rts and ck_cts,
// 1) it's not required since the transmitter starts in a long wait
// state, and 2) doing so will prevent the synthesizer from optimizing
// this pin in the case it is hard set to 1'b1 external to this
// peripheral.
//
// initial	q_cts_n  = 1'b1;
// initial	qq_cts_n = 1'b1;
// initial	ck_cts   = 1'b0;
always	@(posedge i_clk)
	q_cts_n <= i_cts_n;
always	@(posedge i_clk)
	qq_cts_n <= q_cts_n;
always	@(posedge i_clk)
	ck_cts <= (!qq_cts_n)||(!hw_flow_control);

initial	o_uart_tx = 1'b1;
initial	r_busy = 1'b1;
initial	state  = `TXU_IDLE;
initial	lcl_data= 8'h0;
initial	calc_parity = 1'b0;
// initial	baud_counter = clocks_per_baud;//ILLEGAL--not constant
always @(posedge i_clk)
begin
	if (i_reset)
	begin
		r_busy <= 1'b1;
		state <= `TXU_IDLE;
	end else if (i_break)
	begin
		state <= `TXU_BREAK;
		r_busy <= 1'b1;
	end else if (!zero_baud_counter)
	begin // r_busy needs to be set coming into here
		r_busy <= 1'b1;
	end else if (state == `TXU_BREAK)
	begin
		state <= `TXU_IDLE;
		r_busy <= 1'b1;
	end else if (state == `TXU_IDLE)	// STATE_IDLE
	begin
		if ((i_wr)&&(!r_busy))
```

```
				begin	// Immediately start us off with a start bit
					r_busy <= 1'b1;
					case(data_bits)
					2'b00: state <= `TXU_BIT_ZERO;
					2'b01: state <= `TXU_BIT_ONE;
					2'b10: state <= `TXU_BIT_TWO;
					2'b11: state <= `TXU_BIT_THREE;
					endcase
				end else begin // Stay in idle
					r_busy <= !ck_cts;
				end
			end else begin
				// One clock tick in each of these states ...
				// baud_counter <= clocks_per_baud - 28'h01;
				r_busy <= 1'b1;
				if (state[3] == 0) // First 8 bits
				begin
					if (state == `TXU_BIT_SEVEN)
						state <= (use_parity)?`TXU_PARITY:`TXU_STOP;
					else
						state <= state + 1;
				end else if (state == `TXU_PARITY)
				begin
					state <= `TXU_STOP;
				end else if (state == `TXU_STOP)
				begin // two stop bit(s)
					if (dblstop)
						state <= `TXU_SECOND_STOP;
					else
						state <= `TXU_IDLE;
				end else // `TXU_SECOND_STOP and default:
				begin
					state <= `TXU_IDLE; // Go back to idle
					// Still r_busy, since we need to wait
					// for the baud clock to finish counting
					// out this last bit.
				end
			end
		end

		// o_busy
		//
		// This is a wire, designed to be true is we are ever busy above.
		// originally, this was going to be true if we were ever not in the
		// idle state.  The logic has since become more complex, hence we have
		// a register dedicated to this and just copy out that registers value.
		assign	o_busy = (r_busy);


		// r_setup
		//
		// Our setup register.  Accept changes between any pair of transmitted
```

```
// words.  The register itself has many fields to it.  These are
// broken out up top, and indicate what 1) our baud rate is, 2) our
// number of stop bits, 3) what type of parity we are using, and 4)
// the size of our data word.
initial  r_setup = INITIAL_SETUP;
always @(posedge i_clk)
        if (state == `TXU_IDLE)
                r_setup <= i_setup;

// lcl_data
//
// This is our working copy of the i_data register which we use
// when transmitting.  It is only of interest during transmit, and is
// allowed to be whatever at any other time.  Hence, if r_busy isn't
// true, we can always set it.  On the one clock where r_busy isn't
// true and i_wr is, we set it and r_busy is true thereafter.
// Then, on any zero_baud_counter (i.e. change between baud intervals)
// we simple logically shift the register right to grab the next bit.
always @(posedge i_clk)
        if (!r_busy)
                lcl_data <= i_data;
        else if (zero_baud_counter)
                lcl_data <= { 1'b0, lcl_data[7:1] };

// o_uart_tx
//
// This is the final result/output desired of this core.  It's all
// centered about o_uart_tx.  This is what finally needs to follow
// the UART protocol.
//
// Ok, that said, our rules are:
//      1'b0 on any break condition
//      1'b0 on a start bit (IDLE, write, and not busy)
//      lcl_data[0] during any data transfer, but only at the baud
//              change
//      PARITY -- During the parity bit.  This depends upon whether or
//              not the parity bit is fixed, then what it's fixed to,
//              or changing, and hence what it's calculated value is.
//      1'b1 at all other times (stop bits, idle, etc)
always @(posedge i_clk)
        if (i_reset)
                o_uart_tx <= 1'b1;
        else if ((i_break)||((i_wr)&&(!r_busy)))
                o_uart_tx <= 1'b0;
        else if (zero_baud_counter)
                casez(state)
                4'b0???:        o_uart_tx <= lcl_data[0];
                `TXU_PARITY:    o_uart_tx <= calc_parity;
                default:        o_uart_tx <= 1'b1;
                endcase
```

```
// calc_parity
//
// Calculate the parity to be placed into the parity bit.  If the
// parity is fixed, then the parity bit is given by the fixed parity
// value (r_setup[24]).  Otherwise the parity is given by the GF2
// sum of all the data bits (plus one for even parity).
always @(posedge i_clk)
	if (fixd_parity)
		calc_parity <= fixdp_value;
	else if (zero_baud_counter)
	begin
		if (state[3] == 0) // First 8 bits of msg
			calc_parity <= calc_parity ^ lcl_data[0];
		else
			calc_parity <= parity_even;
	end else if (!r_busy)
		calc_parity <= parity_even;


// All of the above logic is driven by the baud counter.  Bits must last
// clocks_per_baud in length, and this baud counter is what we use to
// make certain of that.
//
// The basic logic is this: at the beginning of a bit interval, start
// the baud counter and set it to count clocks_per_baud.  When it gets
// to zero, restart it.
//
// However, comparing a 28'bit number to zero can be rather complex--
// especially if we wish to do anything else on that same clock.  For
// that reason, we create "zero_baud_counter".  zero_baud_counter is
// nothing more than a flag that is true anytime baud_counter is zero.
// It's true when the logic (above) needs to step to the next bit.
// Simple enough?
//
// I wish we could stop there, but there are some other (ugly)
// conditions to deal with that offer exceptions to this basic logic.
//
// 1. When the user has commanded a BREAK across the line, we need to
// wait several baud intervals following the break before we start
// transmitting, to give any receiver a chance to recognize that we are
// out of the break condition, and to know that the next bit will be
// a stop bit.
//
// 2. A reset is similar to a break condition--on both we wait several
// baud intervals before allowing a start bit.
//
// 3. In the idle state, we stop our counter--so that upon a request
// to transmit when idle we can start transmitting immediately, rather
// than waiting for the end of the next (fictitious and arbitrary) baud
// interval.
//
// When (i_wr)&&(!r_busy)&&(state == `TXU_IDLE) then we're not only in
```

```
	// the idle state, but we also just accepted a command to start writing
	// the next word.  At this point, the baud counter needs to be reset
	// to the number of clocks per baud, and zero_baud_counter set to zero.
	//
	// The logic is a bit twisted here, in that it will only check for the
	// above condition when zero_baud_counter is false--so as to make
	// certain the STOP bit is complete.
	initial	zero_baud_counter = 1'b0;
	initial	baud_counter = 28'h05;
	always @(posedge i_clk)
	begin
		zero_baud_counter <= (baud_counter == 28'h01);
		if ((i_reset)||(i_break))
		begin
			// Give ourselves 16 bauds before being ready
			baud_counter <= break_condition;
			zero_baud_counter <= 1'b0;
		end else if (!zero_baud_counter)
			baud_counter <= baud_counter - 28'h01;
		else if (state == `TXU_BREAK)
			// Give us four idle baud intervals before becoming
			// available
			baud_counter <= clocks_per_baud<<2;
		else if (state == `TXU_IDLE)
		begin
			baud_counter <= 28'h0;
			zero_baud_counter <= 1'b1;
			if ((i_wr)&&(!r_busy))
			begin
				baud_counter <= clocks_per_baud - 28'h01;
				zero_baud_counter <= 1'b0;
			end
		end else
			baud_counter <= clocks_per_baud - 28'h01;
	end
endmodule
```