Debugging Code Running on an Embedded ATTiny85, version 1.0

By R. G. Sparber with valuable guidance from Dave Kellogg

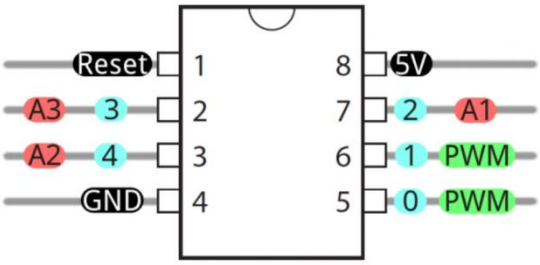
Conclusion

When limited memory, processing power, and uncommitted output pins exist, debugging the code can be difficult. The solution presented here addresses this problem. I look to the community to suggest ways to improve my approach.

Contents

The Problem

The ATTiny85 is an entire computer system in an 8 pin Dual Inline Package[2]. Hidden in there is a powerful computer, persistent memory, dynamic memory, an Analog to Digital Converter (ADC), and even a programmable differential amplifier. It is only missing one thing: pins.



Can't have it all! With such a small package, there are only 5 easily usable Input/Output (IO) pins. A sixth pin can be configured to be a "weak" IO pin but then you give up the ability to reset the device.

When embedded into a project, few if any pins may be free. Yet many bugs only show themselves when the device it actually performing the needed task.

My first project using the ATTiny85 was a perfect fit. I needed two analog inputs (3 and 4) and 3 digital outputs (0, 1, and 2). All went well until it was time to debug the code. Trying to guess at what the program was doing via the single red/green LED connected to the device was maddening. I'm accustom to putting Serial.print() statements in my code to tell me what is going on. Their output goes through a USB cable and into my PC where the information is displayed.

It is possible to add code to the ATTiny85 to simulate Serial.print(). I would also need to free up 2 pins for the USB connection. At that point, there wouldn't be much room left for my original functionality. It would also have a major impact on real time performance since I'm running the clock at only 1 MHz. In the end, I would be watching my code run in a totally different environment. That could hide the bugs I am looking for and generate bugs caused by my new environment.

[2] It is also available in Surface Mount Technology.

The First Step

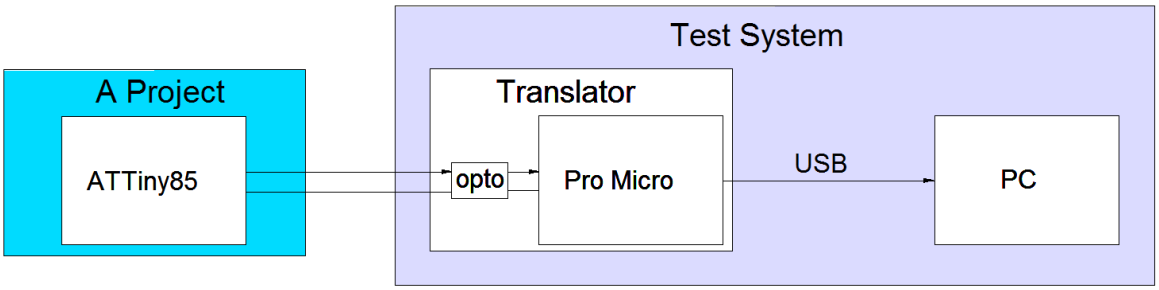
Since Arduino code can run on many devices, I first run my ATTiny85 code on a Pro Micro that is not embedded. I simulate sending signals to output pins with Serial.print() statements.

This lets me debug the hardware and real time independent functionality with full access to the Serial.print() function. However, there is a point when I must move to the actual hardware and real time constrains to continue finding problems.

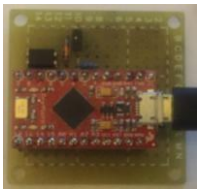
The Built In Solution

The ATTiny85's has a host of powerful machine language debugging functions that drive the single Reset pin. A number of people have written code with custom hardware to access these features[3]. However, I need to debug Arduino code and prefer to not deal with machine language.

My Solution

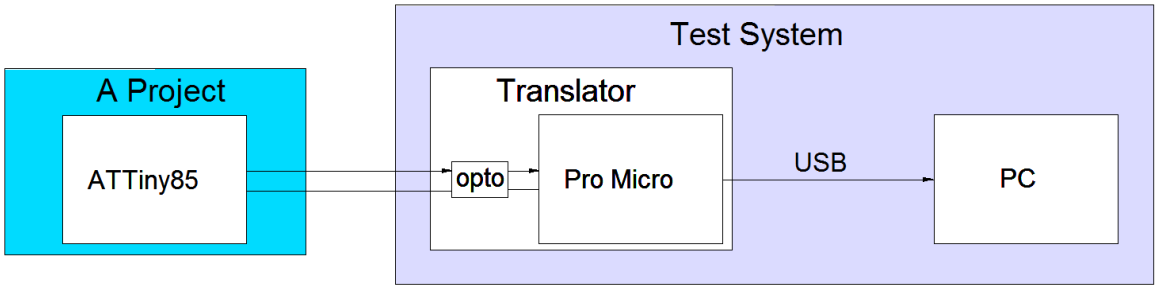


The embedded ATTiny85 drives one, user specified, IO pin. This signal feeds into my Test System. The Test System consists of a Translator and a PC running a Terminal Emulator.



The Translator consists of an opto isolator and a Pro Micro. The opto isolator permits me to run the ATTiny85 and my Test System on different grounds plus protects the PC from possible ground fault currents generated by my project[4].

[3] Search with key words ATTiny85 debugWire. Of particular note is the work by Wayne Holder.
[4] In my first application, the ATTiny85's ground is 2 volts below the Pro Micro's ground.



The translator accepts a nonstandard signal from the ATTiny85 and converts it to bytes. These bytes are sent out the USB where they are displayed on the PC.

The ATTiny85 has 8KB of program memory and 512 bytes of dynamic memory. My transmitter code occupies 14% of this program memory and 7.4% of dynamic memory.

The Pro Micro runs at 16 MHz and can handle an average of one byte every 60 milliseconds from the ATTiny85. However, up to 10 bytes can be queued up in the ATTiny85 before transmission begins.

User's Guide

Initial Setup

First build the hardware shown on page 11. Then download the code shown in Appendix II into the Pro Micro.

Take the code in Appendix I and distribute it into your ATTiny85. It will go into

- the constants section
- the variable section
- setup()
- loop()
- and the rest goes into your subroutine section

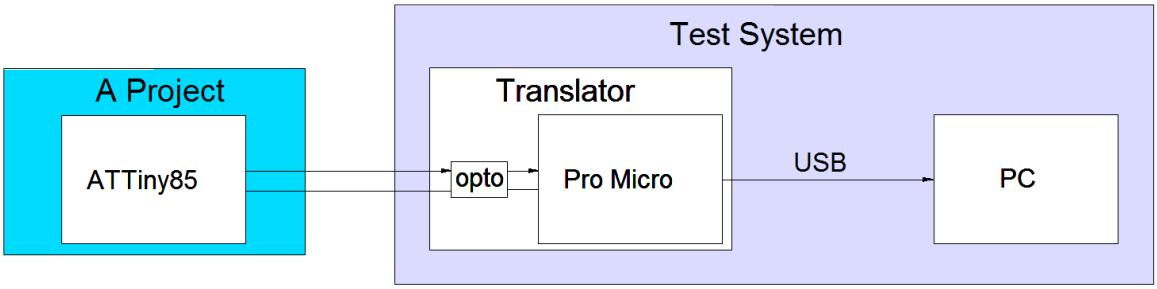
If necessary, you will have to change your ATTiny85 code to free up an IO pin. Then you must tell my software which logical pin to drive.

Using `SkinnyPrint`

Within loop(), you will have the subroutine `SkinnyPrinter()`. Then, you may have up to ten instances of `SkinnyPrintByte()`.

It will take a total of 60 milliseconds for your ATTiny85 to send one byte. If you have 10 bytes in a single burst, `SkinnyPrinter()` needs

$10 \times 60\ milliseconds = 600\ milliseconds$ to get them all out. Furthermore, if you send them faster than `SkinnyPrinter()` can handle, you will get random overflow messages. An overflow message is the byte 255 twice in a row. An overflow event flushes all pending bytes.



You may also randomly see the letter "E" on your screen. It means the Test System had a problem finding the start of a transmission. This often occurs right at power up of the ATTiny85 but if you see it later, something is wrong either with the hardware or with the timing. As long as you run the ATTiny85 at 1 MHz and use the Pro Micro which runs at 16 MHz, I don't expect you will have timing problems. However, there are two constants you can adjust if necessary. See page 10 for details.

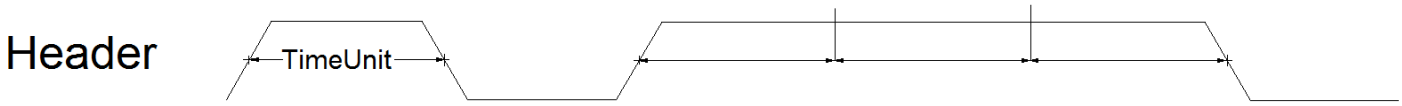
The Serial Signal

Of top priority was to use as few IO pin on the ATTiny85 as possible[5]. I could spare one pin but certainly not more. The strategy I developed with help from Dave Kellogg consists of three elements:

1. A header
2. A logic 0
3. A logic 1

All of these elements are based on a time duration I call a ***TimeUnit***. Information is represented by the ratio of time relative to the TimeUnit. This technique qualifies the signal as "self clocking[6]".

The Header



When idle, the receiving logic is constantly looking for the Header. It signals the start of a byte and provides the TimeUnit.

Each rising edge is potentially the start of the Header. A timer is started on each rising edge and stopped at the subsequent falling edge. This interval becomes our candidate TimeUnit. On the next rising edge we again start a timer. It is stopped at the subsequent falling edge. If this interval equal about three times the TimeUnit, we know we have just read the header.

If the ratio is not around 3:1, we take this new time interval and make it our new candidate TimeUnit. Then the code repeats. Eventually it will detect the header plus have a value for our TimeUnit. This technique qualifies the signal as "self synchronizing[7]".

Although not used in timing, all instances that the signal is low are also one TimeUnit long. This means the Header takes six TimeUnits.

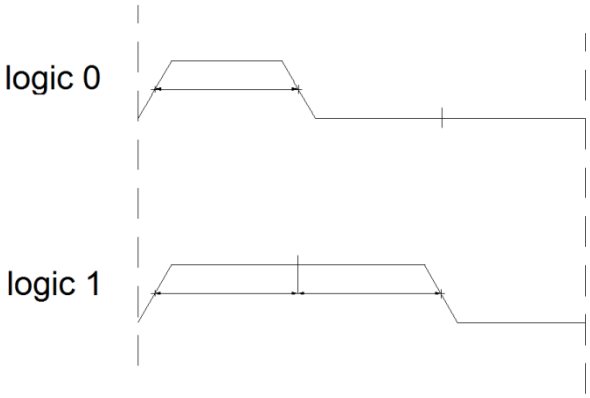
---

[5] I did consider the possibility of having software modulate the devices current drain. If it could shift this current level between two values, I would be able to design a circuit that would extract the data. Then my scheme would use no IO pins which would be very cool.

[6] See https://en.wikipedia.org/wiki/Self-clocking_signal.

[7] See https://en.wikipedia.org/wiki/Self-synchronizing_code

The Bits

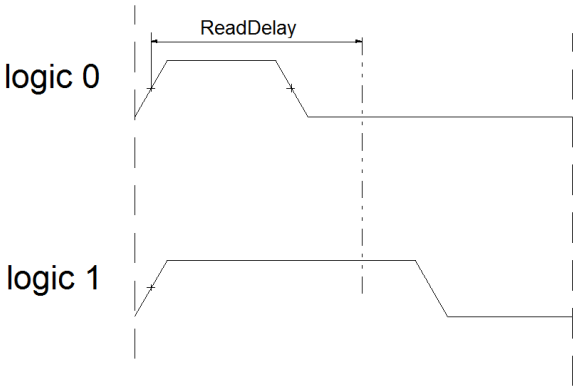


Both a logic 0 and logic 1 begin with a rising edge.

After one TimeUnit, the logic 0 has a falling edge and stays low for two TimeUnits.

The logic 1 stays high for two TimeUnits and then has a falling edge. It stays low for a TimeUnit.

Since neither of these symbols are high for three TimeUnits, no combination of 0s and 1s can be mistaken for the header.



Right after the receiving logic determines the TimeUnit, it multiplies by 1.5 and calls it the ReadDelay.

When a rising edge is detected, it delays by the ReadDelay and then reads the value. If low, it is a logic 0. If high, it is a logic 1. Making this simple helps to keep the code simple and therefore fast.

Each bit takes 3 TimeUnits and 8 bits are passed so sending one byte takes 24 TimeUnits.

The Byte

To keep the code simple (and fast), I only pass single bytes. After the Header has been detected, the code collects 8 bits and then returns to looking for a Header.

It will take time for the receiving code to pass the byte to the PC. To avoid missing the next Header, the transmitter will wait an "Inter-Byte" time.

Software Architecture

The transmitting and receiving code were developed under different constraints so have different architectures.

Transmitting Software Architecture

I have assumed that there will be a burst of data that is real time critical followed by rest.

During the frantic moments, I have SkinnyPrintByte(<1 byte>) which places its byte into a stack and returns. This must run as fast as possible. The present design allows a burst of ten SkinnyPrintByte () calls. If more are needed, the size of the stack must be increased. The penalty is small - one byte of dynamic memory needed for each additional SkinnyPrintByte().

When there is rest, I process the stack using SkinnyPrinter(). This subroutine contains three functions:

MonitorStack();
SendFromStack();
SendStream();

MonitorStack() watches for overflow. If none, it returns. If there is overflow, it prepares to send 0xFF twice to the PC and flushes the rest of the waiting bytes. The user must know enough about the receiving data to determine if "FF, FF" is valid data or trouble.

SendFromStack () waits until the last transmission is done and then takes a byte from the stack and sets it as the next byte to go out.
SendStream () pumps out the waveform. When it gets control during a TimeUnit, it just returns. If a TimeUnit has just passed, it put out the next high or low.

Most of the time, these subroutines do little and then return to the loop().

If you plan to send integers, consider adding

```
void SkinnyPrintInteger(int data){
SkinnyPrintByte(highByte(data));
SkinnyPrintByte(lowByte(data));
}
```

Receive Software Architecture

Here I don't have to share resources with other programs but do have to be as fast as possible. We constantly loop through

```
DetectHeader();
ReceiveByte();
OutputByte();
```

The first two subroutines are Finite State Machines[8]. They look for a transition, do something, look for the next transition, and do something else.

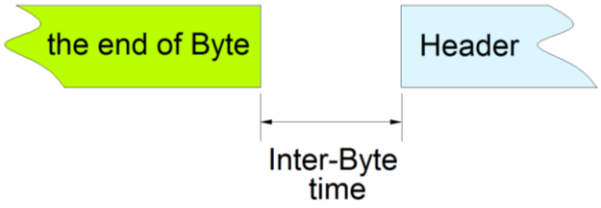
OutputByte() takes the assembled byte and ships it out to the PC with a Serial.print() command.

The code can be found in the Appendices.

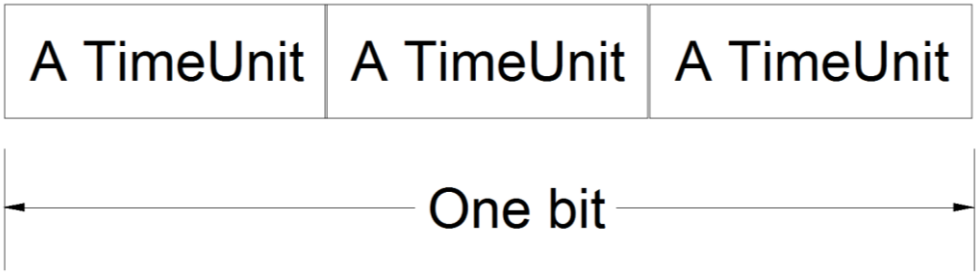
[8] See https://en.wikipedia.org/wiki/Finite-state_machine.

Adjusting Timing

There are two timing constants that set how fast data is transmitted to the Test System by the ATTiny85 code.

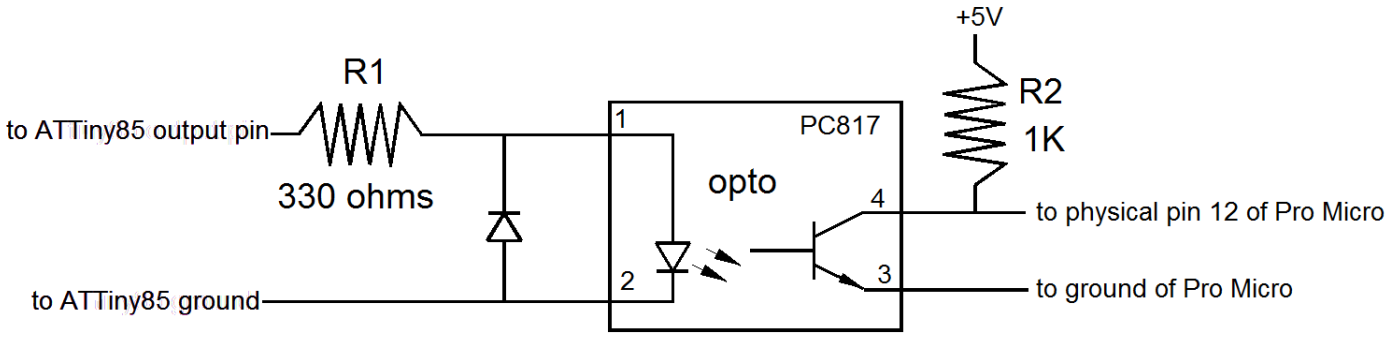


The first is `InterByteTimeMicroS`. It is the number of microseconds of delay between the end of one byte and the start of the next header. This is processing time for the receiver. If you see random E's being displayed on the PC, raise this number and see if it solves the problem. See page 7 for more detail.

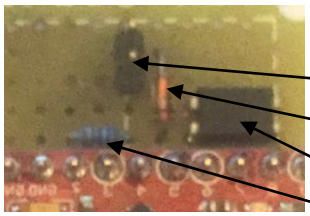


The second timing constant is `TimeUnitMicroS`. It is approximately the width of a TimeUnit although another 200 microseconds are added to it as the code executes. If the bytes you are sending from the ATTiny85 are being corrupted, try increasing this time. Just remember, there are 6 TimeUnits in the Header plus 24 TimeUnits in the byte so increasing this constant by 100 microseconds means it will take 3000 microseconds ***more*** to transmit it. See page 6 for more detail.

The Hardware



This little circuit could prevent you from having a very bad day. Ground from your laptop or PC connects on the right. Ground on the ATTiny85 connects on the left. If for some reason these two grounds are not at the same voltage, an unknown and potentially destructive current can flow.



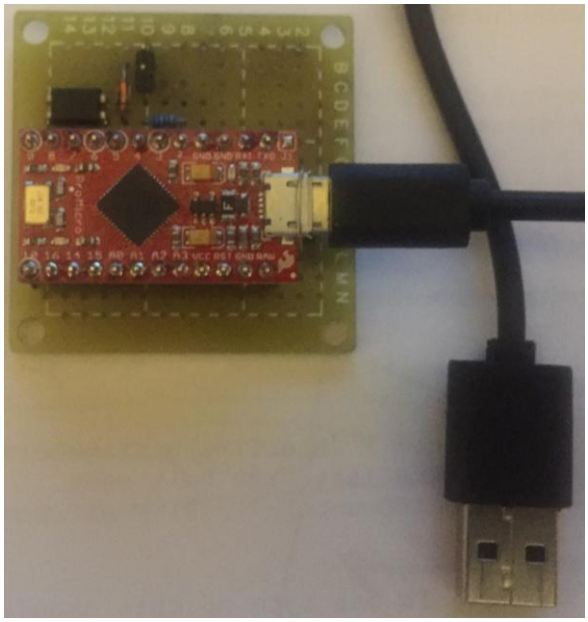
Layout is not critical. Here you see the input connector, diode, the Opto isolator, and the input resistor, R1. Not shown is pull up resistor R2. All parts mount on a board with 0.1 inch spaced holes.

The most expensive part of this circuit will be shipping and handling. The PC817 opto costs under $0.50 although just about any opto isolator will do here. The diode can be any general purpose variety. Both resistors are 0.1watts.

When the ATTiny85 output pin is high relative to its own ground, current will flow through R1 and into the opto's pin 1. This current will come out of pin 2 and return to the ATTiny85's ground.

This will cause the opto's pin 4 to pull down to near the Pro Micro's ground. When the ATTiny85 puts out a low, no current flows and pin 4 rises to near 5 volts.

If you happen to connect the ATTiny85 leads backwards, the diode protects the opto. If an excessive voltage is connected, R1 will blow but all other electronics are protected.



The USB connector on the Pro Micro is delicate. If you have a spare cable, plug it in permanent.

If you choose to not include this circuit, the signal into the Pro Micro will have to be inverted in the code. For example, when the Pro Micro's code reads a HIGH on logical pin 9, it should interpret this as a LOW.

Acknowledgement

Dave Kellogg had the perfect background and willingness to help me get this system to work. He popped up on the homemadetools.net BBS when I posted a second rate way to debug code. I learned a first rate way to do that task plus gained subsequent knowledge about tiny pieces of code that move bytes around. He had done a similar job decades ago and only used 188 bytes. It also used only 2% of the real time. I was happy getting mine down to 1190 bytes but have hope that others can see ways to shrink it further. When I start to use my system, real time usage numbers will be collected.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with just "Article Alias" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org

Appendix I: ATTiny85 Resident Code

```
/*
; ************************************************
; * Skinny Transmitter                           *
; * Version 1.5                                  *
; This work is licensed under the Creative       *
; Commons Attribution 4.0 International License. *
; To view a copy of this license, visit          *
; http://creativecommons.org/licenses/by/4.0/    *
; or send a letter to Creative Commons,          *
; PO Box 1866, Mountain View, CA 94042, USA.     *
; * by R.G. Sparber                              *
; ************************************************
;
; ============================================
;   H A R D W A R E   I N F O R M A T I O N
; ============================================
;
; physical pin  logical         pin name
;      1         5        NotReset, ADC0,PCINT5
;      2         3
;      3         4
;      4         n/a            ground
;      5         0
;      6         1
;      7         2
;      8        n/a             power
; ============================================
;      P O R T S   A N D   L O G I C A L  P I N  N A M E S
; ============================================
*/
const byte SkinnyPrintPin = 2; //logical port that will be used
as output
/*
; ============================================
;      C O N S T A N T S
; ============================================
*/
unsigned int InterByteTimeMicroS = 250; //time from the
transmission of the last bit of one byte to the header of the
next byte
unsigned int TimeUnitMicroS = 1500; //a TimeUnit is actually
about 0.2 milliseconds long than this value
/*
; ============================================
;      V A R I A B L E S
; ============================================
```

```
*/
byte ByteBeingSent = 0;
byte Stack[12]; //holds 10 or fewer bytes waiting to be
transmitted plus the overflow symbol of FF FF
byte LocalTimeUnitCounter = 0;
byte SkinnyTransmitterCounter = 0; //used to shift stack down
byte StackPointer = 0; //next available location in stack

boolean StackOverflowQ = false;
boolean SendingStreamQ = false;

unsigned long start = 0;

byte TimeUnitCounter = 0;

unsigned long TimeLastBitSentMicroS = micros(); //used to tell
when to start sending next byte

unsigned long TimeTimeUnitSentMicroS = micros() -
2*TimeUnitMicroS; //used to tell when to send next TimeUnit.
Initialized to insure immediate transmission

unsigned long  TimeAllOfByteSentMicroS = micros(); // used to
tell when to send next byte. I set it so first byte is sent
without delay.
/*
; ============================================
;      S E T U P
; ============================================
*/

void setup(){
//Serial.begin(9600); //temporary function while running on Pro
Micro

/*
digitalWrite(SkinnyPrintPin,LOW) and was replaced with
Serial.print("_")

digitalWrite(SkinnyPrintPin,HIGH) and was replaced with
Serial.println("-")

_____________________________________________

; physical pin  logical pin name  use
;      1         PB5              NotReset
;      2         PB3        ADC3  negative analog input
```

```
;       3          PB4         ADC2  positive analog input
;       4          ground
;       5          PB0         PB0   opto positive drive
;       6          PB1         PB1   opto negative drive
;       7          PB2         PB2   power control output
;       8          power
_____________________________________________
*/

pinMode(SkinnyPrintPin,OUTPUT); //assignment for debugger

}
/*
; ============================================
;       L 0 0 P
; ============================================
*/
void loop(){

SkinnyPrintByte(0);
SkinnyPrintByte(1);
SkinnyPrintByte(2);
SkinnyPrintByte(3);
SkinnyPrintByte(4);
SkinnyPrintByte(5);
SkinnyPrintByte(6);
SkinnyPrintByte(7);
SkinnyPrintByte(8);
SkinnyPrintByte(9);
start = millis();
hold:
SkinnyPrinter();
if (millis()-start < 600)goto hold; //give time to drain the
stack
}

/*
; ============================================
;       S U B R O U T I N E S
; ============================================
*/


//D I A G N O S T I C  S U B R O U T I N E S


void SkinnyPrintByte(byte data){
```

```
if (StackOverflowQ == true)return;
if (StackPointer > (sizeof(Stack) - 1)){
StackOverflowQ = true;
return;
}

Stack[StackPointer] = data; //populate next available stack
location
++StackPointer; //point to the new free stack location above the
data just added
}

void SkinnyPrinter(){
MonitorStack();
SendFromStack();
SendStream();
}

void MonitorStack(){
if (StackOverflowQ == true){
//flush stack and transmit 255 twice in a row
StackPointer = 2;
Stack[0]=0xFF;
Stack[1] = 0xFF;
Stack[2] = 0; //make backfill of stack 0 rather than random data
StackOverflowQ = false;
}
}

void SendFromStack(){
if (SendingStreamQ == true)return;
if(StackPointer == 0)return; //no byte waiting to be sent
if ((micros() -  TimeAllOfByteSentMicroS) <
InterByteTimeMicroS)return;
PrepareToSendStream(Stack[0]);
SkinnyTransmitterCounter = 0; //shift down stack and adjust
StackPointer
while(SkinnyTransmitterCounter < StackPointer){
    Stack[SkinnyTransmitterCounter] =
Stack[SkinnyTransmitterCounter+1];
    ++SkinnyTransmitterCounter;
    }
--StackPointer;
return;
}

void PrepareToSendStream(byte data){
```

```
SendingStreamQ = true;//we are now Sending a Stream
ByteBeingSent = data; //load supplied byte as the one about to
be sent
}

void SendStream(){
if (SendingStreamQ == false)return;
if ((micros() - TimeTimeUnitSentMicroS) < TimeUnitMicroS)return;
if (TimeUnitCounter <7){ //header is 0 through 6
SendHeader();
}else{ //TimeUnitCounter should be 7 through 30
SendBits();//when done sending 8 bits it returns TimeUnitCounter
to 0 and SendingStreamQ to false
}
if(SendingStreamQ == true){ //if sending stream, advance
TimeUnitCounter but if done, do not advance it
TimeTimeUnitSentMicroS = micros();//prepare to send next
TimeUnit
++TimeUnitCounter;//prepare to send next TimeUnit
}
}

void SendHeader(){
switch(TimeUnitCounter){
case 0:
    digitalWrite(SkinnyPrintPin,LOW);
    break;
case 1:
    digitalWrite(SkinnyPrintPin,HIGH);
    break;
case 2:
    digitalWrite(SkinnyPrintPin,LOW);
    break;
case 3:
    digitalWrite(SkinnyPrintPin,HIGH);
    break;
case 4:
    digitalWrite(SkinnyPrintPin,HIGH);
    break;
case 5:
    digitalWrite(SkinnyPrintPin,HIGH);
    break;
case 6:
    digitalWrite(SkinnyPrintPin,LOW);
    break;
    }
}
```

```
void SendBits(){
LocalTimeUnitCounter =  (TimeUnitCounter - 7)%3; //this is the
relative TimeUnit position within a bit
    if((1 & ByteBeingSent) == 0){ //this will extract the
current LSB and if true we must send a logic 0
        switch(LocalTimeUnitCounter){
            case 0:
            digitalWrite(SkinnyPrintPin,HIGH);
            break;
            case 1:
            digitalWrite(SkinnyPrintPin,LOW);
            break;
            case 2:
            digitalWrite(SkinnyPrintPin,LOW);
            break;
        }
    }else{
    //the LSB is a logic 1
        switch(LocalTimeUnitCounter){
            case 0:
            digitalWrite(SkinnyPrintPin,HIGH);
            break;
            case 1:
            digitalWrite(SkinnyPrintPin,HIGH);
            break;
            case 2:
            digitalWrite(SkinnyPrintPin,LOW);
            break;
        }
    }
if (LocalTimeUnitCounter == 2)ByteBeingSent = ByteBeingSent>>1;
//shift bits to the right one place so bit to the left of the
previous LSB becomes the LSB
if (TimeUnitCounter >29){ //should be 30 after last bit sent



    TimeUnitCounter=0;//prepare for next transmission
    SendingStreamQ = false;
    TimeAllOfByteSentMicroS = micros();//time stamp end of
transmission
    }
}
```

Appendix II: Pro Micro Resident Code

```
/*
; ************************************************
; * SkinnyPrint Receiver for ATtiny85            *
; * this software runs on a Pro Micro            *
; * It reads header plus 8 bits                  *
; * and outputs a byte.                          *
; * Version 0.7                                  *
; This work is licensed under the Creative       *
; Commons Attribution 4.0 International License. *
; To view a copy of this license, visit          *
; http://creativecommons.org/licenses/by/4.0/    *
; or send a letter to Creative Commons,          *
; PO Box 1866, Mountain View, CA 94042, USA.     *
; * by R.G. Sparber                              *
; ************************************************
;
; ==============================================
;   H A R D W A R E   I N F O R M A T I O N
; ==============================================
;
; physical pin  logical pin   use
;      12         D9          serial input
; ==============================================
;      P O R T S   A N D   L O G I C A L  P I N  N A M E S
; ==============================================
*/
const int SerialInput = 9; //the logical port that will receive
the serial stream;
/*
; ==============================================
;      C O N S T A N T S
; ==============================================
*/
In order to get 2 places past decimal point in detection of
header,
I multiply LongOneTime by 10^4, divide it by a time tolerance
times 10^2 and compare it to the tolerance which has been
multiplied by 10^4/10^2 = 100.
const unsigned long MinimumTimeToleranceTimesOneHundred = 250;
// allowed variation in TimeUnit for a LongOne times 10000
const unsigned long MaximumTimeToleranceTimesOneHundred = 350;
// allowed variation in TimeUnit for a LongOne times 10000
/*
; ==============================================
;      V A R I A B L E S
; ==============================================
```

```
*/
unsigned long TimeUnit = 0; //fundamental unit of time in serial
stream
unsigned long TimeUnitTimesOneHundred = 0;
unsigned long ReadDelay = 0;
unsigned long TimeStamp = 0;
unsigned long LongOneTime = 0;
unsigned long LongOneTimeTimesTenThousand = 0;
unsigned int BitCounter = 0;
byte ReceivedByte = 0;
/*
; ============================================
;      S E T U P
; ============================================
*/
void setup(){
//set up physical pin 12 (logical port 9) as digital input
pinMode(SerialInput,INPUT);
Serial.begin(9600); //This pipes text to the PC's Serial monitor
}
/*
; ============================================
;      L 0 0 P
; ============================================
*/
void loop(){
DetectHeader();
ReceiveByte();
OutputByte();
}

/*
; ============================================
;      S U B R O U T I N E S
; ============================================
*/

void DetectHeader(){ // -_-___-
WaitForNegativeEdgeAndTimeStampIt();
WaitForPostiveEdge();
TimeUnit = micros() - TimeStamp; //calculate how long signal was
low and call it my TimeUnit
ScanForLongOne:
WaitForNegativeEdgeAndTimeStampIt();
WaitForPostiveEdge();
LongOneTime = micros() - TimeStamp; //calculate how long signal
was low
```

```
if (FoundLongOneQ() == false){
    //try this time interval out as new TimeUnit and see if next
pulse is the LongOne
    TimeUnit = LongOneTime;
       Serial.println("E");   //tell user of sync error
    goto ScanForLongOne;
    }
}

void ReceiveByte(){
ReadDelay = 1.5*TimeUnit;
//record stream starting with LSB. Always 8 bits
BitCounter = 0;
ReceivedByte = 0; //initialize data to be sent out serial port
while (BitCounter < 8){ //so goes from 0 to 7
    WaitForNegativeEdgeAndTimeStampIt();//wait for negative
edge; either z1 or o1
    PrepareToRecordBit();
    WaitToReadBitValue();//this puts us in the center of the
second TimeUnit
    ReadBitValueAndRecord();
    ++BitCounter; //have recorded a bit so move on to the next
bit
    }
}

void OutputByte(){
Serial.println(ReceivedByte);//send resulting byte out serial
port as a new line
}

void WaitForNegativeEdgeAndTimeStampIt(){
WaitForHigh();
WaitForLow();
TimeStamp = micros();//record time of negative edge
return;
}

void WaitForPostiveEdge(){
WaitForLow();
WaitForHigh();
return;
}

boolean FoundLongOneQ(){
LongOneTimeTimesTenThousand = 10000*LongOneTime;
TimeUnitTimesOneHundred = 100*TimeUnit;
```

```
if
(((LongOneTimeTimesTenThousand/MinimumTimeToleranceTimesOneHundr
ed) > TimeUnitTimesOneHundred) &&
((LongOneTimeTimesTenThousand/MaximumTimeToleranceTimesOneHundre
d) < TimeUnitTimesOneHundred)){
//LongOne width in spec so header has been detected
    return true;
    }else{
    return false;
    }
}

void PrepareToRecordBit(){
ReceivedByte = ReceivedByte >>1; //shift bits over 1 place to
the right  in preparation for next bit being recorded.
}

void WaitToReadBitValue(){
    WaitToReadBitValue:
    if ((micros() - TimeStamp) < ReadDelay){
    goto WaitToReadBitValue;
    }
}

void WaitForHigh(){
WaitingForHigh:
if (digitalRead(SerialInput) == 0)goto WaitingForHigh;
}

void WaitForLow(){
WaitingForLow:
if (digitalRead(SerialInput) == 1)goto WaitingForLow;
}

void ReadBitValueAndRecord(){
    if(digitalRead(SerialInput)== 0){ //this means a 1 was sent
        ReceivedByte = ReceivedByte | 0b10000000; //set MSB to 1
    }
}
```