Debugging Code Running on an Embedded ATtiny85, version 2.1

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

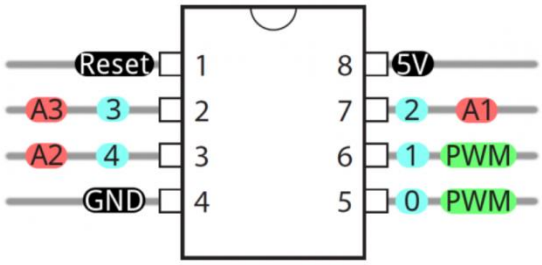
Conclusion

When limited memory, processing power, and uncommitted output pins exist, debugging the code can be difficult. The solution presented here addresses this problem using a single pin to transmit bytes from the ATtiny85 to a PC. These bytes are converted back into decimal values using Excel.

Contents

The Problem

The ATtiny85 is an entire computer system in an 8 pin Dual Inline Package[2]. Hidden in there are 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 is 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 what the program was doing via the single red/green LED connected to the device was maddening. I'm accustomed 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 8 MHz. In the end, I would be watching my code run in a totally different environment. That could hide the bugs I and generate new ones.

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

The First Step

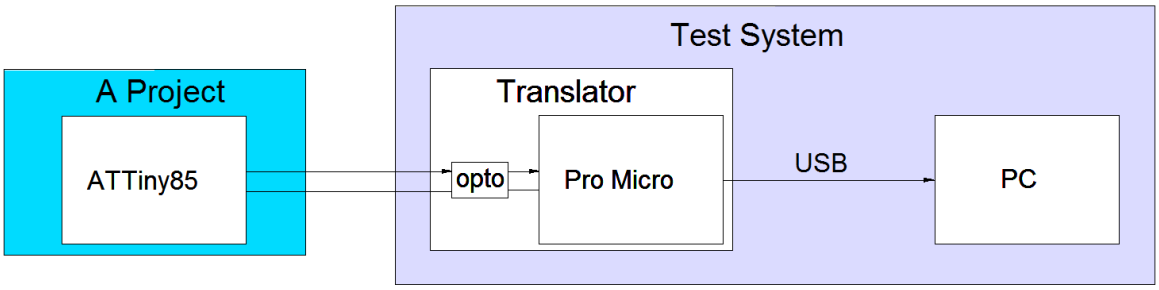
Since Arduino code can run on many devices, I first run my ATtiny85 code on a Pro Micro that is not embedded. I simulate sending signals to output pins with Serial.print() statements.

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 constraints to continue finding problems.

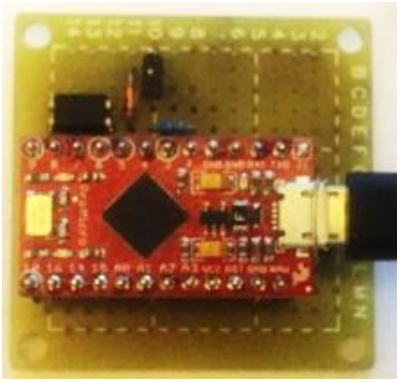
The Built-In Solution

The ATtiny85's has a host of powerful machine language debugging functions that drive the single Reset pin. Several people have written code with custom hardware to access these features[3]. However, I need to debug Arduino code and not deal with machine language or learning someone's debugging system.

My Solution

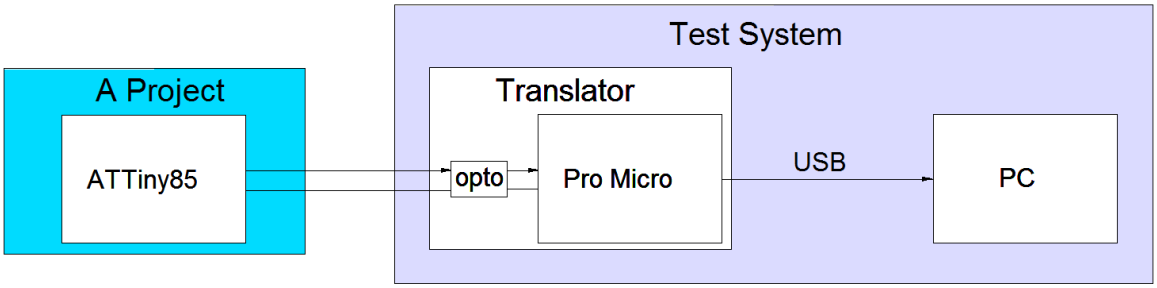


The embedded ATtiny85 drives one, user-specified, IO pin. This signal feeds into my Test System. The Test System consists of a Translator and a PC running a Terminal Emulator.



The Translator consists of an optoisolator and a Pro Micro. The optoisolator 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 128 bytes can be queued up in the ATtiny85 before transmission begins.

User's Guide

Initial Setup

First, build the hardware shown on page 19. Then download this code into the Pro Micro.

Go to your libraries folder, which is probably in your arduinosketchfolder. Create a folder called skinnyPrint and fill it with this .h file, .cpp file, and keywords.txt file. Restart the Arduino IDE before you use this library, so it sees the keywords.txt file.

If necessary, you will have to change your ATtiny85 code to free up an IO pin.

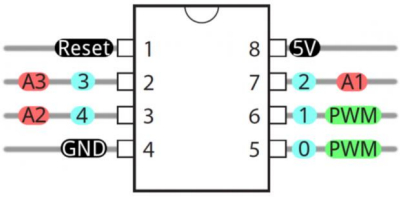
At the top of your program, put

```
#include <skinnyPrint.h>
```

To tell skinnyPrint which logical pin to use, put

```
singlePinPrint name(logical pin number);
```

after the end of setup() and before loop(). ***name*** is what you want to put in front of each of the commands. I suggest you keep it short. ***logical pin number*** is the logical and not the physical pin number to be used on the ATtiny85.



The physical pin numbers are inside the rectangle. The logical pin numbers are shown inside blue circles.

For example, I wrote

```
singlePinPrint sPP(3);
```

which means I have chosen the name sPP and will output data on logical pin 3 which is physical pin 2.

Printing Options

Three functions are used to take bytes from within the ATtiny85 and have them display on your monitor.

To print a single byte, use

name.`PrintN(data)`

It will print Now, which means your program will stop for about 60 ms as the bits go out.

To print a single byte quickly, use

name.`PrintF(data)`

It will print Fast, which means the byte will be put in a queue. This queue can hold up to 128 bytes. When there is time, call

name.Printer()

It will send all of the bytes in the queue to the monitor.

Examples:

```
#include <skinnyPrint.h>

Setup(){
}

singlePinPrint sPP(3);//define the name and the output pin

loop(){
     byte testByte = 55;
     sPP.PrintN(testByte);//55 will appear on the monitor
     for(byte index = 0;index < 67;index++){
          sPP.PrintF(index);//the queue is filled with 0 - 66
     }
     sPP.Printer();//send the queue to the monitor
}
```

The monitor will show 55 and then 0 through 66, one number on a line.

Post Processing with Excel

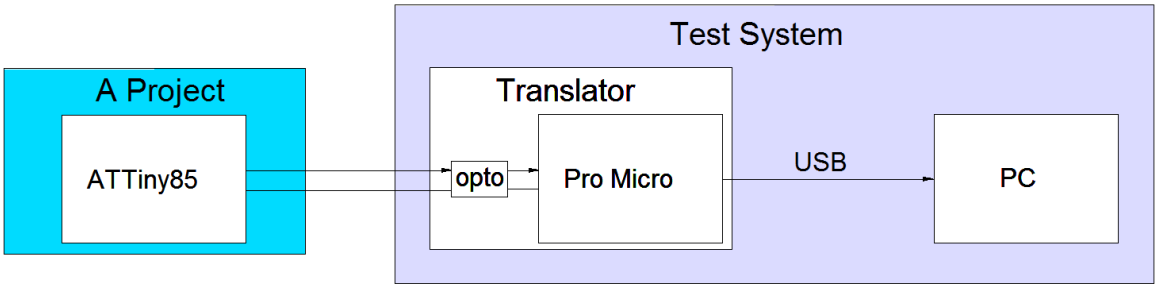
You will likely be evaluating the output of SkinnyPrint more than once. Converting the single bytes into numbers is both drudgery and error-prone. Excel does a great job of avoiding both. Copy the data from the monitor and paste it into this Excel spreadsheet. If you have good security on your PC, this spreadsheet will be flagged because it does contain custom functions.

To convert from four bytes into a long, use the custom function makelong(MSB,byte2,byte1,LSB).

To convert from two bytes into an integer, use the custom function makeinteger(MSB,LSB).

Once these mappings and equations are set up, you only have to copy and paste to get error-free conversions every time. The spreadsheet has a few examples.

Error Codes



You may 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 8 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 18 for details.

If more than 128 bytes are put into the printer queue, the ATtiny85 will send `250` followed by `251` to signal this overflow condition. It will then flush the queue.

Performance

It will take a total of 60 milliseconds for your ATtiny85 to send one byte. This very slow speed is due to the optoisolators. I empirically found this to be a reliable, if not painfully slow, data rate[5].

If you have 10 bytes in the queue, `Printer()` needs

$$10 \times 60\ milliseconds = 600\ milliseconds$$

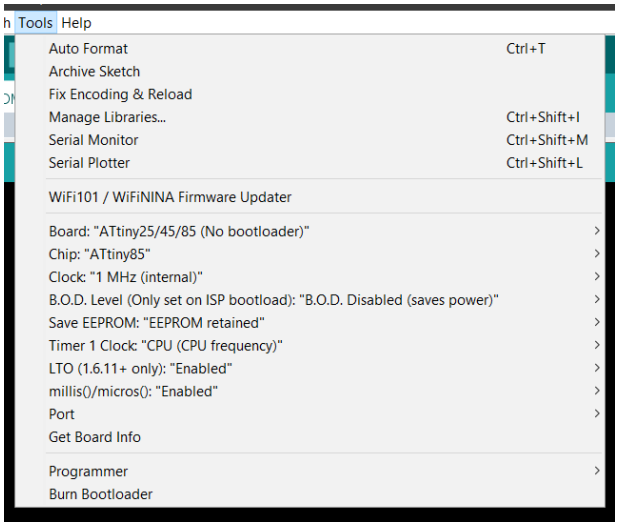
to get them all out. If the queue is full, it will take almost 8 seconds to empty.

[5] I am driving the optoisolator's input at its maximum current of 10 mA. Charge quickly built up in the phototransistor inside the optoisolators and the output goes low. This phototransistor has a very large base-emitter junction so it can collect as many photons as possible from the LED. This gives the optoisolatior decent transfer gain. This also means that a lot of charge is stored in this junction. Most of the turn off delay comes from getting rid of this charge. The ATtiny85 outputs close to zero volts. This discharges the small charge within the LED inside the optoisolators with around $\frac{1.4V}{330\ ohms} = 4.2\ mA$. However, none of the charge within the phototransistor is removed by this current. We must wait for its charge to dissipate without the help of an externally applied current.

Reading the ATtiny85's Internal EEPROM

One use for skinnyPrint is to be able to read the ATtiny85's EEPROM. The key to this puzzle can be found by searching the web using "Spence Konde ATtinyCore." You will find a superb collection of documentation and software tools that can be installed on the Arduino IDE.

The needed configuration is shown here.



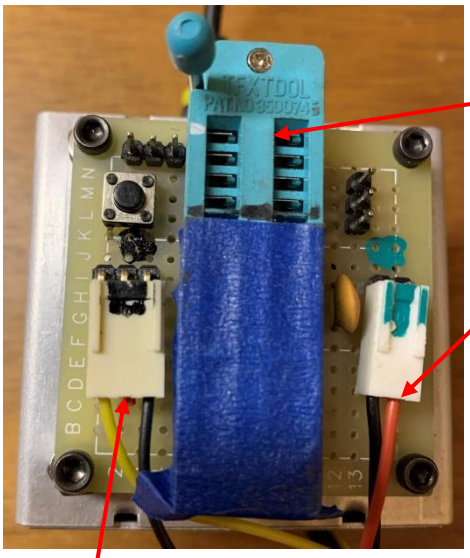
The Programmer should be set to USBtinyISP (slow).

Notice that it has the option

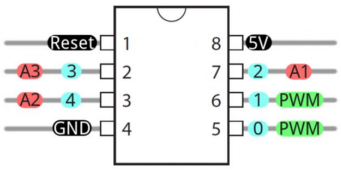
Save EEPROM: "EEPROM retained"

This enables me to change the software on the ATtiny85 without disturbing the EEPROM. As explained in his documentation, you must run "Burn Bootloader" once per device for this to work. This will corrupt the EEPROM, so be sure to do it before any useful data is stored there.

I have the SkinnyPrint code installed in my program under test, but it is only visible to the compiler by uncommenting `#define skinnyPrint`. If I need to read the EEPROM or get any other information from the ATtiny85, I enable this code. All related code is also wrapped in `#ifdef skinnyPrint`, so they only take up memory and real-time when needed.



I added a circuit board on top of my SkinnyPrint hardware. It holds a Zero Insertion Force (ZIF) socket, which makes connecting to an ATtiny85 quick and easy. I drop in the device and pull the lever down. There is no chance of bent pins.



The red and black wires attached to the right connector carry power to the device. The black wire connects to pin 4 of the ZIF, while the red wire connects to pin 8. A 0.1 *uf* capacitor also connects to pins 4 and 8. It provides instantaneous power to the ATtiny85.

The connector on the left has a black and yellow wire. These are the SkinnyPrint input conductors. The black wire connects to pin 4, and the yellow wire connects to pin 2. This means I must specify logical pin 3 as my output pin.

To use skinnyPrint without this new hardware, I unplug the black/yellow wire connector and run the wires to the device under test.

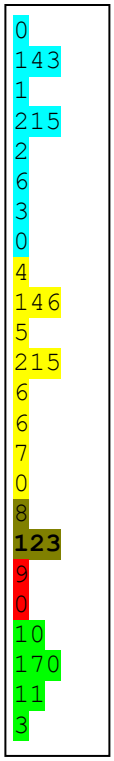
That square on the upper left side with the black circle in it is the Reset button. It connects pin 1 to pin 4 while it is being pressed, and that causes the device to reset.

On the upper right side are 3 test pins arranged vertically. The top pin collects to pin 7, the middle pin goes to pin 6, and the bottom pin goes to physical pin 5.

On the upper left side are 3 test pins arranged horizontally — the pin on the left ties to pin 4, which is ground. The middle pin connects to pin 3, and the right pin goes to pin 2, which is also the output of the skinnyPrint signal.

These test pins let me connect instruments to the ATtiny85's signal pins without interfering with the ZIF.

When I want to dump a block of EEPROM, I find it useful to print the address followed by the data. The incrementing address is easy to identify, so there is no risk of confusing the address and the data.



For example, I wanted to dump addresses 0 through 11. The highlighting lets me see each group of addresses. Bytes 0-3 and 4-7 are longs. Bytes 10 and 11 form an integer.

I write to the EEPROM in a rather clunky but straightforward way. A second utility program called writeEEPROM.ino contains all of my writes. I then set up the Arduino IDE with this program in one window, and the program under test is in another window. It is easy to switch between windows and load into the ATtiny85 either program. I also wrote a simple program called readEEPROM.ino that is in a third window.

Say I want to verify that my program responds correctly to EEPROM address 8. As shown here, it is set to 123. I use writeEEPROM.ino to write this address to 122. Then I load this program via my USBtinyISP.

Next, I load the program under test. If needed, I would move the updated ATtiny85 from this Programmer to my project and let the software run. If it doesn't need the rest of the hardware, I can let it run in the Programmer.

To read the EEPROM and see it on the screen I use readEEPROM and skinnyPrint.

A more elegant way to read and write the EEPROM would be to run I2C between the ATtiny85 and the Pro Micro. This would only work if there were no optoisolators between them. The ATtiny85 would unplug from its hardware and plug into the socket on top of the SkinnyPrint box. This socket would get power from the Pro Micro and have I2C with pull-ups connected to dedicated pins on the ATtiny85. The Pro Micro would be the master, and the ATtiny85 would be the slave. The ATtiny85 would have to run SoftWire because it does not have I2C hardware.

The user would load a program into the ATtiny85 using the Programmer and then move the device to the SkinnyPrint's socket. Then the user would be able to send commands to the Pro Micro and receive back data. It would be possible for the user to ask for any EEPROM address to be dumped or written.

This is a fair amount of software development, and the benefits are small. With the clunky approach, I have to load one or two programs into the ATtiny85 and then move it to the SkinnyPrint socket or connect the SkinnyPrint cable to the device plugged into its hardware. The elegant approach requires me to load one program into the ATtiny85, move it to the SkinnyPrint socket, and have full access to the EEPROM. If I need to read data from the ATtiny85 while it is plugged into its hardware, I can't use this approach.

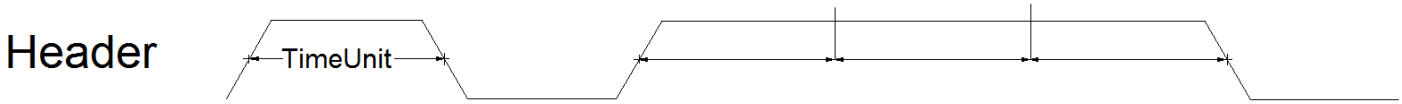
The Serial Signal

Of top priority was to use as few IO pins on the ATtiny85 as possible[6]. 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.[7]"

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 equals 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[8]".

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

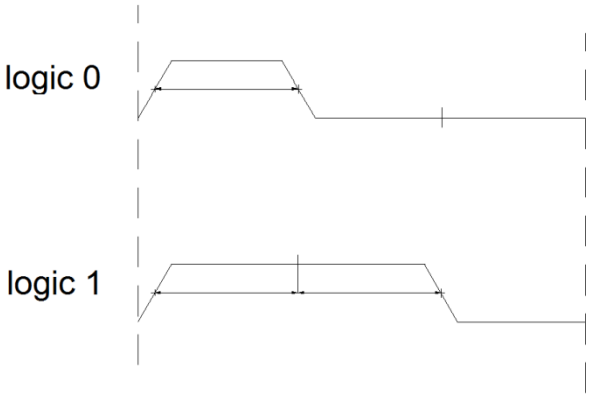
---

[6] 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.

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

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

The Bits

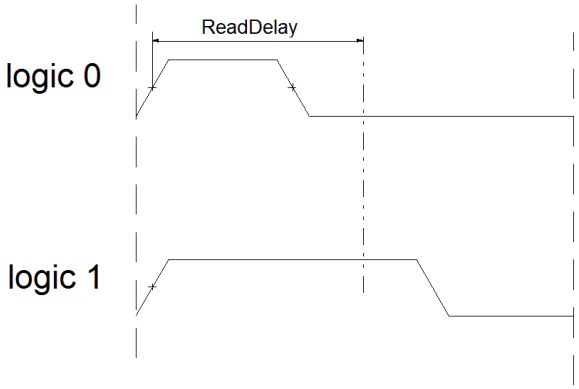


Both a logic 0 and logic 1 begin with a rising edge.

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 they 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 is 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().

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[9]. They look for a transition, do something, look for the next transition, and do something else.

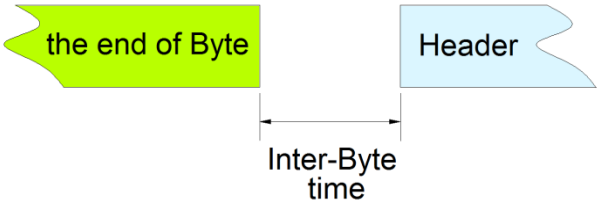
OutputByte() takes the assembled byte and ships it out to the PC with a Serial.print() command.

The Pro Micro's code can be found here.

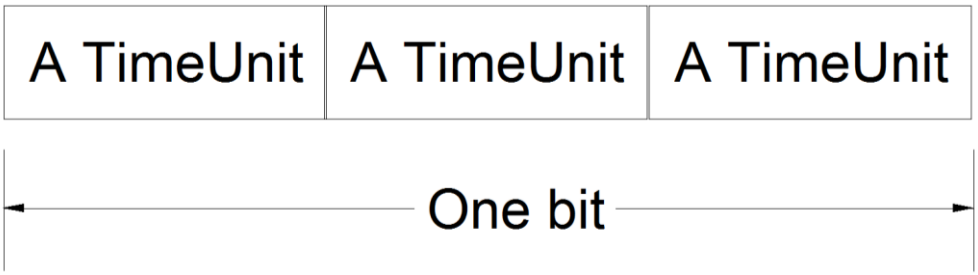
[9] See https://en.wikipedia.org/wiki/Finite-state_machine.

Adjusting Timing

There are two timing constants that set how fast data is transmitted to the Test System by the ATtiny85 code.

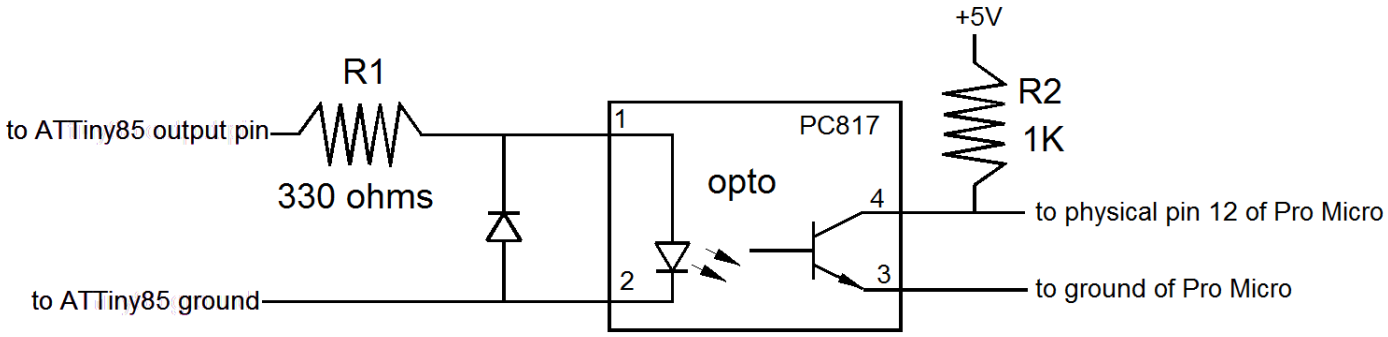


The first is `InterByteTimeMicroS`. It is the number of microseconds of delay between the end of one byte and the start of the next header. This is processing time for the receiver. If you see random E's being displayed on the PC, raise this number and see if it solves the problem. See page 15 for more detail.

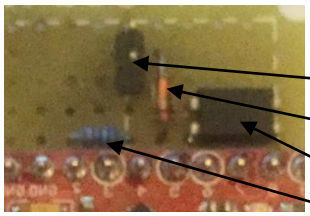


The second timing constant is `TimeUnitMicroS`. It is approximately the width of a TimeUnit, although another 200 microseconds are added to it as the code executes. If the bytes you are sending from the ATtiny85 are being corrupted, try increasing this time. Just remember, there are 6 TimeUnits in the 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 14 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.



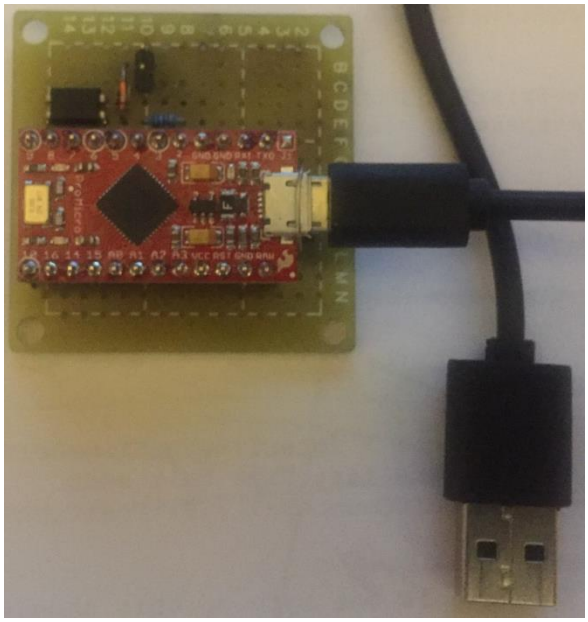
Layout is not critical. Here you see
the input connector,
diode,
the optoisolator, and the
input resistor, R1. Not shown is pull up resistor R2.
All parts mount on a board with 0.1 inch spaced holes.

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 backward, 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.

Acknowledgment

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.

Thanks to Spence Konde for his excellent documentation and code.

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. In order to minimize the number of unwanted articles I send, please tell me which of these subjects interest you:

1. Metalwork
2. Electronics and software
3. Kayaking
4. Electric bikes

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