Microcontrollers For Everyone!

Lowering the barrier to entry to useful microcontroller development for casual users.

Table of Contents

Goal, Requirements, and Anti-Requirements

The goal of the Microcontrollers For Everyone (MFE) project is to encourage new sets of casual users, including students (middle school, high school, and up), hobbyists, and other non-career users, to learn about, have fun with, and build useful projects with ***state-of-the-art microcontroller technologies***. The goal of MFE is explicitly ***not*** to hide or abstract the microcontroller technologies, but rather, to make them ***more approachable, transparent, and forgiving*** so that casual users can learn the same fundamental concepts that are used by career users, but without the career investment.

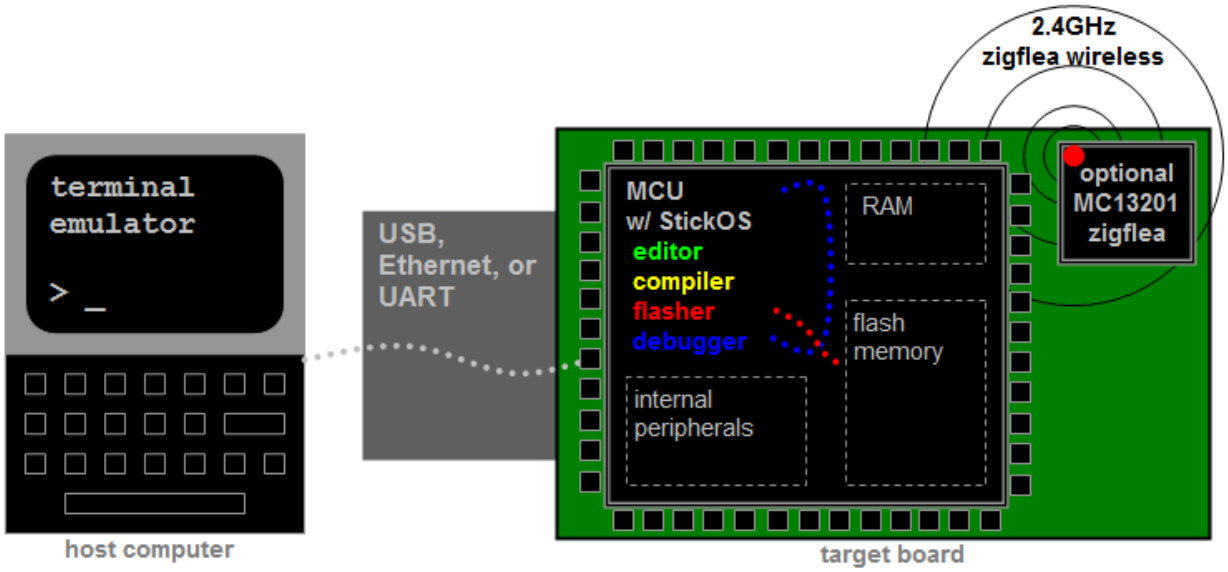
To achieve these goals, the following requirements must be met:

- ***the microcontroller must be approachable and transparent*** -- the casual user must be able to approach the microcontroller in the same hands-on way that they would approach a lawn sprinkler system in their yard; the user must be able to touch and manipulate the inner workings of the MCU, to see and feel what is going on with every pin and every peripheral, step-by-step, and in real time.
- ***the microcontroller must be forgiving*** -- the casual user must be able to incrementally adjust a program and watch the resulting change, in the same way that they would adjust a lawn sprinkler system in their yard, adjusting one sprinkler head at a time until it had the desired coverage, and then moving on to the next head as the system continues to run.
- ***the user must be able to build the microcontroller system "from scratch"*** -- if the casual user is to truly learn the fundamental concepts of microcontrollers, they can't start with a "black box" that does everything for them; "wizards" may be conducive to getting a small set of canned tasks accomplished, but they rarely provide the learning and fun of truly understanding what you are doing.
- ***the user must get nearly immediate results, encouragement, and feedback*** -- effort and results/fun must go hand-in-hand; after the first five minutes of effort, the user should see five minutes of results/fun, to thereby encourage the user to put in more effort; after the first hour of effort, the user should see an hour of results/fun, and so on.

The first two of these requirements can only be met in a truly interactive system -- ***interactivity, by its very nature, is conducive to learning!*** With interactivity, you can experiment with the microcontroller and immediately correlate your actions with the results of those actions -- if something doesn't work, you can immediately try something else. Furthermore, in a truly interactive system, you can build your solution in pieces, programming the pieces that you understand and "running" the remaining pieces "by hand", experimenting, until you figure out the entire solution. And on your way to programming the entire solution, you can change your program on the fly, back it up, or skip it forward, never losing state!

Ironically, if there are any ***anti-requirements*** for a system like this, it would be the traditional "reset-and-run" methodology of debugging, as well as the traditional "edit-compile-debug" loop of software development -- both of which are exemplified by nearly all "introductory" microcontroller systems today, such as ***Arduino and BASIC Stamp***. The "reset-and-run" methodology of debugging basically is the opposite of "approachable and transparent" -- instead, the bottom line of reset-and-run is "the program didn't work, somewhere, would you like to try again?". The "edit-compile-debug" isn't much better and is the opposite of "forgiving" -- basically every time the program doesn't work, you get to lose all your state (both in the microcontroller and in your brain!) and try again, from the very beginning.

The StickOS Approach



StickOS® BASIC is an entirely MCU-resident patented interactive programming environment, which includes an easy-to-use editor, transparent line-by-line compiler, interactive debugger, performance profiler, and flash filesystem, ***all running entirely within the MCU and controlled thru an interactive command-line user interface.*** In StickOS, external MCU pins may be bound to "pin variables" for manipulation or examination, and internal MCU peripherals may be managed by BASIC control statements and interrupt handlers. A StickOS-capable MCU may be connected to a host computer via a variety of transports and may then be controlled by any terminal emulator program, ***including a web-page terminal emulator running on a Chromium-based web browser***, with no additional software or hardware required on the host computer. Once program development is complete, the MCU may be configured to autorun its BASIC program autonomously.

Most importantly, though, StickOS is entirely interactive... From the terminal emulator, the user communicates directly with the MCU and can ask the MCU to manipulate or examine any of its internal state, the state of any of its output or input pins, or the state of any of its internal peripherals! You can execute BASIC statements in "immediate mode" directly from the command prompt (for immediate feedback!), write and run a program, or even interactively debug a program.

The interactive debugger of StickOS supports ***all the same fundamental concepts used by career users***, including:

- trace or single-step program execution,
- use sampling profiling to see where the program is spending its time,
- use breakpoints, assertions, and watchpoints,
- use live variable (and pin) manipulation and examination while the program is stopped, and
- even edit-and-continue the program!!! (OK, career users actually can't do this! :-)

The StickOS BASIC language similarly supports ***most all of the same fundamental concepts used by career users***, including:

- main program loop
- interrupt handlers, maskable and disablable
    - 4 periodic timer interrupts
    - pin interrupts on any pin
    - peripheral interrupts
- block structured programming constructs
- parameterized subroutines
- bit manipulation operators
- variable sized data types, arrays, and strings
- external pin manipulation
    - digital input/output
    - analog input/output (PWM)
    - servo output
    - frequency output
- serial input/output
    - uart
    - spi
    - i2c
- raw MCU register examination/manipulation

StickOS BASIC also has other features traditionally missing from other development environments:

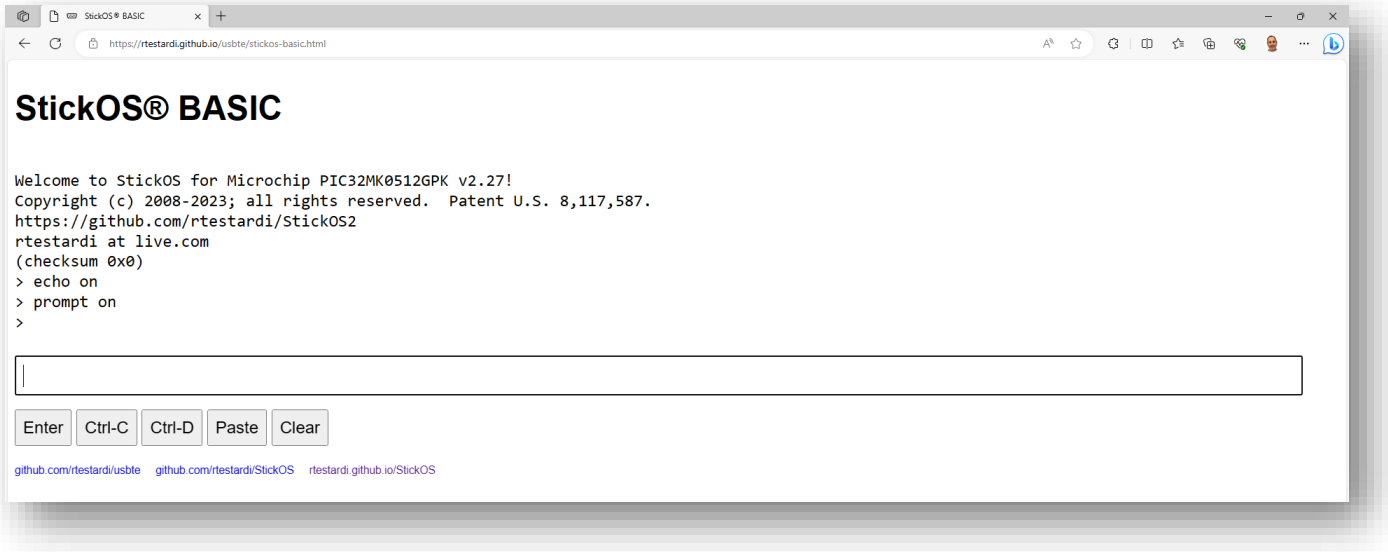
- 2.4GHz zigflea wireless transport (using an MCU-external zigbee chip)
    - remote control via a telnet/rlogin-like interface
    - remote variable access in BASIC
    - wireless BASIC program update

For more information on StickOS, see: CPUStick™ and StickOS® -- Embedded Systems Made Easy (rtestardi.github.io). For an easy language overview, see the quickref.v1.90.pdf (rtestardi.github.io).

How Easy is StickOS, Really?

Using a Flea-Scope™, logging into StickOS is as easy as:

1. Connect your Flea-Scope™ to the host computer via USB.
2. Open the web-page terminal emulator at `https://rtestardi.github.io/usbte/stickos-basic.html`
3. Click the “Connect” button, select your Flea-Scope, and click “Connect” again to authorize USB access:



Once you are logged into StickOS, you're ready for action! The following examples, which give immediate results, encouragement, and feedback, use a Flea-Scope™ with the PIC32MK0512GPK064 MCU as an example of the kinds of trivial things you can do with StickOS.

1 of 6: Blink an LED

As the "hello world!" of embedded programming, let's get the green LED “e2” to blink on the a Flea-Scope™ board:

```
> 10 dim led as pin e2 for digital output
> 20 while 1 do
> 30   led = !led
> 40 endwhile
> run
_
```

Line 10 of the program dimensions (declares) a variable named "led" that is bound to pin e2 of the Flea-Scope™ board, which is configured for digital output; from then on, any manipulation of the variable is immediately reflected at the pin. Lines 20-40 of the program form the main loop of the program. Line 30 simply inverts the state of the led on pin e2 of the Flea-Scope™ board, inside the loop.

Oops! Something is wrong -- the LED isn't blinking -- but maybe it seems to be half on...

So let's stop the program with a “<Ctrl-C>”, and then try single-stepping the program by entering the “step on” command:

```
<Ctrl-C>
STOP at line 30!
> step on
> _
```

Then, with single-step mode enabled, whenever we press “<Enter>”, we single-step one line of code... Go ahead and press <Enter> a few times and you will see the LED actually turns on and off and on and off, as we step thru lines of code:

```
<Enter>
STOP at line 20!
> <Enter>
STOP at line 30!
> <Enter>
STOP at line 40!
> <Enter>
STOP at line 20!
> <Enter>
STOP at line 30!
> <Enter>
STOP at line 40!
> <Enter>
STOP at line 20!
> <Enter>
STOP at line 30!
> _
```

Oh! We forgot to add a delay! So let's disable single-step mode with the “step off” command, fix the program by adding a 500ms delay at line 35, just after inverting the state of the LED, and then continue from where we left off:

```
> step off
> 35 sleep 500 ms
> cont
_
```

Much better!

OK, now let's get a bit more interactive with the MCU and stop the program again with a <Ctrl-C>... Then we'll examine the state of the LED (i.e., print the value of the "led" variable) and then blink the LED by hand...

```
<Ctrl-C>
STOP at line 35
> print led
0
> led = 1
> print led
1
> led = 0
> print led
0
> _
```

Notice that when we change the variable, the LED on the Flea-Scope™ board changes state!

2 of 6: Blink an LED with a Timer ISR

Now that our LED is blinking, let's graduate to using a timer ISR (Interrupt Service Routine) rather than a programmed loop -- this way we can do other things with the "main loop" in the future. We'll keep line 10 of the program, but wipe out all the lines that followed, list the program, and start from there:

```
> delete 20-
> list
  10 dim led as pin e2 for digital output
end
> 20 configure timer 1 for 500 ms
> 30 on timer 1 do led = !led
> 40 halt
> run
_
```

Line 10 still dimensions (declares) a variable named "led" that is bound to pin e2 of the Flea-Scope™ board, which is configured for digital output; line 20 configures timer #1 to pop every 500ms; line 30 enables the timer interrupt and says that every time the timer pops, we should run the statement "led = !led" (if we had more work to do we could have called a subroutine); finally line 40 just puts the main loop to sleep -- we just service interrupts from then on!

Wow, that worked on the first try! :-)

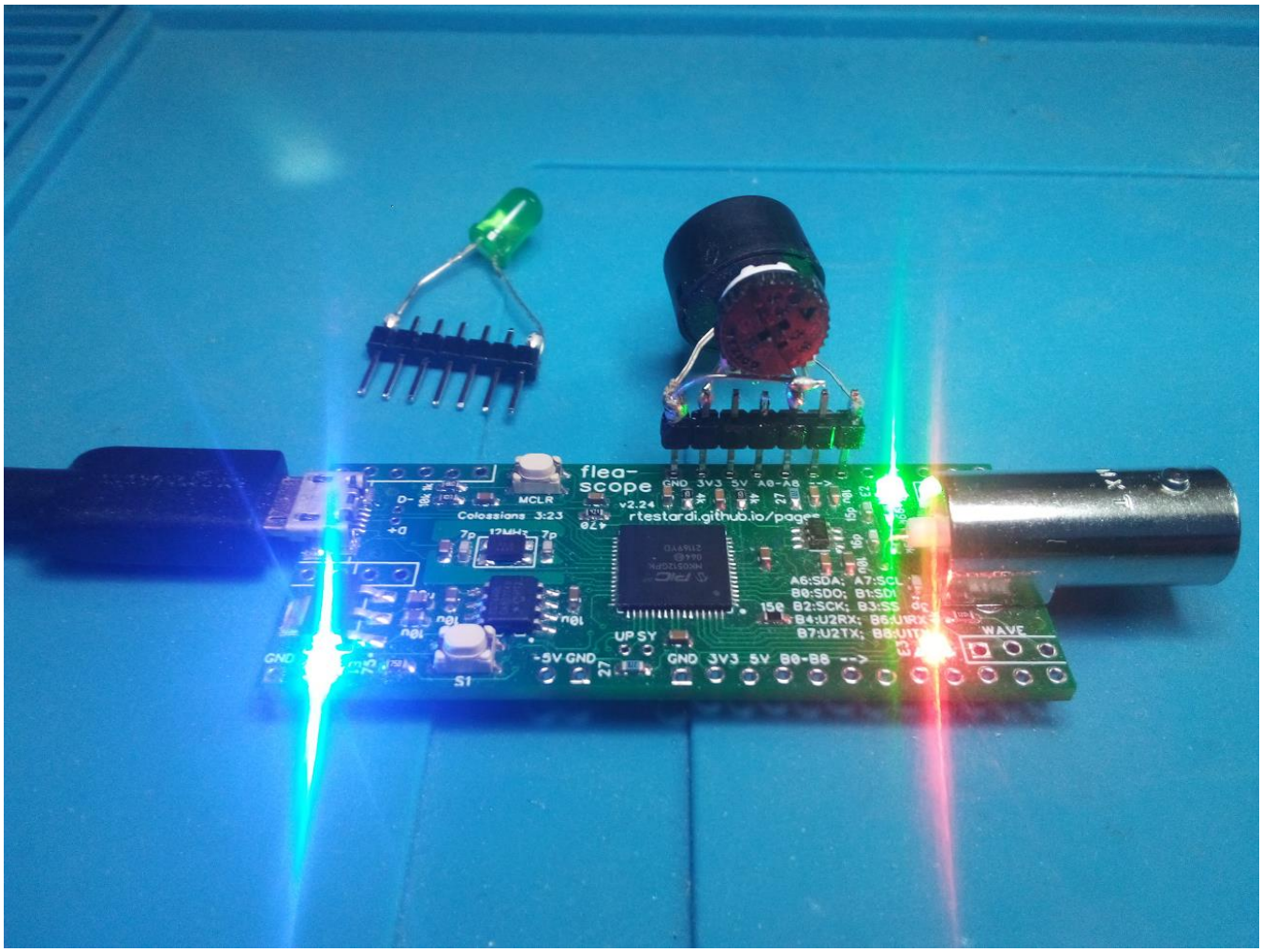
3 of 6: Blink an LED with an Output Compare Module

Now that we understand timers a bit, let's try using an MCU "output compare module" peripheral so we can blink the LED without using the CPU at all! Again, these are all the ***same fundamental concepts used by career users****!!!*

Unfortunately, we can't use the existing LED on the Flea-Scope™ board for this experiment, as the pin it is connected to does not have output compare functionality (since I saved the extra functionality for pins that could be used by the user), but we can simply attach a LED between pin "a3" and ground, being careful to make sure the cathode is on ground. (If your LED is not able to handle 3.3V input voltage, you will also need a current limiting resistor.)

To do this, I just used a header pin strip with 7 positions to plug into the Flea-Scope. Note that the pads on the Flea-Scope are offset with a small zig-zag, so you can just press the header pin strip in without soldering, and it will stay there and form good electrical connections!

My pin strips for the LED in the experiment now, and potentiometer and buzzer for the experiment below look like:



We have to stop the program again with a <Ctrl-C>, and then we'll just do this at the command line (not even writing a program) because we're a bit unsure of ourselves...

```
<Ctrl-C>
STOP at line 40
> dim hz as pin a3 for frequency output
> hz = 1
> _
```

The first line there dimensioned (declared) a variable named "hz" that is bound to pin a3 of the Flea-Scope™ board, which is configured to receive the output of an output compare module; from then on, any manipulation of the variable is immediately programmed as the output frequency, in Hz, of the output compare module. The second line sets the "hz" variable to 1, resulting in an almost 1 Hz signal on pin a3 of the Flea-Scope™ board.

Unfortunately, the output compare modules of various MCUs have lower limits of how slow they can go, and the one on PIC32MK only goes down to 3 Hz, which we can see by querying the variable after we set it -- it was silently rounded up to a value of 3 (note that “?” is a shortcut for “print”):

```
> ? hz
3
> _
```

Wow, that is cool!!! Let's make it go faster:

```
> hz = 10
> _
```

And faster:

```
> hz = 100
> _
```

I can barely see it blink if I move my head... I want to hook up a buzzer to this thing, but before we go any farther, let's try another experiment...

4 of 6: Read a Potentiometer with an A/D Converter

For this experiment, we'll hook up the middle lead of a potentiometer (maybe 1k ohm to 10k ohm, the exact value does not matter) to pin a1 of the Flea-Scope™ board, and the outer leads of the potentiometer to ground and 3.3V. Let's again do this at the command line (no program) so we can see how it works:

```
> dim pot as pin a1 for analog input
> print pot
1876
> _
```

The first line there dimensioned (declared) a variable named "pot" that is bound to pin a1 of the Flea-Scope™ board, which is configured for analog input thru the A/D converter; from then on, any reference of the variable reflects the current number of millivolts read on the pin. The second line prints the value of the "pot" variable, displaying the number of millivolts on pin a1 of the Flea-Scope™ board.

Now let's turn the pot a bit and try it again:

```
> print pot
1201
> _
```

Cool!!!

5 of 6: Tying It All Together -- An Analog to Frequency Converter!

OK, let's replace the LED on pin a3 of the Flea-Scope™ board with a small buzzer... And now let's write a program using all of the skills we just learned to create an "analog to frequency converter" (which is no small feat in the real world)! We'll start out by wiping out everything we've done so far with a "new" command, and then move from there:

```
> new
> 10 dim buzzer as pin a3 for frequency output
> 20 dim pot as pin a1 for analog input
> 30 configure timer 1 for 100 ms
> 40 on timer 1 do buzzer = pot
> 50 halt
> run
_
```

Now turn the pot and listen to the buzzer -- it goes from 0 to 3300 Hz, updated from the pot every 100ms!

We rock! :-)

6 of 6: Let's Go Crazy -- A Wireless Analog to Frequency Converter!!!

And if we have a Zigflea external 2.4GHz wireless transceiver based on the MC13201 attached to each of two MCUs, at about a $5 cost in parts per MCU, we can even make a wireless embedded system! We can connect the potentiometer to one MCU and the buzzer to the other! "Remote Variables" in StickOS easily allow one MCU to manipulate variables (or pins) on another MCU, with just a single line of code!

Assuming one of our MCUs has been configured as nodeid 0 (with the "nodeid 0" command) and has the buzzer on pin a3, and the other MCU has been configured as nodeid 1 (with the "nodeid 1" command) and has the potentiometer on pin a1, ***we can just physically connect nodeid 0 to the host computer and do all the work from there.***

Our terminal emulator is connected to nodeid 0; we'll start out by logging into nodeid 1 from nodeid 0, pressing <Enter> for a prompt:

```
> nodeid
0
> connect 1
press Ctrl-D to disconnect...
<Enter>
Welcome to StickOS for Microchip PIC32MK0512GPK v2.27!
Copyright (c) 2008-2023; all rights reserved.  Patent U.S. 8,117,587.
https://github.com/rtestardi/StickOS2
rtestardi at live.com
(checksum 0x0)
> _
```

Then we'll write the "buzzer" part of the program on nodeid 1:

```
> nodeid
1
> 10 dim buzzer as pin a3 for frequency output
> 20 halt
> run
_
```

Notice that nodeid 1 just dimensions (declares) a variable named "buzzer" that is bound to pin a3 of the Flea-Scope™ board, which is configured to receive the output of an output compare module; from then on, any manipulation of the variable, ***even if initiated wirelessly from a remote MCU***, is immediately programmed as the output frequency, in Hz, of the output compare module. Then nodeid 1 simply halts program execution, just waiting for remote variable manipulation from nodeid 0. Finally, we start the program running on nodeid 1.

Then we'll disconnect from nodeid 1 by pressing <Ctrl-D> (its program is still running), and we're back on nodeid 0:

```
<Ctrl-D>
...disconnected
> _
```

Finally we'll write the program on nodeid 0, just like before, but instead of dimensioning the buzzer variable locally as we did before, we'll dimension it as being remote on nodeid 1.

```
> nodeid
0
> 10 dim buzzer as remote on nodeid 1
> 20 dim pot as pin a1 for analog input
> 30 configure timer 1 for 100 ms
> 40 on timer 1 do buzzer = pot
> 50 halt
> run
_
```

Now turn the pot on nodeid 0 and listen to the buzzer on nodeid 1 -- it goes from 0 to 3300 Hz, updated from the pot every 100ms!

This is just crazy easy! And fun!

Curriculums, Etc.

If we get this going, I can come up with curriculums that take the user thru lots of projects, step by step like we did the voltage to frequency converter, above, including:

- TBD

Conclusion

StickOS® BASIC and makes quick work of simple microcontroller experiments! And even more complicated ones! More importantly, it is fully interactive, so users can manipulate their circuit right from a web-page user interface, while it is live! They can single-step programs, even change code and continue from where they left off! Interactivity is a fundamental requirement for deep understanding of circuits, and we hope StickOS® BASIC and

If you want to build a toaster oven temperature controller for SMT reflow soldering at home, see:
`https://rtestardi.github.io/usbte/toaster.pdf`

About the Author

Richard Testardi has a wife and 17yo daughter and lives in Colorado. He is grateful most of the time and Christian. He loves anything outdoors or math/science related. In the future, he hopes to be teaching high school students. He lives without a cell phone (well, except a sim-less phone for interoperability testing!)!