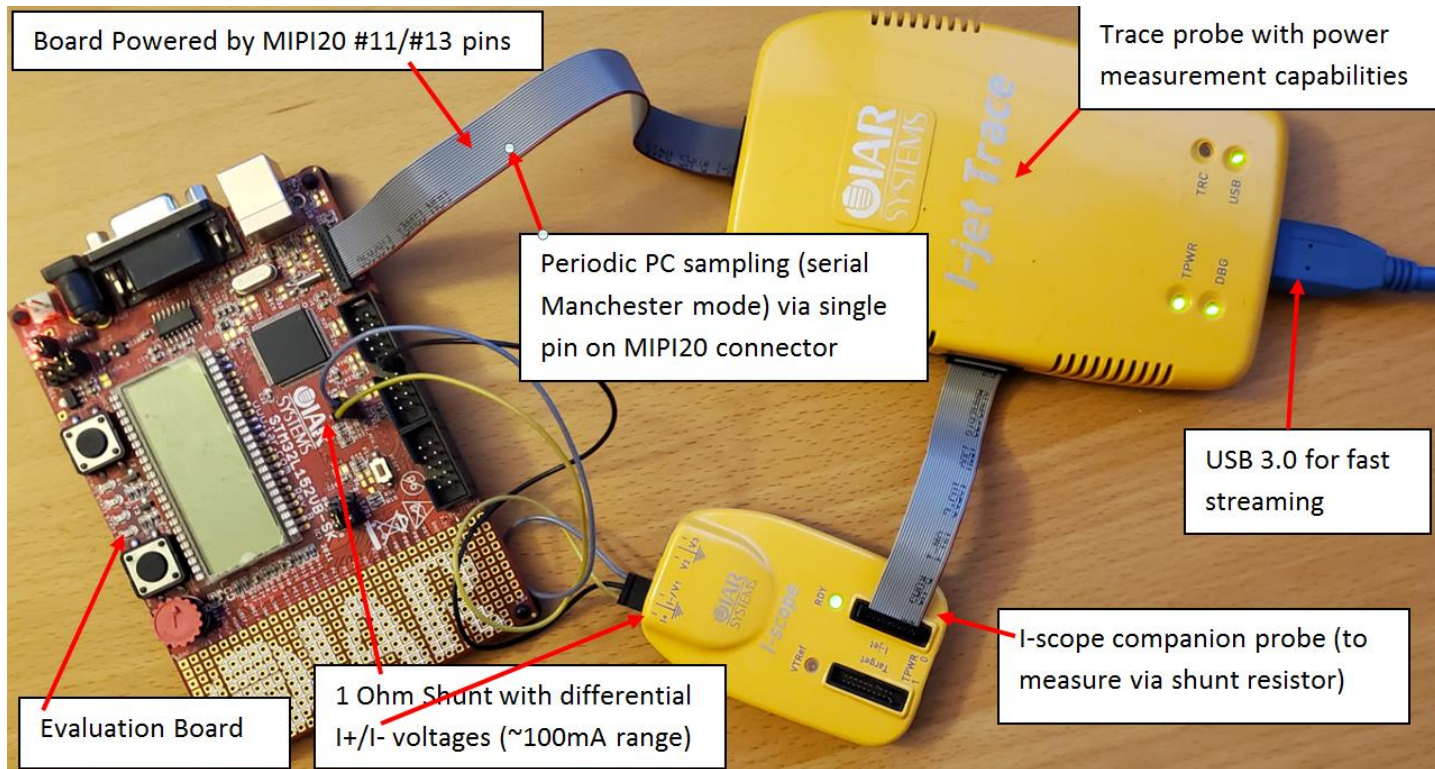


Demo Setup (I-jet Trace + I-scope)



Power measurements principle:

1. Probe is providing power to the target and is measuring it.
2. Probe is performing periodic PC sampling
3. PC samples are correlated to power measurements

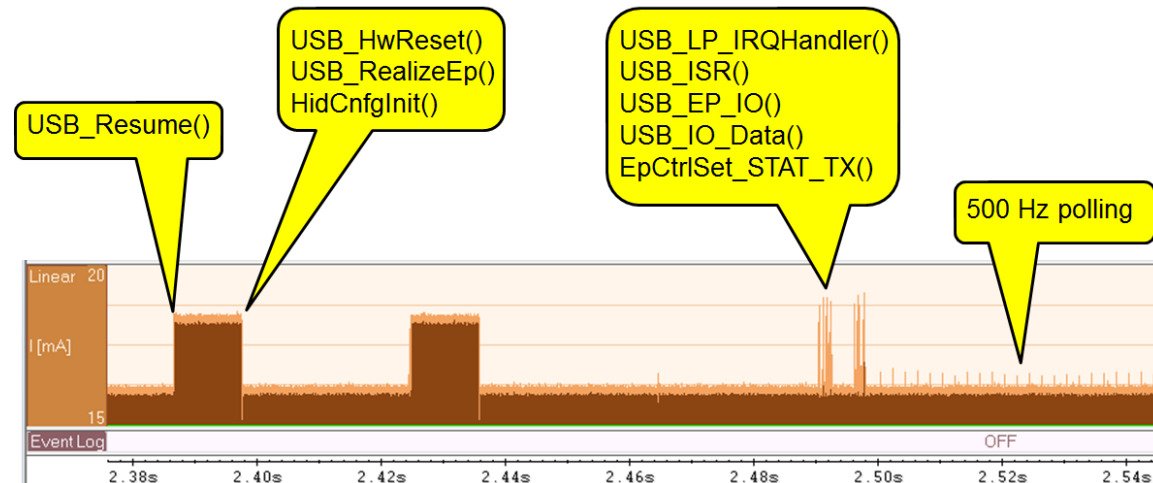
Option (extra I-scope probe/adaptor)

- Measurement can be performed via shunt-resistor directly on MCU power-rail
- It may also work with non-trace, simpler I-jet

Demo (using Embedded Workbench)

- Differences between 'power-delivered' vis 'shunt' ...
- What can be really seen (more than most think ...).
- Some interesting use case (can we see power of fetch-logic?).
 - Are (some?) RISC-V implementations have same 'feature'?
- Some practical snapshot:

- USB mouse application
- Profile from cable connect (enumeration)



What is needed on RISC-V side?

- Power delivery via MIPI-20
 - **Nexus Trace TG defines that MIPI-20 connector option** (two TgtPwr pins).
- Periodic PC sampling
 - PC can be periodically sampled by reading dedicated register – it is hard to have it ‘uniform’ ...
 - PC can be sent (via trace) - **Nexus Trace provides full, periodic PC packets**. Can be configured to ONLY enable these (every ‘N-cycles’).
 - Also **Nexus Trace Control defines slow serial** (UART and Manchester) trace option (good for small IoT size MCUs).

NOTES:

- Periodic PC sampling can be used for 100% non-intrusive statistical profiling.
- It was also a question (by Mark) about some ‘random-sampling’. Very non-regular is certainly not OK, but some variation (like spread-spectrum ...) should not be hard.
- Power is considered as ‘integral of power-curve’, so if one instruction takes a lot of power, it will be visible in total consumption.
- Have debugger active will affect measurements – but for example Manchester encoding is same as far as GPIO activity regardless of byte value, so offset can be ‘constant’.

Screen-shots (as presented during live-demo)

- Demo code (setting more and more LEDs, so power goes up).

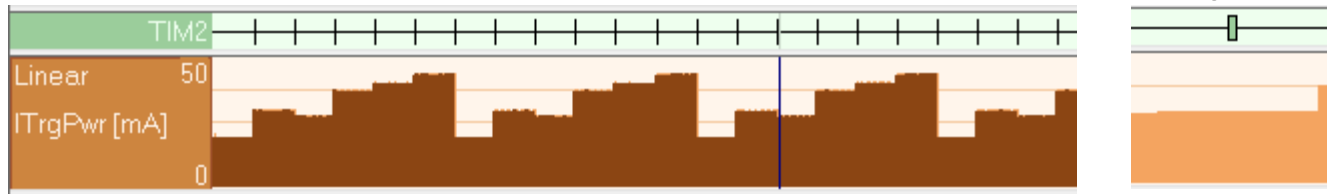
```
while(1)
{
    // LEDs pattern 0000
    while(!s_Tick);
    s_Tick = 0;
    LEDsSet(0x1);
    // LEDs pattern 0001
    while(!s_Tick);
    s_Tick = 0;
    #if 1                // Enable to see something interesting ...
    while(!s_Tick);
    s_Tick = 0;
    #endif
    LEDsSet(0x3);
    // LEDs pattern 0011
    while(!s_Tick);
    s_Tick = 0;
    LEDsSet(0x7);
    // LEDs pattern 0111
    while(!s_Tick);
    s_Tick = 0;
    LEDsSet(0xF);
    // LEDs pattern 1111
    while(!s_Tick);
    s_Tick = 0;
```

NOTES:

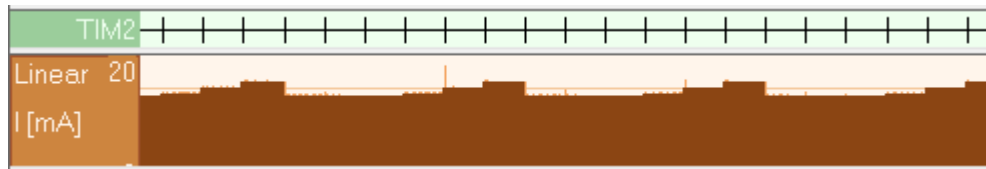
- Variable 's_Tick' is set to 1 in periodic interrupt handler. Main code is looping waiting for that change by doing 'while (!s_Tick);'
- That '#if 1' section will make one LED pattern to be 2x as long (but it also makes some other differences)

Screen-shots (as presented during live-demo)

- Power for entire board (I-scope disconnected – big zoom on right shows delay from interrupt to elevated consumption). It is partially due to code need to set LED on, but it is also related to power not being seen ‘instant’:



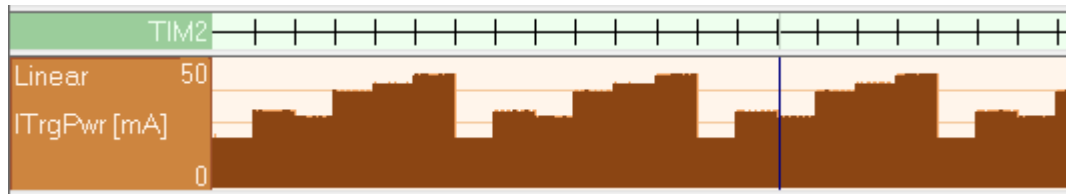
- Same graph with MCU 3.3V rail is monitored via I-scope (and shunt)



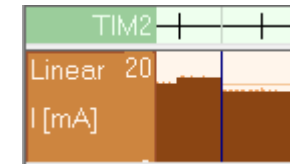
- Power is smaller (as LEDs itself are not ‘counted’, but GPIO current still is – that graph is ‘reversed’ as GPIO current is higher when LED is OFF!).
- Clicking on graph is synchronizing code to that spot – TIM2 is periodic timer interrupt

Screen-shots (as presented during live-demo)

- 2x wider LED level here:



- and here:



- This vertical line shows location of #if 0 in code. Assembly code is identical:

	while(!s_Tick);		
	0x800'020e: 0x6820	LDR	R0, [R4]
	0x800'0210: 0x2800	CMP	R0, #0
→	0x800'0212: 0xd0fc	BEQ.N	0x800'020e
	s_Tick = 0;		
	0x800'0214: 0x2000	MOVS	R0, #0
	0x800'0216: 0x6020	STR	R0, [R4]
	while(!s_Tick);		
	0x800'0218: 0x6820	LDR	R0, [R4]
	0x800'021a: 0x2800	CMP	R0, #0
	0x800'021c: 0xd0fc	BEQ.N	0x800'0218
	s_Tick = 0;		
	0x800'021e: 0x2000	MOVS	R0, #0
	0x800'0220: 0x6020	STR	R0, [R4]

- We have IDENTICAL loops. One is looping at address 0x800'020E and second at address 0x800'0218.
- First loop takes more energy (~2ma which is ~10% of MCU power!) than second one.
- Branch destination is NOT 32-bit aligned and it seems it 10% less efficient!
- Compiler may take advantage of this, but it only makes sense if code is spending a lot of time there (trace, pragma?)