Real Time Debugging

What to do when a breakpoint just won't do!

CyberData Corporation
Innovation For a Global Future

Lloyd Moore, President

Lloyd@CyberData-Robotics.com

www.CyberData-Robotics.com

Northwest C++ Users Group, November 2013

Overview

Definition of Real Time

Problems with Breakpoints

printf() – the old standby

Memory Buffers

I.C.E. and Trace Buffers

On Chip Debugging

Protocol Sniffers

Logic Analyzers and Scopes

Techniques & Best Practices

Summary

Definition of Real Time

MANY definitions depending on the context of the problem and who you talk to!

All generally refer to a condition where the answer is wrong if it is "late"

Soft Real Time: Refers to a condition where you have a guarantee that you will get into some portion of the code in a bounded time (latency/delay).

- Typically the ISR routine based on an hardware interrupt
- Generally what is intended if an operating system is in use

Hard Real Time: Refers to a condition where you have a very tight latency requirement, often on the same order as the machine cycle of the processor. Jitter may also be a requirement – cannot trigger too early either.

- *OS CANNOT be involved as it is just too complex*
- *Typically used with smaller microcontrollers or programmable logic*

Real Time Problems are fundamentally different from other problems because the answer was typically correct at some point in time, just not *this* point in time.

Problems with Breakpoints

- Breakpoints stop the thread of the process in which they are contained
- Only thread execution containing the breakpoint is guaranteed to be stopped, though typically the containing process is stopped as well.
  - Not all threads will stop at the same time in any case, particularly on multiple processors!
- Other aspects of the system being debugged are typically <u>NOT</u> stopped
  - Hardware generating interrupts
  - Timers and real time clocks
  - Other processes which may be communicating over some port/pipe
  - Other processors in the system
- The root of the problem is that part of the system you are debugging is stopped and part of the system continues to run.
- ALL Real Time guarantees just went out the window! Any answers are now likely late and therefore, wrong!
- Leads to inconsistent state of the overall system as some things continue running and others are stopped.

Some Real World Implications

In machine issues:

- Buffer overflows / overwrites for shared memory buffers
- Interrupts get missed and DMA status become corrupted
- Timeout values expire without even being considered

Between machine issues:

- Communication port overflows – data being sent not removed/received
  - Have actually had this crash commercial USB drivers forcing PC reboot!
- Upstream data sources time out waiting for responses
- Downstream data sinks may enter unexpected conditions
- These are generally good test cases anyway as they are real world anyway – **find em & fix em!**

Hardware / physical issues:

- Motors may continue to run!! (REALLY BAD!)
  - Your robot runs off the table and/or into a wall! (Bigger bots may go through a wall!!!)
- Processes that are being controlled may enter unsafe operating conditions.
  - Heater may continue to heat once desired temperature is reached
- Also really good to simulate as they *could* happen in production – **fix any safety issues!!**

Well then just use printf() or similar!

- This is what most developers do when faced with a breakpoint issue
- Advantages:
  - Quick, simple, common practice
  - Keeps the process running
  - Works in many cases
- Disadvantages:
  - Still alters program timing, sometimes considerably – consider serial port debugging even at 115200 baud – 868+uS to send 10 chars
  - Needs some other resource to be activated which can radically alter system state
    - I/O stream – generally the fastest but will trigger other processes
    - Disk access – may or may not block based on buffering, other process activated
    - COM port – Physical port slow, virtual COM port radically alters USB environment
  - Even non-blocking calls will alter thread behavior and timing
  - Simply not fast enough for many real time applications!
  - In small embedded applications may not have needed resources available
  - May alter hardware interrupt patterns and possibly DMA usage
- Tracepoints are basically breakpoints tied to printf()

Memory Buffer Debugging

The practice of reserving a section of memory to write debugging information into. May be a linear array or circular buffer.

- Advantages:
  - Much faster than printf() but will still take some time
  - Much less likely to alter gross thread behavior or other system state – new threads not spun up
- Disadvantages:
  - Usually have to write your own, or at least mess with much more code than printf()
  - Data is hard to get at – still need to stop system at some point or transmit memory
    - At least you have the option of doing this outside a critical time window
  - Limited data storage, especially on a microcontroller (maybe a couple K)
  - May have to alter your memory map for a small device to allocate memory

The Real Issue

Fundamentally EVERYTHING you do to the execution environment will somehow alter the program execution environment!

- Normally not a big deal for general application development as resources are plenty and processors are fast! Also works for some soft real time issues.
- For hard real time applications you are typically near the limits of the processor to start with
- For embedded applications, resources are much more scarce
- Race conditions are extremely sensitive to timing changes
- Examples:
  - Program timing altered as program instructions and flow also altered
  - Resources that are not normally used being called into service
  - Program alignment in memory can be altered (PIC processors use paged memory)

I call this the: Heisenberg Uncertainty Principle for Software

Getting around HUPfS

What needs to be done is to add resources that ARE NOT part of the system we are debugging – at least as we intend to use it in production!

Yes the added resources will each have their own impact on the system – current draw, line inductance, line capacitance, etc. These impacts will be small as compared to methods we have talked about so far.

Furthermore if you are seeing problems introduced by these debugging methods it is an indication of something that may be marginal in the design anyway and should at least be investigated if not fixed outright.

I.C.E. and Trace Buffers

I.C.E. stands for in-circuit emulator. This is a piece of hardware and software that is typically inserted between the processor and the rest of the board to accomplish debugging.

Consists of a “pod” that holds or replaces the processor and a “chassis” that contains specialized debugging hardware, including a trace buffer.

Trace buffer is a big chunk of memory that records some or all of the processor signals in real time. These can then be analyzed to determine program flow and variable values.

More on the I.C.E.

Advantages:

- TRUE real time debugging! (At least for microcontrollers)
- In *most* cases has NO impact on executing program
- Can read out data in real time while program is running

Disadvantages:

- Can be VERY expensive – start at about $1K and go up to over $20K!
- Sometimes require a special "bondout chip" to access internal processor signals
- Generally not usable for high speed processors as line inductance off the main board alters signal timing significantly
- Often times making the physical connection is problematic or impossible

Generally not used all that often today, but still available.

On Chip Debugging

On Chip Debugging was developed to address the issues with ICE, in particular connectivity, bondout chips and cost.

Places many of the functions of the ICE onto the die of the processor and connects to these resources with a special serial connection.

Advantages:

- Typically has no impact on running program
- Comes integrated with the chip and the tool chain
- Solves about 80% of the debugging issues
- More complex implementations can also have a trace buffer and a serial wire debugging output

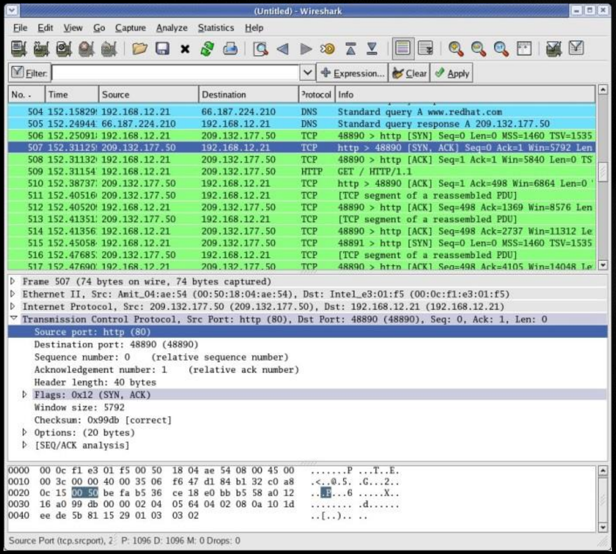
Disadvantages:

- Most have little support for real time debugging issues
- Trace buffer, if present, is small
- Serial wire debugging, if present, is typically limited to 1Mbps
- Does raise the cost of the chip in production

Protocol Analyzers / Sniffers

A protocol sniffer is a piece of hardware that sits between two communicating devices, or on a common bus, with the devices being debugged and captures the communication data.

- WireShark (free!!) is likely the best known one of these, allowing you to monitor and record network communications in real time! (Hardware is an additional PC and switch with port mirroring)
- Other products available for USB, RS-232, wireless, SPI, I2C, I2S, etc.



```
No. -  Time       Source          Destination     Protocol Info
  504 152.15829! 192.168.12.21   66.187.224.210  DNS      Standard query A www.redhat.com
  505 152.24944  66.187.224.210  192.168.12.21   DNS      Standard query response A 209.132.177.50
  506 152.25091  192.168.12.21   209.132.177.50  TCP      48890 > http [SYN] Seq=0 Len=0 MSS=1460 TSV=1535
  507 152.31125! 209.132.177.50  192.168.12.21   TCP      http > 48890 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len
  508 152.31132  192.168.12.21   209.132.177.50  TCP      48890 > http [ACK] Seq=1 Ack=1 Win=5840 Len=0 TS
  509 152.31154  192.168.12.21   209.132.177.50  HTTP     GET / HTTP/1.1
  510 152.38737  209.132.177.50  192.168.12.21   TCP      http > 48890 [ACK] Seq=1 Ack=498 Win=6864 Len=0
  511 152.40516  209.132.177.50  192.168.12.21   TCP      [TCP segment of a reassembled PDU]
  512 152.40520  192.168.12.21   209.132.177.50  TCP      48890 > http [ACK] Seq=498 Ack=1369 Win=8576 Len
  513 152.41351  209.132.177.50  192.168.12.21   TCP      [TCP segment of a reassembled PDU]
  514 152.41356  192.168.12.21   209.132.177.50  TCP      48890 > http [ACK] Seq=498 Ack=2737 Win=11312 Le
  515 152.45058  192.168.12.21   209.132.177.50  TCP      48891 > http [SYN] Seq=0 Len=0 MSS=1460 TSV=1535
  516 152.47685  209.132.177.50  192.168.12.21   TCP      [TCP segment of a reassembled PDU]
  517 152.47690  192.168.12.21   209.132.177.50  TCP      48890 > http [ACK] Seq=498 Ack=4105 Win=14048 Le

▷ Frame 507 (74 bytes on wire, 74 bytes captured)
▷ Ethernet II, Src: Amit_04:ae:54 (00:50:18:04:ae:54), Dst: Intel_e3:01:f5 (00:0c:f1:e3:01:f5)
▷ Internet Protocol, Src: 209.132.177.50 (209.132.177.50), Dst: 192.168.12.21 (192.168.12.21)
▽ Transmission Control Protocol, Src Port: http (80), Dst Port: 48890 (48890), Seq: 0, Ack: 1, Len: 0
    Source port: http (80)
    Destination port: 48890 (48890)
    Sequence number: 0    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
    Header length: 40 bytes
  ▷ Flags: 0x12 (SYN, ACK)
    Window size: 5792
    Checksum: 0x99db [correct]
  ▷ Options: (20 bytes)
  ▷ [SEQ/ACK analysis]

0000  00 0c f1 e3 01 f5 00 50  18 04 ae 54 08 00 45 00   .......P ...T..E.
0010  00 3c 00 00 40 00 35 06  f6 47 d1 84 b1 32 c0 a8   .<..@.5. .G...2..
0020  0c 15 00 50 be fa b5 36  ce 18 e0 bb b5 58 a0 12   ...P...6 .....X..
0030  16 a0 99 db 00 00 02 04  05 64 04 02 08 0a 10 1d   ........ .d......
0040  ee de 5b 81 15 29 01 03  03 02                     ..[..).. ..

Source Port (tcp.srcport), 2   P: 1096 D: 1096 M: 0 Drops: 0
```

Protocol Analyzers / Sniffers

Advantages:

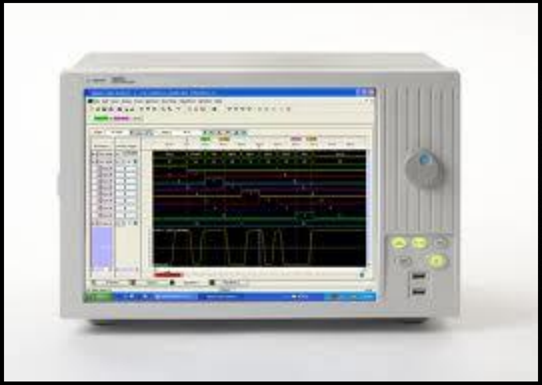
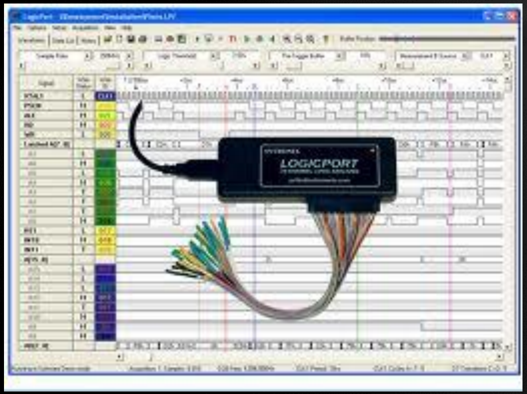
- Free to low cost (<$1000) for many applications
- Near ideal solution if you have a communication problem
  - Can capture in real, or near real time with NO impact on running programs

Disadvantages:

- Can sometimes be temperamental to use, each one seems to have it's own quirks
- Does impact the electrical signals – if these are marginal to start with it could create additional issues
  - In this case you need to solve these issues anyway, and it is better to detect them now before the product goes into production!
- Generally dedicated to one particular protocol – may need a collection of these depending on what you are doing

Logic Analyzers and Oscilloscopes

- Logic analyzers record groups of digital signals at a fixed rate, and then make these signals available for viewing (8-64 channels typical)
- Oscilloscopes plot voltage vs. time curves instead of just binary values, typically on fewer channels (2-4 channels typical)
- Can also purchase "mixed signal" models that have both in one unit
- Two "flavors" are typically available – standalone or USB
- Newer models also have multiple protocol analyzers built in, and with some you can write your own.

Logic Analyzers and Oscilloscopes

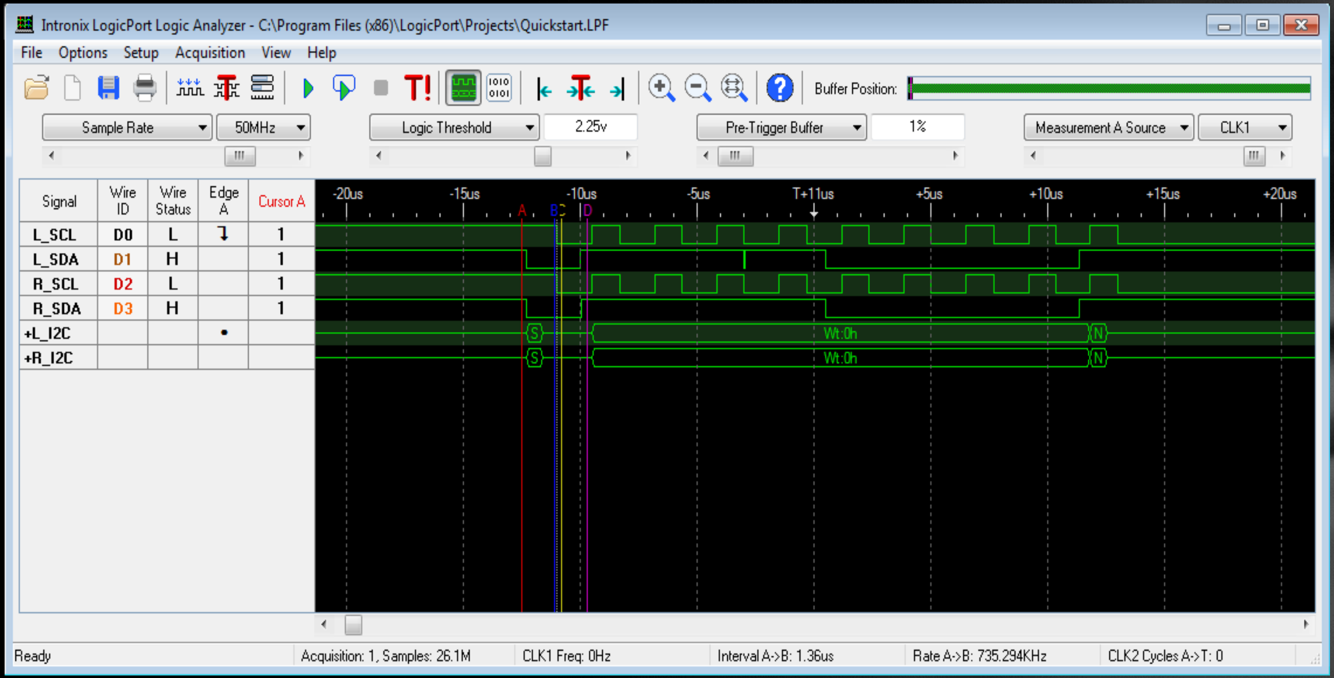
Advantages:

- Usually pretty cost effective, especially for USB models ($200 - $2500)
- Zero to low impact on running program
  - May need to insert code to output debug values or states, toggle a bit to watch
- Very flexible tool for debugging real time and hardware issues, particularly if there are protocol analyzers available
- Most have very complex triggering modes to help manage the deluge of data you can get with them

Disadvantages:

- Can be a real pain to hook up!
- Can be complicated to use, and trigger properly
- High level information can be hard to interpret, particularly on a oscilloscope

Logic Analyzer w. I2C Protocol



Intronix LogicPort Logic Analyzer - C:\Program Files (x86)\LogicPort\Projects\Quickstart.LPF

Sample Rate 50MHz | Logic Threshold 2.25v | Pre-Trigger Buffer 1% | Measurement A Source CLK1

Signal	Wire ID	Wire Status	Edge A	Cursor A
L_SCL	D0	L	↧	1
L_SDA	D1	H		1
R_SCL	D2	L		1
R_SDA	D3	H		1
+L_I2C			•	
+R_I2C				

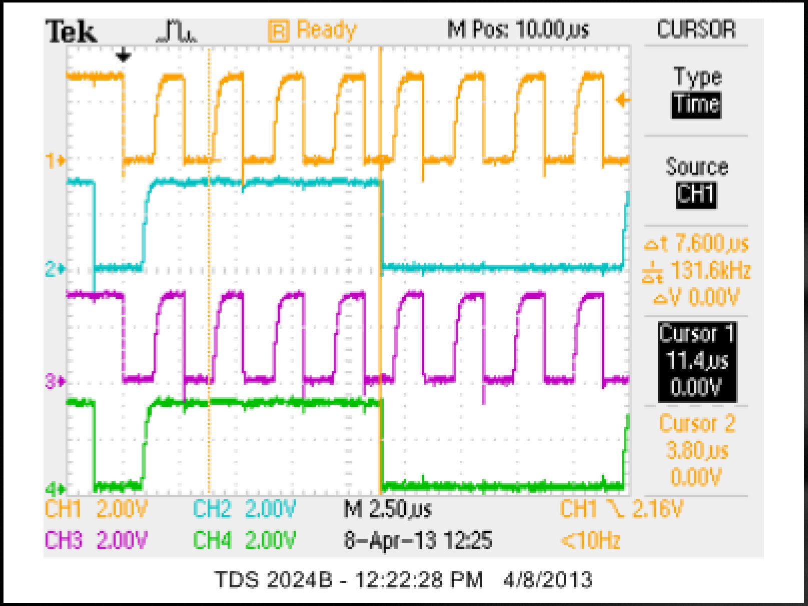
Ready | Acquisition: 1, Samples: 26.1M | CLK1 Freq: 0Hz | Interval A->B: 1.36us | Rate A->B: 735.294KHz | CLK2 Cycles A->T: 0

Sample screen capture of a USB based logic analyzer showing two I2C captures and the resulting byte values from the I2C interpreter.

Notice that the "N" at the end of the interpreted traces – indicates a NAK from the device, so something is clearly not correct here!

The Wt: indicates a byte being written which should be 60 (decimal) – clearly getting interpreted as zero and NAK'd by the I2C slave as well.

I2C Capture on an Oscilloscope



This is a capture of the same four waveforms on an oscilloscope.

Notice that the leading edge of each trace is curved instead of squared off. While the logic analyzer only captures digital levels, the oscilloscope captures analog levels allowing you to debug finer details of the communication.

Techniques

1. Communications Issues
   a) Simply use one of the protocol monitors to capture the communications directly
   b) Timing information is also typically obtained here
   c) If you allow for "debug frames/packets" then you can send additional information
2. Use simple "bit twiddling" to output state information and watch on scope or LA – use this much like you would a printf() statement
   a) Very common tactic is to toggle a I/O pin in software and record on scope
   b) Make sure this code is in the form of a macro or in-lined, do NOT use a function call here!
   c) Can also toggle two pins at different points to measure order or timing
   d) Can output a stream of state information to a port if available (8 bit port is great for this)
   e) Program flow can be traced by putting out numbers to a port and then capturing on a logic analyzer
3. By knowing where your code is NOT sensitive to interruption you can often place debugging actions (breakpoints, debug code) in those areas

More Techniques

- Most real time issues on a PC or "large" hardware involve communications with real world or another computer
  - Tap into and monitor this communication link
- Check to see if the processor you are using has a debugging module built in with a trace buffer.
  - May need to use a special debugging package to access this
- Place debugging instrumentation AFTER the "critical window"
  - May be able to stop program and then look at historical data such as data buffers or other program state to determine what happened during the "critical window"
  - Usually will have to restart the debugging session after one attempt but can move you forward
- May be able to instrument or measure a "surrogate" to help determine timing
  - Power consumption is common here as peripherals spin up and down
  - RF emissions may change as radios go on and off (AM radio was used on the olden days – won't likely help today!)

Intermittent Problems and Statistics

- Intermittent problems require special attention to ensure changes that you are seeing are not do simply to chance
- Need to know your failure rate and track this as you make changes
- Consider a case where you see a failure on 3 out of 10 trials:
  - Failure rate is 30% (3/10 or 0.30)
  - Success rate is 70% (1.00 – 0.30)
  - Now you make a fix, test again for 10 trials and see zero failures! Did this happen by chance?????
  - Compute the chance (probability) that you got 10 successes by chance:
    - Chance of success on one trial is 0.70 (70%), raise to the power of the number of trials
    - 0.70 ^ 10 = 0.028 or 2.8% chance you didn't really fix the bug!!!
- Can extend this method to measure impact of instrumentation and multiple fixes

Intermittent Problems and Statistics

General process:

1. Determine the specifics needed to repeat the problem
2. Determine the percentage of the time the problem occurs, and possibly a confidence level – yes you have to do math here!
3. Add instrumentation (code and/or hardware) to begin investigating the problem
4. Ensure that you HAVE NOT changed the result from step 2! If you have determine if you are still looking at the same issue and can continue. If there are any doubts find a different way to instrument!
5. Enter a proposed fix for the problem and re-run step 2.
6. Interpret the results from step 5.
   1. If you think you have solved the problem remove the instrumentation and test AGAIN! Then use the math from the previous slide to see if you fixed something by chance.
   2. If you have not solved the problem remove the fix attempt and try again!
   3. You may end up accumulating several fixes that need to be considered together – make these a SEPARATE trial, do NOT just start adding stuff until it works!

Best Practices

1. Use the right tool for the job! None of these solutions are one size fits all!
2. If you have “hard real time” requirements partition the problem at design time to isolate those requirements accordingly. (No OS, no processor at all)
3. Design for test
   a) Leave extra resources available in the develop design for debugging, may not have in production (memory, I/O, connectors, prototype area, extra UART)
   b) Have a modular design that can be instrumented between major components
   c) If possible provide a standard connector for hooking up a protocol analyzer, logic analyzer or scope
4. Purchase instruments that are flexible and integrate with the PC
5. Build the system up in layers and fix problems as soon as they are found to prevent interactions or having to “peel the onion”
6. Use statistical methods to measure impact of instrumentation and fixes
7. Be prepared to give up the “divide and conquer” debugging methodology, some real time problems by their nature require a holistic approach

Summary

1. Real time debugging requires a variety of techniques to address different types of issues
    1. Hard real time issue will require additional hardware to debug properly due to the Heisenberg Uncertainty Principle for Software
2. Design your system to allow for test and debug
3. The most complex issues will require both hardware and software approaches to debug effectively
4. Some problems will require statistics to estimate actual results
5. Every time you modify software be aware of the resources you are using – even if you don’t think it matters! (Ya it usually does!)

Questions?