How to debug embedded systems

Ilias Alexopoulos - December 11, 2012

Embedded systems design is a challenging field. Each project is unique with diverse needs and constraints. This is a tutorial discussing, methods, tips and tricks for helping debugging embedded systems firmware using logic analyzers and digital oscilloscopes. There are many ways to debug embedded systems. Tools that help include simulators, in-circuit emulators, JTAG/BDM debuggers, custom hardware, LEDs and switches, as well as serial or other communication ports. Depending on budget and complexity, designers choose the best tools that fit their needs. Although they are primarily targeted at hardware design, you can use digital or analog oscilloscopes and logic analyzers for firmware development as well. The equivalent firmware tool for these is the trace buffer.

Embedded firmware has many differences in respect to traditional software development. Apart from limited resources in hardware (memory, speed, and tools), embedded systems always have the parameter of time. We often use the term real-time system to describe embedded systems. If response timing fails, then the system is not considered working even if function-wise it is perfect. For example, if you drive some excitation, like motors, it is not easy to stop your program and execute code step-by-step to determine the problem. It may be also an intermittent problem that occurs only when running full speed. One way to overcome these problems is to use a trace buffer [1, page 33.]

A trace buffer records actions in a memory buffer. After the event, you can stop the debugger or gather the data for analysis through a communication port (e.g. UART, SPI, Ethernet). This method has advantages. For instance, since it is a firmware solution, it offers flexibility. In addition, you can access to any internal variable. However, trace buffers need memory, some management code, and they may have difficulties correlating to external hardware events. Although newer processors offer more memory, this is often used by the application.

Another method is the action codes [1, pg 43]. Instead of capturing data in a system’s RAM, you could send the codes to an external register or pins and capture the outputs with the help of a logic analyzer. This overcomes the memory problem, and the required code is very small and fast. In this case, the limitations are the output pins and the instrument’s memory capacity. Fortunately, external event synchronization is performed easily, and you have a very strong triggering capability if you use a relatively modern logic analyzer. This article includes techniques and examples for this approach.

Example Case 1
On a project I was involved in the mid ‘90s, we had a system with two devices: a microcontroller for doing the I/O work of the board, and a microprocessor running the application code. We were a team of two hardware engineers and three software engineers developing the product. The two microprocessors were communicating through an 8-bit latched register port. The master processor (application) polled the slave (I/O processor) for events. The communication was based on interrupts (request) and acknowledgement from the slave processor side. The hardware engineers (including
me), were responsible for the hardware and to provide a hardware abstraction layer for the application. The I/O controller, for which I was writing the firmware, was standalone firmware communicating with the application processor.

The software team occasionally complained that they had missing keyboard events during their testing. As you might expect, we (the smart hardware engineers), never investigated the issue because we believed that probably there was an error on their side. After a few months and repeated complaints, we decided to investigate the issue. We used in-circuit emulators, and we implemented a trace buffer on the application processor. We could not track the problem from the software though. Each of us concluded that the software worked functionally as expected. The suspect was the interprocessor interface, so we arranged to have a logic analyzer.

The specific instrument had 4K bytes of storage memory. We attached the probes quickly and started looking at the signals. Soon we realized that the memory was not enough to investigate this issue. And even worse, there was no other analyzer with much more memory at the time. As the senior engineer was disappointed, I started to work with the trigger mechanism. I was motivated to accomplish the task with what we had. It was the second time I used a logic analyzer. I never imagined that I would use this kind of instrument at all, especially for low-end embedded systems. I was so wrong! I managed to do a multi level triggering of about 10 stages. And then it happened. The instrument triggered, and we saw the problem (albeit the small memory). The I/O controller responded too fast for the application processor. The problem was resolved with a few ‘no operations’ (NOPs).

Lessons learned: Firmware usually has many states (memory) and actions that are correlated. You need to have as much memory as you can to debug it. Also a complex triggering mechanism is essential for more complex problems. **Example Setup**

On Figure 1 you can see an example setup for debugging an embedded system.

**Figure 1: Example setup of a debug testbench for embedded systems.**

Debugger and consoles are the most common approach. LED indicators and switches are another
example of hardware assisted firmware development. Often an oscilloscope is used to resolve a timing issue. Custom debug hardware can be an FPGA or CPLD logic that decodes some signaling and provides either oscilloscope outputs or visible status of the system, e.g. a counter that displays the clock frequency of the bus. Many companies do not have the budget for logic analyzers. However, if one is available, it is an indispensible debug tool. Some engineers use very inexpensive logic analyzers or modules that are, obviously, limited in capabilities as compared with traditional logic analyzers. Another category not illustrated here is the possibility to use FPGA-based logic analyzers. These are used mostly for in-chip FPGA debugging (like Xilinx Chipscope). If your board has an FPGA, you may use this to capture bus activity. Despite the fact that your memory and triggering capabilities will be limited, it is a good alternative.

**First Line of Defense: Digital Oscilloscopes**

Although this article is mostly about using logic analyzers and firmware debugging, I feel the need to discuss the common method of using digital oscilloscopes for debugging too.(It is possible to use analog oscilloscopes as well, but you would need to have set up a loop.) Digital oscilloscopes offer the storage capability needed for investigating aperiodic or periodic events. Usually, you pulse 1-4 pins (depending on how many channels you oscilloscope has) on specific points of your software, as pass points. For example, you can set an output when you enter a function and reset the output on exit. That way you know when your function started, what was the execution duration, and when it finished. You can also correlate timing with another function, such as an interrupt or external event (see Figure 2).

![Figure 2: Oscilloscope showing a communication line and the corresponding ISR](image)

The first trace (upper, blue) is a communication line while the second (bottom, cyan) trace is the deferred function (interrupt service routine; ISR) that processes the event. The cursor measures the time between the two events (finish of communication to start of deferred processing). As you may observe, the pulse is very small compared with the communication timing. This means that your
pulse may not be visible if your processor is fast enough and your event scale is large. Then you
either need to place a few NOPs to enlarge the pulse width, or you may toggle the pin on each
execution.

Toggling the pin while entering a function works well with oscilloscope. You lose the information of
duration, but probably this is not an issue in this case. However this technique may be trickier to use
on logic analyzers or custom hardware because you actually need double-edge triggering to capture
all events. Also, your firmware should have toggling capability of the output ports. This means that
either your hardware will offer a toggle bit function, or the firmware should execute a read-modif-
-write cycle.

**Flagships: Logic Analyzers**

Logic analyzers are complex instruments. There are many parameters that affect your decision on
instrument selection. As the cost of such an instrument is high, you will likely need to merge
specifications for both the hardware and firmware teams. The most obvious specs hardware
designers are interested in are the number of channels available (how many signals you can track
simultaneously), sampling frequency, voltage levels etc. From the firmware side, important factors
include deep memory recording (millions of samples) and trigger levels support. With mixed-signal
oscilloscopes, you get a trade-off between an affordable instrument and features like deep memory
and advanced triggering capabilities. This means you must be smarter to do the job! On the positive
side, you can directly correlate analog and digital signals. A standard logic analyzer though, can pay-
off the difference.

**Modes of Operation**

Logic analyzers have two main modes of operation: timing and state. (Timing mode is subdivided to
transitional storage or full storage.)

**Timing Mode**

Timing mode is the simplest operation to understand. It has an internal clock is running at the
specified settings (normally user selectable from a series fixed of options). At each internal clock
cycle, the instrument samples the signals and stores the data to memory. So, if your memory depth
for the given channels is say 2M samples, and you sample at 200MHz, then your maximum time span
will be 10mS (2M/200MHz). This may not be very useful for firmware development. Of course, you
can use the triggering capabilities to begin capturing at the interested interval and thus overcome
this problem. In a subset of this mode, transitional storage, the analyzer samples data at each own
clock rate like in timing mode. However it does not store the signals unless there is a change. This
means that if signals change every hour you will need 1 sample every hour. You would need 2M
hours to fill your logic analyzer’s memory!

**State Mode**

This mode is not very different apart from the fact that the clock is external. Sampling is performed
on either the rising or falling edge of the external clock (provided by your system). Then the signals
captured are like states of the system. The advantage of this method is that you captured sequence
is glitch free. As the instrument samples only on the clock edge, signal settling should be final and
you will not see spurious transitions or glitches of your signal. Usually the state mode has half the
timing sampling rate; ie. If your logic analyzer has a maximum of 800MHz sampling rate in timing
mode, it may have 400MHz in state mode. If your memory is 2M and you sample with 20MHz clock
(you CPU clock rate for example) you time span will be 100mS. This is ten times longer than with the
timing mode above. **Triggering Model**

In contrast with oscilloscopes, logic analyzers have a much more complex triggering system. This
system allows the user to write a sequence of events that will trigger data capture. For example, you
can count for an event 102 times before a second signal activates for more than say a time interval of 10 ms, and more than 18 seconds have passed before initiating storage and fill your memory. Or, even better, you can conditionally record samples only when your desired trigger event happens. You also have the capability to write `<if-then-else>` statements, or even go back to your triggering state (reset) if a condition is not met. I would need a whole chapter to explain such complex systems, but I can assure you that this is a very powerful method for capturing difficult situations. You will need time to master this, but modern analyzers offer simple patterns that you may copy and then modify, which make your life easier. After you master this system you will probably not need these so often. You may not find such capabilities on mixed-signal oscilloscopes or low-end devices though.

**Display**

More or less everybody knows what kind of display to expect from a logic analyzer. Individual signals or groups of signals by means of bus are displayed against time. This kind of display is also common to hardware engineers on the FPGA/CPLD simulators as well. However, things can be more interesting. You can assign to a group of signals logic names or labels for specific values. For example, you may have a group name called “Tasks,” and mark code 0x06 to “RxCOMM,” 0x07 to “TxCOMM,” 0x10 to “Timer” etc. When the bus takes these values, the display does not show the raw value but the corresponding label unless there is no assignment for this code. This is a great feature as it helps you understand your view, instead of struggling to translate bits and bytes to something meaningful.

An alternate method to display the data is the listing mode. In this mode, you get a list that each row represents a time sample. This is easiest to follow in some instances, especially if you use the symbolic representation stated above. See Figure 3 for example.

![Figure 3: Example combination of timing and listing mode display](image)

Another display mode that is often offered is the histogram, similar to Figure 4. The histogram displays what it says: It displays occurrences or durations of codes. If you have assigned labels, then you get a bar-graph of labels. This is useful if you want to see statistics. For example, you may observe execution times of your tasks or functions.
Some instruments offer the option to display an analog representation of the digital signal. This is useful if you output codes to the logic analyzer that represents values written to a digital to analog converter (DAC) or read from an analog to digital converter (ADC). An example of this style of display is shown in Figure 5. Upper trace is the actual analog signal. Note, this instrument has a digital sampling oscilloscope (DSO) as well as a logic analyzer. The second trace is the digitized word in analog form.

**External Trigger**

In some instances, you could use an external event to trigger the logic analyzer. For example, you may use an oscilloscope attached to an analog or digital signal that you want to trace. You may use
the trigger output of the oscilloscope to trigger the logic analyzer. This offers the option to trigger on signals that logic analyzer does not have access for some reason.

**Debug Procedure Description**

Of course, the use of external instruments does not cancel the use of other debugging methods like debuggers, ICE etc. Engineers are free to combine any methods they have in order to accomplish their tasks. For example, I have successfully used my debugger with a mixed-signal oscilloscope to extend my debug range. In order to begin, you need to have the required hardware, including a logic analyzer, mixed-signal oscilloscope, PC-based logic analyzer, etc.

**Flow Summary**

**Flow Summary**

You will need to find the available I/O or output only pins of your design in order to assign them to the instrument. You may also use pins that are already used. Once I was working on a system with external EPROM for data, we utilized unused pins combinations on an address that the logic analyzer captured. For example, using a WR pulse to an EEPROM without programming voltage enabled not harm the EPROM while the logic analyzer could monitor a specific address and capture the 8-bit data on the data bus. This made sufficient number of pins and I/O available to the task. The more pins you have, the more easily you can output data from your firmware. The truth is, you must be creative. Spare pins are usually scarce or not easily accessible by an instrument. It is a good design practice from the hardware engineers to have a provision of a connector with a few spare pins. This can make things much easier.

After you have decided which pins are you going to capture, you need to make the corresponding debug macros that exercise the pins to the required value. As pins can be less than 8, you will not be able to output byte values directly. You will need to split the values in nibbles. If you plan to output pass-points even 3-4 lines are more than enough. Again specific needs and requirements will instruct the methodologies.

The next step is to put your debug macros inside the inspected code. This is straightforward, but depending how you want to see the output on the logic analyzer, you may need to trim the debug macros or/and the logic analyzer’s capture mode.

The next part is trickier. You must setup your logic analyzer and decide which mode is the preferred to capture the data. Also, this may effect which debug macros you will need. For example, if you have a state mode, you will need to output a pin as a clock. Otherwise, this pin can be used as an extra data pin. After you are comfortable with the data representation and time span, you will need to set your trigger. This is application specific. Next, you are ready to run the system!

**Debug Procedure Details**

**Step 1 of 6: I/O pins**

Determine the number of pins that you need. You should be able to control the output of these pins by the software. Ideally, you should have made a provision to connect your spare pins to a debug connector that can be used for this purpose. It is very convenient to unplug/plug a connector and have your setup ready to debug. If you plan to use existing bus signals to drive the logic analyzer, you should take account any signal integrity problems you may have. Connect the pins to your logic analyzer and mark down the connections. It will help to know which pin goes to which logic analyzer input later. Don’t forget to assign the proper signal level to the instrument.

**Step 2 of 6: Debug macros and functions**

For simplicity I will assume that the system under test is a single-threaded application with interrupts. Debug macros are the code that executes a debug output. As you may notice, there isn’t
one debug macro, but rather a set of them. In a simple version, the example code is written in C and used in PIC microcontrollers. You may easily adapt this code to your architecture. Initially you must define the pins as it is seen in Figure 6.

```
/* Contents and definitions */

/* Pin definitions for I/O */
#define DEBUG0_PIN LATDbits.LATD4
#define DEBUG1_PIN LATCbits.LATC3
#define DEBUG2_PIN LATCbits.LATC4
#define DEBUG3_PIN LATCbits.LATC5
#define DEBUG4_PIN LATDbits.LATD1
#define DEBUG5_PIN LATDbits.LATD2
```

**Figure 6: Code fragment for I/O macro definition**

Here we define macros for each pin. Thus we may change the pins in the future.

Next you define another macro like in Figure 7.

```
/* Debug output codes, here you can replace with your preferred debug routine */
if (CFG_DEBUGLA == 1) 
#define c_DEBUGSTATE(x) f_DebugLA(x)
else
#define c_DEBUGSTATE(x)
#endif
```

**Figure 7: Debug macro definition**

This macro defines a name for your debug function and maps it to the real debug function. Disabling the CFG_DEBUGLA macro (set to 0), eliminates the debug output from your production code. Also if you need to change the debug function to a different one, you can do it easily, at a single point.

Figure 8 shows the simplest function possible to do the task.

**Macro 1:**

```
void f_DebugLA(uint8_t v_dbgword)
{
  if (v_dbgword & 0x01) DEBUG0_PIN = 1;
  else DEBUG0_PIN = 0;
  if (v_dbgword & 0x02) DEBUG1_PIN = 1;
  else DEBUG1_PIN = 0;
  if (v_dbgword & 0x04) DEBUG2_PIN = 1;
  else DEBUG2_PIN = 0;
  if (v_dbgword & 0x08) DEBUG3_PIN = 1;
  else DEBUG3_PIN = 0;
  if (v_dbgword & 0x10) DEBUG4_PIN = 1;
  else DEBUG4_PIN = 0;
  if (v_dbgword & 0x20) DEBUG5_PIN = 1;
  else DEBUG5_PIN = 0;
}
```
This macro accepts a byte but actually uses only 6 bits. Your routine may use even less bits. The unified 1-byte input can make this code interface to be common amongst projects and on each project you may have different physical output (more or less pins).

When you call this function, each pin will take its own state one by one. If your pin-out is a single register write (i.e., a single port) then this code above is redundant. The advantage of the above code is that it is more generic and adoptable on different systems.

The next function in Figure 9 is a little more advanced. It disables the interrupts during code output and restores the interrupt enable to its previous state. This is useful if you need more strict timing on transitions or you plan to output codes from the interrupt service routines. In this case, you don’t want your background output codes to be scrambled by interrupt output codes (if you have placed such debug macros on interrupts).

**Macro 2:**

```c
void f_DebugLAIrdis(uint8_t v_dbgword) {
  uint8_t v_previrh;

  if ((INTCONbits.GIEH|0U) v_previrh = 1;
      else v_previrh = 0;
      
  Interrupts_Disable();

  if (v_dbgword & 0x01) DEBUG0_PIN = 1;
      else DEBUG0_PIN = 0;
  if (v_dbgword & 0x02) DEBUG1_PIN = 1;
      else DEBUG1_PIN = 0;
  if (v_dbgword & 0x04) DEBUG2_PIN = 1;
      else DEBUG2_PIN = 0;
  if (v_dbgword & 0x08) DEBUG3_PIN = 1;
      else DEBUG3_PIN = 0;
  if (v_dbgword & 0x10) DEBUG4_PIN = 1;
      else DEBUG4_PIN = 0;
  if (v_dbgword & 0x20) DEBUG5_PIN = 1;
      else DEBUG5_PIN = 0;

  if (v_previrh == 1) Interrupts_Enable();
}
```

**Figure 9: Debug macro with interrupts disabled (atomic)**

Figure 10 shows the clocked version of the previous code. The difference here is that we use a pin as a clock and after setting the individual pins high or low we drive a pulse to the clock pin. This is useful if we are doing state-mode capture with the logic analyzer.

**Macro 3:**
The above code provides a small pulse. If the length of the pulse for some reason is small and you need to increase it, you may use the toggling clock version, like below.

Macro 4:
This time the clock is alternating on each clock which makes it easier to see it in the timing or transitional timing modes. However, the state mode capture may have difficulties, because you will need to capture signals at both clock edges; something which might be impossible. Note that v_DebugClkState should be a global variable for the debug function.

**Differences in RTOS**

In RTOSes you may have to output codes from different tasks. This is the same as the interrupt version of the code. You need to protect each output from other tasks. Instead of disabling interrupts, you may call an OS specific call to forbid context switch during this interval. Unfortunately, this may freeze your system for a fraction of time. Another option is to pace the debug codes from the kernel itself. This will provide system information (i.e. tasks execute) but not a task specific datum. Another easiest case is to trace a single task. Then you might forbid context switch in order to have more uniform timing display. Of course you should have the pin resource available for the debugged task. A different approach could be to have a debug server who queues debug messages and outputs the codes in a FIFO time sequence, maybe with a discrimination of task/data. This is a more complex scenario of course. Virtual machines can be covered using the same principles as well.

**Step 3 of 6: Select debug points & data in your code**

You can use the debug macros in two ways. You can assign pass-codes to verify when the program flow passes through a specific part of the code, or you may output results from a part of the code. If your debug pins are few, you will need to split the output byte (assuming you want to output bytes) to smaller units that will fit your debug pin capacity. You can, of course, inter-mix both methods. You
will only need to assign codes in a way that you will be able to tell, if what you see is a pass-code or an actual value. See action codes assignment example on Figure 12, where I also output the reset reason of the microcontroller.

![Figure 12: Debug action codes to output](image)

```c
#define c_DBG_WDTWR 0x01 /* Opcode for Watchdog timer */
#define c_DBG_ENDLOOP 0x00 /* Opcode for Main Loop end */
#define c_DBG_IRQLOW 0x05 /* Opcode for irq Low */
#define c_DBG_IRQHRX 0x06 /* Opcode for irq high Rx UART */
#define c_DBG_IRQHTX 0x07 /* Opcode for irq high Tx UART */
#define c_DBG_FRAMEVR 0x0A /* Opcode for Rx Frame Overrun */
#define c_DBG_SRST 0x0C /* Opcode for Reset Values */
#define c_DBG_CMRS 0x0D /* Opcode for denoting Reset condition */
```

![Figure 13: Application of the debug routine for reset determination](image)

In this example in Figure 13, I want to output the two variables v_status1, v_status2. In order to understand when these codes will be output (obviously after reset, but it may not be easy to have access at this pin), I selected code 0x0C (c_DBG_SRST), which is the least probable to be the same as one of the v_status1/2 variables. Furthermore as there are two variables to track, it is highly improbable that all codes will be the same (0x0C). Thus, I would know in the rare occasion that 0x0C code was output, that one of the variables had the same value. The second variable signal change will provide a time scale of the output code duration.

Having a macro for the debug function allows you to select at one point in the code which debug function you will need for the specific tests you want to conduct. Now you are almost ready to fire your system up and begin gathering data.

Step 4: of 6: Setup acquisition mode and select appropriate debug function
This step explains the various macros along with the corresponding logic analyzer settings.
Combination 1: LA mode: Timing/Macro 1 or 2
This is the simplest configuration. You configure your logic analyzer in timing mode and you use
Macro 1 or 2 in your code. Timing mode will fill your available memory very fast, so you should use a “smart” triggering to capture the events around your point of interest. Figure 14 is a simulated example showing how your logic analyzer will display the output. The valid action codes are 0x06, 0x0B, 0x10.

**Figure 14:** Simulation of Logic Analyzer capture of action codes in timing mode

As you may observe, there are codes between each action code transition that are cluttering the display. You have to measure their timing in order to be sure which transitions are valid codes and which are ghost codes coming from bit changes. In Figure 15 you can see the bit transitions in more detail. I am showing the bus at the bit level to make easier to see the bit transitions that causes the trouble.

**Figure 15:** Detail of Figure 14, showing the bit changes between the valid action codes

As you can see, although the action codes can be a vital tool to resolve a problem quickly, the situation may require some experience to identify the proper points.

Combination 2: Logic analyzer mode: Transitional / Macro 1 or 2

In this mode you will have a similar display as above. The only difference will be the memory consumption, which in this case will be much smaller. It may take hours to fill your memory depending on your event rate. If yours LA supports transitional mode there is no major reason to use the timing mode for software debugging.

The timing and transitional modes are useful if you need to correlate software action codes with external hardware events (i.e. Interrupts, ADC or DAC signals). For example, you can trace a hardware interrupt line and see the correlation of an interrupt handling routine with the external physical trigger signal. You may also observe the timing duration of your handler. In addition you may modify Macros 1 or 2 (Figure 8, Figure 9) to provide a validation signal. This signal will be high if the debug data is stabilized.

**Figure 16:** Action code validation through a separate signal

At the expense of a signal, you have an easy way to identify the valid debug codes. Here in Figure 16 the validation signal is named LA_DebugClk (outside of the debug data).

Combination 3: Logic analyzer mode: State / Macro 3 or 4

The state mode provides a cleaner display. It uses an external signal as a clock to capture the data on one of its edge. You need to provide adequate clock pulse width and setup time of your signals for this to work properly. Normally logic analyzers are very fast, so the software clock width will not pose a problem. You may extend the clock width a little using a few NOPs. Alternatively if your instrument supports double-edge (both rising and falling edge) triggering, you may use the toggling
signal as a clock; the clock toggles after data is stabilized.

Figure 17 depicts how the previous examples would show on the state mode.

Notice that the clock signal is always high when you use rising edge capture (the opposite is true for falling edge). This is normal. If you could sample the clock as a signal, it would be always be high in this case. So, you should never see your clock line changing if you have set your logic analyzer in state mode. Now you see clearly that the debug codes are 0x0B, 0x06 and 0x10. Memory is again preserved, as only clocked events are recorded.

If your logic analyzer supports state mode, then it is the best mode to work for your software development. I have used this setup to track running tasks in an embedded system and see if any deadlines were missed. I could monitor both execution latency and handler duration. 

**Interrupts or multitasking problems**

What will happen if there is an interrupt that uses the same macros, the same time as the background task? If the debug routine is unprotected and an interrupt which outputs another debug code is activated, then you may miss the interrupt code or even worst you may have a corrupted background task code. This is why I have interrupt protected functions. These forbid an interrupt to intervene in the output code, until the code is finished. Here the validation signal would be of significant merit in order to identify valid codes. For RTOS cases you will need to add your OS function to forbid task switching during the same intervals. You need to take this action only if you place debug codes on many tasks or interrupts. You do not need protection for simple foreground-background loop (ie. Non-RTOS) architectures.

**Step 5 of 6: Select appropriate trigger event**

As you begin exercising this method and learning your logic analyzer you will use simple triggers. However later on the development you will need more advanced tools to pin-point the problem. For example you may leave the LA active all night to capture a missed deadline or a task execution time which is above the limits. In the past I have used the LA for such problem identification. In a system I was working on we had co-operative multitasking. Every task should execute during a 2ms interval and then stop to allow a task change. I configured a trigger to identify an action code larger than the 2mS interval. This setup ran on a target system for days without a problem. Then one day we did a change and we captured immediately a failure of this condition. First thing is to define as accurately as possible the fault condition and what discriminates the faulty events against normal operation. This information is vital to finding the problem fast and without relying on “luck”. If such condition is not unique then you may try to place debug codes in your program flow that will make the fault events unique and thus you can define the appropriate trigger. After you have defined the proper condition you may ask a hardware engineer on how to setup your instrument, or consult your manual/help files.

Make baby steps and if possible verify each condition you add (by running a trigger). This will help you understand better the trigger logic. When you finish your trigger configuration you may be amazed of the complexity. Don’t worry; it is not unusual to have 5-8 trigger stages. For difficult problems you may push your instrument’s limits.
Step 6 of 6: Run system and gather data.
Ok, everything is setup and you only need to fire the test. You may need a few runs to see that triggering is working well, signal display is fine, LA mode is properly setup etc. Don’t forget to assign debug labels to bus signals in order to have human readable plots (see Figure 18). The labels may assign a particular value or range of values to an ASCII label. Instead of seeing hexadecimal numbers you see the corresponding label. If a number is not matched against the label setup, you will see it as a number. Thus unassigned codes are still visible. In the below example the address bus is the task or action code, while the data bus provide a debug value that is specific to each particular event. This way you cannot confuse the output, as each data output is on the context of the corresponding action code.

![Figure 18: Logic Analyzer output after setup](image)

Also the histogram in Figure 19 provides statistical insight:

![Figure 19: Histogram display of the Logic Analyzer](image)

Sometimes the listing window provides an easier data access for debugging, more familiar to the programmers like the screenshot of Figure 20.
After that, I can only say two words: Good Luck! **Example Case 2**

**Example Case 2**

In this test case I will describe debugging with the Tektronix MSO2024. The MSO is a mixed signal oscilloscope. This instrument has 4 analog channels and 16-digital lines. We had also the RS-232 decode modules, which could use either analog or digital lines. MSOs are cost reduced instruments as compared to a logic analyzer. They did not offer transitional or state modes of operation. Also triggering is basically similar to what oscilloscope instruments have. The connection setup was as per Figure 21.
There was an intermittent error of an invalid response occurring once in a while. In order to attack the problem, I setup the above system. I asked our software engineer to modify a version of his software and stop sending further data on the communication link when the software encountered the bad response. Then the last captured RS-232 data would be stable at my oscilloscope’s screen. This was important as the MSO did not have a huge memory or advanced triggering capabilities. We exercised the system while running the debugger, but the behavior was different. I decided to program the microcontroller with the code and see the behavior in this case. The behavior changed! So there was a difference in running from the debugger or from the chip itself. From some clues I decided that an internal reset may cause the behavior. I placed the example code for reset reason capturing of Figure 13 in the beginning and ran the code. The MSO captured the data of Figure 22.
Reset Diagnosis

Figure 22: MSO in action; Reset codes capture

Aaah! The codes captured indicated that a watchdog reset happened. Watchdog timeout is disabled in ISP debugging (for obvious reasons: you cannot place breakpoints or do step-by-step actions while the watchdog timer is running) so I could not find this problem with the conventional debugger-only methods. But what was causing the watchdog timeout? We had very relaxed timings to have such a condition. I tried to trigger before the reset. The RS-232 activity was evident when the problem occurred. Also debug code activity is stalled after the last Rx event as seen in Figure 23. I have omitted here a series of “program-test-replace debug codes” cycles to approach the problematic code.
Figure 23: Last event before crash capture
Looking in more detail just in the receive event, it was evident that there was something wrong in the Rx IRQ (interrupt service routine). Figure 24 revealed what was the last activity before the crash.

Just Before Stuck
Then the next time moment after the Rx IRQ shown, there was a frame overrun error detected by the IRQ. However in this case the microcontroller was looping inside the interrupt, clearly visible at Figure 25.

![Rx IRQ Error Code](image)

**Figure 25: Deeper understanding of IRQ failure. Debug macros have been moved to suspect code.**

After code inspection, it was revealed that the problem was a variable that was not declared as volatile. The compiler optimized the register read that cleared the interrupt pending flag. Also, the difference between the debugging environment and the runtime made this error harder to catch. Notice also that I co-operated with the software guy (software of another layer) in order to help me pin-point the error time. This part was crucial to resolve this problem. You notice that there are signal transitions on the parallel bus (my debug codes) between valid codes. I discriminated these by the relative time duration between codes and transitions. Normally, debug codes stay stable much longer than the transitions. I was able to resolve this issue in a couple of hours. (This was a problem that was detected by the upper layers, but nobody could found the root cause for a long time, using the “usual” methods.) Even when I revealed the WDT time-out, I had a hard time convincing people that this was actually happening.

**Example Case 3**

You think that these tools need a real target in order to be useful? Think again! I have used these techniques successfully when I was involved in a SoC design. I was doing the embedded controller hardware and firmware that controlled our baseband. As the hardware was not ready yet (I was responsible for this as well to some extent), I used the simulator to design our firmware. We were using an ARM946 core and the ADS1.2 development suite from ARM. ARM had a very capable instruction set simulator that was cycle accurate. But one of the most important features was that
you could create C models of your custom hardware and simulate the behavior. I made all the custom hardware I needed (I would make the actual VHDL models later). As I learned the procedure to add our own custom virtual hardware, I knew that we would need a debug connector for this kind of tests.

So, I added a virtual logic analyzer debug connector with clock and data. In the real target we would have the same connector and hardware, so we could compare executions and other data between physical system and the simulator. Our logic analyzer was a 1690 series from Agilent. This instrument exported its capture in XML format. So, I made my virtual debug port to write to file time and data information. Every register write of the debug register concluded in a file write recording time of event and actual data. I then created a script that got this file as an input and converted to XML. Then, using an additional Tk tool (Tk is a graphic interface used by TCL or other languages), I read the XML file and have a visual of what was happening inside. In addition, another virtual component recorded transmitted and received frames, so the debug tool could aggregate the data and also see external events (frames) in respect to internal execution (tasks) of the system. An example of the output is seen below. The good thing was that I could use the exact same tools with the physical system as well.
The original tool served its purpose. It was a little buggy, but for my first tool implementation it did its job. I since re-wrote this tool in Python, but I did not manage to have good performance. In order to get rid of the majority of problems, I decided to re-write the tool once more in Python. This time, the performance is much better and the small bugs are not very annoying. The difference between this tool and a classic viewer is that you can display and correlate two different XML files (i.e. tasks, events). Also double-clicking an event or task allows you to see their statistics or additional information. Here is a screenshot of the latest version 3.01. Of course this tool is open source and you can use it freely.

![Figure 27: Screen Shot of the Latest TaskAnalyzer V3.01](image)

The tool along with its manual can be found at:
[
]

**References**


**About The Author**

Ilias Alexopoulos has more than 15 years of experience in embedded systems, DSP, and FPGAs. Commitment to engineering excellence and enjoying the engineering journey is always his aim. Ilias is also a contributor to the open source community with articles, code, etc. You can find more information at www.ilialex.gr