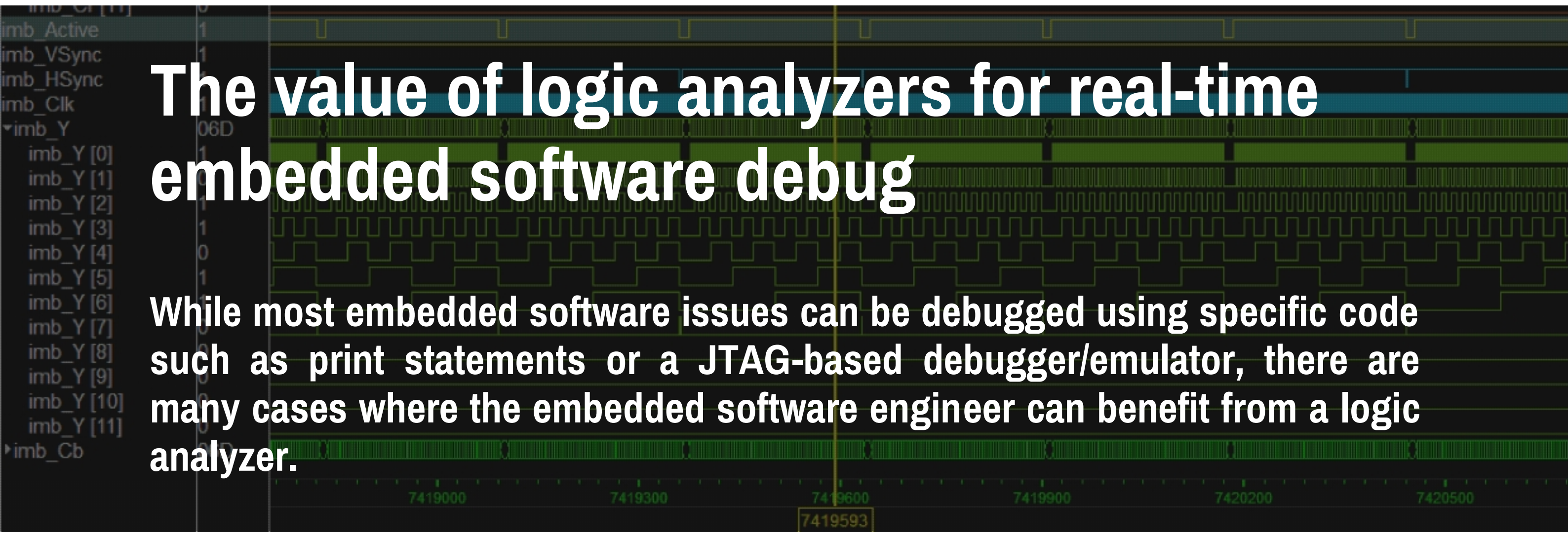


The value of logic analyzers for real-time embedded software debug

While most embedded software issues can be debugged using specific code such as print statements or a JTAG-based debugger/emulator, there are many cases where the embedded software engineer can benefit from a logic analyzer.

Typical cases are:

- **Problems related to the interrupt handler**, for interrupts happening several thousand times per second. Here, adding debug code - typically 'printf' - and/or breakpoints will severely affect the real-time execution of software.

- **Problems related to synchronization in real-time systems.**
  Race conditions, priority inversion,... Problems affecting the order and priorities of the access to embedded system shared resources - all of this cannot be troubleshot easily with software-based solutions only.

Logic Analyzers can help

The logic analyzer is traditionally used by hardware engineers to debug digital circuit. By providing enhanced visibility of an embedded system at the signal level, it may also be the software engineer's best companion.

Usually, software engineers feel that 'logic analyzers are not for them'. The logic analyzer is seen as something for 'hardware guys', something expensive, usually not available in the lab and difficult to setup and use. In this paper, we note that:

- Software engineers can benefit from a LA, especially if it features sufficient memory and advanced features like data filtering.
- Acquisition price can be reasonable - especially with PC-based logic analyzers.
- Availability is not an issue if everyone is equipped with a low-price PC logic analyzer.
- There is a learning curve to understand how to select and use a logic analyzer, but this is offset by the accompanied benefits.

Keep in mind that any debugging tool aims at providing clues for the source of a problem. An engineer should be considered as an investigator that uses an array of evidences to correct a problem. Understanding the features of a logic analyzer and being able to define which ones will really help is the first job of the debug engineer.

Logic analyzers are like cameras

You can think of logic analyzers as a 'specialized camera for embedded systems': they record the movie and history of events that happen in embedded systems. The main characteristics of a logic analyzer are: **the**

maximum sampling frequency, the memory size and the number of channels. They can be compared to specific characteristics of a camera, as summarized the table below.

Logic analyzer	**Motion camera analogy**
Sampling frequency	Movie frame rate
Memory	Maximum movie length
Number of channels	Screen size

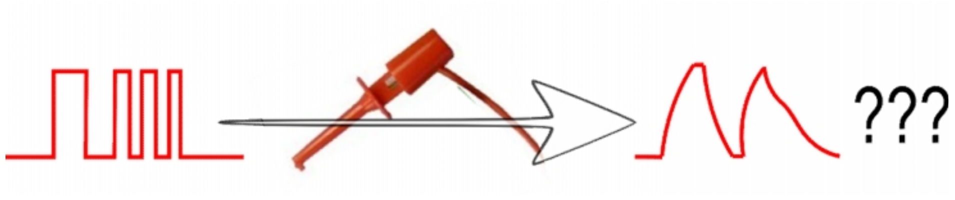
Selecting a logic analyzer can be challenging

Unfortunately, it can be very challenging to select a logic analyzer based on the above characteristics only. Sampling frequency, memory size, and number of channels are essentially hardware features. Moreover, many logic analyzers claim misleading hardware specifications which prove to be of no use to the software engineer.

A few examples:
- Many low-cost logic analyzers (PC-based) claim a maximum 'sampling frequency' of 500 MHz to 1 GHz. Such a frequency can be desirable, especially if the system that has to be observed features high-speed busses. Shannon's theorem states that you need at least 2 samples to detect a transition or that your sampling frequency must be at least equal to twice the highest signal frequency that you would like to observe. In other words, you will not be able to correctly observe a 250 MHz bus unless you sample it at 500 MHz or more.

While it can be true that the internal circuitry of a logic analyzer is able to sample data at such a high frequency, the probes provided to sample the signals are often simple plastic clips and wires which will filter the high frequency content of your signal. So, you will end up with a practical sampling frequency as low as 20 MHz (and even below) just because the 'bandwidth' of the clips and wires is too low (see Figure 1 below).

Figure 1 : Because of limited probe / clips bandwidth, the signal seen at the input of the logic analyzer (on the right side of the picture) may have lost its digital information, despite claimed high sampling frequencies.



- Similarly, claimed specifications about 'memory' must be considered carefully.
PC logic analyzers are composed of a piece of hardware connected to a PC through USB connection, for instance. Such logic analyzers often claim virtually infinite memory and storage because the collected data is stored into the PC memory and hard disk. The reality may be different. All logic analyzers must contain 'some' hardware memory buffer where data is stored temporarily. A very common technique consists of flowing the data stream to the PC. The connection with the PC inserts a bottleneck in the data path. For instance, a USB connection controlled with non-real-time PC OS usually does not exceed a total sustainable bandwidth of 30 MByte per second. If memory resources of your logic analyzer are too limited, you will quickly create an overflow and you will not actually be able to benefit from the huge storage capacity of your PC (see figure 2).

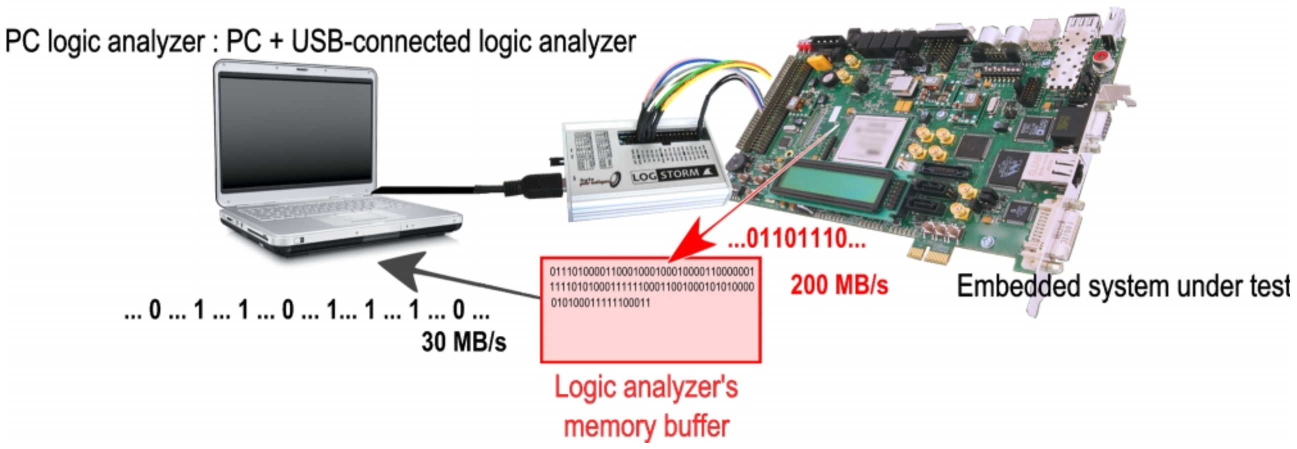


Figure 2 : Example: the logic analyzer samples 16 bits at 100 MHz. The logic analyzer's memory is filled in at the rate of 200 MB per second. On the USB link, the data buffered in the logic analyzer transits at 30 MB per second. Even if PC memory resources are virtually infinite, it is the memory size in the logic analyzer that matters.

Keep the target in mind

There are many cases where a logic analyzer helps troubleshoot real-time software. We shall limit ourselves to one of them in this paper: tracking a race condition and/or a violation of the priority of events in an embedded system.
To understand why such a condition occurs, the software engineer will have to:

- avoid breakpoints;
- avoid inserting too much debug code.

In short, try to avoid anything that would break the 'real-time' execution of 'real' software application.

Typically, debugging consists of observing one or multiple microcontroller busses and probing interrupts in order to understand the history of events that has led to a software bug. The source of the bug is supposed to be 'functional': the system is electrically ok and the bug originates from the execution of an illegal sequence of software events.

Once the movie of events is recorded, the software engineer can examine it, correlate it with its software execution and try to spot where something went wrong.

In short, there is a need for a tool that provides the maximum visibility over the embedded system at logic level.

What would be the essential qualities of a logic analyzer to do this?

- **The sampling frequency** essentially depends on the microcontroller used on the board and the frequency used in the system. Many microcontrollers today run at a few megahertz up to 50 MHz. For all of these, it will not be hard to find a logic analyzer - even a PC-based logic analyzer - with a maximum sampling frequency around 100 MHz.
  If your bus system runs at higher speed, then you should look for logic analyzers in the 200 MHz to GHz sampling frequency range, which can prove more expensive. And don't forget to check the probe bandwidth too!

- **The number of channels** depends on how many digital signals you need to get a sufficient understanding of the history of events. This would be essentially a budgetary decision: in the end, in a 'brute force' approach, you would like to collect as much information as you can. However, the acquisition price of the logic analyzer will quickly rise if you want more than 16 or 32 channels. Please note that larger busses also have an impact on the total memory needed in a logic analyzer.

- **Finally, memory depth is 'THE' highly desirable element for a bug hunt.**
  Memory depth represents how many samples can be stored in the logic analyzer before its memory fills up. For example, if you use a sampling frequency of 1 million samples per second, an analyzer with 32-KSample depth will fill up in just 32 ms, while an analyzer with 8-MSample Depth can run for eight second. The latter is extremely useful because it allows you to see the succession of events that happen long after a triggering condition (that is, the conditions that you use for starting the logic analyzer). If you sample 'longer', you'll have a better chance to capture the clue that helps you to spot and correct a failure.

Now, what if you need to understand the course of events that caused a failure one hour or one day after you have started your software?

With the above example, 1 hour would represent a depth of... 3.6 G samples !
24 hour = 86.4 G sample. If say, your bus is 16 bit you can calculate the memory needs to 7.2 GByte per hour of run. A simple look at the logic analyzers available today will show you that this exceeds most usual specs.

To overcome this, all logic analyzers have more or less sophisticated 'triggers' and the ability to 'rearm' the triggers.

Simply put, a 'trigger' is a logic condition built on the sampled data that starts the logic analyzer. For real-time embedded software debug, you'll typically use a specific data on the bus (e.g. a given printf value) or a system interrupt to tell the logic analyzer that to start recording.

Once the logic analyzer memory is full, 'rearming the trigger' lets you automatically run an new capture upon occurrence of the same trigger event. Of course, if you want to keep all the recorded data, you'll have to empty the logic analyzer's memory before allowing a new trigger.

Typically:

Figure 3 is a timing diagram that represents the digital traffic transmitted on an embedded system's bus. The colors indicate that the traffic originates from various sources. The message in red represents a failure - e.g.: a message that violates a priority rule on this bus.

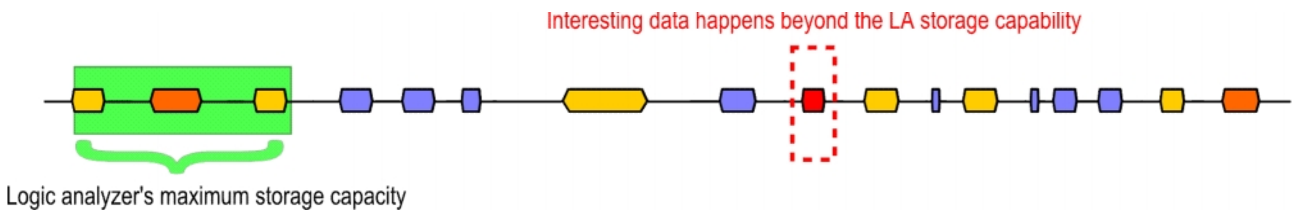

Interesting data happens beyond the LA storage capability
Logic analyzer's maximum storage capacity

Figure 3 : Simple logic analyzer run. The logic analyzer is triggered (started) upon the occurrence of a 'yellow' message on the bus. The green area represents what is 'seen' by the logic analyzer. It is limited by its maximum storage capacity. Once the logic analyzer's memory is full, the logic analyzer stops. Note that a logic analyzer usually records samples even if the bus is idle.

Now, with trigger rearm:

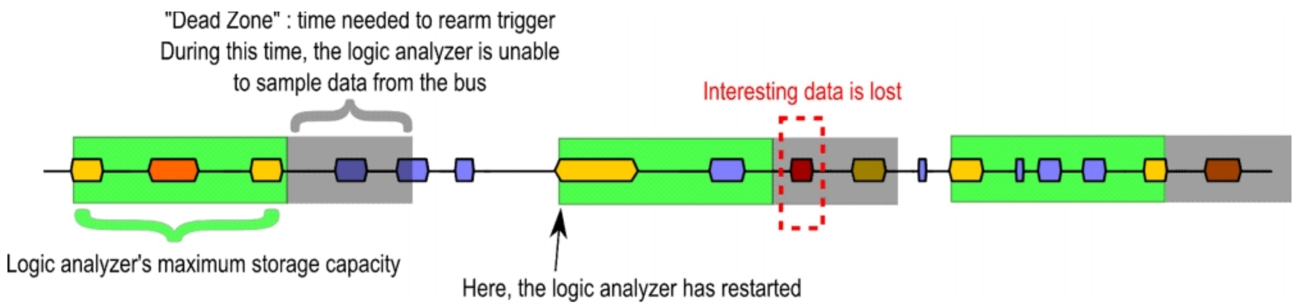

"Dead Zone" : time needed to rearm trigger
During this time, the logic analyzer is unable
to sample data from the bus
Interesting data is lost
Logic analyzer's maximum storage capacity
Here, the logic analyzer has restarted

Figure 4 : Logic analyzer with trigger rearm. Once its memory is full, the same trigger is rearmed. The 'dead zone' represents the time needed to rearm the trigger - that is, to transfer the logic analyzer's memory content to a safe place and/or to rearm its trigger condition. Even in this case, there is a risk in 'missing' the interesting data - if it occurs during the 'dead zone'.

Rearming trigger extends the reach of a logic analyzer over time and increases the total visibility over embedded system.

Do you really care about idle times?

Several strategies come to the rescue to overcome the memory limitation of logic analyzers. It must be noted that a memory limit will *always* be reached, even if you have deep pockets and can afford the largest available memory option for your logic analyzer.

The table below summarizes these strategies.

Strategy	Comment
- Extend the available memory in the logic analyzer.	Good idea. Beyond the budgetary problem that this represents, having a large 'hardware' memory depth lets you collect larger sampling windows. This increases your visibility even if you need to rearm your trigger to extend your reach. Remember that even PC logic analyzers need a large memory buffer because transferring your data to the PC memory or hard disk will take some time, during which the LA memory can overflow.
- Use a complex trigger to carefully select what you need to record.	Good idea, but this requires an a priori knowledge of what you are looking for. This is advisable after a first run, when you are trying to narrow down your search. When you do not have a clue, you'll prefer a 'brute force approach' and not limit your search.
- Use data compression	This can help save on memory, but data compression is unreliable because it is data-content dependent. Data compression does not always compensate for insufficient memory specs. Remember that software debug often requires long runs.
- Use a logic analyzer with data logging behavior	This last point is explained in the next paragraph. It consists in recording data only when data is interesting. This also requires some trial-and-error

Strategy	Comment
	strategy and/or some 'a priori knowledge' of what you are looking for. Bus idle times are filtered, it is a kind of 'smart retriggering' or 'data filtering'. It is also called 'data qualification' by some manufacturers.

Logic analyzer with data logging behavior

Data loggers, data monitors and data recorders differ from **logic analyzers** because they are primarily interested in 'data', not 'signals'. Actually, you should not bother too much about **what the tool 'is'**, but rather about **what it is useful for**. The point here is that troubleshooting real-time software is about viewing the data that is generated and processed by the embedded system. As we have seen, collecting information always involves using some memory to store it. At some point, you'll want to use your memory resources wisely. Here is another strategy to do this: qualifying **data (filtering data) upfront in hardware.** It is explained with the figure below.

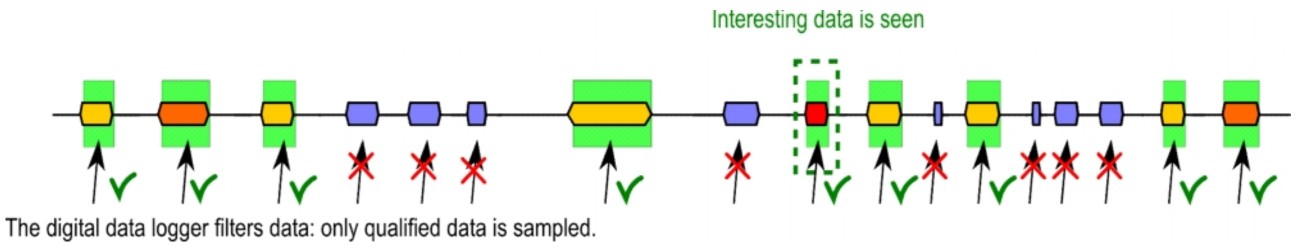

Interesting data is seen
The digital data logger filters data: only qualified data is sampled.

Figure 5 : Logic analyzer with data logger behavior. Filtering capabilities allow selecting the bus traffic that needs to be recorded. In the example above, the 'blue traffic' needs not to be viewed for debug. Bus Idle times are also filtered. As a consequence, the logic analyzer's memory is used for a limited data quantity, allowing to spread the recording over a longer period of time.

Conclusions

In this article, we have explained why logic analyzers should be used for real-time embedded software debug. Typical scenarios that can benefit from the use of logic analyzers include: interrupt handlers-related problems or problems related to the synchronization of access to shared resources. In general, logic analyzers help when the traditional use of debug code such as 'printf' affects the real-time behavior of software execution. Like investigators, the software engineer can use logic analyzer to provide an array of evidences that will ultimately lead him/her to debug software.

We have also shown that it can be difficult to choose the right logic analyzer. Raw specifications inherited from the 'hardware world' can be hard to understand and can sometimes be misleading.

Having sufficient memory in the logic analyzer together with strategies to save memory is definitely advisable. These strategies include: repeating / rearming trigger conditions and data qualification (data filtering). Such features expand the 'observability' of a system over time. As a consequence, the ideal companion device for software troubleshooting is a logic analyzer that features a large embedded memory and data logging (data filtering / data qualification) capabilities.