Thoughts on Data Collection and Analysis

by Brian Johnson, 2018

History

For decades devices have been designed and sold for the purpose of collecting data of various kinds and making the data available to the user with varying degrees of interpretation. Such devices have been called data loggers, oscilloscopes, logic analyzers, signal generators, spectrum analyzers, and various other devices.

All of these devices have as their purpose the collection of some kind of data, recording that data in some form, and making the data savable, visible and/or analysable. The details of all of these capabilities have varied greatly, and the scope of such devices has varied between single purpose instruments to much more versatile multi-use instruments. In recent years, the most versatile of all implementation platforms, the computer, has increasingly been utilized to implement such devices, bringing with it the possibility of expanding the versatility to new heights.

Current Practice

The current trend includes a large sub-category where a general purpose computer is used with a variety of attached devices that serve as the interface to the data being collected, internal or external disk drives for saving the data, and software to make the data visible and/or to analyze the data.

As in the past, such systems range from single purpose to versatile, and from proprietary to open. In the most proprietary form, one or more front end devices are available from a single source, and coupled with software from the same source. The devices are not intended to be used with any other software, and the software does not support any other devices. In some cases, the communications mechanisms between the devices and the software are also proprietary, in hardware, in protocols, or both. There are many examples of this architecture.

Open systems are usually designed to facilitate the connection of a diverse set of front end devices using openly defined hardware and communication protocols, and software that is capable of being configured in a way that matches the capabilities of an individual device. In a fully open system, both the hardware details and the software source code are openly published so that the details are visible to anyone who cares to look for it.

Front End Devices

For a fully open architecture, front end devices are all conceptually the same. Each is a (usually) simple device which collects some kind of data and forwards it to the software in the computer. Examples of such data include the following:

- low level signals, usually as individual conductors where the voltage represents the data, perhaps encoded in analog format or in a digital (usually binary) format
- encoded signals, such as current loop, temperature, pressure, voltage, etc. or (either in analog or digital formats)
- data streams, including video, UART, SPI, I2C, and a multitude of other communication protocols

There is much overlap between these categories, including the fact that ultimately, all of them come down to, or can be converted to, a timed sequence of individual sampled voltage values, either in analog or digital form. This conversion is essential to being able to convey the data to the computer over either an electrical or optical connection, and to be meaningful, this conveyance must include both the voltage samples and the time relationships. While the voltage samples generally must be explicit, the time relationships might be either explicit or implied.

The primary purpose of the front end device is to do the encoding of input data into records that can be forwarded to the software using the physical interface provided. Depending on the original nature of the data, the records might represent single data points or clusters of data points. Depending on its capabilities, it might forward data records either constantly (periodically, or whenever data is available) or conditionally (depending on whether some constraint is satisfied). The latter case is sometimes specified in the form of a trigger criterion. It might occur just once, or multiple times. A device that only reports out-of-spec data is an example of the latter case.

Examples of Front End Devices

Logic Analyzer

The logic analyzer is one of the most ubiquitous devices in the modern age. They are usually intended to connect to a set of low level digital signals, periodically sample them, and forward them to the computer. In most cases, the sample rate is implemented internally, but the rate used can be selected by the software from a set of rates supported by the hardware. These devices vary greatly in detail in the following ways:

- time sample rate range, lowest to highest
- number of signals monitored
- input voltage threshold, sometimes selectable

- buffered or streaming
- support for internal triggering, ranging from none to simple to complex triggering
- usage modes implied by any internal state machines
- command set
- data encoding formats

The simplest such device would collect sampled data at a fixed (possibly selected) rate and transmit it continuously to the computer software. It would be up to the software to decide whether and when to process the data, including the trigger concept. This would be a streaming mode with no internal support for triggering. The minimum useful command set probably includes streaming enable, sample rate select, channel enables, voltage threshold.

The very highest sample rates may require buffering and/or limited sample counts, and therefore a triggering mechanism, when the communications channel to the computer cannot support the data stream data rate directly. These devices also usually have internal state machines that must be understood in order to support management and operation of the device.

Oscilloscope

The oscilloscope is almost identical to the logic analyzer, except that the input voltage(s) are digitized by an ADC and the resulting multi-bit data values are forwarded to the software, rather than individual binary samples. The range of variable details is otherwise almost the same as for the logic analyzer, though the simpler forms are not currently very useful, so most support buffering and internal triggering, and have internal state machines that must be understood in order to support the device. Nearly all oscilloscope front end devices support configuring the pre-ADC gain and sample rate.

Data Logger

Traditional data loggers typically monitor various physical (generally electrically encoded) signals and sometimes serial communications protocols. Those that monitor physical signals are frequently configurable to determine which samples to forward based on internal criteria such as min/max ranges, etc. As such, they are usually not thought of as “triggered” in the same sense as the logic analyzer or oscilloscope, but do have internal state machines that decide which data to forward. The forwarded data is usually encoded into some form of sequence of records, where the records include the data, and usually some form of timestamp, because they frequently are not issued on a fixed time basis.. They are frequently encoded in the form of blocks of serially encoded text strings, but the possibilities are quite varied. Internal state machines are usually simple and don’t need much external support, beyond configuration protocols. Most data loggers have less need of extreme data bandwidths than do signal analysis tools.

Meter

There is a growing availability of connectible multimeters, usually serial port or USB connected. Such devices are relatively slow speed and have varying degrees of usually fairly limited control interfaces. They tend to be very low bandwidth and typically provide constant updates at a low repetition rate. They are logically a subset of the data logger category.

Commentary

Streaming protocols are vastly to be preferred whenever possible because they avoid the requirements for large blocks of memory, complex communication protocols, and complex state machines. This makes the devices cheaper, more reliable, and easier to support with software.

Interconnection

The means of interconnection between front end devices and the computer varies widely. In the past, serial RS-232 ports, GPIB (aka IEEE-488), and IEEE-1394 were in common use. Today, these are rapidly fading away due to better alternatives. None of these is commonly available on modern desktop and laptop computers, though there are ways to accommodate them. Today's most ubiquitous choices include USB, GigE Ethernet, PCIe, and eSATA, with a lesser smattering of other high speed serial channels, like FireWire. Of these, USB is the most popular today, in the form of USB2, and this will surely increase as USB3 and beyond are more widely adopted. Most computers made today support USB3 and GigE Ethernet out of the box.

USB3, with its bandwidth of approximately 5 Gb/s, can provide higher transfer rates than previously conveniently available, permitting the wider use of streaming protocols, and the resulting simplification of the front end devices for an increasingly higher performance category.

I expect that in the future we will see the adoption of fibre links capable of even higher speeds even in consumer-level computers, but that is barely on the horizon today.

USB3 can likely support streaming logic analyzer devices with 32 bits at 100 MHz, 16 bits at 200 MHz, and 8 bits at 400 MHz. It can provide even higher performance with effective data compression (which is usually required to be lossless for this application).

It could also support streaming oscilloscope devices with similar speeds, depending on the ADC precision and number of channels.

Software

Versatile software is essential for the success of this open approach. Today that effort is somewhat fragmentary, but shows much promise. It is a huge undertaking. Since nothing today fully meets my goals, I am deferring the software discussion to the Future Directions section.

Future Directions

This section lays out my personal wish for the future of fully open data collection systems. It is an aggressively versatile vision which necessarily entails a large amount of up-front design effort, and a non-trivial implementation, but which has enormous benefits for all.

Front End Devices

I think front end devices are on track. As more devices are made to take advantage of higher communication speeds, e.g. USB3 and beyond, there will be less need for trickery and complexity. Trickery and complexity both lead to decreased reliability, so this is a positive direction.

In addition, I expect there to be more and more devices that loosely fall into the IoT camp. So network connection capability will become more relevant. Most such devices probably will not fall into the extreme speed category.

Interconnection

Interconnection is on track. USB3 is adequate for most needs, and is here today. There will be improvements and advancements, but it is not holding us back.

In addition, basic network connection capability will become more relevant because some devices will be wireless in some fashion, perhaps BlueTooth, or maybe WiFi. Such devices will likely manifest in being on the local network in some fashion, perhaps via adapters in the case of BlueTooth. These technologies are already adequate, if not completely pervasive

Software

I have a lot to say here. To complete this vision, current software has a long way to go. And it is a huge undertaking because it is central to everything, and is subject to needing to be adaptable to a wide range of devices, including devices that have not been thought of yet.

Here is a discussion of desired characteristics, as I see it:

Support for a wide range of devices

This requirement dictates that the software has a simple but adaptable interface to the devices, via a number of technologies, including USB and network. Currently support tends to be “compiled in”. This needs to change to a “plug-in” technology. This would allow most of the complexity for supporting a given device to be separately developed, and that the requirement to support device internal state machines can be excluded from the overall software by putting it into the individual plugin module as required.

Perhaps this should include some or all of the “driver” code as well. It is nice to envision the software

providing access to the available interconnection technologies, but allowing the individual plugins to actually negotiate their own connections to the devices they support. This has long been the case for traditional networking, which has a highly standardized API, and for which no application software has to actually support the hardware interface directly. It is true to a lesser extent for USB, which still tends to need specialized low level drivers that are at least partially known about by the application. This tends to be more true for higher speed interconnections, because it still is not standardized enough.

Currently, the multiplicity of USB drivers competing to support the particular combination of device and application is a major impediment, because there are no reliable solutions to this issue whenever there is more than one application wanting to access a particular device but via different drivers.

State machine coupling

To support a wide variety of devices, it is essential that the global state machine of the application be as simple as possible, largely limited to attaching the wanted device(s). Those devices that require state machine support should be supported by the associated plug-in. In particular, there should be no global requirement for the need for a “trigger”, although it would be desirable to be able to support that in some cases. Not all devices need triggering. The most basic state machine requirements for any device include some or all of the following:

- attaching/detaching individual devices to the software application
- device configuration
- starting, maintaining, and stopping data capture
- managing or performing triggering

In particular, the last two requirements should not be mixed together globally, although some specific devices may require a single state machine to deal with both issues. It would be best for such a state machine to be implemented in the plug-in.

Capture modes

There are two primary modes of data capture, continuous and gated.

Some devices may only be capable of providing data continuously. Some meters are in this category, and some low end logic analyzers are as well.

Some devices may be incapable of providing data continuously, and can only provide data in length limited bursts, with the bursts being commanded either by the application software, or perhaps by the device itself. This latter case is typical of internally triggered buffered mode devices. Including some logic analyzers and oscilloscopes.

Some devices may support doing data capture either way.

Triggering and gating support

Trigger mechanisms generally serve one or both of two separate purposes. One purpose is when the device can generate data too rapidly to just send all data continuously to the application software, so triggering is used to determine which data is to be captured, and to buffer it until it can be sent to the application software. The other purpose is to reduce the amount of data that needs to be analyzed by capturing data when it is thought to be of interest, according to some criteria.

Some devices need a trigger mechanism, and some don't. In some cases, it may be mandatory for the device to implement a trigger mechanism internally. In others, triggering may be performed in the application software. In yet others, there may be no hard need for triggering at all.

Where needed, triggering can sometimes be implemented internally to the device, or externally in the application software, and in a few rare cases perhaps partially in both.

Triggering is most likely to be needed for high speed logic analyzers and oscilloscopes, mostly because such devices tend to be internally buffered. In these cases, the plug-in must manage setting up the trigger, and any complexities that arise in the data capture action.

If the device generates data at a lower rate than the communication interconnect can handle, then the application program may be able to just turn capture on and off as required, or as requested by the operator. If desired, then either triggering, or gating, or both could be performed in the application software.

Gating typically refers to the application deciding, based on some criteria, which data to forward to analysis, and which data to just toss. It would usually be used with a continuous source such as a multimeter, and the gating criteria might be used to forward only data that is outside of configured limits, such as voltage too high. Like triggering, gating can also be done within the device, or in the application software. Some traditional data loggers did it the internal way, whereas most multimeters (not all) don't support that internally.

Subject to the effective data speed limit, almost all devices could theoretically be used in continuous data production mode, allowing the application software to pick and choose which data to analyze by applying trigger criteria, gating criteria, or whatever to the captured data. This is the most general case. Other modes are used primarily to reduce the performance requirement on the interconnect, the software, or both. Triggering and gating are also used to efficiently focus the attention of the data analyzer or human user on specific portions of the captured data.

In some cases, multiple triggering events are desirable. This is almost the definition of gating,

Almost since inception, oscilloscopes have provided a choice of three primary operating modes: single sweep, normal, and automatic.. In these modes, the first two are defined on the basis of having a trigger criteria, and allowing data capture either exactly once (single sweep), or (almost) every time the trigger criteria is met. In the second case, it is common to use a trigger holdoff timer to limit the rate of

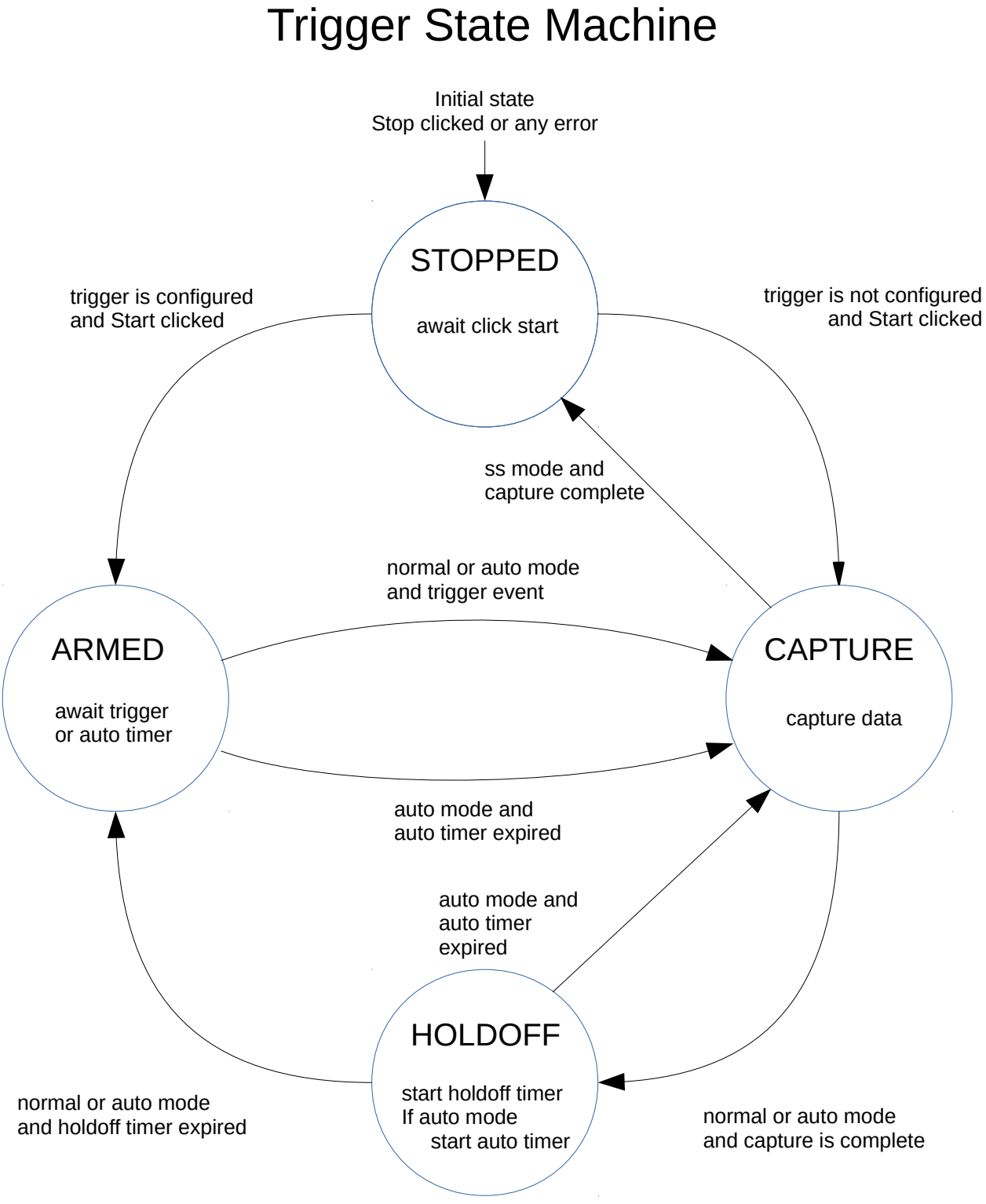

Figure 1 – Example of the UI view of a reasonable Trigger State Machine

retriggering. Automatic mode is an extension of normal mode, still responding to trigger events, but also enabling capture occasionally, even when there is no data satisfying the trigger criteria. The first mode is typically used to analyze events that are rare, while the two latter modes are normally conveniences for a human operator.

The first two modes are equally applicable to a logic analyzer. The third mode is also applicable, even though the logic analyzer does not have the issue of the trace possibly being off the viewable part of the viewport as with an oscilloscope. It is still useful though as it allows seeing whether there is any ongoing activity at all, without the necessity of performing manual triggering.

If the device operates in a pure streaming mode, then the concept of triggering in software may also be used to determine whether to keep or discard individual frames of data.

Figure 1 shows the usual UI view of triggering. While this is drawn as a state machine, it is a virtual state machine in that it is how the user sees it as working. It would likely not be implemented literally this way, because of differences in the capabilities of the various devices, some of which require doing triggering internally, and others require it to be done externally. In this figure, “capture complete” is to be interpreted as “configured capture length satisfied”.

This virtual state machine supports 4 modes of operation:

1. Normal mode
2. Single capture mode
3. Auto capture mode
4. Untriggered capture mode

Untriggered mode was not previously discussed because it doesn’t use a trigger. In this mode, the user can direct that capture start immediately without awaiting a trigger, and run either until a configured capture length has been satisfied, or until the user manually stops the capture, whichever occurs first. This mode would be useful if the captured data were then saved, and later analyzed by other means. This might be done if the user does not yet have any idea what to look for.

Support for multiple data buffers

It is traditional that when these repeating modes are used (including manually requesting a new single capture), no data is kept between one capture and the next. However, in the future it would be desirable to optionally alter that behaviour. If the last N captures were saved internally, it would be very convenient to be able to stop and go back and view past captures. Most software today allows saving any particular capture, but it is a tedious process. Providing a running historical queue of a few past captures, even though limited, could make the process much more efficient for the human. After all, most use of these devices is for the purpose of solving some problem.

Another benefit is that even if N is only 2 (it should be configurable from 1 to N), then the view would

not go blank during the time it takes to capture another data set. This would enhance the ability to visually detect small differences between one capture and the next. Most oscilloscopes today already sort of work that way (except for not being able to go back to the previous one).

Support for a scrolling viewport

In the case of long term captures at a low data rate (like a meter), it would be nice to be able to have the viewport scroll in time. Criteria similar to triggering and gating could be set up to stop the scrolling and alert the user. Also, for analog signal trace displays, it would be nice to be able to color code portions of the trace that are outside configured value limits, with or without stopping scrolling.

Timestamps

In the case of slow data captures like multimeters that do not provide timestamped data, it would be desirable for the application software to be able attach timestamps to the collected data, particularly if gating is supported. This would be appropriate whenever clock time has adequate resolution for a timestamp.

Decoders

Signal decoders are great for bridging the gap between uninterpreted data and interpreted data. They are already common today. It is desirable that decoders can also be added as plug-ins not requiring recompilation of the application software, and that users can create their own decoders.

Support for multiple simultaneous devices

This is where it can get really complicated. Because there is much diversity in the world of front end devices, there may be very little inherent compatibility between any two devices. It would still be desirable to support them as much as possible.

Some questions that arise in supporting multiple simultaneous devices include the following:

- Tabbed interface

In the simplest model of support for multiple simultaneous devices, the devices would all be considered as independent of each other, sharing a single application presence, probably using a tabbed interface similar to a web browser. Multiple instruments would be treated as completely independent of each other sharing nothing except the ability to select one to work with by clicking on its tab. Each device would be using its own lower level driver and state machine plug-in. To support multiple instances of the same device, each such plug-in would need to be able to support multiple connection instances, probably in distinct execution threads. In general, separate threads is probably always a good idea for the lower levels. There would be some linkage between the global GUI system and these worker threads. This model might permit only viewing and manipulating a single device at a time.

The next model up would provide mostly the same capabilities, except that it could provide for

multiple simultaneously visible viewports. This would allow the human operator to watch the operation of several devices at once, rather than only one. It might also provide dockable or separable viewports. There are multiple possibilities for this sort of multiple document interface.

- Cross-arming and cross-triggering

The next model up allows some limited interaction between the multiple devices. This might include crossover trigger arming, so that a detection of an event of interest by one device could cause other device(s) to begin collecting data (triggered or otherwise). Note that this does not imply any tight time synchronization of the individual capture activities, only an automation of actions similar to what a human might do given the same situation. One example is running a data test when the temperature reaches a certain value. This is the first level of support for multiple devices where the same effect is not equally achievable simply by running multiple separate instances of the application software.

- Synchronization

For low speed devices, like multimeters, adequate time synchronization of the respective data sets can probably be achieved simply be allowing one device or other event to initiate capture on several of them. For very high performance devices this is probably impractical, although cross-arming may still be useful for such devices, especially if they all monitor a single signal that can cause a trigger to occur.

- Shared viewports

Some combinations of multiple devices might be able to share the same viewport. This includes any combination of devices where the effective data synchronization is adequate for the purpose. Meters are a clear example. At around one sample per second, clock time likely suffices for all, and a set of meters displaying in the form of traces could share a single viewport. In such a display, hovering over a data point could bring up the actual reading. Even high speed devices could probably share the same viewport if they were all triggered by the same event, such as by a shared signal.

Even where the time synchronization is imperfect, it might be possible to provide a means for the operator to manually time skew the data from a device based on knowledge about the causal relationship between it and other devices. For some purposes this would be sufficient.

Saving and restoring configuration

Saving and restoring complex configurations is always desirable. Supporting multiple types and number of devices makes it more challenging to implement.

Saving and restoring data sets

Saving and restoring data sets is always desirable, especially if further and different analysis can be done following restoration of the data set(s). Supporting multiple types and number of devices also makes this capability more challenging to implement.