Distributed Monitoring Architectural Reference Guide

12/9/2013

Table of Contents

OVERVIEW and BACKGROUND

This guide provides a recommended architecture for applications involving embedded, distributed monitoring applications using the National Instruments LabVIEW graphical system design software. The architecture discussion in this document specifically focuses on the NI Compact RIO platform, however, is applicable to other National Instruments platforms including the NI Compact DAQ Standalone line of products and PXI.

Definition and Examples

A distributed monitoring system is defined as several self-contained Data Acquisition and Analysis Nodes (DAAN's) dispersed in geography and collecting synchronized or unsynchronized data. Distributed monitoring is an umbrella term and encompasses several different applications.

One example is Machine Condition Monitoring (MCM) where health information about various machines (pumps, motors, blowers, etc.) is collected with a goal of predicting failure. A second example is pipeline monitoring where data acquisition nodes are distributed along gas pipelines and used to monitor leaks and theft of oil within the pipeline. The graphic in Figure 1 highlights other applications areas within the distributed monitoring umbrella.

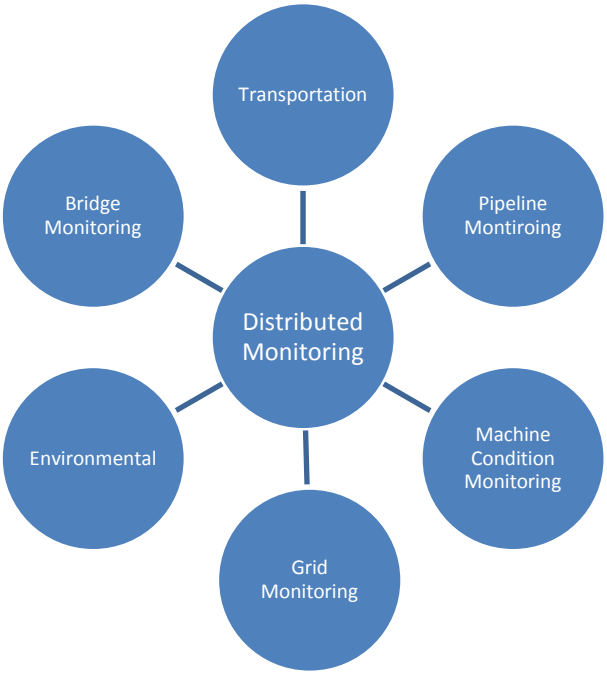


Figure 1 Distributed Monitoring Applications

Business Drivers

There are numerous business drivers and benefits for implementing a distributed monitoring system. The obvious factors include loss of revenue from unplanned downtime and maintenance expenses. One

estimate places the amount of unnecessary maintenance expenses at $18billion dollars out of a total of $100 billion in US maintenance spending representing 3% in lost revenue[1].

Outside of the obvious cost and revenue impacts, there are also human factors driving business decisions. One recent article describes how the percentage of unplanned downtime will worsen over the coming years due to the retirement of seasoned operation and production personnel. "In excess of 40% of current production staff will reach retirement age by 2015. The loss of these staff along with the associated loss of "intellectual capital" is worth noting"[2].

Safety is a primary concern with today's workforce. Machines, assets, and structures are increasingly being installed in remote, hard to access locations. Offshore wind farms, the Canadian oil sands, and remote pipelines are all examples of hazardous locations where it is both dangerous and costly for humans to access. Distributed monitoring solutions significantly reduce the number of worker trips to these dangerous environments because operators can move from a reactive (run to failure) or preventive maintenance (time-based) approach to a more predictive maintenance approach.

[1] Mtelligence webcast: http://www.mtelligence.net/mtelligence-webcasts/

[2] Dennis Nash, "When will it Fail? Anticipating Equipment Failure Using Predictive Analytics", http://www.sustainableplant.com/assets/Sustainable-Plant/PDFs/Predictive-Analytics-When-Will-It-Fail.pdf

System Diagram

The diagram in Figure 2 provides a high-level view of a typical distributed monitoring application.

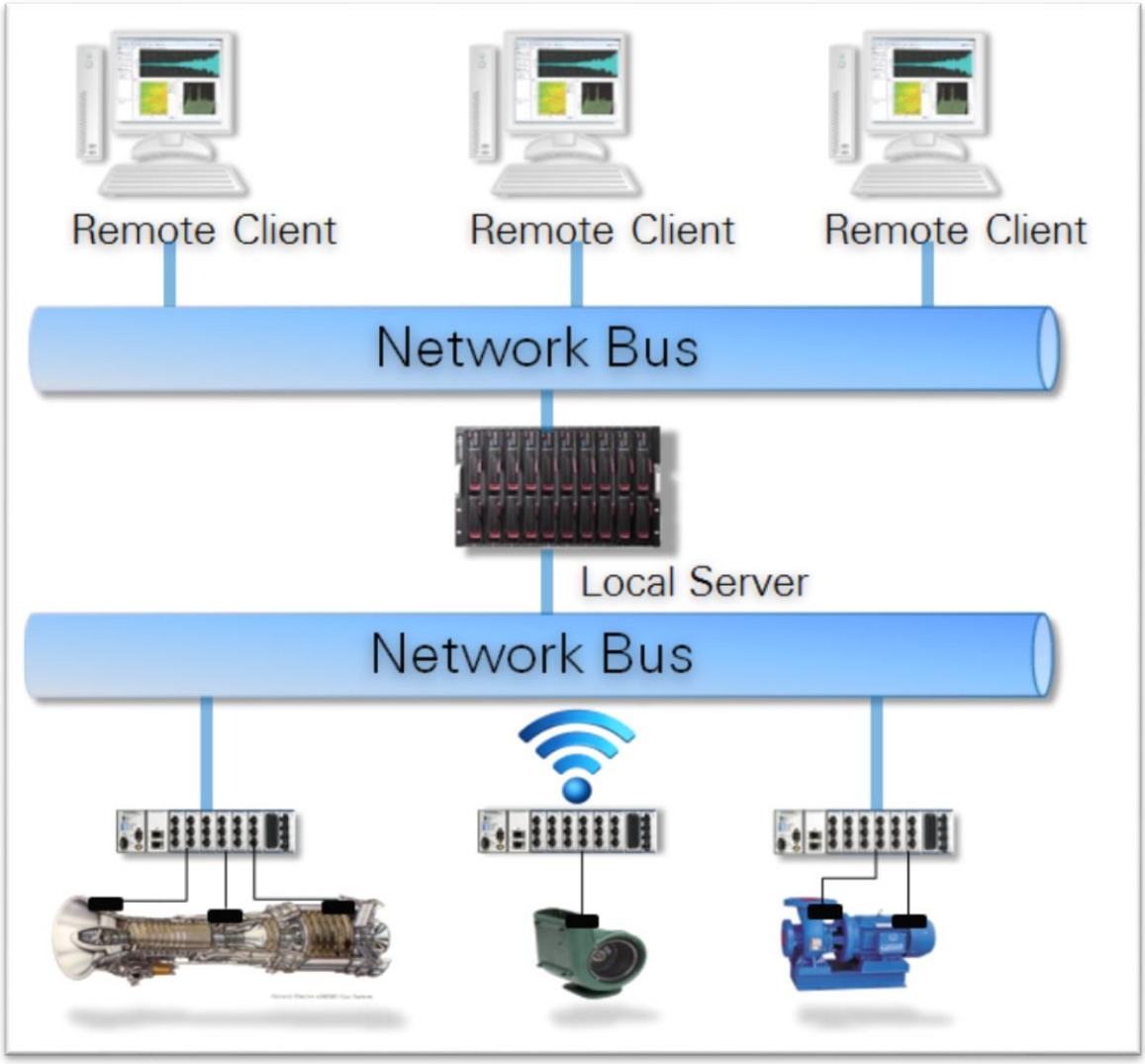


Figure 2 Distributed monitoring system architecture

Several Compact RIO Distributed Acquisition and Analysis Nodes (DAAN's) are connected to various types of machinery (a turbine, a pump, and a motor). These machines can be found in a typical power generation facility and are normally distributed across a plant covering several miles. Each machine is instrumented with a wide variety of sensors for measuring physical phenomena such as vibration, temperature, pressure, sound, etc. The DAAN's measure, collect, and with Compact RIO can perform on board signal processing of the information coming from the various sensors. The information collected is either stored locally for future retrieval or continuously streamed to a centralized server/processor.

Data is forwarded to a centralized server(s) via a network bus. The networking bus can be wireless or wired Ethernet, cellular, and radio (900MHz) or a combination of all of these. Remote clients then request data from centralized server(s) for visualization and further processing of the data.

Background Information

As previously stated, the goal of this document is to discuss a typical application pattern in embedded, distributed monitoring applications. The document focuses primarily on the embedded software architecture and won't cover how one might architect a remote client for viewing and parsing of the data from the server. Code samples, however, will be provided when relevant showing how to access the embedded software.

The author assumes the reader has some knowledge of embedded systems and specifically the National Instruments cRIO platform. Many of the concepts build upon the information discussed in the NI LabVIEW for CompactRIO Developer's Guide. It is recommended for those new to the CompactRIO platform to read this document before proceeding.

Some of the code samples and projects utilized in this document are based off of National Instruments Systems Engineering code. A complete list of all code utilized is listed in Appendix A with appropriate links.

Finally, the architecture drawings were created using a vector-based drawing program called yEd Graph Editor, which a free design tool downloadable from yworks.com

HARDWARE DETAILS

The DAAN's are built on top of the National Instruments cRIO platform. The Overview and Background section of the NI LabVIEW for CompactRIO Developer's Guide provides further information about the capabilities of the platform. The code in the provided LabVIEW projects is built on top of the NI cRIO-9074. This DAAN is an 8 slot chassis with an integrated controller.

The first six slots are dedicated for vibration, tachometer and proximity probe measurements and utilize the NI 9232 3 channel, industrial vibration, cSeries module. At least one of these modules must be present, but not all six, in order for the code samples to function properly. Optionally, slot 7 can be populated with the NI 9205 16 channel, differential, general voltage input cSeries module. Use of this module is for non-vibration sensors having a voltage output that can be scaled to engineering units. Optionally, slot 8 can be populated with the NI 9219 4 channel universal input cSeries module. This module can be configured to accept inputs from sensors including Thermocouples, RTDs, strain gauges, general bridge based sensors, and sensors with a 4-20mA output. In the current configuration, with the samples provided all channels are setup for 4-20mA input.

In all, this system allows for a maximum of 18 vibration, tachometer or proximity probe measurements, 16 general purpose voltage inputs, and 4, 4-20mA sensor inputs. This configuration covers a majority of distributed monitoring applications requiring a mixture of high-speed vibration data in conjunction with lower speed temperature, pressure, strain, etc. data. Of course, each application is unique and there

are dozens of other available cSeries modules available to acquire data specific to a particular application.

It is important to note while other configurations are possible, because of the nature of the FPGA on the cRIO any chassis, controller, or module changes requires an update to the provided sample code.

Module Classifications

This section describes how the different types of cSeries I/O modules used in this project are classified. It is important to understand how these modules are designed to properly implement timing and synchronization. The basic types of module classifications are presented in Figure 3

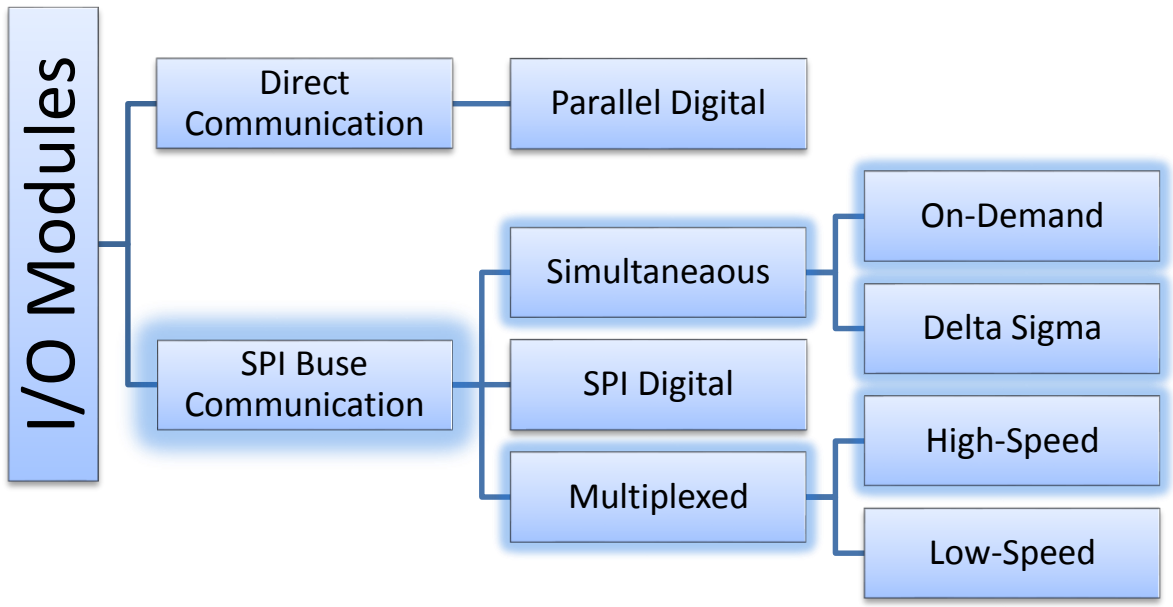


Figure 3 C Series Module Classification Organizational Tree

In this application only SPI>>Simultaneous>>On-Demand (9205 and 9219) and Delta Sigma (9232) modules are utilized. Therefore, these classifications are the focus of the remaining part of this section.

SPI Bus Communication

The Serial Peripheral Interface bus, or SPI, is a standard 4-wire communication protocol set up for master/slave, full duplex (two-way simultaneous) communication. The bus clock rates typically range from 1 MHz to 70 MHz. The SPI clock rate CompactRIO hardware uses to communicate with C Series modules varies but usually operates around 10 MHz. The common architecture of SPI bus modules contains a complex programmable logic device (CPLD) that, on the module side, controls the timing to and data collection from the ADC/DAC chips. On the chassis side, the CPLD communicates via SPI bus back to the FPGA. Based on the specific module used and slot location, the LabVIEW API knows how and

where to communicate to the individual CPLD on the module. This is why new module support is added in new versions of LabVIEW.

SPI Bus Challenges

Regardless of front-end circuitry, each SPI module communicates to the FPGA in a CompactRIO chassis over a single dedicated SPI bus. This means even though commands to the module (from multiple loops on a LabVIEW block diagram) or data retrieval (from modules with multiple ADCs) can happen in parallel, those commands must be interlaced through a single SPI bus. LabVIEW and the NI-RIO driver can properly prioritize multiple calls coming in at the same time, but if a call comes in while the previous call is still being handled this may introduce unwanted jitter into the control or data acquisition loop. Another way to convey this concept is to say I/O node calls to a single module from different parts of a LabVIEW FPGA program create a "hardware race condition." This in no appreciable way limits the abilities of the module or a CompactRIO system, but it does require arbitration of I/O node calls within the application. The easiest way to do this is to keep all of the I/O node calls in the same loop, or utilize sequence structures or semaphores. A reference design for semaphore implementation in LabVIEW FPGA is discussed in the NI Developer Zone document Semaphore Reference Design for LabVIEW FPGA.

As one continues to read about the different classifications of modules, note all modules utilizing SPI communication must adhere to the same caveat of hardware arbitration.

Simultaneous Modules

Modules which are simultaneous have one ADC per channel and acquire data with no appreciable skew between channels. The two subcategories of simultaneous modules, on-demand and delta-sigma, transfer data via the SPI bus and are subject to all of the specifications and challenges of other SPI bus modules.

On-Demand Conversion

On-demand modules, like the 9219, have few specific challenges when it comes to programming with LabVIEW FPGA. This makes them some of the easiest modules to program. The biggest caveat involves arbitration, which is shared by all modules using SPI communication.

Data on Demand

Data on demand is less of a caveat and more of a feature. On-demand simultaneous C Series modules have the ability to return data when the I/O node is called at any interval down to the minimum conversion time as listed in the manual. This means the acquisition can be clocked by external, irregular clocks. The Δt does not need to be a constant for an acquisition with an on-demand simultaneous module.

Pipelined Simultaneous Data or User-Controlled I/O Sampling

Some modules designed for high-speed measurements exceed the data throughput capabilities of the FPGA I/O node. In these situations, implement user-controlled I/O sampling functions to communicate with a module. These add complexity to the program but dramatically increase the bandwidth from the module.

Delta-Sigma Modulation

Many C Series modules, like the NI 9232 are designed for high-speed, dynamic measurements and utilize delta-sigma (ΔΣ) converters. To better understand how these modules work, it is important to understand the fundamentals of delta-sigma modulation. The Greek letters delta and sigma are mathematical symbols for difference and sum respectively. The modulation circuit of a delta-sigma converter compares the running sum of differences between the desired voltage input, Vin, and a known reference voltage, Vref. The output from the comparator becomes a bitstream that is sent into a digital filter and a 1-bit DAC. Because of this negative feedback, the differences oscillate around 0 V, or ground. The digital filter effectively keeps track of how many times the difference is above 0 V and based on this count, along with the reference voltage, it is possible to determine the input voltage. This modulation loop runs at a much higher frequency than the actual output frequency of the converter.

C Series modules with delta-sigma converters feature an oversample clock which runs the modulation circuitry. Oversample clocks, which run at 12 MHz or faster, affect timing, synchronization, and programming paradigms. The following list provides insight into the specific challenges of C Series modules using delta-sigma modulation.

- **Need a Sync Pulse to Reset**—Oversample clocks need to be "reset" before they are used. This is why there is a LabVIEW FPGA I/O node to send a "start" event to the module.
- **Time to Data Ready Is Non-Zero**—The time between the "start" event and data availability is specified as "time to first data." This time may vary slightly between homogeneous delta-sigma-based modules and greatly between modules of other types. On-demand modules have a "time to first data" of zero. Information on how to align data sets can be in KnowledgeBase 4DAEUNNQ: How do I Compensate for Different Group Delays with C Series Modules in LabVIEW FPGA and Knowledgebase 53CHLD6C: What is the Best Method to Synchronize Two Different DSA Modules in LabVIEW FPGA?
- **Sample Rates Are Discrete and Specific**—Because of the oversample clock and the digital filter, delta-sigma modules can run only at discrete sample rates. These sample rates are a function of a divisor and the oversample clock. This is why the "rate" input for a delta-sigma module is an enumerated data type of predetermined sample rates. If a non-supported sample rate is input, then it is rounded to the next highest available sample rate.
- **Minimum Sample Rates Are Greater Than 1 k**—The minimum sample rate for delta-sigma modules is often over 1 kHz. Use averaging, filtering, or some form of decimation to further reduce the data set beyond the rate the digital filter on the module outputs.
- **No Irregular or External Clocks**—Delta-sigma modules cannot report data "on demand" and thus do not work with irregularly timed I/O node calls because the iterative process must complete a large number of loops before returning accurate data. The I/O node for a delta-sigma module always blocks for the exact (Δt) for which the module has been set to acquire. An I/O node call at an interval less than Δt must wait until the full sample period is complete. This gives the oversample circuitry enough time to calculate an accurate value. To compensate for this, implement a resampling algorithm on the FPGA before processing data or using it in a control loop.

- **Physically Share Oversample Clock to Synchronize ΔΣ Modules**—Two delta-sigma modules needing synchronization must share the same oversample clock. To synchronize modules, the oversample clock from one module must be exported from the right-click properties menu of the "source" module in the LabVIEW project. It must be imported from the same menu of the other "client" modules. Any module can be a "source" or a "client" module—it is up to the programmer's discretion. Remember, changes made to a module property window from the project view cause a recompile and cannot be changed at run time.

Multiplexed

One of the more expensive components of a module is the ADC. By using a multiplexer, also known as a mux, to route multiple channels through a single ADC, multiplexed modules offer higher channel counts at lower per channel prices than simultaneous modules.

Before learning how to program these modules, it is important to understand some specification-level details. First, the sample rates are often listed as the total rate of all channels put together, also known as an aggregate rate. From the module hardware standpoint, all channels selected must run at the same rate (aggregate rate divided by number of channels), but from the program standpoint, FPGA processing can remove samples from select channels. Second, it is important to note there is an interchannel delay, or skew, between all channels in a multiplexed module. Implementing processing in the FPGA to compensate for this skew via shifting or data resampling is an option, but most systems incorporating multiplexed modules are not impacted by this small offset. If hardware-based phase alignment is important to the system, select a module with multiple ADCs.

There are two main subsets of multiplexed modules to choose from: high speed and low speed. The NI 9205 is a high-speed multiplexed module and therefore will only focus on this particular subset.

High Speed

High-speed multiplexed modules implement a double pipeline to increase the throughput of data to the chassis. With a double pipeline, the first valid data cannot be returned until the process has run two complete iterations. Once the first two iterations have run to "prime" the pipeline, the subsequent iterations begin to yield valid data. When using an I/O node to sample channels, the pipeline is automatically managed by the FPGA I/O node, and the channels within the FPGA I/O node are sampled in numerical order regardless of the order they appear in the node. If the first two channel requests in the FPGA I/O node do not match the two channel requests stored in the module pipeline, there is a delay before the first channel sample occurs. This delay is caused by the FPGA I/O node automatically updating the module channel sample pipeline, which takes two channel sample cycles. This situation could happen if there is an I/O node addressing the same module from different parts of the block diagram. For example, if in an "init" case of a Case structure, channels 0, 1, and 2 are read from an I/O node and then in the "acquire" case channels 5, 6, & 8 are read, one incurs the pipeline updating penalty because the module is already primed for channels 0, 1, and 2 and needs to flush the pipeline. If this delay causes problems, the workaround is to acquire all of the channels in the same step and place data from channels 5, 6, and 8 into a FIFO to be called upon later.

I/O Sample Method for High-Speed Multiplexed Modules

Some high-speed multiplexed modules, such as the NI 9205 and NI 9206, have an alternate method for programming. This method, referred to as the "I/O sample method," is more difficult to implement but takes up less FPGA space and provides an easier input into DMA nodes on a LabVIEW block diagram. This lower-level access to the module does not account for the double-pipeline architecture; therefore, the first two data points returned from the I/O sample method must be explicitly discarded because they are invalid. The most difficult step is building the channel array to feed into the sample method. Search the example finder for "NI 9205 Basic I/O Sample Mode" for an example of this method.

Synchronizing Modules

Some applications, such as vibration or sound measurement, require a high level (sub 100 nS) of synchronization between channels. This section discusses timing and synchronization of both delta-sigma-based modules and scanned (SAR) modules. Any NI C Series I/O module not classified as delta sigma is classified as SAR.

Synchronizing Delta-Sigma Modules

To synchronize delta-sigma modules in CompactRIO hardware, the oversample clocks and start triggers need to be physically shared between all of the modules. Most module manuals provide specifications for the phase match between channels and between modules. Delta-sigma-based modules may be synchronized even if they are not the same model number.

1. Select one of the modules, "the master," to export the clock to the backplane. The other modules are set to import this clock from the backplane. Whichever module selected as the master (typically module one) overrides both the timebase and available rates. For example, if the desired system rate is 51.2k, select the NI 9234 or NI 9232. If 50k rates are desired, select the NI 9237. Modify the import/export properties from the properties window accessed via the right-click menu of the module in the LabVIEW Project Explorer. Import/export settings cannot be set programmatically because this information is compiled.

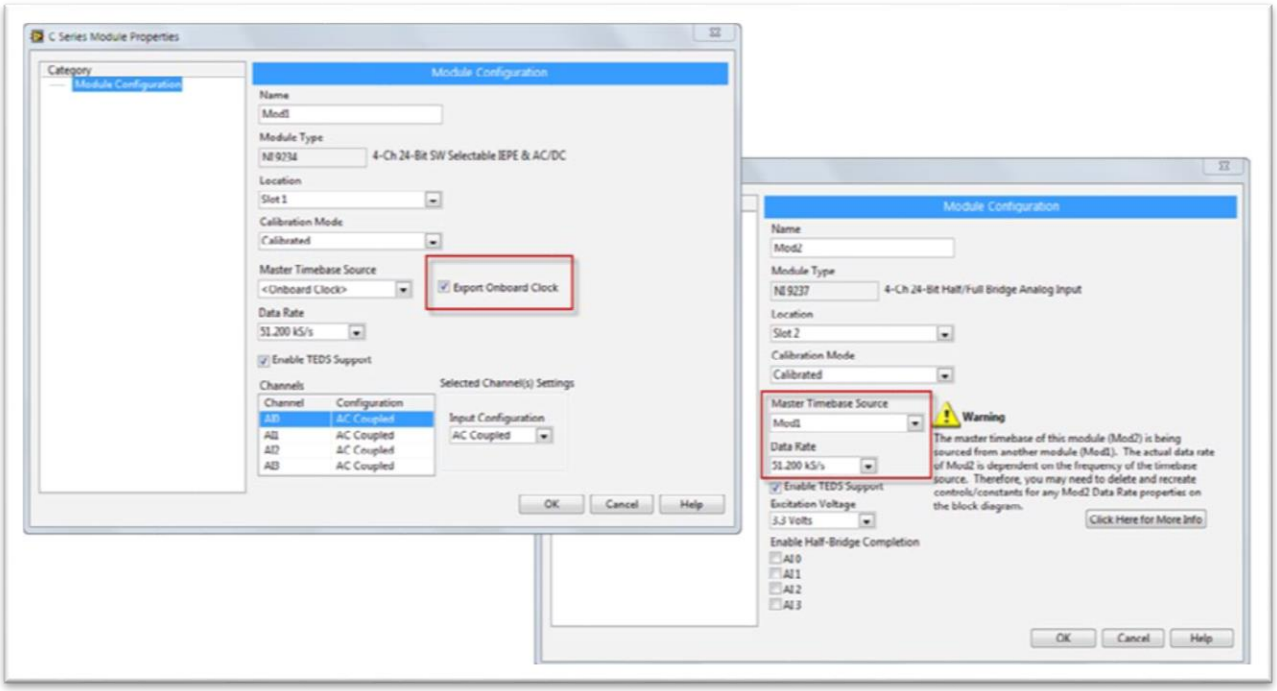


Figure 4 Export the clock from the chosen master module (left) and the Master Timebase Source from all subsequent modules to be synchronized (right)

2. On the block diagram, create a property node for each I/O module and use the Data Rate enumeration to specify the rate, as shown in Figure 5. Note even though the I/O modules share the same sampling rate, one must create a unique Data Rate enumeration for each property node (right-click on the property node for each module and select “create constant”). This ensures the enum integer is properly matched to the expected rate for the specific I/O module.

3. Create a Start Trigger for each I/O module and place them into the same I/O node. This ensures the start triggers are properly routed.

4. Place all channel reads from all synchronized modules in the same I/O node as seen in Figure 5. Using this process, it is possible to mix and match any of the existing simultaneous delta-sigma modules.

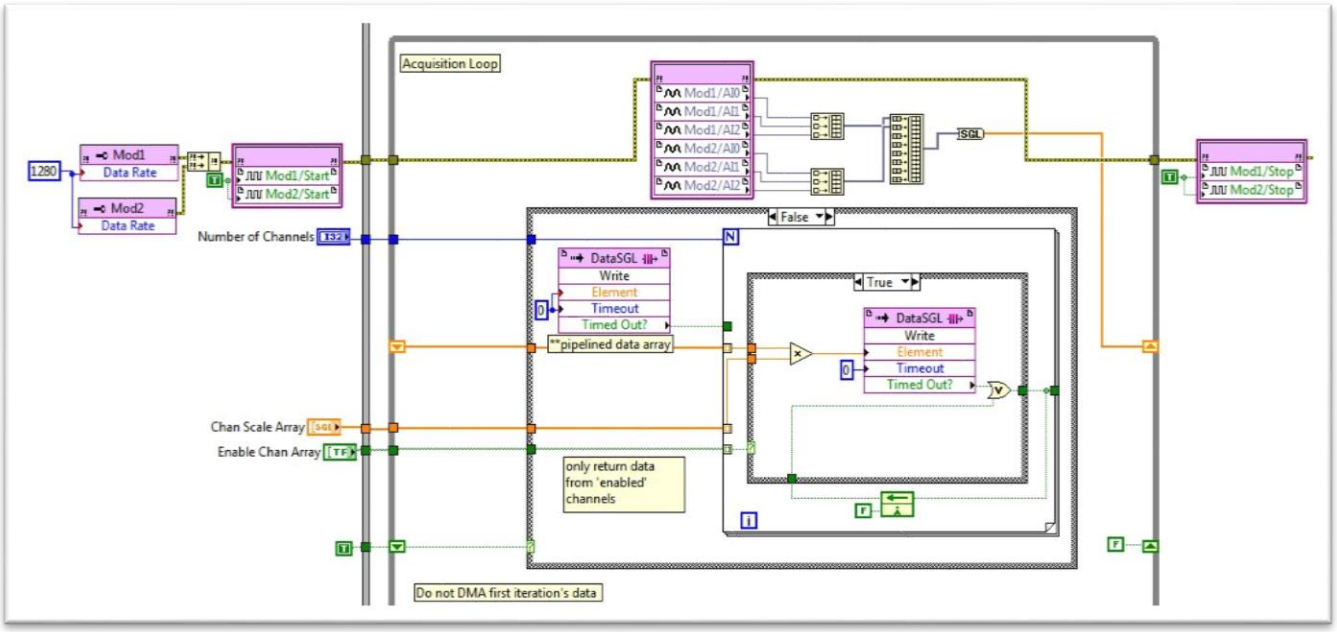


Figure 5 Block Diagram for synchronizing NI 9232 modules

The best method for synchronizing different delta-sigma modules in LabVIEW FPGA is to have the I/O nodes for each module in the same While Loop. If the I/O nodes for the different modules are placed in parallel While Loops, then additional startup delays need addressing. Group delay needs to be accounted for with each module, because the modules acquire data at the same time when in the same loop. KnowledgeBase 4DAEUNNQ: How to Compensate for Different Group Delays with C Series Modules in LabVIEW FPGA provides tips on how to compensate for different group delays within C Series modules in LabVIEW FPGA.

Synchronizing Simultaneous On-Demand Modules

This process is easier than delta-sigma modules because there is no oversample clock to share. These modules are clocked by a convert pulse originating from the programmed I/O node on the FPGA. To synchronize the convert pulse, place all channel reads or updates in the same I/O node call. It is then possible to mix analog input, analog output, and digital channels in the same I/O node with minimal skew.

Synchronizing Multiplexed Modules

Multiplexed modules sharing the same model number operate in “lock step” as they move through the channels. Channel 0 on each module is synchronized as are channels 1 through n. This is more informational than instructional because multiplexed applications rarely are affected by interchannel delay.

Synchronizing Delta-Sigma and Scanned (SAR) Modules

Synchronizing delta-sigma modules with SAR (non-delta-sigma) modules is slightly more complex. This is because delta-sigma modules have their own timebases, while SAR modules are a slave to the FPGA clock. From a programming perspective, loop timing with delta-sigma modules is determined by the

data rate node; While Loop timing with SAR modules is determined by using the FPGA loop timer. The best way to synchronize delta-sigma with SAR modules is to design the application for delta-sigma timing (similar to Figure 5) and then add the I/O blocks for the SAR modules as a separate I/O node shown in Figure 6. This requires synchronizing SAR modules to a delta-sigma module clock.

Note: Delta-sigma-based modules and SAR modules should not share the same FPGA I/O node because they execute in series and limit the maximum data rate of the system.

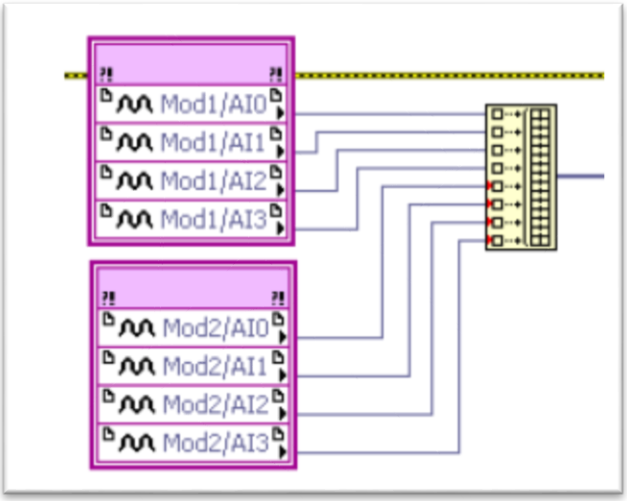


Figure 6 FPGA I/O Nodes for Hybrid Applications (Delta-Sigma and SAR Modules)

An application becomes more complicated when it requires multirate synchronization between delta-sigma and SAR modules. Since these modules have different timebases, they cannot share a clock when they are separated into two or more loops executing at multiple rates. The best option is to separate them into multiple loops and then expect some drift between the timebases over time, in addition to a phase offset caused by the varying startup times.

As stated earlier, many of the concepts of this section are described in further detail and provide further information about the various hardware implementations of a cRIO system in the NI LabVIEW for CompactRIO Developer's Guide .

Pipelining Code Execution

Another important discussion topic is pipelining the acquisition code to optimize the FPGA code. Pipelining is an extension of the parallel code execution concept that works within a single process. Instead of partitioning the process, pipelining is utilized to achieve parallel code execution by partitioning the code sequence into smaller segments executing over multiple iterations of the loop. As with parallel loops, the smaller code segments run in parallel in the same loop. By reducing the length of the critical path (longest code segment) in each loop iteration, the loop execution time is decreased.

The following diagram illustrates how a process consisting of A and B code segments can be pipelined to reduce the length of each loop iteration. Passing data from one loop iteration to the next (from A to B) is

easily implemented using the shift register.

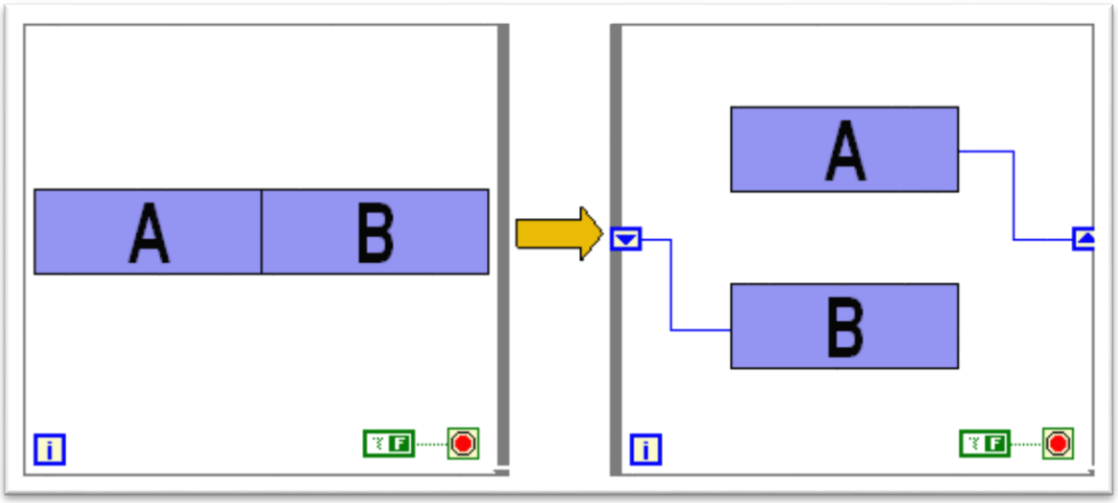


Figure 7: Illustration of pipelining in a LabVIEW While Loop

Refer to Figure 8 which is the same code discussed in the synchronization section, and the code contained in the supplied application. This code implements pipelining. Instead of serializing the acquisition of data and the scaling of the data the application pipelines those processes. On the first iteration of the loop data is acquired and set in a shift register. On the subsequent loop iteration data is again acquired, but this time data from the initial iteration is scaled to engineering units and written to the DMA buffer. By utilizing pipelining the code is able to run at a faster rate.

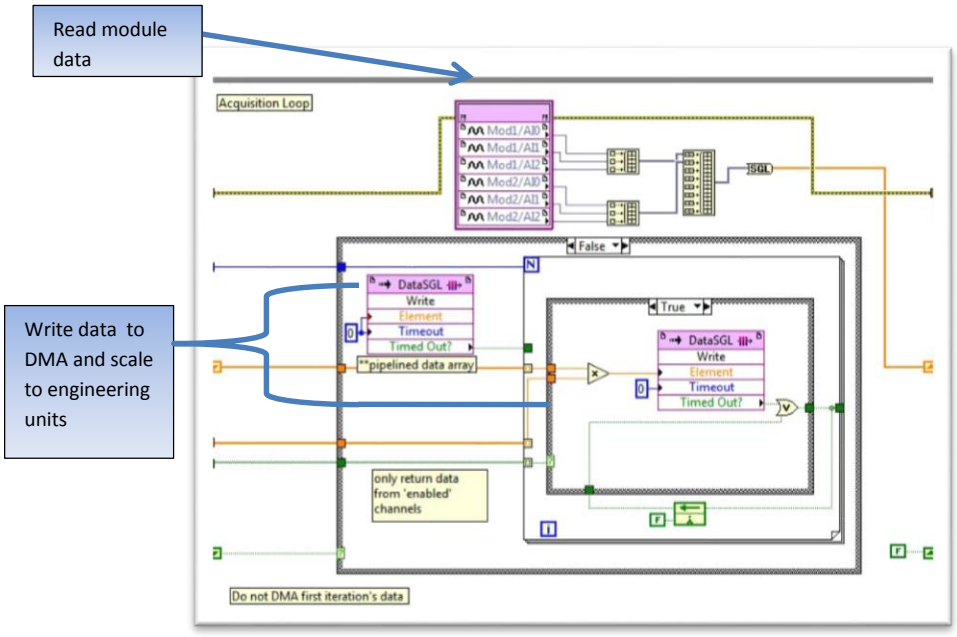


Figure 8 Pipelining of acquisition data

TECHNOLOGIES

As previously shown a typical distributed monitoring system might be described by the following figure.

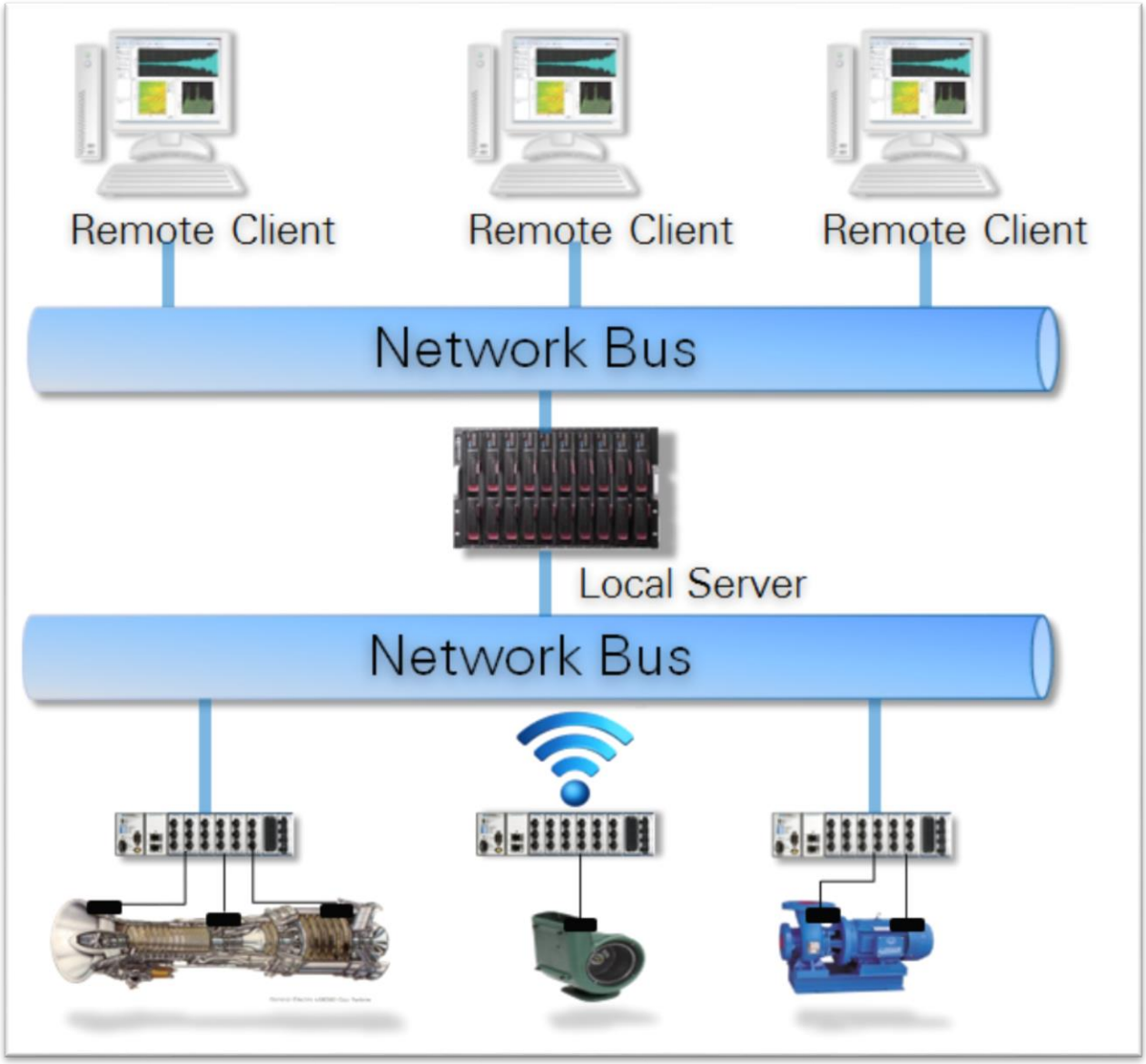


Figure 9 Typical Distributed Monitoring System

By zooming in on this picture and breaking down further how the pieces interact, there are several technologies utilized to get sensor data from the Data Acquisition and Analysis Node to the Remote Client for future or interactively viewing data on the DAAN at a given time. The technologies discussed here are being run within either the Real Time Operating System on the cRIO and/or on the Remote Client machine.

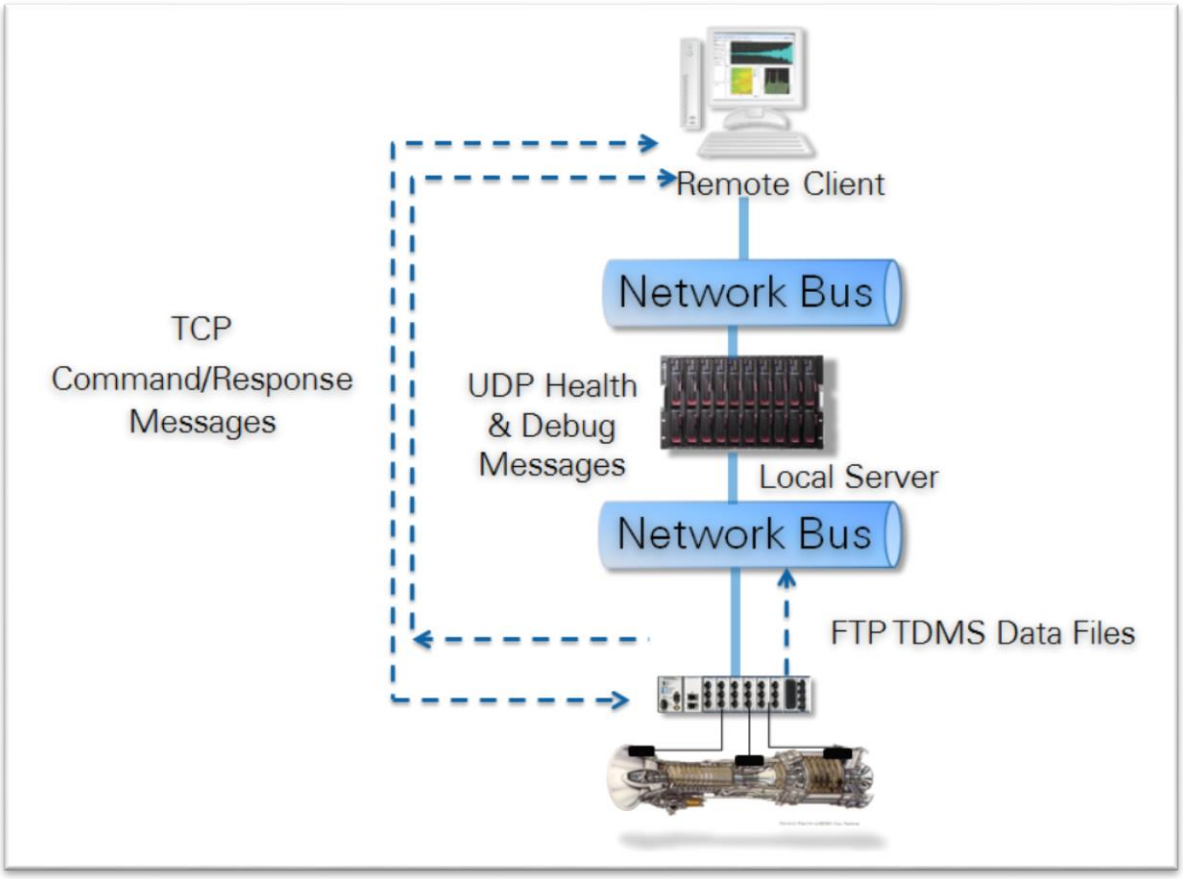


Figure 10 Architecture Technologies

Raw Ethernet (TCP/UDP)

TCP and UDP are the low-level building blocks of all Ethernet standards. Tools for raw TCP and UDP are natively supported in virtually every programming environment including LabVIEW. They offer lower-level communication functions which are more flexible but less user friendly. Developers must handle functions such as establishing connections and packaging data at the application level. National Instruments systems engineers developed a Simple TCP/IP Messaging (STM) architecture which abstracts some of the complex, lower level functions of TCP/IP. The STM protocol utilized in this architecture is discussed in later sections.

TCP or UDP are good options if it is necessary for very low-level control of the communications protocol or if designing a custom protocol. They are also recommended for streaming data to third-party applications.

TCP provides point-to-point communications with error handling for guaranteed packet delivery. TCP communication follows a client/server scheme where the server listens on a particular port to which a client opens a connection. Once establishing a connection, data is freely exchanged via basic write and read functions.

With TCP functions in LabVIEW, all data is transmitted as strings. This means flattening the Boolean or numeric data to string data to be written and unflattened after being read. Because messages can vary in length, it is up to the programmer to determine how much data is contained within a given message and to read the appropriate number of bytes. Refer to the LabVIEW examples “Data Server” and “Data Client” for a basic overview of client/ server communication in LabVIEW.

As stated previously, TCP provides a good mechanism for point-to-point communication with error handling for guaranteed packet delivery. In this architecture, TCP is used to send commands which must be guaranteed to be delivered to the sender. Commands like rebooting the remote acquisition node, sending a user initiated trigger from the client computer, and sending live data back to the client computer upon request are all examples of the types of information which are critical to ensure are received; or at the very least error if there is some sort of network miscommunication.

UDP can broadcast messages where multiple devices can receive the same information. UDP broadcast messages may be filtered by network switches and unlike TCP do not provide guaranteed packet delivery. In this architecture UDP sends health and debug information about the DAAN. Examples include how much memory is currently being utilized by the DAAN, if any local errors occurred, and what the current health of sensors connected to system are. While important information, information like this is not critical to the overall running of the DAAN or the client machine, therefore, if packets are dropped the system continues operating normally.

FTP

FTP (File Transfer Protocol) is, as the name implies, a network protocol for transferring files from one host to another. FTP is an additional abstraction on top of TCP (like STM), providing a robust mechanism for ensuring the transfer of data files across a network. Handshaking, guaranteed packet delivery, and error handling are all supported by the FTP standard.

FTP follows a client server architecture where one or many servers provide data to one or many clients on the network. The National Instruments cRIO devices natively support being an ftp server, and for this architecture do indeed act as the server of information (files). For a given facility there is typically one local server (or PC) which acts as the FTP client. In other words, the client (local PC) requests at given intervals the DAAN’s send the data files they have captured via their FTP server.

TDMS

The Technical Data Management Solution (TDMS) is an open source file format specifically designed to store and easily retrieve high-speed streaming data. This is a natural fit for distributed monitoring applications which typically record some form of vibration or high-speed data. For example, sample rates for vibration acquisition are on the order of 5,000 to 20,000 samples/second. For a full system of 18 vibration channels acquiring for 5, 10, or 20 seconds there is going to be tremendous amounts of data being produced. This is a Big Analog Data™ problem. TDMS is a great format for such Big Analog Data™ applications because of the way it saves high fidelity data in an optimized manner while still being able to retrieve the data quickly.

The TDMS data model offers three levels of hierarchy, as shown in Figure 11 – file, group, and channel. The file level can contain an unlimited number of groups, and each group can contain an unlimited number of channels. Because of this channel grouping, data is organized in an easy to understand hierarchy. For example, there might be one group for raw data and another group for analyzed data within one file. Or, there might be multiple groups corresponding to sensor types.

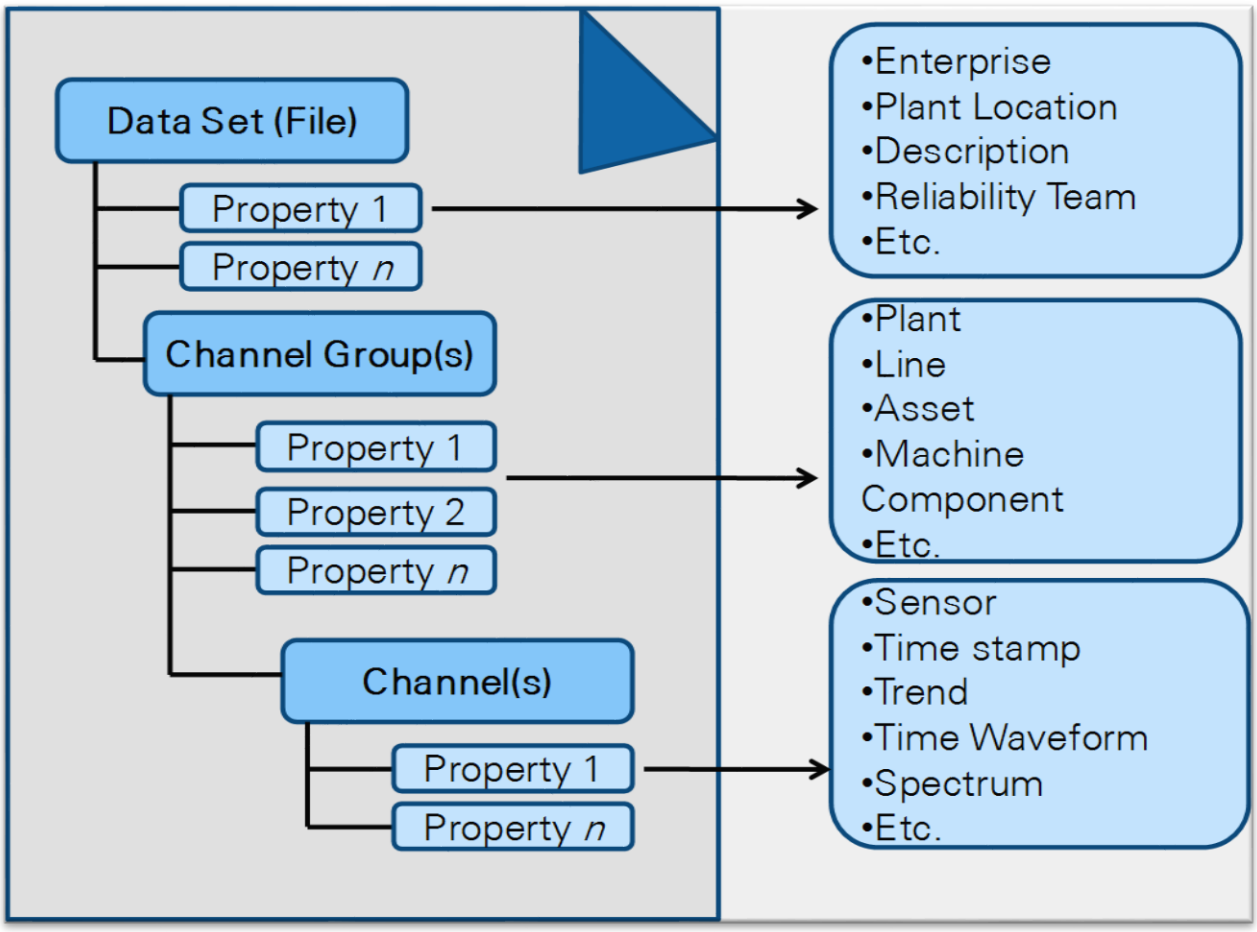


Figure 11 TDMS File Hierarchy Description

User defined custom properties may be inserted at each of the three levels. Each level accepts an unlimited number of custom-defined attributes to achieve well-documented and search-ready data files. The descriptive information located in the TDMS file, a key benefit of this model, provides an easy way to document the data without having to design a customized header structure. As documentation requirements increase, the application does not need redesigning; simply extend the model to meet new system requirements.

Further information about TDMS and the NI Technical Data Management platform can be found at NI Technical Data Management. Appendix C outlines the TDMS schema utilized in this application.

STORE and FORWARD

There are many pieces running on the Data Acquisition Analysis Nodes (DAAN's), but perhaps the most important piece is the Data Acquisition state machine. Fundamentally, this piece is crucial in order to collect sensor data; and without meaningful data the system is essentially useless.

We have devised a measurement scheme termed "Store and Forward". The state machine for this Store and Forward implemention is illustrated in Figure 12.

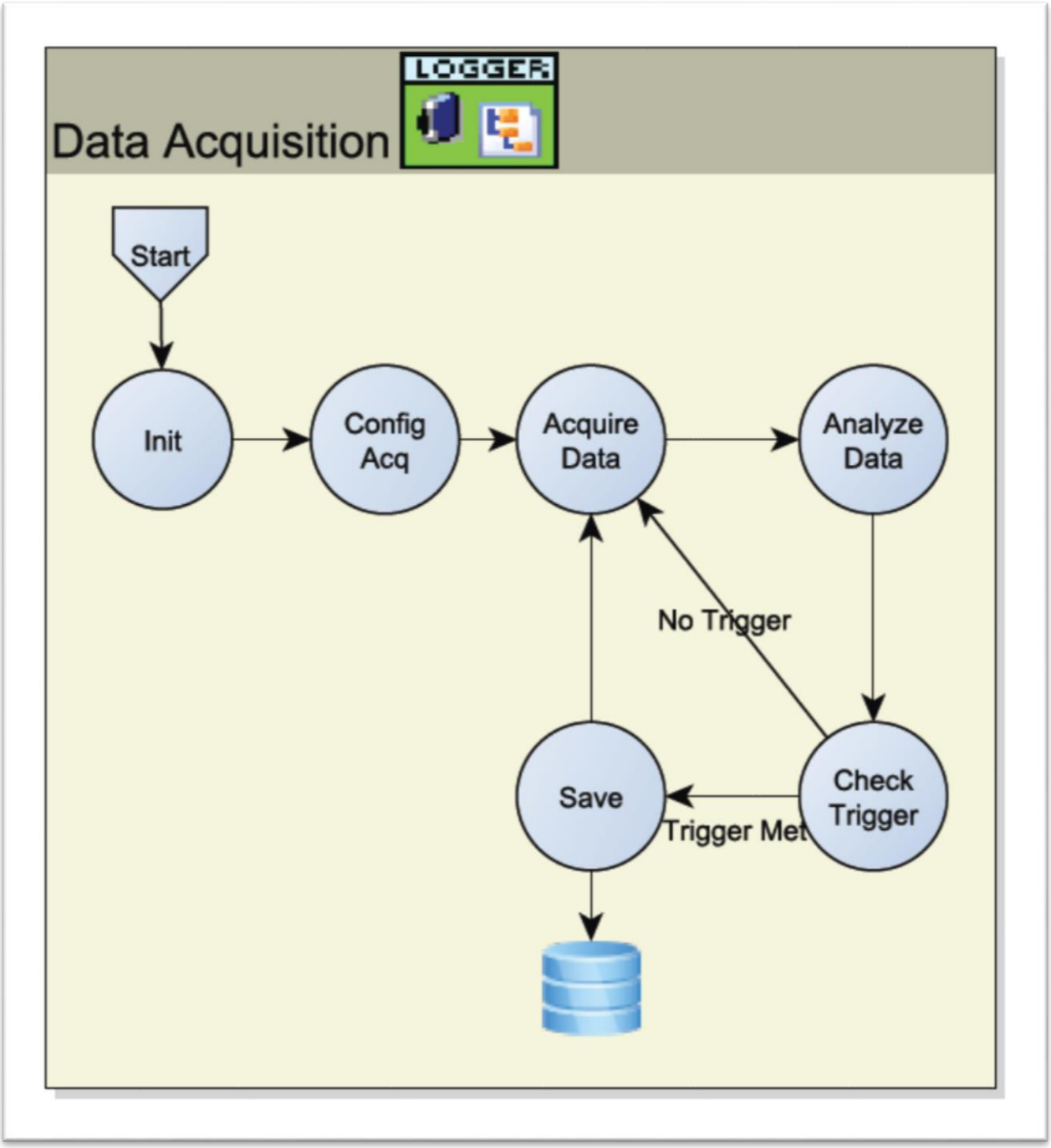


Figure 12 Store and Forward Measurement Architecture

Store and Forward State Machine

A subsequent section will describe in more detail how a state machine is implemented in LabVIEW, but for now all that is important to know is a state machine is quite simply a recipe for how a system reacts to a given set of inputs and a set of possible actions or output events resulting from new states(s).

After starting the DAAN the measurement hardware must be initialized. Note, there is a system wide initialization routine which happens before this block and will be described in more detail later; but for now just assume initialization has occurred without error.

After initialization and configuration of the measurement hardware, data starts collecting (Acquire Data) in an internal buffer. This is internally called the "pre-trigger" buffer and is typically expressed in units of seconds or blocks (one second, two blocks, etc). The pre-trigger buffer is implemented as a lossy queue of fixed size. If no trigger occurs before the pre-trigger buffer is filled then the oldest data is removed and replaced with the most recent data.

The pre-trigger buffer is often important because operators want to know what was happening in the machine right before some event occurred. If data is captured only AFTER an event occurred, then in most situations, the most valuable data is already lost. After each block of data is captured it is both put in the lossy queue and then sent to the analysis state for further processing.

The analysis state contains any analysis the user requires. Some of the analysis functions performed in the included architecture include integration, windowing of the signal, calculating speed of the machine, determining a given signal level, etc. The ability of the cRIO hardware to perform processing on board is very powerful because decisions are made as close to the sensor as possible about whether the data is interesting and should therefore be saved for future review. There is no delay sending the information to a host machine first for data analysis. After analyzing a given block of data the check trigger state is called.

Several types of triggers can be supported and are key to knowing when an event of interest occurred, and then being able to save off the critical data. Triggers can be as simple as time (trigger every one hour) to as complicated as triggering on a pre-defined level of the power within a given spectral band. In the current architecture three types of triggering are supported: time, delta RPM, and forced. Delta RPM detects sudden changes with the running speed of a given machine. In order for Delta RPM to be affective one of the sensor inputs needs to be from a tachometer (sensor for measuring speed). Forced trigger is a special trigger initiated by an external user and not by the DAAN. Many times as analysts are reviewing the data coming from a DAAN or inspecting the machine, they might see or hear something interesting and want to capture an additional snapshot of data without having to wait for a pre-defined trigger (time, delta RPM, etc.) to occur. The analyst is able to issue a command directly to the DAAN instructing it to immediately save a pre-defined amount of data.

If none of the given trigger conditions are met the data acquisition node goes back and acquires another block of pre-trigger data and stores it in its internal buffer. If, however, a trigger condition is satisfied the entire pre-trigger buffer and an additional number of seconds (or blocks) of data, previously defined by the user, are saved to disk in a TDMS file.

Benefits

There are other architectures available to the user which could be implemented, but each has specific limitations not found in the store and forward architecture. For instance, one approach is to stream all of the data across the network and store the information on a local system (a PC). This architecture

presents a couple of problems. First, it requires a tremendous amount of hard drive space on the local machine.

When acquiring vibration data the volume of data being produced quickly becomes unmanageable and costly. Machines typically don't change quickly and the majority of data the user would be storing is repetitive, yet the analyst would have to crawl through a mountain of data trying to find anomalies (the classic needle in haystack problem). Finally, what happens if the network goes down, which at least for this author happens frequently? As long as the network is down data is not being collected, and if one believes in Murphy's Law then something critical is going to happen to the machine while the network is down.

Another approach is to continually store all of the data locally on the remote Data Acquisition Node. At least in this method if the network goes down all of the data is being captured at the DAAN and COULD be retrieved at a future date. However, there still is the problem of the large hard drive space needed and the unnecessary data being collected. This approach is amplified on embedded machines which typically have very small (relative to PCs or servers) hard drives. External drives on some embedded DAAN's could be added, but most are not rated to the harsh, industrial environments where wide temperature swings, high humidity, and often high vibration levels are common.

The Store and Forward architecture addresses all of these problems. It doesn't have a dependency on an external network and it only collects data when it sees something interesting or at periodic times; thus reducing the number of files the analyst has to review.

OVERALL ARCHITECTURE

Figure 13 illustrates the LabVIEW implementation of the embedded monitoring system architecture. This architecture follows a standard design pattern in several National Instruments control and monitoring applications.

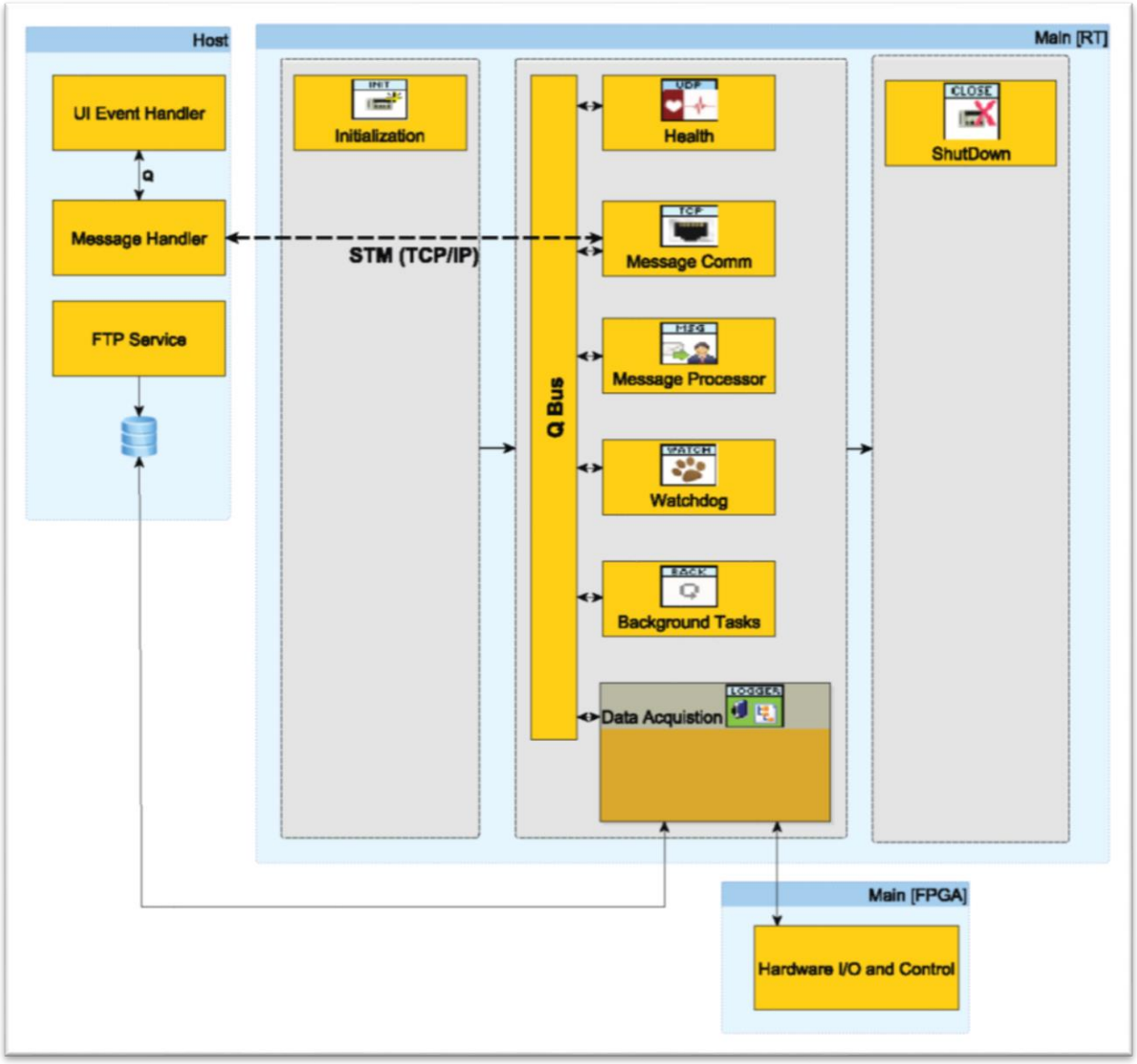


Figure 13 Embedded Monitoring System Diagram

Vibration Logger Reference Architecture and Waveform Reference API

It is important to note that this example has its roots based off of the Vibration Logger Reference Architecture and the Waveform Reference API. The Waveform Reference API shown in Figure 14 is an API created for acquisition of waveform data from various CompactRIO modules. The API has both an FPGA component and a Real-Time Component.

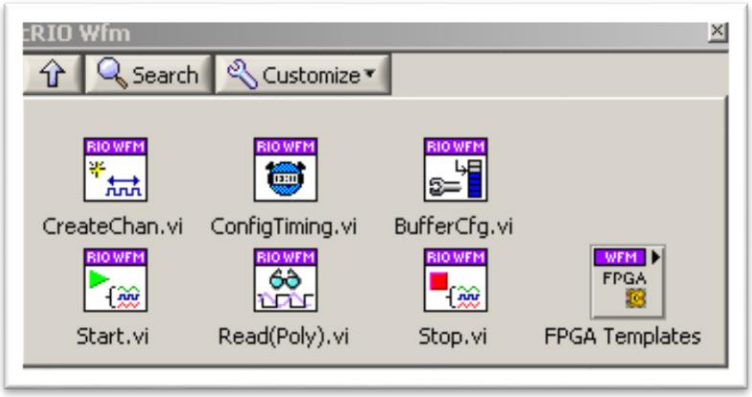


Figure 14 Waveform Reference API

The Vibration Logger Reference Architecture is a simple example which builds on top of the Waveform API and presents the various pieces necessary to log waveform data from a CompactRIO controller. This includes host code, messaging architecture, and logging strategies. Many of the concepts in the Vibration Logger are similar to what is presented in this document and this document should be seen as an extension describing a real world application. This document also covers in greater detail the concepts behind some of the technology decisions, architecture, and implementation details.

Components

Host

The host serves three primary functions. First, it provides a user interface application to the embedded hardware or DAAN. The user interface at the highest level either generates a set of commands to send to the Data Acquisition and Analysis Node (DAAN) or receives commands/information coming from the DAAN and processes them appropriately. Primarily this is accomplished via a technology termed Simple TCP/IP Messaging or STM. STM will be specifically discussed in the section outlining the message communication thread, but quite simply it is a methodology for sending and receiving commands between the host application and the DAAN.

Second, the host user interface application is responsible for retrieving the TDMS files off of the distributed DAAN's. This is accomplished via FTP.

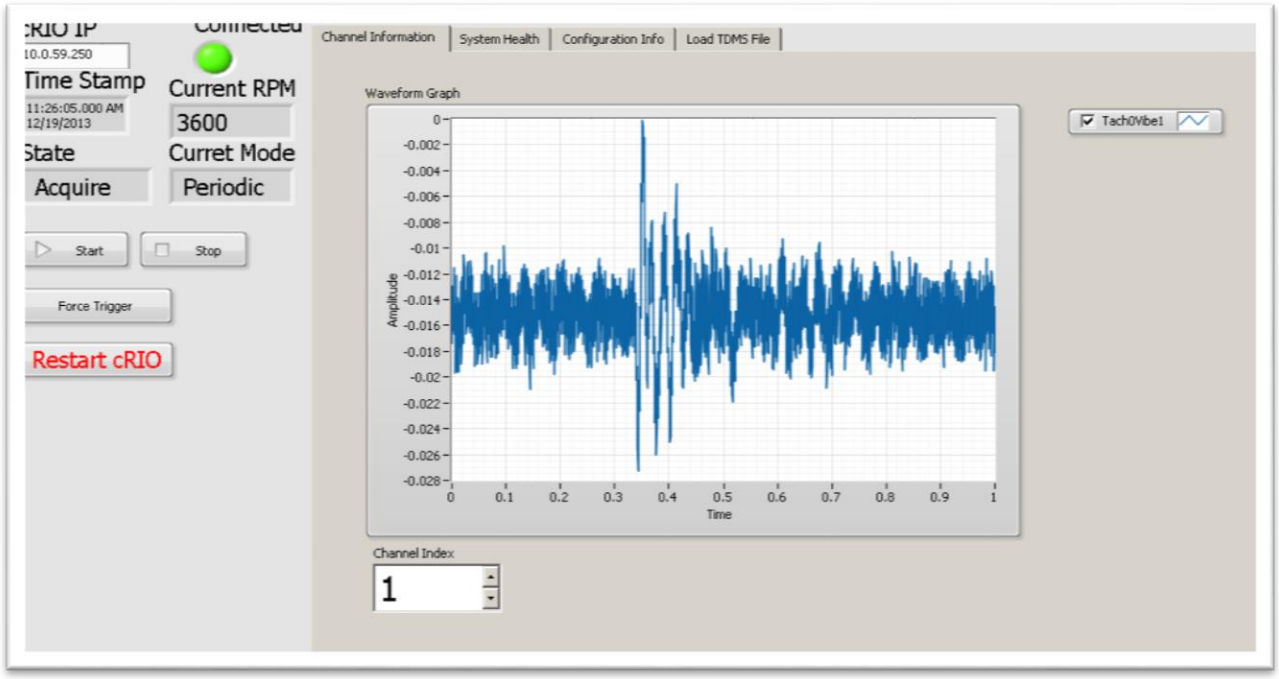


Figure 15 Host Monitoring Application

The final host piece is a separate utility for configuring all of the DAAN's. Each distributed node needs to operate in the absence of a network connection. That requirement enforces a static configuration file to reside on each distributed node. Quite simply, the configuration utility creates the configuration file and can ftp the given file to the remote node. The static configuration file is important, because if the remote node were to lose connection to the host server and a reboot of the DAAN happened for some unexpected reason if there wasn't a locally stored static configuration the DAAN wouldn't know how to operate after re-initialization.

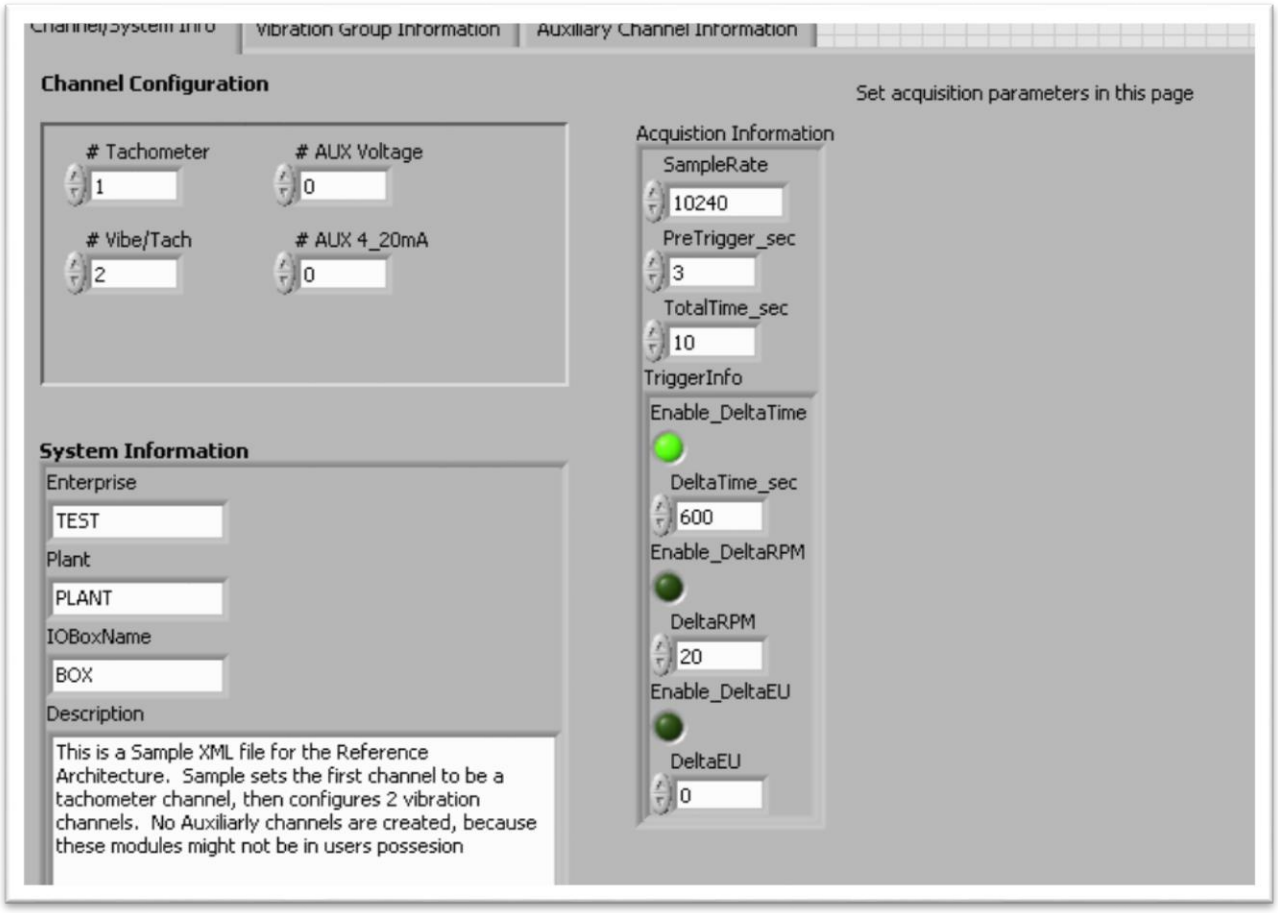


Figure 16 Remote Configuration Tool

Real-Time (RT)

The Real-Time or RT software is often referred to as the firmware. This firmware resides on the distributed DAAN and is the real work horse in the system. It is comprised of several software threads (which will be discussed later) each performing a specific operation such as communication, acquisition, health monitoring, etc.

A RT Operating System (RTOS) is able to reliably execute programs with specific timing requirements, which is important for projects. The key component needed to build a Real-Time system is a RTOS.

Precise timing

For many applications running a measurement or control program on a standard PC with a general purpose OS installed (such as Windows) is unacceptable. At any time, the OS might delay execution of a user program for many reasons: running a virus scan, updating graphics, performing system background tasks, and so on. For programs needing to execute at a certain rate without interruption (for example, a cruise control system) this delay may cause system failure.

Note this behavior is by design—general-purpose OS'es are optimized to run many processes and applications at the same time and provide other features like rich user interface graphics. In contrast,

real-time OS'es are designed to run a single program with precise timing. Real-time systems make it easier to implement the following:

- Perform tasks within a guaranteed worst-case timeframe
- Carefully prioritize different sections of the application
- Run loops with nearly the same timing on each iteration (typically within microseconds)
- Detect if a loop missed its timing goal

Reliability

In addition to offering precise timing, RTOS'es can be set up to run reliably for days, months, or years without stopping. This is important not only for engineers building systems needing 24/7 operation, but also for applications where downtime is costly. A "watchdog" feature is also typically included in real-time systems to automatically restart an entire computer if the user program stops running. Furthermore, hardware used in real-time systems is often designed to be rugged and sustain harsh conditions for long periods.

FPGA

The final component in the system is the FPGA application. The FPGA is directly connected to the I/O for high-performance access to the I/O circuitry of each module and timing, triggering, and synchronization. Because each module is connected directly to the FPGA rather than through a bus, the application experiences almost no control latency for system response compared to other controller architectures.

By default, the FPGA automatically communicates with I/O modules and provides deterministic I/O to the real-time processor. Out of the box, the FPGA enables programs on the real-time controller to access I/O with less than 500 ns of jitter between loops. The FPGA can also be directly programmed to further customize the system. Because of the FPGA speed, the National Instruments CompactRIO platform is frequently used to create systems incorporating high-speed buffered I/O, fast control loops, and/or custom signal filtering.

For instance, using the FPGA, a single CompactRIO chassis can execute more than 20 analog proportional integral derivative (PID) control loops simultaneously at a rate of 100 kHz. Additionally, because the FPGA runs all code in hardware, it provides the highest reliability and determinism, which is ideal for hardware-based interlocks, custom timing and triggering, or eliminating the custom circuitry normally required with nonstandard sensors and buses.

In the distributed monitoring application the FPGA creates a high-speed buffered interface between the I/O modules and the real-time operating system. The particular implementation of the high-speed buffered interface is through the use of DMA FIFOs.

DMA FIFOs

If it is necessary to stream large amounts of data between the FPGA and real-time processor, or if the application requires a small buffer for command- or message based-communication, consider using DMA FIFOs for data transfer. DMA does not involve the host processor when reading data off the FPGA;

therefore, it is the fastest method for transferring large amounts of data between the FPGA target and the host.

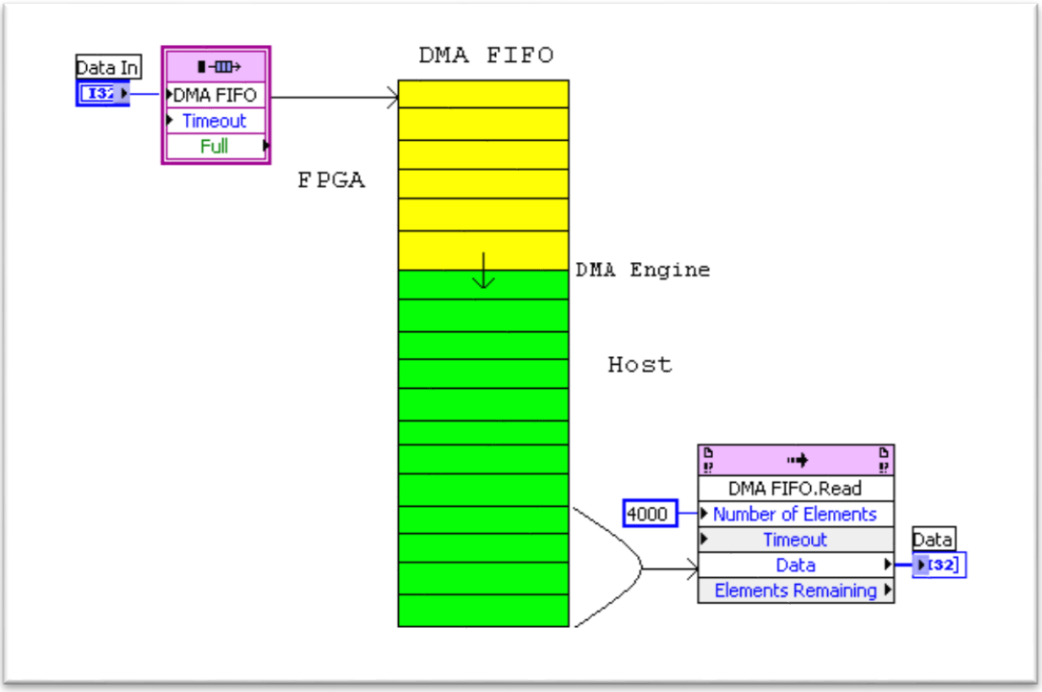


Figure 17 DMA uses FPGA memory to store data and then transfer it at a high speed to host processor memory with very little processor involvement.

The following list highlights the benefits of using DMA communication to transfer data between an FPGA target and a host computer:

- Frees the host processor to perform other calculations during data transfer
- Reduces the use of front panel controls and indicators, which helps save FPGA resources, especially when transferring arrays of data
- Automatically synchronizes data transfers between the host and the FPGA target

Threads

The real time portion of the embedded firmware is the most complicated portion of the software. For reference, the real time portion of the architecture is highlighted again in Figure 18. Each block in the ***middle section*** is an independent software "thread". A thread is simply an independently running piece of code designed to execute a particular function.

In this implementation each thread is designed based on the LabVIEW state machine design pattern.

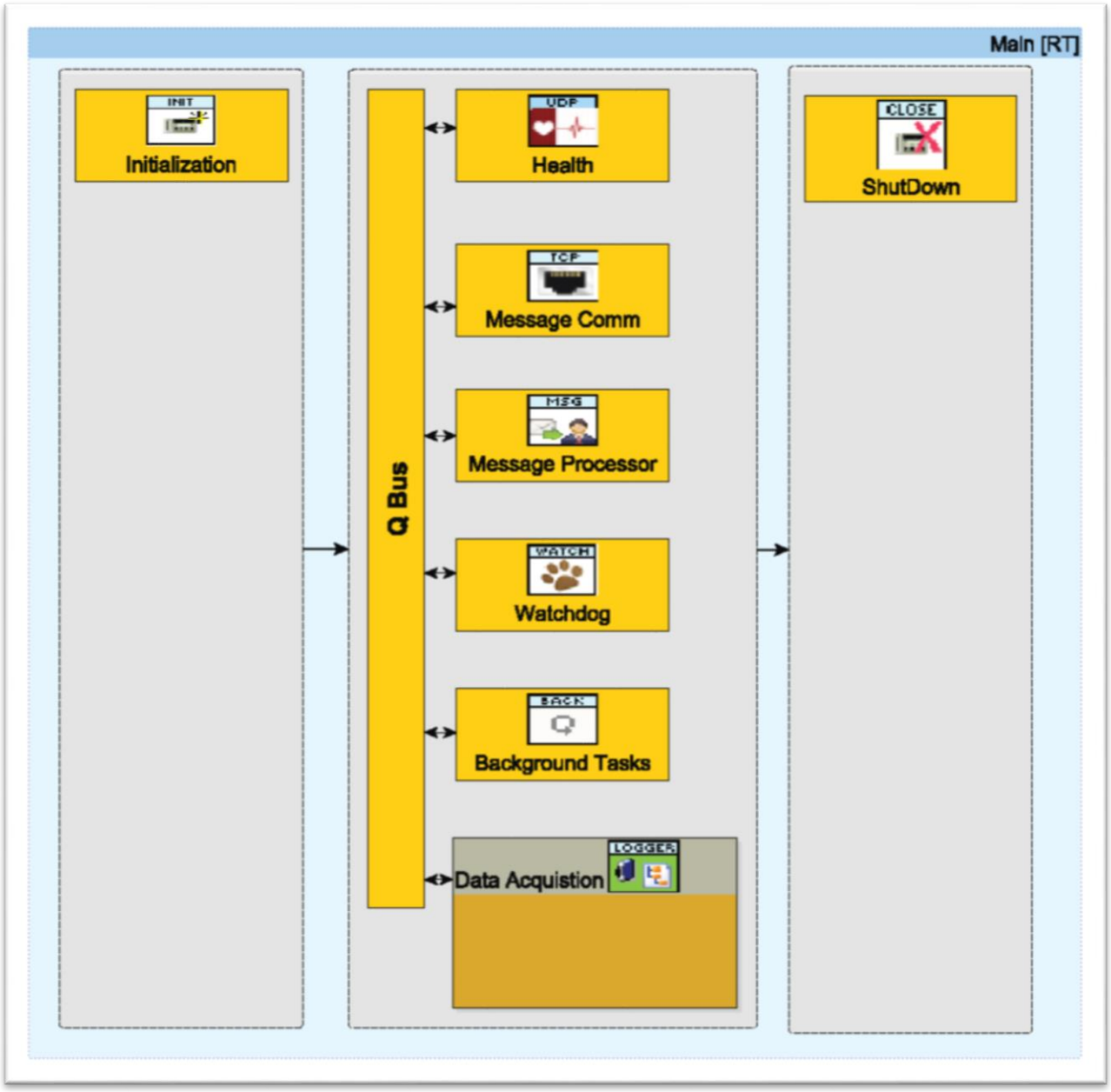


Figure 18 Real Time firmware diagram

State Machine

A state machine is a common and useful design pattern. Use a state machine to implement any algorithm that can be explicitly described by a state diagram or flowchart. A state machine usually illustrates a moderately complex decision-making algorithm, such as a diagnostic routine or a process monitor. To learn the basics of the state machine architecture in LabVIEW, see the NI Developer Zone document Application Design Patterns: State Machine.

A state machine design pattern may contain both periodic and event-based synchronization. A simple example of a state machine is a bioreactor recipe engine, which is described by the flowchart in Figure 19.

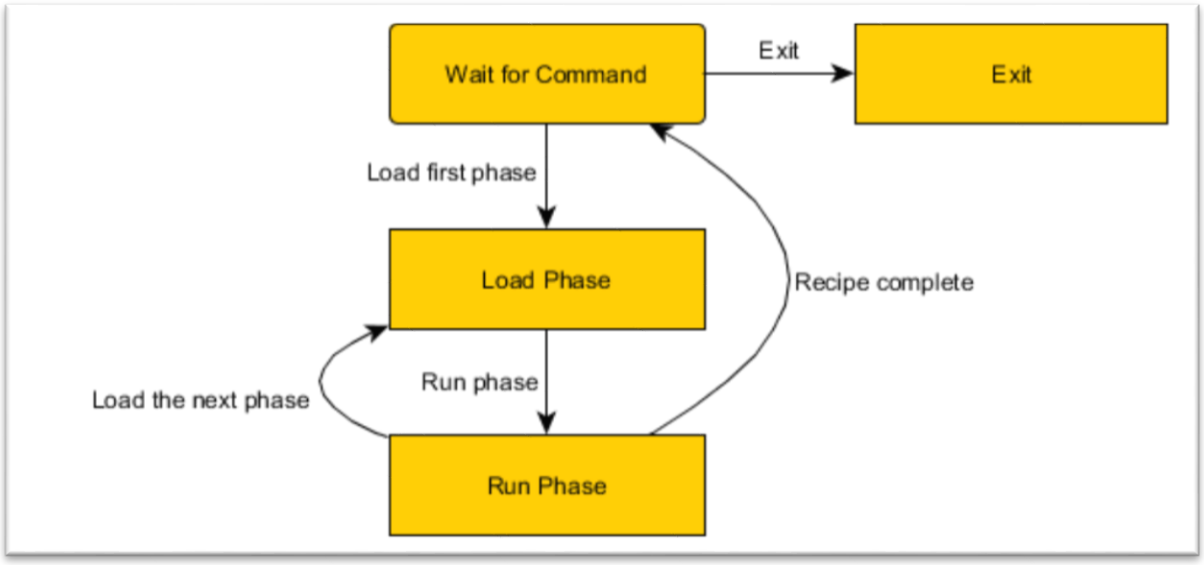


Figure 19 Flowchart Used for a Bioreactor Recipe Engine

The state machine starts in the Wait for Cmd state. This state waits on a command from the command parser loop. It uses a queue with a default timeout (-1) to block until a command is received.

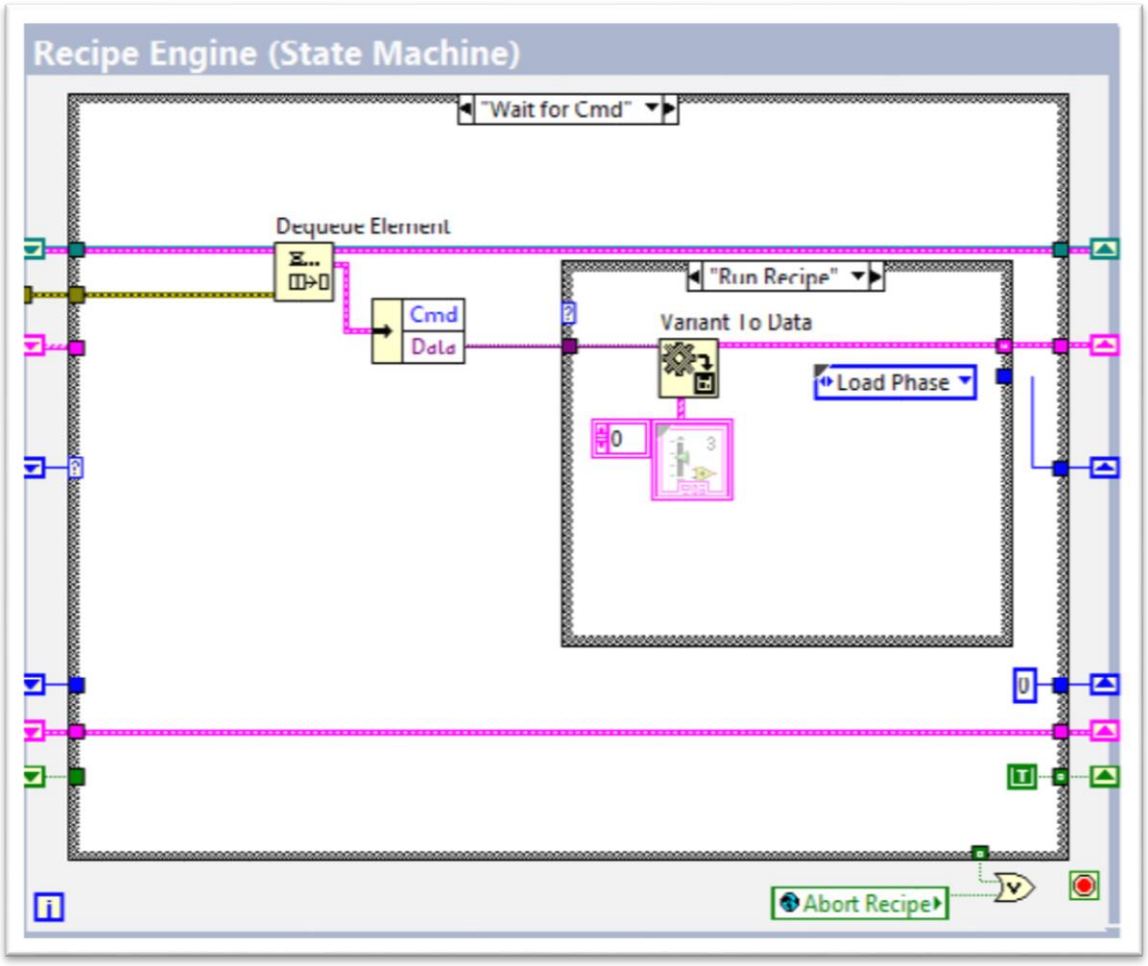


Figure 20 Example of a State with Event-Driven Synchronization

The Load Phase state does not involve any type of synchronization. It simply executes a task and moves to the next state when the task is complete. The Load Phase state checks how many phases are in the recipe and loads one phase each time it is called.

The Run Phase state features periodic synchronization. It uses a Sequence Structure with a Wait until Next ms Multiple function to control the timing. This state holds each tag at the value defined by the recipe until the specified time has elapsed. When the time has elapsed, the recipe engine loads the next phase. If there are no more phases to execute, it returns to the Wait for Cmd state.

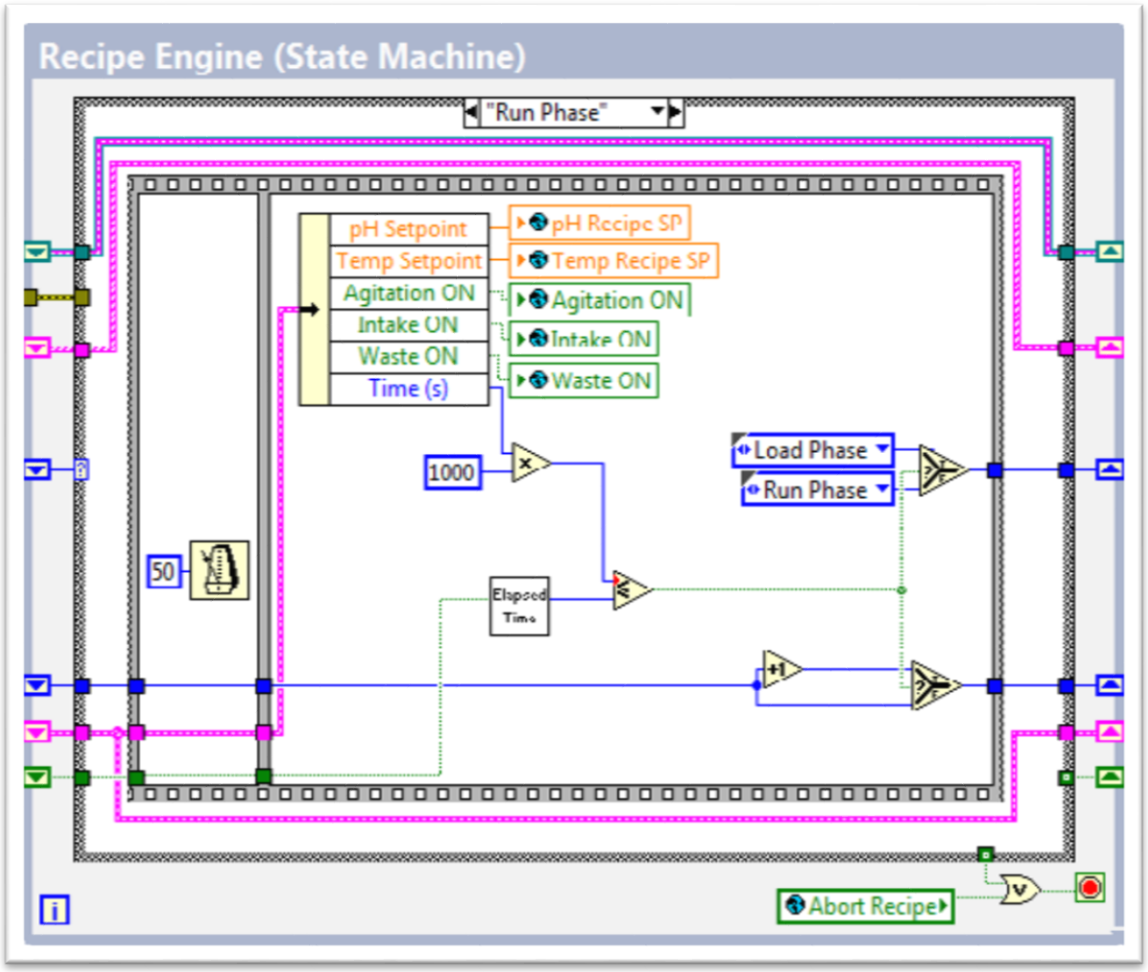


Figure 21 Example of a State with Periodic Synchronization

While the state machine implementation in the distributed monitoring application is more complicated, it utilizes the same high level techniques described here. One important difference is the mechanism in which the messages are read. Figure 22 is the block diagram for the TCP Send Thread in the included application.

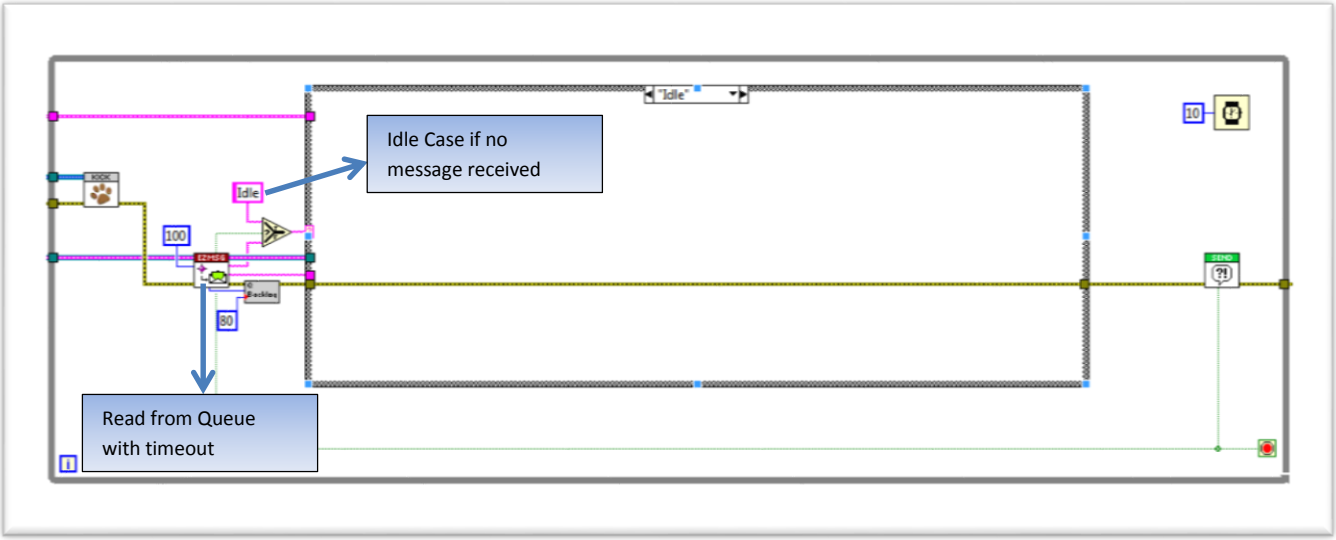


Figure 22 Non-blocking State Machine

Unlike the simple state machine, because this state machine is intended to be an independently running thread it can't wait until a command is received. The code must run independently. An example of the importance of this, is if this thread blocks (i.e. waits until a message is put in the queue) then the watchdog code (will be explained in a later section) for this application won't execute; and the system will reboot. Therefore, a timeout on the queue is necessary. In this example, if the queue read doesn't receive a command within 100ms then the idle case is called. If a message was, in fact, in the queue then the appropriate case within the state machine is called.

Interprocess Communication between threads

While each thread can be independently run, often the states within a thread are dependent on various states within the entire system. Therefore, there needs to be some mechanism to communicate between individual threads while maintaining their independence. A great example of this is a global stop. If an error occurs within the system all of the threads need to be notified of the error, and they should perform some clean-up within their thread (a possible state in the state machine) before finally exiting (a required state within the state machine). This communication between threads is referred to as interprocess communication.

There are numerous ways to accomplish this communication. Global variables are a simple, yet naïve way of doing interprocess communication. Asynchronous Message Communication (AMC) is another, quite popular, LabVIEW design pattern for implementing interprocess communication. AMC at its core is implemented on top of LabVIEW queues where each queue has a message (a state) and a data value. Each thread then becomes a unique queue and data is transferred to an individual thread by callings its unique queue.

For our particular application we utilized a library called EZ Msg. EZ Msg is a simpler, queue based implementation of AMC. For this particular application, all of the functionality of AMC was unnecessary

and therefore a simpler library was created. Again, at its core EZ Msg is based on top of a queue based mechanism for passing data between threads. This queue based method of sending data between threads is represented by the Q bus in Figure 18. The Q bus is quite simply the unique queues created for each thread where data is transferred by calling the unique queue.

APPLICATION SPECIFIC THREADS

A discussion of each thread is important as they each play a critical role in creating an embedded distributed monitoring application.

System Health 

Troubleshooting and debugging system errors or issues while running on a headless, embedded application can be quite challenging. The System Health thread provides a methodology for communicating system information and any debugging messages to the host where the information can be logged and/or analyzed.

It was discussed previously in the technologies section UDP is utilized to communicate non-critical, though not necessarily unimportant, information to a central server. Parameters like CPU utilization, memory utilization, and remaining drive space are all information the developer (or end user) might be interested in knowing while the remote node is running.

The implementation in this application uses a protocol known as syslog. Syslog is a protocol for logging information built on top of UDP. It was originally developed as part of the Sendmail project. As its popularity gained the protocol was implemented in several networking and computer peripherals including as printers, routers, switches, etc. There is a LabVIEW component freely available which implements the Syslog protocol in G. Utilizing the Syslog protocol the developer can use any number of third party tools available for receiving, displaying, filtering, and logging Syslog message while not requiring the developer to create their own unique viewer. An example output from the DAAN captured in Syslog Watcher is shown in Figure 23.

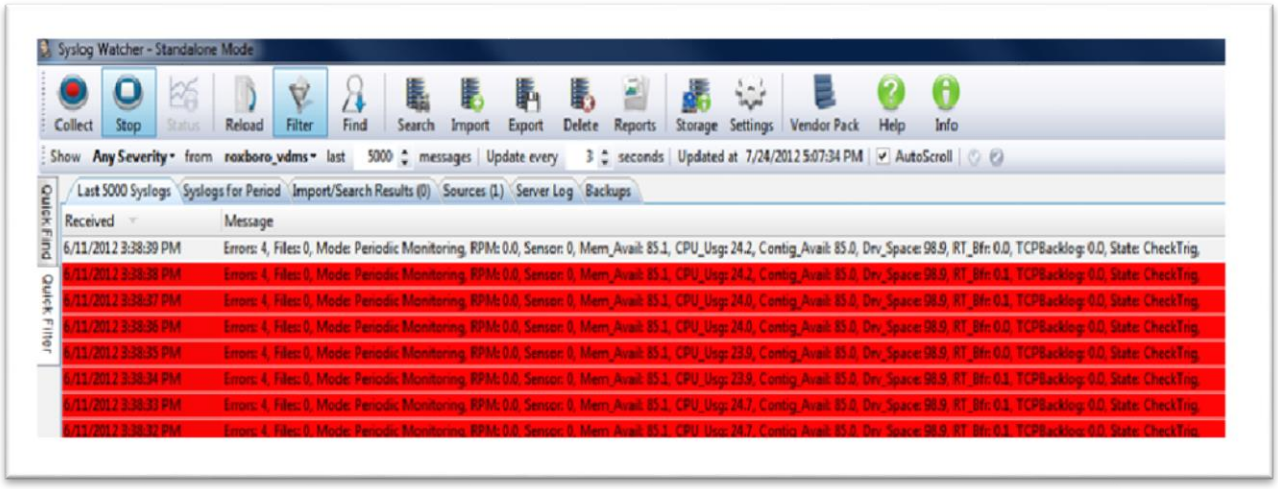


Figure 23 Syslog Watcher Capture of Syslog Messages from DAAN

How are the Syslog messages generated? Again we go back to the concept of the System Health "thread" which is built on top of the LabVIEW state machine design pattern. Figure 24 shows the block diagram of the system health state machine.

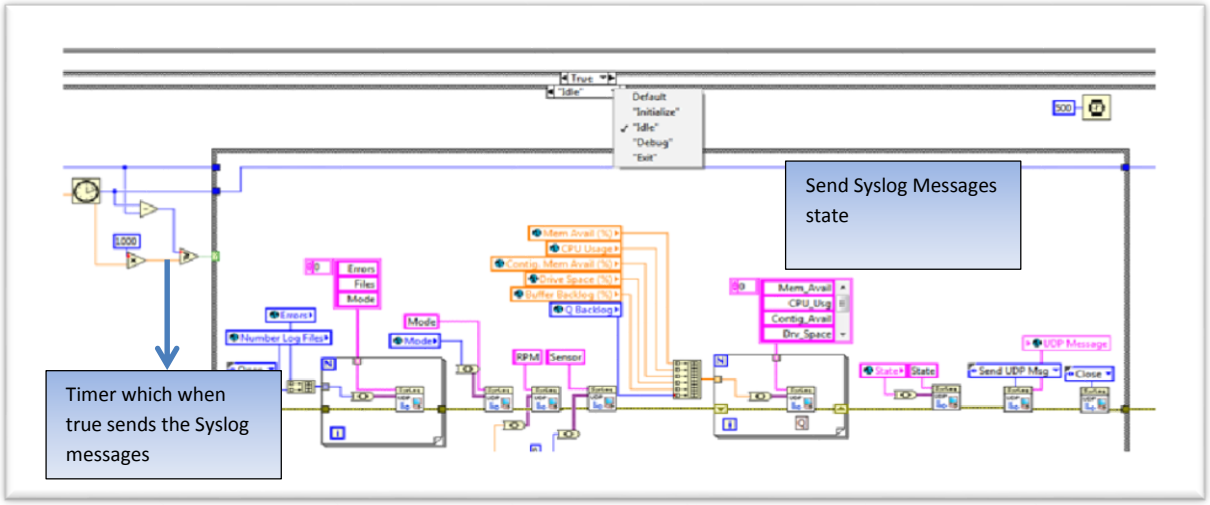


Figure 24 Block Diagram of System Health State Machine

Notice in this diagram, as opposed to the TCP Send thread shown in Figure 22, some work is done in the idle case, but not all the time. The loop rate for this particular thread is 500ms. In almost all cases it makes no sense to send status or health information every 500ms because information is not changing that fast. Therefore, we have implemented a system where the user can set the update frequency read from a configuration file and the timer code keeps track of elapsed time. When the timer meets the user specified frequency, the True state in the "Idle" case is called and the syslog messages are sent to the host. If the timer is not met the false state is called and no work is done.

Notice the "Debug" state in the list of possible states in Figure 24. Again, utilizing Syslog our code is implemented to provide debug information. This is a fairly complex architecture involving several different threads all running in parallel. That is a pretty difficult thing to debug on a full blown windows

system where breakpoints, probes, and even a monitor are available to us. While all of these tools are available to the developer thanks to the rich capabilities of LabVIEW RT, many times there are run-time only issues which must be solved while the application is in its final, deployed, and headless state. How can we accomplish this? Simple.

By creating a single VI referenced in Figure 25 which specifies three things we can utilize it throughout the code. First, the VI gets a reference to the System Health thread. Second, it calls the Debug state in the System Health thread, and finally it sends the debug information.

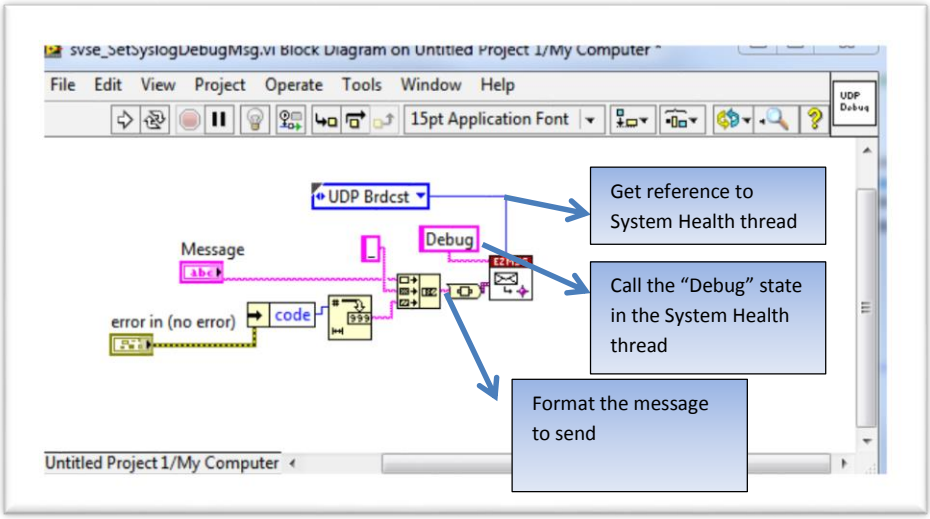


Figure 25 Debug VI

When this VI is called anywhere in our code it sends a message via the interprocess communication (i.e. a queue) to the System Health thread, which then forwards on the message to the host system where the data can be logged and analyzed. The Syslog protocol also allows the user to specify different message "types" which can then be filtered by most third party viewers. This allows the developer to separate the true system health messages from the debug messages.

Message Communication

Before diving into the specifics of the message communication thread in this application it is important to familiarize oneself with the various best practices discussed in Chapter 4 of the Compact RIO Developer's guide.

The architecture chosen for this thread is a command sender/parser architecture built on top of the Simple TCP/IP Messaging (STM) protocol.

Simple TCP/IP Messaging (STM)

STM is a networking protocol NI systems engineers designed based on TCP/IP. It is recommended for sending commands or messages across the network if communicating to a third-party API or the

application requires a standard protocol. It makes data manipulation more manageable through the use of formatted packets, and improves throughput by minimizing the transmission of repetitive data.

Metadata

Metadata is implemented as an array of clusters. Each array element contains the data properties necessary to package and decode one variable value. Even if only the Name property is defined, a cluster to customize the STM by adding meta properties (such as data type) can be used according to application requirements. The metadata cluster is a typedef, so adding properties will not break code.

Figure 26 shows an example of the metadata cluster configured for two variables: Iteration and RandomData

Figure 26 Metadata Array of Strings

Before each data variable is transmitted, a packet is created that includes fields for the data size, the metadata ID, and the data itself.

Figure 27 Packet Format

The metadata ID field is populated with the index of the metadata array element corresponding to the data variable. The receiving host uses the metadata ID to index the metadata array to get the properties of the message data.

API

The STM API is shown in Figure 28. For basic operation it consists of a Read VI and a Write VI. The API also features two supplemental VIs to help with the transmission of the metadata, but their use is not mandatory. Each of the main VIs is polymorphic, meaning they can be used with different transport layers. This document discusses STM communication based on the TCP/IP protocol, but STM also works with UDP and serial as transport layers. The API for each layer is similar.

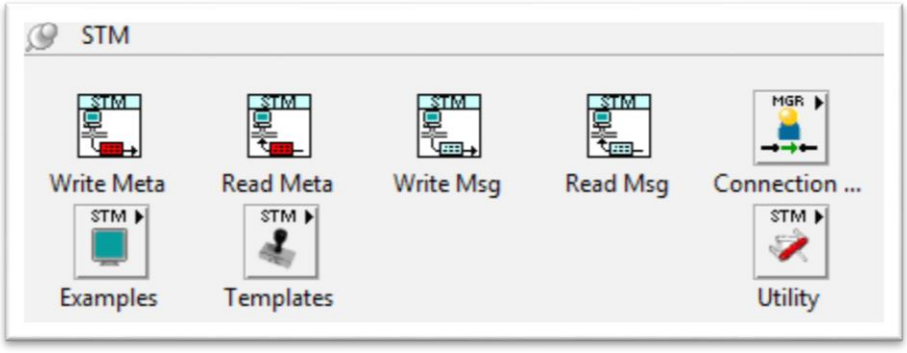


Figure 28. STM Functions

STM Write Meta

Send metadata information to a remote host with this VI. For correct message interpretation, the metadata must be consistent on both the receiving and sending side. Instead of maintaining copies of the data on each host, maintain the metadata on the server and use this VI to send it to clients as they connect.

STM Read Meta

Receive metadata information from a remote computer with this VI. It reads and unpacks the metadata array, which can then be passed to the read and write VIs.

STM Write Msg

Send any type of data to a remote host with this VI. It creates a packet based on the data, the data name, and metadata information. When this VI is called, it retrieves the index of the variable specified by the name in the metadata array. It then assembles the message packet and sends it to the remote host via TCP/IP using the connection ID.

The data must be in string format for transmission. Use the Flatten to String function to convert the message data to a string.

STM Read Msg

Receive any type of data from a remote host with this VI. It reads and unpacks the metadata index and flattened string data. It looks up the meta element and returns it along with the data string. The application can then convert the flattened data to the message data type using the name or other meta properties as a guide. In the example in Figure 29, the variable named “RandomData” is always converted to an “Array of Doubles” data type.

This VI is usually used inside a loop. Since there is no guarantee data will arrive at a given time, use the “timeout” parameter to allow the loop to run periodically and use the “Timed Out?” indicator to know whether to process the returned values.

Example

Figure 29 shows a basic example of STM being used to send RandomData and Iteration data across a network. The server VI is shown in Figure 29, and the client VI is shown in Figure 30. Notice the server VI sends the metadata, implemented as an array of strings, to a remote host as soon as a connection is established. The example writes two values: the iteration counter and an array of doubles. The metadata contains the description for these two variables.

Wire only the variable name to the STM Write Message VI, which takes care of creating and sending the message packet back. Because of this abstraction, data can be sent by name while hiding the underlying complexity of the TCP/IP protocol.

Also note the application flattens the data to a string before sending. For simple data types it is possible to use a typecast which is slightly faster than the Flatten to String function. However, the Flatten to String function also works with complex data types such as clusters and waveforms.

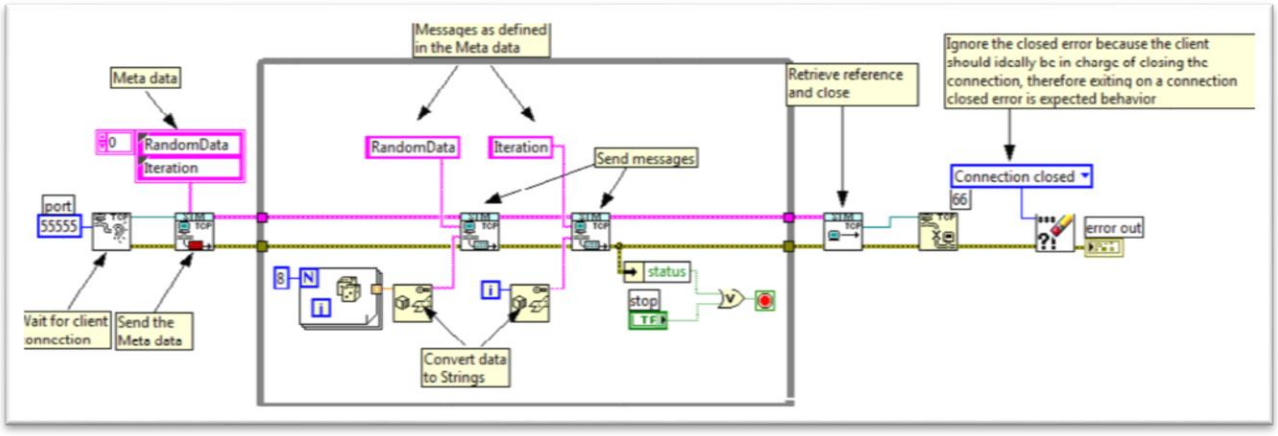


Figure 29 RT Target VI Using STM Communication to Send Data to a Client

The protocol can be customized and expanded to fit the specific needs of the application requirements. As more variables are added, simply add an entry to the metadata array and a corresponding STM Write Message VI for the new variable.

Receiving data is also simple. The design pattern shown in Figure 30 waits for the metadata when the connection is established with the server. It then uses the STM Read Message VI to wait for incoming messages. When it receives a message, it converts the data and assigns it to a local value according to the metadata name.

The Case structure driven by the data name provides an expandable method for handling data conversion. As variables are added, simply create a case with the code to convert the variable to the right type and send it to the right destination.

Note an outer Case structure handles timeout events.

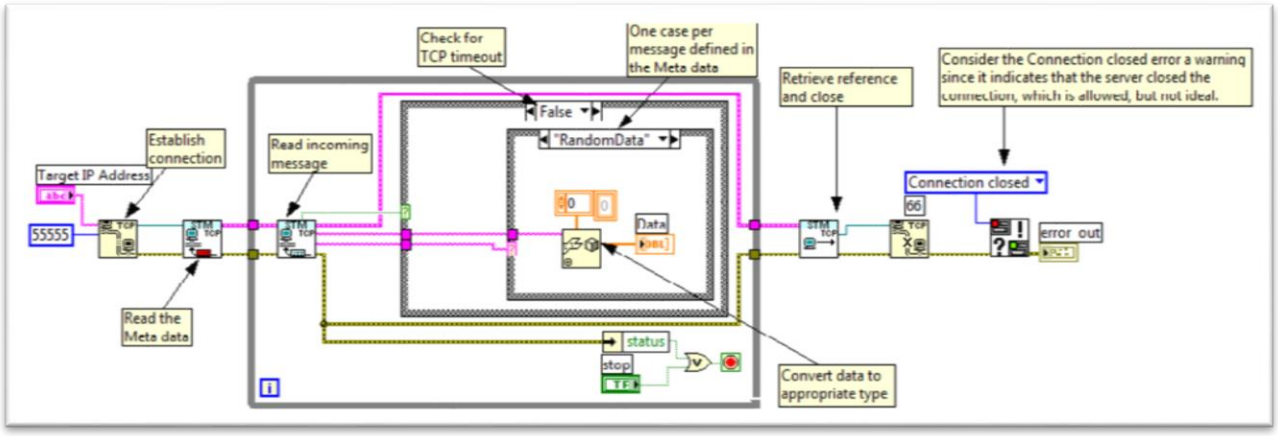


Figure 30 Host VI Using STM Communication to Read Incoming Data

One advantage of this design pattern is it centralizes the code receiving data and assigns it to local values.

Note: Since the client has no knowledge of the metadata until run time, be sure the application handles all possible incoming variables. It is a good practice to implement a "default" case to trap any "unknown" variables as an error condition.

Message Communication Details

The message communication block diagram is divided into two main processes: the TCP Send Process and the TCP Receive Process. Each of these processes are based on the STM pattern described above.

TCP Send Process 

As the name implies the TCP Send Process is responsible for sending messages from the embedded system **to** any connected clients. When the "TCPSend" command is received from any messages which have put their information into the TCPSend thread the first thing is to check if a valid connection exists.

If a valid connection exists then the MetaData, which in this case is the message name (GetSensorFaults, GetVersionInfo, GetSystemErrors, etc.), and the value or data associated with that particular message are sent via the STMWriteMessage.VI.

TCP Receive Process 

The TCP Receive Process is a little more complex than then TCP Send Process, but in general this thread is responsible for receiving any messages which come **from** any connected client and passing the information from the message to the appropriate thread/process on the embedded device.

This thread in the idle condition first checks if there is a valid connection to any clients. If no clients are connected the thread waits a specified amount of time for a client to connect. If a client connects then its ID is added to a lookup table. If after the specified time a connection has not been established, the TCP/IP VIs will timeout with Error 56. It is important to note even though an error occurs this is an acceptable error, because it simply means no client has connected; but this shouldn't cause the embedded system to stop functioning. The error handling strategy is to acknowledge the error and then clear it.

If a connection is valid the application code checks to see if there are any messages using the STM Read Message.vi. If the VI times out (indicating no messages) then again a TCP/IP connection timed out error (error 56) is generated. As in the previous paragraph, this is an acceptable error because it simply means the client wasn't sending any messages at the time and the embedded system should not stop functioning.

If a client does send a message then that message is "looked up" in the case structure. It is important all messages be defined and any new messages added must be included in the look-up table else the code doesn't understand how to parse the given message.

Example—Set System Time

Let's look at a couple of examples which should help illustrate better the Communication Thread. In example one (Figure 31), the client (Windows) application sends a request/message to change the system time on the embedded system. The TCP Receive Message has already established a connection to the client and is in the idle case waiting for a message to show up. The "SetTime" message is received and the "SetTime" case is executed.

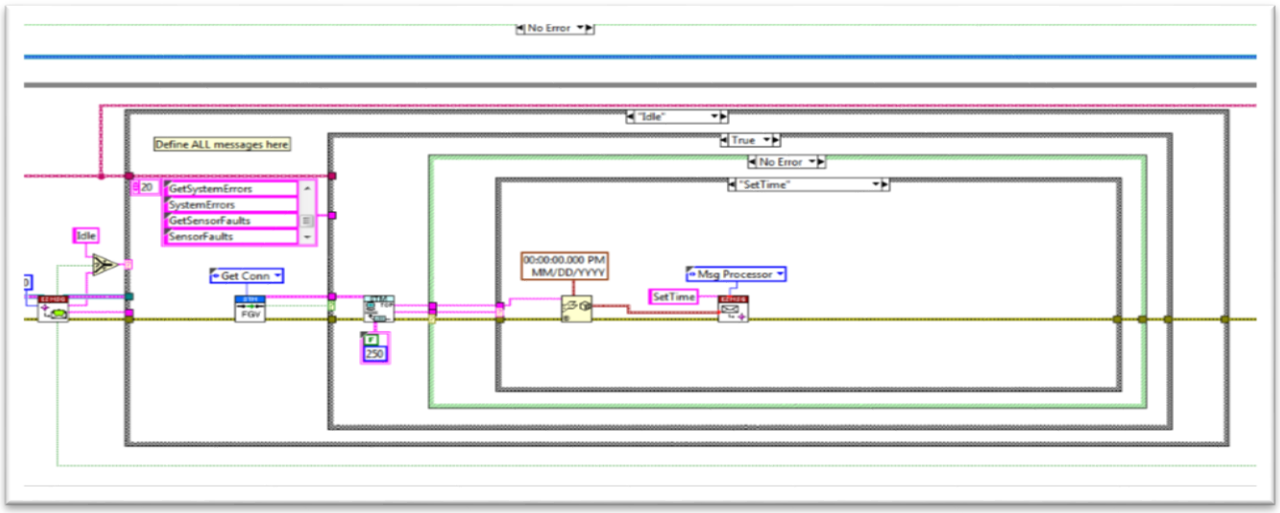


Figure 31 Set Time Example

The timestamp the client wanted to set the system to is parsed from the data string of the STM package and converted to a timestamp. Note: all messages are sent and received as flattened strings. Next, the Msg Processor thread is called with the "SetTime" command and the data.

The Message Processor thread will be discussed next, but in general it is used to handle miscellaneous commands from other threads within the application. While this seems overly complex, the design decision was made that the TCP Send and Receive processes would ONLY be responsible for the parsing of messages and wouldn't do any of the work. It is the Message Processor thread the actual work to set the system time is performed.

Example—Get Version Info

In this example the client (Windows) application requests information about the current firmware version running on the embedded system. Just like in example one, a connection to the client is assumed. Now the "GetVersionInfo" message is received by the client and the code looks-up in the case structure what should be done. In this case the global version information is read and the "VersionInfo" command is sent to the TCP Send Process with the corresponding information. Note: the EZMsg VI pictured in Figure 32 is simply a wrapper with added functionality for calling the TCP Send Process.

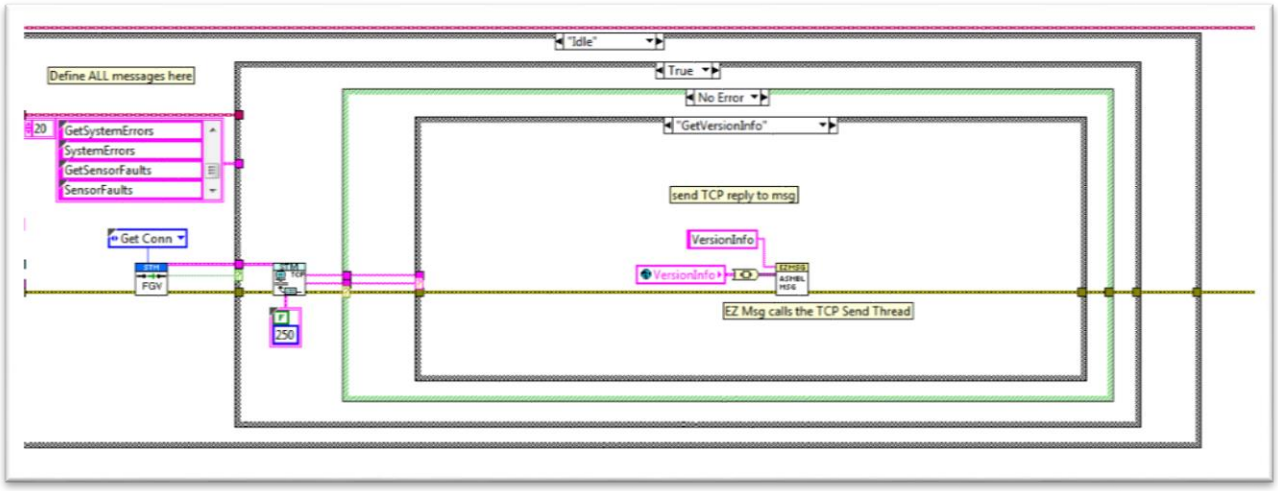


Figure 32 Get Version Info Example

Remember there is a separation between the TCP Receive Process and the TCP Send Process. Thus far in this example, we have been examining the TCP Receive Process. Now, let's look at the TCP Send Process in Figure 33.

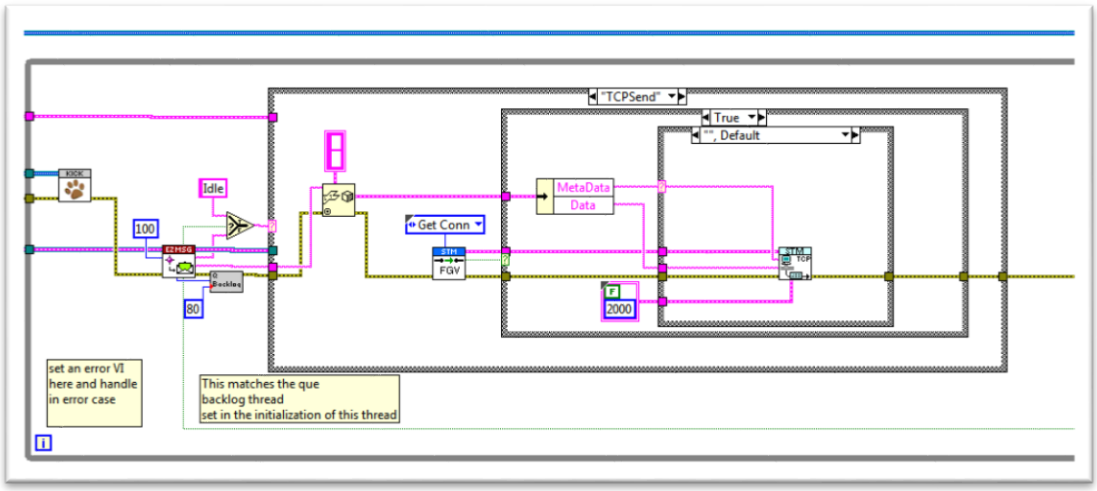


Figure 33 TCP Send Process

Remember this was the less complicated of the two messaging processes. In this example the TCP Receive Process sent a message to the TCP Send Process for "VersionInfo". We assume a valid connection is made and the MetaData (or command), which equals "VersionInfo", and its corresponding Data, which equals the version information string, are sent to the client application.

What isn't shown in both of these examples is the client example. Quite simply the client app is built on a very similar TCP Send and Receive architecture. Therefore, in the client application the TCP Receive process gets the "VersionInfo" message which it has subscribed to, parses the version information string, and displays the information to the user.

Message Processor 

The message processor is responsible for taking commands/messages from various threads and doing work. As previously highlighted, one utilization is to take the messages from the TCP Send Process and perform the actual implementation. For instance, going back to our first example in the Message Communications Example where the user was trying to set the System Time; a message was set in the TCP Send Process to "SetTime" and the client requested time was passed as data in the form of a flattened string. Figure 34 shows what happens to the message in the Message Processor. The flattened string is unflattened, converted to a timestamp, and then sent to the system function responsible for updating the OS timestamp.

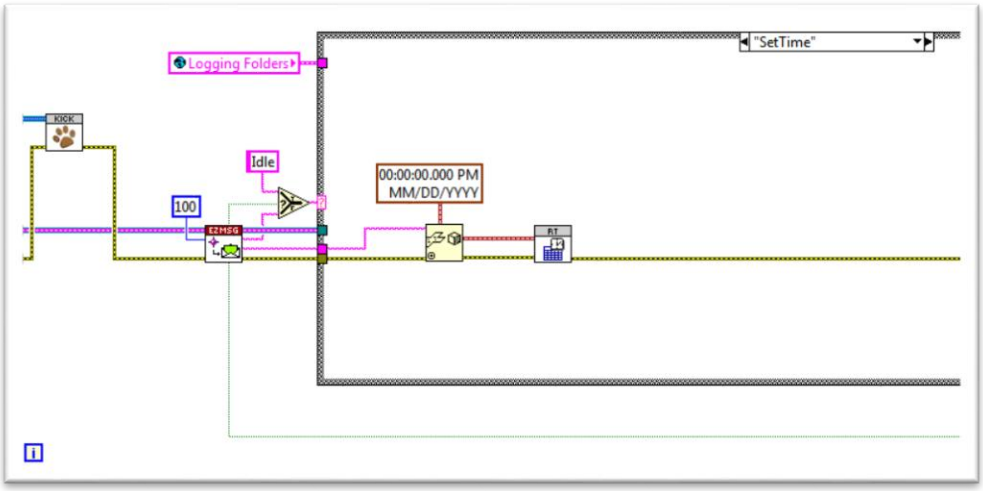


Figure 34 Setting System Time in Message Processing Thread

In an embedded system with a single processor this is a method for offsetting where action occurs so higher priority threads can run more important tasks. A good example of this is another command in this same VI called "Manage Drive". Manage drive is one of the states in the data acquisition state machine. Its purpose is to make sure there is enough hard drive space to save the next block of data. While the data acquisition loop initiates the command, it is more important for the data acquisition loop to not block waiting for this expensive operation to occur. Instead, the data acquisition loop can continue doing the actual acquisition, which is the most critical function of the entire embedded system. The message processor, however, can get around to this function whenever there are available CPU resources available.

Background 

The background's purpose is similar to the message processor. It is a thread which can do background tasks while giving priority to higher level tasks. The difference between the background and message processor thread is the message processor receives commands from various threads within the

application while the background runs tasks at set intervals. Things like writing to log files, updating system information, updating system status lights, etc. are all examples of housecleaning actions which need to occur on a regular basis, but aren't high priority.

One exception to this is the exit task which does take a command from any number of threads. The exit task could have just as easily been implemented in the Message Processor, but was not because in some simpler applications the Message Processor may not be necessary. In almost all applications, however, the background thread is necessary and thus rather than implementing the exit case depending on whether the message processor was available or not the decision was made to just implement the Exit case in a thread which is almost guaranteed to be present.

Figure 35 shows the block diagram of the Background Process.vi. Like all the other threads discussed in theis application they are based on a state machine. The interval time for each action is defined at the beginning of the loop. The CheckTimers case is similar to the Idle case implemented in most of the other threads. It simply checks how much time has expired since the last time the loop has executed and if one of the actions times has expired CheckTimer calls that specific case, else call the sleep case. Sleep simply waits for a short period of time and then goes back to the "CheckTimers" state.

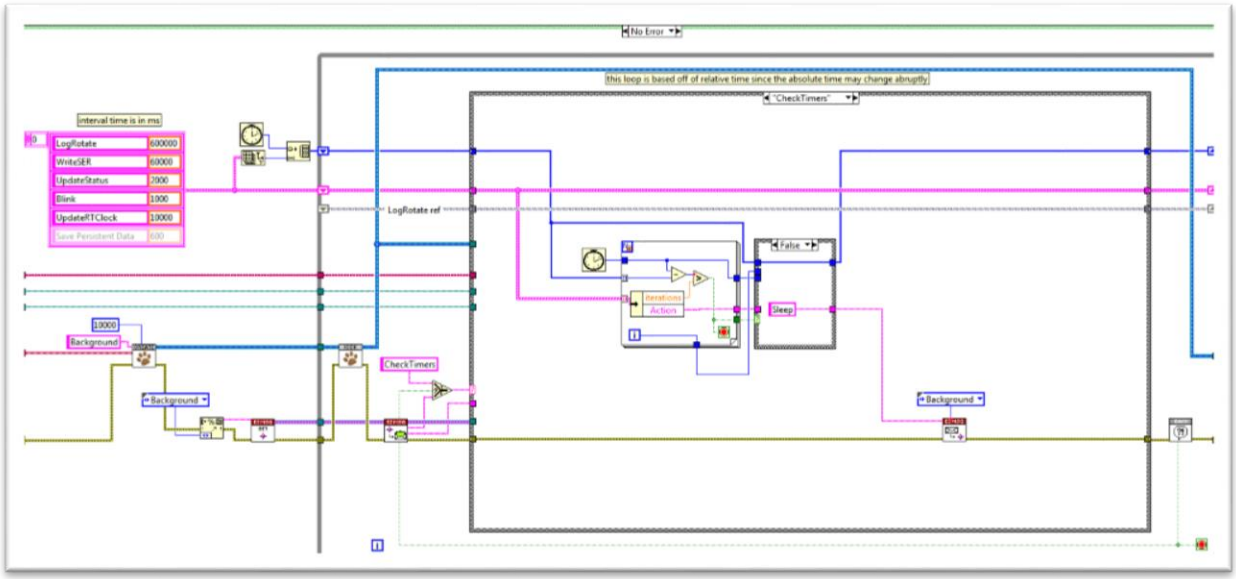


Figure 35 Background Thread Implementation

Watchdog 

Ensuring Reliability with Watchdog Timers

Imagine one has just developed an application for NASA which will be part of the system responsible for getting a lunar rover to the surface of Mars. Tens of thousands of hours have been spent writing, testing

and debugging code and then something unexpected happens. One can't very well catch the spacecraft to reboot the system.

According the Michael Lyi in the *Handbook of Software Reliability Engineering, 1996 "*The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them. … It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope, and it is the growth in utilization of software components that is largely responsible for the high overall complexity of many system designs."

Even in a report done by NASA on *Study on Flight Software Complexity, 2009* they state "An exceptionally good software development process can keep defects down to as low as 1 defect per 10,000 lines of code. This means that a system containing 1 million lines of code will have 100 defects, some of which will manifest during mission operations. "

While it is out of the scope of this document to go into all of the various software engineering practices needed to develop a robust system, one mechanism which is frequently utilized is Watchdog Timers.

A watchdog timer is a hardware counter interfacing with the embedded software application to detect and recover from software failures. An example of a software failure is an application running out of memory, causing the application to hang or crash. Even if one follows best practices for managing memory, it's always important to have a back-up plan.

All CompactRIO and Single-Board RIO controllers include a hardware timer accessible from the LabVIEW Real-Time Module. During normal operation, the software application initiates the hardware timer to count down from a specific number at a known increment and defines the action to take if the timer reaches zero. After the application starts the watchdog timer, it periodically resets the timer to ensure the timer never reaches zero, as shown in Figure 36.

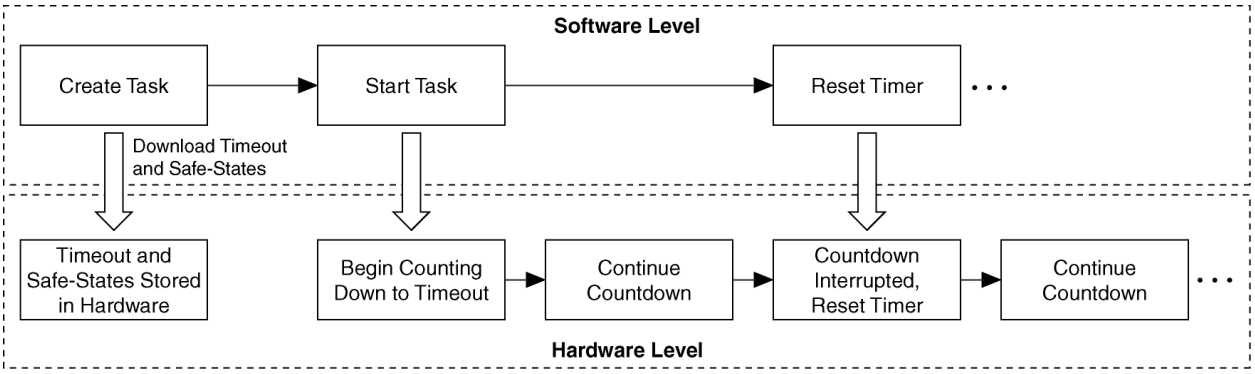


Figure 36 Application periodically resets the watchdog timer when the application responds on time

If a software failure prevents the application from resetting the timer, the timeout eventually expires because the hardware counter is independent of the software and thus continues to count down until it

reaches zero. When the watchdog timer expires, the hardware triggers the recovery procedure, as shown in Figure 37.

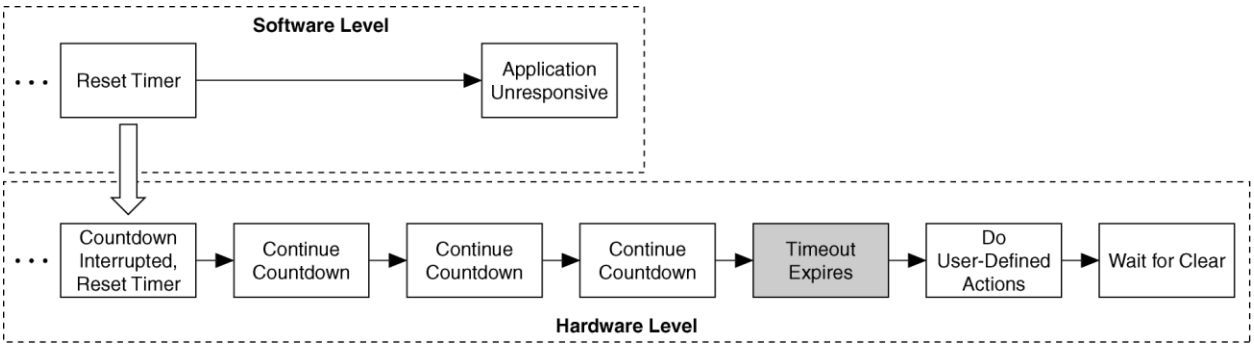


Figure 37 The hardware timer triggers a recovery procedure when the watchdog timer expires

When preparing an embedded system for deployment, there are two options for implementing a hardware-based watchdog timer within LabVIEW. The built-in watchdog hardware available in all NI CompactRIO and Single-Board RIO controllers can be accessed using the LabVIEW Real-Time Module, or once can implement a customized watchdog timer using the LabVIEW FPGA Module. If writing to any hardware outputs occurs from within the FPGA, it is beneficial to implement the watchdog timer in LabVIEW FPGA. If something goes wrong, all hardware outputs can immediately and reliably be put into a safe state. Each option is described in further detail in the following sections.

LabVIEW Real-Time Watchdog

The LabVIEW Real-Time watchdog uses a hardware counter built-in to the real-time controller which interfaces with the embedded software application. The RT Watchdog API can be found in the Real-Time palette.

Figure 38 The RT Watchdog API interfaces with a hardware counter built into NI CompactRIO and Single-Board RIO controllers

When programming with the RT Watchdog API, the first step is configuring the watchdog and set a timeout value. The appropriate range of timeout values depends on the specific performance characteristics and up-time requirements of the embedded application. Set the timeout long enough so it does not expire due to acceptable levels of system jitter, however, set the timeout short enough so

the system can recover from failure quickly enough to meet system up-time requirements. In Figure 39 the watchdog timeout is set to 10 seconds.

Next configure the expiration actions. Specifically, how should the system respond to a watchdog timeout? One option is to reset the target, or a second option is to trigger an occurrence which can execute another piece of code if the watchdog loop becomes unresponsive.

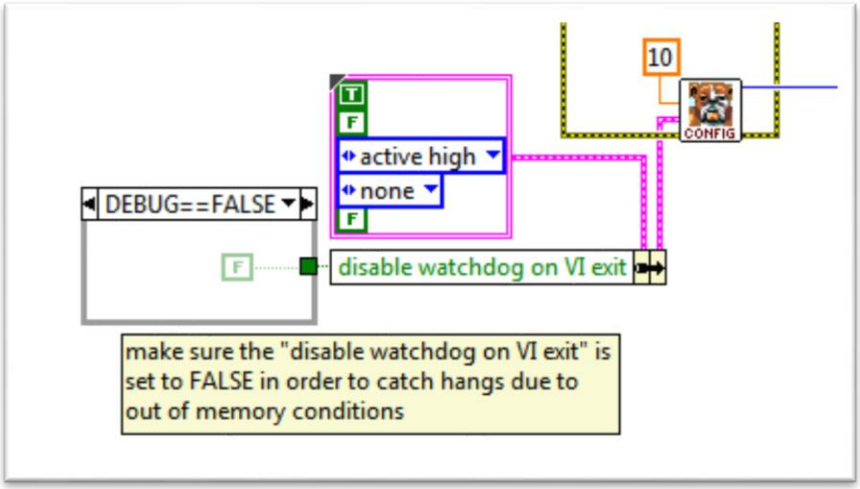


Figure 39 Configure the timeout value and expiration actions with the Watchdog Configure.vi

Finally, acknowledge (sometimes called whacking the dog, petting the dog or servicing) the watchdog periodically. Every time the watchdog is acknowledged the watchdog timer is reset. If something in the overall system causes the watchdog loop to become unresponsive (low memory, low CPU bandwidth, etc.) the watchdog timer is not reset and the expiration actions defined during initialization will execute.

LabVIEW FPGA Watchdog and Fail-Safes

While our particular application doesn't utilize any hardware outputs, if the embedded application uses LabVIEW FPGA for any hardware outputs, a watchdog timer implemented on the FPGA fabric should be considered. The FPGA watchdog will increase reliability of the system and ensure all of the hardware outputs are put into a safe state upon a software failure. When controlling dangerous or critical machinery, it is necessary to implement fail-safes to ensure the machine operates safely even when elements of the control hardware or software fail.

Figure 40 shows an example of how to implement logic in LabVIEW FPGA that determines when a system should go into a safe state. Note one of the conditions we are monitoring is whether or not the watchdog is safe.

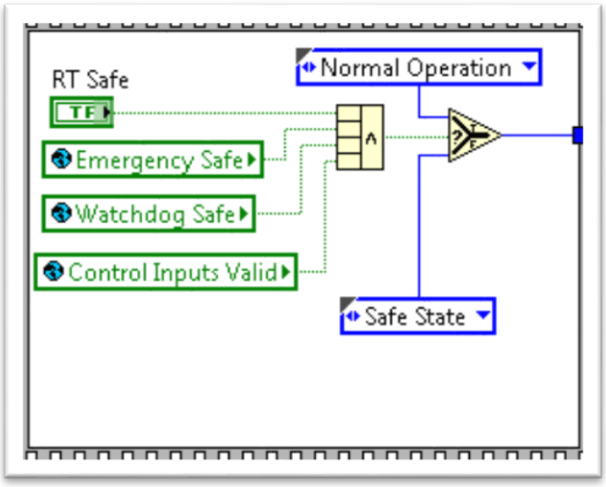


Figure 40 Define failure conditions within LabVIEW FPGA when implementing safe states

Listed below are two reference designs available to help implement a LabVIEW FPGA based watchdog timer and safe states:

- The Fail-Safe Control Reference Design for CompactRIO whitepaper – this reference design written by NI Systems Engineering provides a framework demonstrating FPGA safe states and FPGA monitored watchdogs for the real-time controller.
- The LabVIEW FPGA Control Sample Project – this Sample Project included in LabVIEW 2012 is based upon the Fail-Safe Control Reference Design linked above.

Software Watchdogs

In conjunction with implementing hardware-based watchdogs, software-based watchdogs can additionally be implemented. The Fail-Safe Control Reference design has multiple software loops in conjunction with a software watchdog loop, and an additional LabVIEW FPGA based watchdog. If any of these loops become unresponsive, the software watchdog can take action to fix or reboot the system. This software watchdog loop then checks-in with the hardware watchdog in case something happens to it or to the entire system.

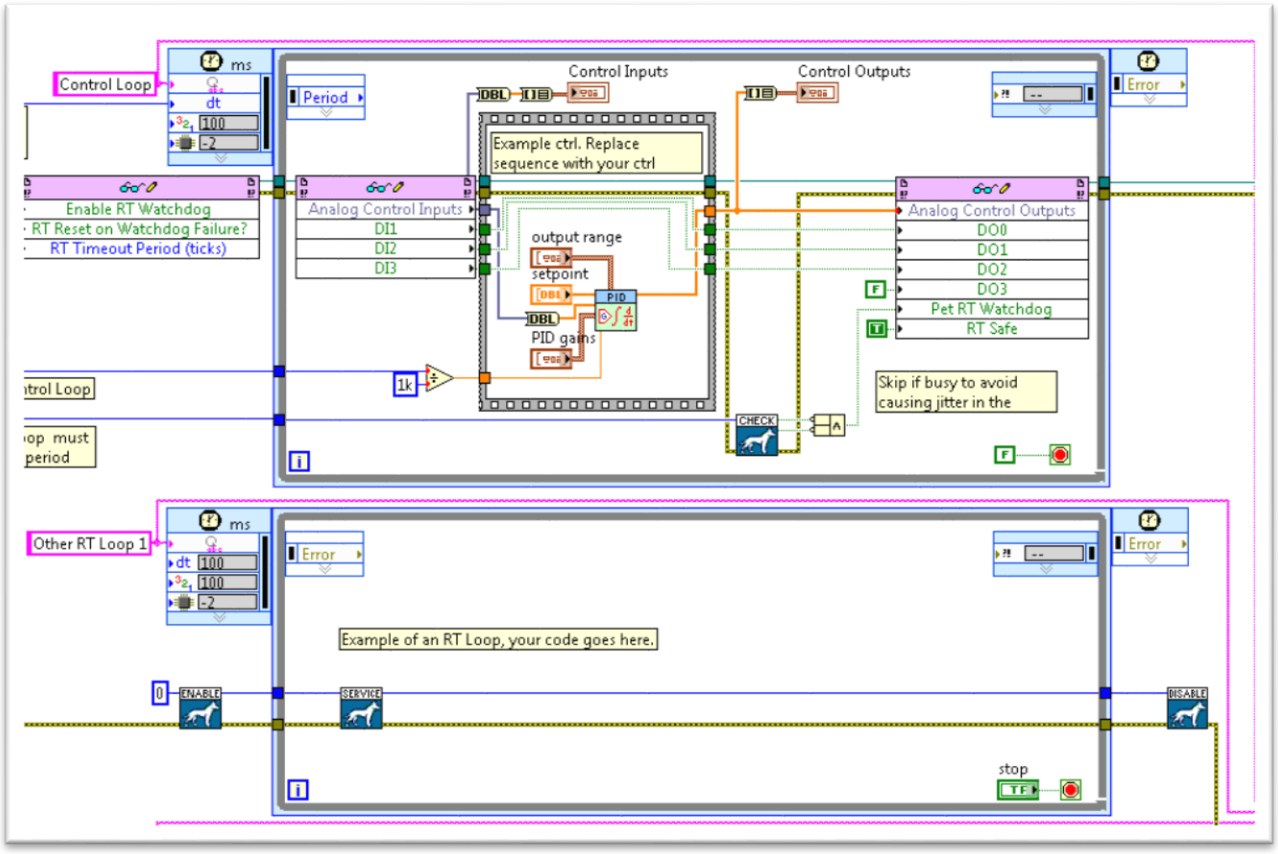


Figure 41 A software watchdog can take action if any loops become unresponsive, and check-in with the hardware watchdog in case something happens to the entire system (From Fail-Safe Reference Design)

Data Acquisition

The data acquisition loop is the most important loop in the entire embedded application. It is the module responsible for delivering the data which the user will make decisions on. Remember the acquisition loop is based off of the Store and Forward methodology referenced again in Figure 42.

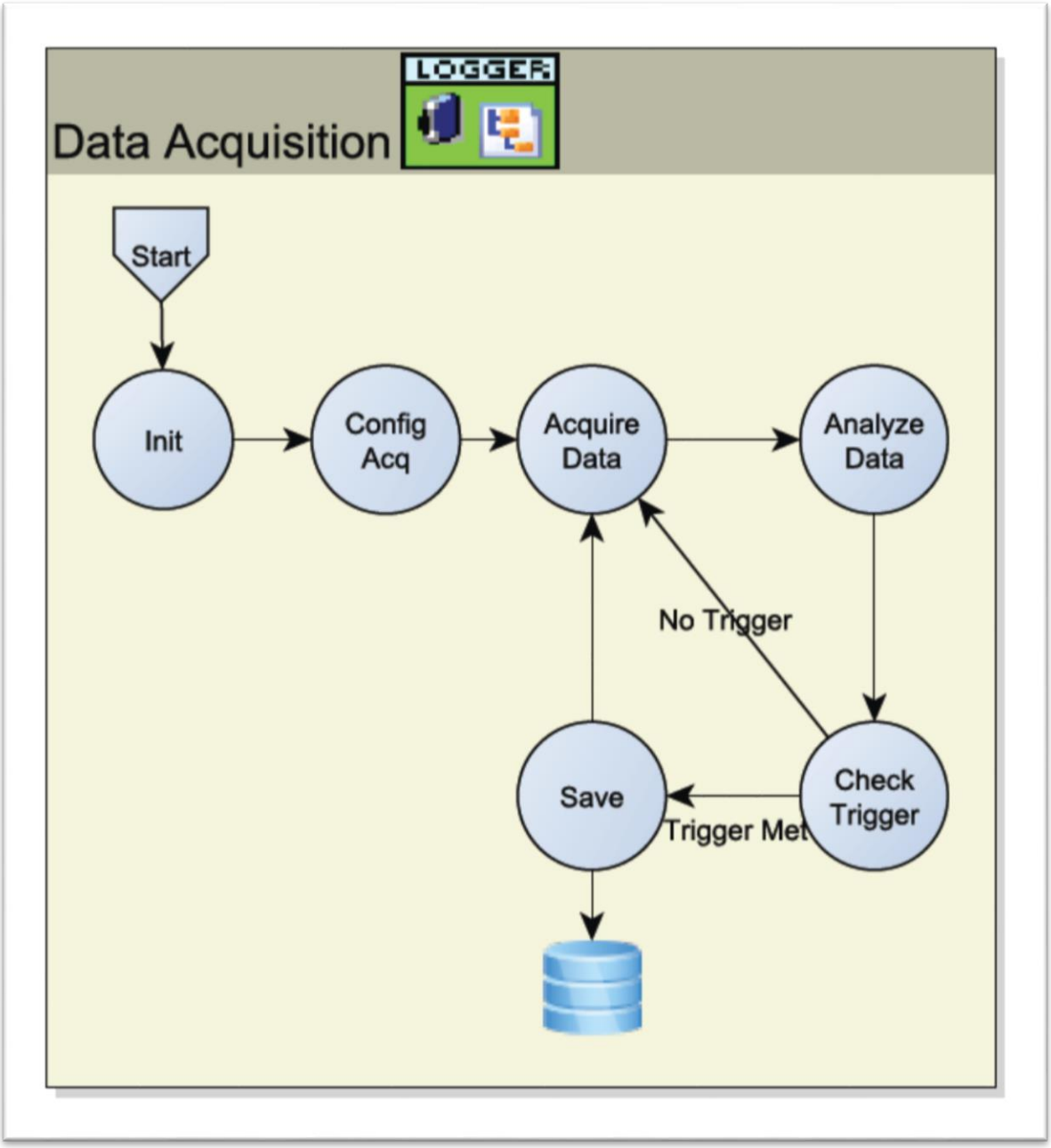


Figure 42 Store and Forward Architecture

The reader should familiarize themselves with this architecture again as this section won't talk in detail about how or why the state machine exists, but will focus more on the implementation details.

The acquisition can be broken down into two parts: the Real Time operation and the FPGA implementation. The Real Time code is oftentimes referred to as the firmware. This firmware resides on the Distributed Acquisition & Analysis Node and is where the majority of the code for the embedded system is implemented. It is where all of our previous threads have lived.

The other component of the acquisition system is the FPGA application. The FPGA is directly connected to the I/O for high-performance access to the I/O circuitry of each module including timing, triggering, and synchronization. Because each module is connected directly to the FPGA rather than through a bus, the application experiences almost no control latency for system response compared to other controller architectures.

Real Time Data Acquisition Implementation

As stated previously this is where the acquisition state machine is implemented. First, the information about which vibration and auxiliary channels are configured within the system is read and verified to ensure proper channel configuration, scaling, etc. Additionally, all of the static property information is written to the TDMS file. Because, by definition, the information is static it doesn't make sense to write static properties again while the acquisition is running; thus causing unnecessary delays to the acquisition loop.

Start is called next and essentially this is the "Config Acquisition" bubble in Figure 42. Items like buffer size, acquisition rate, number of channels, etc. are all collected and used to configure the FPGA acquisition. The FPGA portion will be discussed next, but remember the FPGA is actually what directly accesses the I/O modules.

Data is then acquired or read off of the FPGA buffers. The data is added to a circular buffer of N block sizes where N is a parameter defined by the user. In this example each block size is one second, so for a 10 block acquisition it takes 10 seconds to acquire the complete set of data which is eventually written to disk. Remember data is constantly being collected and stored into the buffers. Only when a trigger (discussed later), or unique event occurs, does the requested amount of data get written to disk.

Analysis is performed on the data and feature information is provided to the client which can subscribe to it if desired. The analysis is important, because it helps determine if a trigger condition has been met. Many times a user wants to trigger the capture and storage of the waveform information based on analysis parameters like overall level in the system, speed, speed change, crest factor, etc. The analysis must first be performed on the data to see if one of those trigger conditions has been met.

Checking of triggers is the next state in the state machine. In this example, the trigger conditions are defined by either a given elapsed period of time or a defined change in speed.

Finally, if a trigger condition is met the information about the waveform and the calculated parameters are saved to the TDMS file. There is a special case called "Finish File". While all of the current

information is written to disk immediately we have implemented a similar circular buffer for the Files which get stored. If an embedded device were to be offline for a considerable amount of time the hard drive would fill up and thus cause the device to fail. To prevent this the Manage Drive command is implemented in the Message Processor thread. The manage drive function simply looks at how much drive space is left, what threshold the user has set, and then deletes old files making room for a new file to be saved. Originally, this functionality was native to the acquisition loop, but after benchmarking it was found the manage drive functionality took a considerable amount of time; thus slowing down the data acquisition loop and creating buffer overflows. Therefore the decision was made to move this to the asynchronously running Message Processor thread.

FPGA Data Acquisition Implementation

As previously stated, the other component of the acquisition system is the FPGA application. The FPGA is directly connected to the I/O for high-performance access to the I/O circuitry of each module including timing, triggering, and synchronization. Because each module is connected directly to the FPGA rather than through a bus, the application experiences almost no control latency for system response compared to other controller architectures.

One challenge with this type of system where each module is directly connected to the FPGA, is the FPGA needs to be compiled for the EXACT hardware configuration; thus limiting flexibility. As an example, imagine having three available hardware modules of the same type. This can represent 8 different possible hardware configurations. It would be quite cumbersome to expect a user to manage 8 different firmware images based on the specific hardware configuration. One mitigation for this is utilization of the RIO Scan Interface Mode.

RIO Scan Interface Mode

Using the RIO Scan Interface mode, one can program the real-time processor of the CompactRIO system but not the FPGA. In this mode, National Instruments provides a predefined personality for the FPGA that periodically scans the I/O and places it in a memory map, making it available to LabVIEW Real-Time. The RIO Scan Interface Mode allows the user to dynamically add and remove modules into the system without having to update any FPGA code. The RIO Scan Interface is sufficient for applications requiring single-point access to I/O at rates of a few hundred hertz.

This sounds promising, however, as the last sentence states Scan Interface Mode is really only beneficial when doing single-point type access. Distributed Monitoring applications doing any sort of vibration measurements will need to run in the 1kHz to 10s of kHz range and thus Scan Interface Mode is an inappropriate solution.

Adaptable FPGA Architecture

One way to merge the dynamic addition of adding/removing modules of the RIO Scan Interface Mode while at the same time getting the speed and latency benefits of utilizing the FPGA fabric is to create an Adaptable FPGA Architecture. While it is not 100% dynamic it can greatly reduce the number of necessary firmware images which must be supported.

Going back to our example where we have three modules of the same type. By using the Adaptable FPGA Architecture the number of images can be reduced from **eight to one**. If the developer were to have a completely different application with 3 different, yet homogenous modules, the firmware developer does have to develop another firmware image bringing the total number to two. Unlike RIO Scan Interface Mode, the developer can't completely get out of developing new firmware when the module mixture changes, but the number can greatly be reduced.

Let's look at a real world application and then discuss how the Adaptable FPGA Architecture is constructed to meet this solution. In power generation facilities there are several ancillary components (pumps, fans, boiler feed pumps, chillers, etc.) whose primary functions are to ensure the electricity generating turbines continue running. These components are oftentimes referred to as Balance of Plant Assets.

Balance of Plant Platform

The example code developed and provided with this document was for a typical balance of plant Distributed Acquisition & Analysis Node. The hardware breakdown for this balance of plant solution is described in Figure 43:

Model	Description	Qty.
NI cRIO-9074	NI cRIO-9074 Integrated Controller Backplane	1
NI 9232	NI 9232 Accel/Prox w/ Screw Terminals, 3 Channels	1-6
NI 9205	NI 9205 w/ Spring Terminals, 16 Differential Channels, up to +/10V input	0 or 1
NI 9219	NI 9219 Universal Input Module, 4 Channels	0 or 1

Figure 43 BOP Hardware Description

The solution includes modules to support vibration (NI 9232) as well as modules to support general voltage (NI 9205), temperature (NI 9219) and 4-20mA (NI 9219) input sensors. This platform is designed to monitor many of the standard sensors necessary to implement a robust predictive maintenance program.

Architecture

The first step in developing the Adaptable FPGA Architecture is passing information about how many modules the user has configured and defined in a configuration file to the FPGA code.

Note: There is logic in the firmware to ensure the expected number of modules and channels defined in the configuration file matches what is actually present in the hardware.

Figure 44 shows a snapshot of the FPGA code

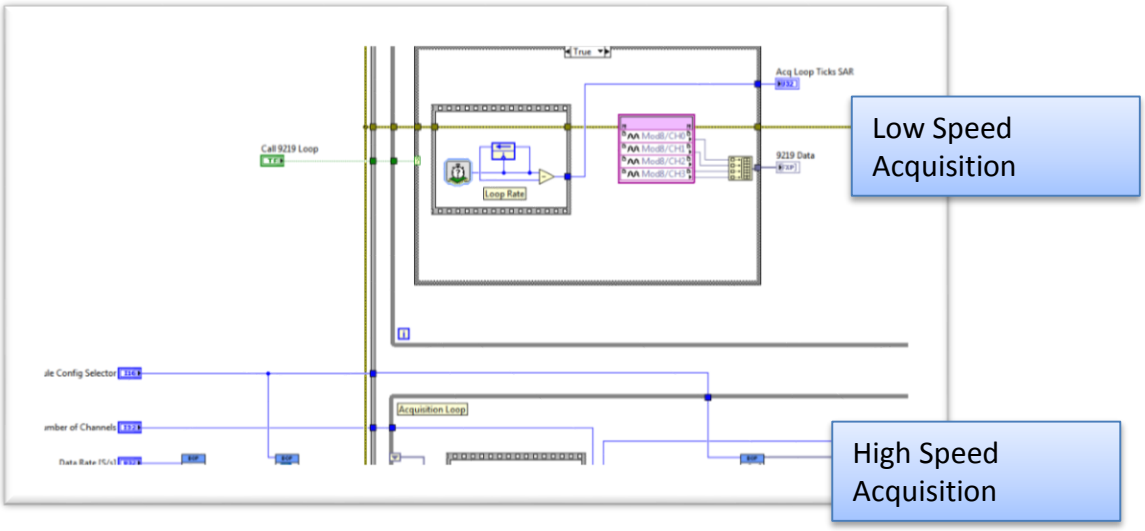


Figure 44 Adaptable FPGA Architecture

The architecture is broken into two separate pieces: a High Speed Acquisition Loop and a Low Speed Acquisition Loop. The low speed acquisition loop executes once per high speed acquisition loop but rather than taking several readings per loop, as in the high speed loop, only one point is returned per loop execution in the low speed loop. The low speed loop is where the general purpose voltage, temperature and 4-20mA sensors are read. For those type of sensors reading a few points per second is acceptable.

High Speed Acquisition

The high speed loop reads information from Delta Sigma modules which were discussed in previous chapters. The Delta Sigma modules all share the same time domain clock and thus are all synchronized. Delta Sigma modules must be explicitly configured, read, and then stopped. How is this accomplished if we don't know how many modules might be in the system at run-time? Simple. Figure 45 shows a snippet of the Start and Read functionality. Notice in each of these functions the block diagram contains a case structure numbered 1-6. This represents the number of vibration (9232) modules which could be in the system. Remember, at run-time the number of configured modules is passed to the FPGA code. This number is the look-up for the case structure and the appropriate number of reads, starts, and stops is selected based on the number of modules.

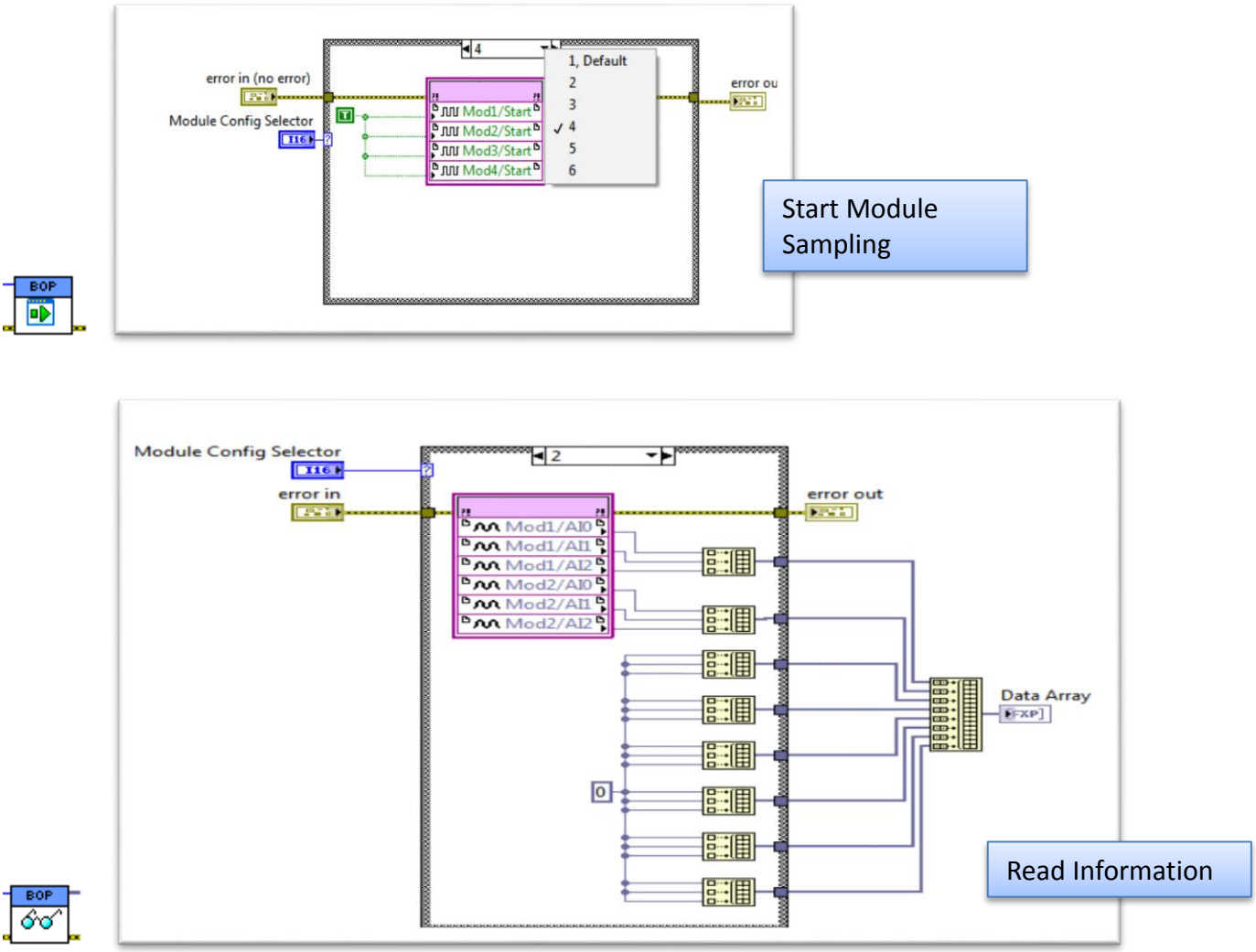


Figure 45 FPGA Start and Read

This allows the developer to have flexibility in the number of modules which "could" be present in the system. The con to this approach is there are several cases which never get called, but still take up fixed FPGA space. It is up to the developer to balance the amount of space available on the FPGA with how flexible they want to make the application.

Low Speed Acquisition

The low speed loops are much simpler to program. As previously discussed both the 9205 and 9219 modules utilize "On Demand Conversion". There is no need to start, synchronize reads, or stop these modules. Simply call the appropriate FGPA I/O node for the given module and read the data. There are two "low speed loops" in our application: one for the 9205 module and one for the 9219 module. Remember, information was passed to the FPGA about whether these modules were present in the system. A simple Boolean case structure either turns on or off the corresponding acquisition loop.

Note: For this application we do enforce the NI 9205 module must be in slot 7 and the NI 9219 cSeries module must be in Slot 8).

APPENDIX A—EXTERNAL CODE LINKS

This is a complete list of all of the software utilized in developing this application and will be necessary for modifications or deployment

Software Development Environments:

All Software comes with a 30 day free trial:

LabVIEW 2012 SP1
LabVIEW Real-Time Module 2012 SP1
LabVIEW FPGA Module 2012 SP1
Sound & Vibration Measurement Suite 2012

Embedded Driver Software:

NI-RIO 2012
NI System Configuration 5.5

Package Files:

All of the following packages can be found by searching through the JKI VI Package Manager Utility.

NI CompactRIO Waveform Library 4.1.0.9
NI Asynchronous Message Communication (AMC) Library 3.3.0.21
NI Simple Messaging (STM) Library 3.0.0.4
NI GXML 1.4.1.7
NI Software Watchdog 1.0.0.24
NI LogRotate 1.0.0.18
NI Event Logger Library 1.0.0.8

APPENDIX B—REFERENCED WHITE PAPERS and DEVELOPER ZONE DOCUMENTS

NI LabVIEW for CompactRIO Developer's Guide
yEd Graph Edit
NI cRIO-9074
NI 9232
NI 9205
NI 9219
Semaphore Reference Design for LabVIEW FPGA
How to Compensate for Different Group Delays with C Series Modules in LabVIEW FPGA
What is the Best Method To Synchronize Different DSA Modules in FPGA?
Optimizing your LabVIEW FPGA VIs: Parallel Execution and Pipelining
Best Practices for Saving Measurement Data
Application Design Patterns: State Machines
Asynchronous Message Communication (AMC) Library
LabVIEW Syslog Protocol Reference Library
Syslog Watcher
LabVIEW Simple Messaging Reference Library (STM)
Fail-Safe Control Reference Design for CompactRIO
FPGA Control on CompactRIO Sample Project Documentation

APPENDIX C—CURRENTLY SUPPORTED MESSAGES

STM messages are a high level wrapper sitting on top of basic TCP/IP functionality. Each STM message consists of two strings: The message command/metadata string and the message data string. The actual information in the strings can be of any data type (i.e. Boolean, Integer, Enumeration, etc) which is flattened into a string and then transmitted. On the receiving end the "type" of the string must be known and reconstructed back to the appropriate data type.

C.1 Host/Client Application Messages to Embedded Hardware

Message Name: RestartHardware

Variable Name	*Datatype*	*Possible Values*	*Function*
RestartHardware	Boolean		Allows appliction to force system reboot of embedded hardware.

Message Name: ForceTrigger

Variable Name	*Datatype*	*Possible Values*	*Function*
ForceTrigger	Boolean		Allows operators at any given time to force a data file recording regardless of the current on-board triggering settings.

Message Name: ChannelInfoIndex

Variable Name	*Datatype*	*Possible Values*	*Function*
ChannelInfoIndex	U8	0-255	Specifies the index of the channel to return when the RequestedChannelInfo command is issued.

Message Name: SetTime

Variable Name	*Datatype*	*Possible Values*	*Function*
SetTime	Timestamp		Sets the local time on the cRIO to the designated time.

Message Name: GetVersionInfo

Variable Name	*Datatype*	*Possible Values*	*Function*
GetVersionInfo	Boolean		Request the cRIO to send the VersionInfo message. Message contains information about the current firmware loaded on the cRIO including (Name, Version, Description, and Date Created)

Message Name: GetSystemHWInfo

Variable Name	Datatype	Possible Values	Function
GetSystemHWInfo	Boolean		Request the cRIO to send the SystemHWInfo message. Message contains information about the cRIO controller (model and SN), the Chassis (model and SN), and Module Information (an array of cRIO modules and SNs)

Message Name: GetPrimaryNICSettings

Variable Name	Datatype	Possible Values	Function
GetPrimaryNICSettings	Boolean		Request the cRIO to send the PrimaryNICSettings message. Message contains information about the network settings on the cRIO

Message Name: GetSystemErrors

Variable Name	Datatype	Possible Values	Function
GetSystemErrors	Boolean		Request the cRIO to send the SystemErrors message. Message will return the error message if the cRIO is currently in SafeMode. Otherwise it returns "No Error".

Message Name: SetPrimaryNICSettings

Variable Name: NetworkSettingsCluster. This applies the settings for the primary network interface card (NIC) on the cRIO

Variable Name	Datatype	Possible Values	Function
MACAddr	String	(not used…cannot set)	Mac address of the NIC
IP Address Request Mode	U32	1 – Static 2 – DHCP/Link Local 3 – Link-Local Only 4 – DHCP Only	
IPAddr	String		IP Address
SubnetMask	String		Subnet mask
Gateway	String		Gateway
DNSServer	String		DNS server address

Message Name: ClearSafeMode

Variable Name	Datatype	Possible Values	Function
ClearSafeMode	Boolean		If the cRIO is in safe mode, it will delete the error log and restart the system.

C.2 Server Application Messages to Client Application

Message Name: State

Variable Name	*Datatype*	*Possible Values*	*Function*
State	String	Initialize, CheckTrig, NewFile, AcqSave, FinishFile, MngDrive	Returns current state of the embedded hardware

Message Name: PrimaryNICSettings

Variable Name: NetworkSettingsCluster. *This gets the settings for the primary network interface card (NIC) on the cRIO*

Variable Name	*Datatype*	*Possible Values*	*Function*
MACAddr	String	(not used…cannot set)	Mac address of the NIC
IP Address Request Mode	U32	1 – Static 2 – DHCP/Link Local 3 – Link-Local Only 4 – DHCP Only	
IPAddr	String		IP Address
SubnetMask	String		Subnet mask
Gateway	String		Gateway
DNSServer	String		DNS server address

Message Name: Timestamp

Variable Name	*Datatype*	*Possible Values*	*Function*
Timestamp	Timestamp	Current Time	Provides the current time on the embedded hardware

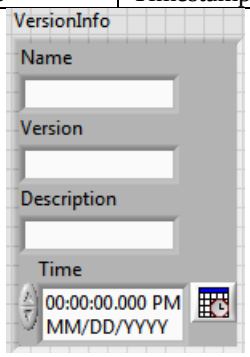
Message Name: SystemErrors

Variable Name	*Datatype*	*Possible Values*	*Function*
SystemErrors	String		Returns information on the current errors on the system when in safe mode.

Message Name: VersionInfo

Variable Name: VersionInfoCluster

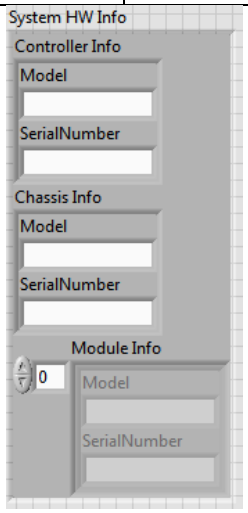
Variable Name	*Datatype*	*Possible Values*	*Function*
Name	String		Name/product ID of the system (i.e. BOP_1). This is not user configurable.
Version	String		Firmware version
Description	String		General description field
Time	Timestamp		Creation time of firmware



Message Name: SystemHWInfo

Variable Name: SystemHWInfo

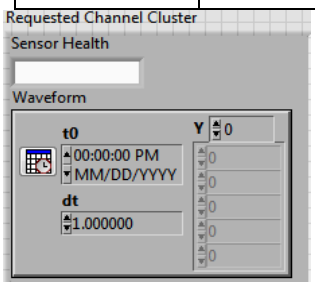
Variable Name	*Datatype*	*Possible Values*	*Function*
Controller	HWInfo.ctl This is a cluster of 2 strings		Model name and serial number of controller
Chassis	HWInfo.ctl This is a cluster of 2 strings		Model name and serial number of chassis. Uses same HWInfo.ctl but renamed to Chassis Info
Module	Array of HWInfo.ctl		Model name and serial number of modules. Named Module Info



Message Name: RequestedChannelInfo

Variable Name: Requested Channel Cluster

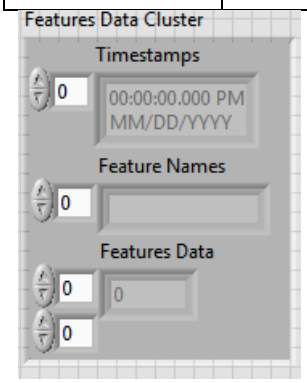
Variable Name	*Datatype*	*Possible Values*	*Function*
Sensor Health	String	Open, Short, Out of Range, Unverified	Provides information about the status of a given sensor specified by ChannelInfoIndex command.
Waveform	WDT of Singles	Channel Waveforms	Provides decimated, waveform data about the individual channel specified by ChannelInfoIndex command



Message Name: FeaturesData

Variable Name: Features Data Cluster

Variable Name	*Datatype*	*Possible Values*	*Function*
Timestamps	1d Array of Timestamps	TimeStamp	Provides the timestamp for when the Features are calculated
Features Names	1d Array of Strings	Documented Elsewhere	Names of the Features which were calculated specified in the form <channel>_<feature>.
Features Data	2d Array of Singles	Features Information	Calculated feature data. Rows correspond to the information in features names array. Columns represent time published in timestamps array.



APPENDIX D—TDMS SCHEMA

File naming convention

The file has the following naming convention: [YYYY][MM][DD]_[HH][min][sec].tdms (example: "20110217_235959.tdms"), date/time being the creation date of the TDMS file.

Because one of the stated purposes of this example is to generate log files suitable for data mining and post processing, the file, group, and channel properties are particularly significant.

Table 1—File Properties

Name	Datatype	Comments
Enterprise	String	
description	String	
DateTime	timestamp	
SV_TDMSFormatVersion	String	Version number of the file format ("1.0")
AlarmDetected	Boolean	OR of AlarmDetected property of all groups in file

Table 2 — Group Names

	Name	Display Name	Comments
Waveform	Waveforms Waveforms_1 Waveforms_2 …	Waveforms Waveforms 1 Waveforms 2 …	within a group, waveform data must be continuous for each channel. Time continuity not known apriori between same channel in different group/file.
Supported Features (measurent type)	RMS	RMS Level	
	Peak	Peak Level	
	MaxMin	Max-Min Level	
	Band_000 Band_001 Band_002 …	000 Band Power 001 Band Peak 002 Band Level …	The integer identifier assigned to the Band does not correspond to the band range.
	Speed	Speed	
	OrderMagnitude_000 OrderMagnitude_001 …	000 Order Magnitude 001 Order Magnitude …	The integer identifier assigned to the Order does not correspond the order(s) being tracked.
	OrderPhase_000 OrderPhase_001 …	000 Order Phase 001 Order Phase …	The integer identifier assigned to the Order does not correspond the order(s) being tracked.
Timestamps	Features_Timestamps	Features Timestamps	timestamp channel not displayed in turnkey applications; instead, it is associated with the feature channels and used to plot feature trends accurately. Features Timestamps can be used as the default timestamp channels for all features that do not

			have a dedicated timestamp group (i.e. RMS_Timestamps)
	<Feature>_Timestamps	i.e. RMS Timestamps Peak Timestamps Max-Min Timestamps 000 Band Timestamps Speed Timestamps 000 Order Magnitude Timestamps 000 Order Phase Timestamps …	dedicated timestamp channel for each feature requires more data management and disk space, but may be necessary when using some logging engines. Dedicated timestamp group always takes precedence over "Features_Timestamps"
Alarms	<Feature>_AlarmStates <Feature>_AlarmLevels <Feature>_AlarmComments	i.e. RMS Alarm States RMS Alarm Levels RMS Alarm Comments	

Calculated channels, whether they be filtered waveforms or features computed from derived channels shall be added with unique channel names to existing groups. Therefore, files written with groups of waveform data and groups of extracted features can look identical for the following two use cases:

1. Multiple measurement types with physical sensors at each measurement node
2. Multiple measurement types with physical sensors and additional derived (calculated) channels. A derived channel could be a filtered instance of a physical channel such as Integration (i.e. acceleration to velocity).

It is the responsibility of the logger to uniquely name any derived channels when writing channel data to a group. The file reader should not rely on channel name to describe the derivation, but should look to the channel properties to qualify the 'physical meaning' of the waveform data and/or extracted features.

Table 3 – Group Properties

	Name	Datatype	Comments
Common	<ID Properties>		
	Author	String	Propose Author would be populated with computer name and name of logging VI
	Operator	String	person with monitoring responsibility
	SiteID	String	Preferred location is at group level, but adding these properties at the channel level is supported and may be required for some measurement system topologies
	AreaID	String	
	AlarmDetected	Boolean	Logical OR of AlarmDetected property of all channels within group
Waveforms	<Feature>	depends on feature	
Features	MeasurementDuration	DBL	
	BandDomain	String	i.e. "1: Frequency (Hz)"

			Feature = Band
	BandStart	DBL	Feature = Band
	BandStop	DBL	Feature = Band
	NI_FFTSize	I32	Feature is any spectral measurement such as Band_000
	NI_WindowSize	I32	Feature is any spectral measurement such as Band_000
	NI_Window	String	i.e. "1: Hanning" Feature is any spectral measurement such as Band
	Order	DBL	Feature = OrderMagnitude_000

Table 4 – Waveform Channel Properties

Name	Datatype	Comments
AlarmDetected	Boolean	
SiteID	String	Preferred location is at group level, but adding these properties at the channel level is supported and may be required for some measurement system topologies
AreaID	String	
LineID	String	
MachineID	String	
ComponentID	String	
ChannelID	String	DAQ:"Dev1/ai1"
ChannelGUID	String	stronger than ID; user does not assign GUID same GUID ➔same channel (continuity guaranteed when spanning groups and files)
ChannelAlias	String	user-assigned string for identification and display. Does not have to be same as TDMS channel name or <name> channel property.
SensorID	String	this should be sufficient to uniquely identify the sensor within a table, file, database recommend format <Make>Model<Model>SN<SerialNumber> IMIModel622B01SN33254 BSWAModelMPA416SN440186 MonarchModelPLT200SN418
SensorType	String	use DAQmx measurement type long name as template and standard name for DAQmx types supported: Voltage Voltage RMS Current Current RMS Thermocouple RTD Thermistor Resistance Strain Gage Force (IEPE) Bridge LVDT RVDT Proximity Probe Accelerometer Velocity (IEPE) Microphone
NI_SensorSensitivity	DBL	mV/EU can be used to convert between unscaled voltage

		and engineering units; NI_ namespaced to preserve compatibility with waveform attributes written by SVL Scale Voltage to EU VIs
MeasurementLocation	String	
name	String	
description	String	
unit_string	String	
ReferenceChannel	String	This property can be used to link vibration channel to corresponding tachometer channel. Note, this can be used in compliment with the NI_SV_Tachometer property used to tag the tachometer channel
ReferenceType	String	'Tachometer', 'Speed', 'FRF Reference', 'Frequency'
RMS	DBL or SGL	units are same as waveform
Peak	DBL or SGL	units are same as waveform
MaxMin	DBL or SGL	units are same as waveform
UpperLimit	DBL or SGL	units are same as waveform
LowerLimit	DBL or SGL	units are same as waveform
UpperLimitClearance	DBL or SGL	units are same as waveform
LowerLimitClearance	DBL or SGL	units are same as waveform
LimitTestPass	Boolean	
Speed	DBL or SGL	
Speed_Units	String	'RPM'
Band_000_Domain	String	'Frequency', 'Order'
Band_000_Start	DBL or SGL	
Band_000_Stop	DBL or SGL	
Band_000	DBL or SGL	
Band_000_Units	String	'g^2 rms', 'in/s pk'. If not written, band values assumed to be same units as waveform (peak)
Order_000	DBL or SGL	
OrderMagnitude_000	DBL or SGL	
OrderMagnitude_000_Units	String	
OrderPhase_000	DBL or SGL	
OrderPhase_000_Units	String	
maximum	DBL	
minimum	DBL	
monotony	String (increasing, decreasing, not monotone, not calculated)	Always 'not calculated'. Actually assumed to be not monotone for SV Applications.
length	Int	
novaluekey	String (Yes, No, Not calculated)	Always 'No'
label	String	Unit info
unit_string	String	Unit info
wf_xname	String	Unit info
wf_xunit_string	String	Unit info
name	String	Function info
NI_LabVIEWDatatype	String	Function info
NI_XDimension	String	Function info
NI_FunctionSubType	String	Function info
NI_FunctionType	String	Function info
NI_SV_SensorLocation	String	
NI_SV_Pair	I32	Sensor location is converted to I32 and written as pair for eddy current proximity probes
NI_SV_SensorOrientation	String	
NI_SV_ProbeAngle	DBL	SensorOrientation is converted to DBL and written as probe

		angle for eddy current proximity probes
NI_SV_Tachometer	Boolean	Indicates tachometer channel.
NI_SV_PulsesPerRevolution	I32	written on tachometer (reference) channel
NI_SV_RotationDirection	I32	written on tachometer (reference) channel
NI_SV_TachometerThreshold	DBL	written on tachometer (reference) channel
NI_SV_TachometerSlope	I32	written on tachometer (reference) channel
NI_SV_MaxOrder	I32	written on tachometer (reference) channel
NI_SV_AnalogSWTrigger	Boolean	written on analog software trigger (reference) channel
NI_SV_TriggerLevel	DBL	written on analog software trigger (reference) channel
NI_SV_PretriggerSamples	I32	written on analog software trigger (reference) channel
NI_SV_TriggerSlope	I32	written on analog software trigger (reference) channel
NI_SV_TriggerHysteresis	DBL	written on analog software trigger (reference) channel

Table 5 - Feature Channel Properties

Name	**Datatype**	**Comments**
AlarmDetected	Boolean	
SiteID	String	Preferred location is at group level, but adding these properties at the channel level is supported and may be required for some measurement system topologies
AreaID	String	
EquipmentID	String	
ChannelID	String	DAQ:"Dev1/ai1"
ChannelGUID	String	stronger than ID; user does not assign GUID same GUID ➔same channel (continuity guaranteed when spanning groups and files)
ChannelAlias	String	user-assigned string for identification and display. Does not have to be same as TDMS channel name or name channel property.
SensorID	String	
name	String	
description	String	
unit_string	String	
ReferenceChannel	String	This property links vibration channel to corresponding tachometer channel. It is necessary to use this property to support multiple tachometers. For the most common use case of one tachometer, all channels are not marked with this property. Instead the tachometer channel is marked with NI_SV_Tachometer
ReferenceType	String	'Tachometer', 'Speed', 'FRF Reference', 'Frequency'
NI_SV_ Tachometer	Boolean	Indicates tachometer channel.
NI_SV_PulsesPerRevolution	I32	written on tachometer (reference) channel
NI_SV_RotationDirection	I32	written on tachometer (reference) channel
UpperLimit	DBL or SGL	
LowerLimit	DBL or SGL	
UpperLimitClearance	DBL or SGL	
LowerLimitClearance	DBL or SGL	
TrackingOrder	DBL or SGL	
LimitTestPass	Boolean	
maximum	DBL	

minimum	DBL	
monotony	String (increasing, decreasing, not monotone, not calculated)	Always ‘not calculated’. Actually assumed to be not monotone for SV Applications.
length	Int	
novaluekey	String (Yes, No, Not calculated)	Always ‘No’
label	String	Unit info
unit_string	String	Unit info
wf_xname	String	Unit info
wf_xunit_string	String	Unit info
name	String	Function info
NI_LabVIEWDatatype	String	Function info
NI_XDimension	String	Function info
NI_FunctionSubType	String	Function info
NI_FunctionType	String	Function info

Only relevant channel properties are written. For example, NI_SV_TrackingOrder is only written for order tracking groups of channels.

AlarmDetected is a property of File, Groups, and Channels. At each level, the property is an OR of all the levels below.

SiteID and AreaID are supported at either the group or the channel level. Group level is preferred because it reduces property redundancy in the file. Some measurement topologies or the need to consolidate multiple files can require groups with different ID properties get combined. Any group properties that differ should get pushed down to the associated channels.

Readers of the SV_TDMSFormat should be robust enough to read files including any or all groups or properties. Some files will not contain waveform groups; some may contain a small subset of feature groups.