Web-based Parameter Control and Real-time Waveform Display for the GRIFFIN Experiment

Bryerton J. Shaw, Pierre-André Amaudruz *Member, IEEE*, and Daryl P. Bishop

***Abstract*—New data acquisition electronics is being developed at TRIUMF for the Gamma-Ray Infrastructure For Fundamental Investigations of Nuclei (GRIFFIN) spectrometer. Current FPGA capabilities have allowed opportunities for providing a more user friendly, web-based, hardware control interface that can be used without requiring additional custom software. Several software and firmware components are being developed, including a real-time waveform viewer, parameter control and read back, diagnostic counters, and a template-based configuration system utilizing MIDAS, and Javascript. This paper discusses the various protocols that were investigated, the benefits and challenges of the choices made, and the details of the interface implementations.**

I. Introduction

THE Gamma-Ray Infrastructure For Fundamental Investigations of Nuclei (GRIFFIN) [1] experiment uses the GRIF-16 Digitizer to read the signals from the GRIFFIN HPGe detectors, plastic scintillator detectors (with a preamplifier), and Si(Li) detectors. To enable ease of use of the device, a web-based control system was built. This control system enables the user to see the digitized waveforms, update parameters, as well as readback status and statistics. All parameter changes are synchronized with a Maximum Integrated Data Acquisition System (MIDAS) [2] Online Database (ODB) in real-time, and locked during the running of the experiment.

The control system itself runs on a NIOS II embedded processor [3] instantiated within the Arria V FPGA on the GRIF-16 digitizer. The web content is served using Mongoose [4], and the FPGA registers and waveforms are accessed via custom Avalon [5] Memory-Mapped slaves. Communication to the MIDAS mhttp server is handled by a custom http client. All parameter control and readback flows through multiple MSCB [6] nodes running within the uC-OS II [7] on the NIOS II processor. On the web browser client, a combination of HTML, Javascript, AngularJS [8] and Angular Materials [9] are used to display the content and format it for desktop or mobile.

Over the course of the integration and development on the various components that make up the control system on the GRIF-16 digitizer, some limitations were uncovered, and additional functionality was desired for future projects. Specifically, a lack of self described input validation, and the ability to route messages between devices. A new protocol is being developed in an attempt to overcome these deficits, and enable TRIUMF to improve the functionality, reliability and usability of its electronics.

Manuscript received June 2, 2016.

B. Shaw is with Electronics Development, TRIUMF (bryerton@triumf.ca).
P. Amaudruz is with DAQ Group, TRIUMF (amaudruz@triumf.ca).
D. Bishop is with Electronics Development, TRIUMF (daryl@triumf.ca).
TRIUMF, 4004 Wesbrook Mall, Vancouver, BC V6T 2A3 Canada

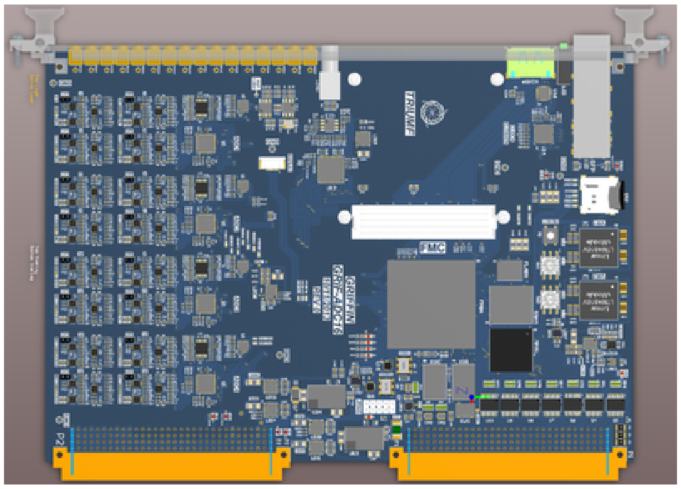


Fig. 1. 3D Model of GRIF-16 Digitizer

II. Overview of the Control System

The web-based control system can be broken into six main components: MSCB Nodes, HTTP server, MIDAS HTTP Client, Web Interface, Avalon Memory-Mapped slaves, and Trigger Memory. These components are integrated together to produce both real-time parameter control and waveform display.

A. MSCB Nodes

MSCB is a simple, self-documenting protocol [10], with each parameter providing its data type, SI unit, and unit prefix (ie: Voltage in milliVolts). Each MSCB node can be reached directly, or through an MSCB submaster also running as a process in the uC-OS II environment. The MSCB nodes are responsible for all the parameters on the device. They provide access to the signal processing registers, clock status, link status, the on-board DACs, and other system resources.

Each node runs an update loop at 10Hz, performing periodic tasks such as checking the run status bit and verifying link connections. If an event has occurred requiring attention, the node will take appropriate action and update the relevant parameters. If the eSATA cable providing the reference clock and run status is disconnected, the MSCB node will flag the eSATA disconnected bit, and switch back to the internal clock. When the cable is reconnected, it will attempt to establish lock to the reference clock, and flag that the cable has been reconnected.

When commands are issued to an MSCB node, they forward the appropriate request to the Avalon Bus or C function as required. If the command is a write request, checks are done to ensure the input is valid, and that the run is not currently occurring. If the checks are passed, the parameter and any appropriate hardware registers are updated. If the command is a read request, an MSCB reply is generated and sent back to the MSCB submaster.

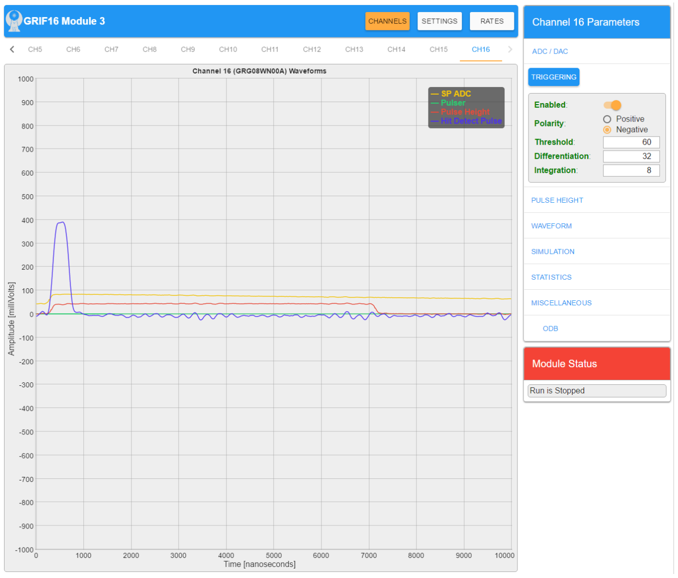
B. *HTTP Server*

The Mongoose web server was modified to run under the uC-OSII environment by stripping down its feature set. The modified Mongoose web server has three purposes: serving static content, handling MSCB messages, and returning waveform data. Static content is served directly from a 1Gbit Flash memory chip located on the GRIF-16. The altera_zipfs [11] filesystem is used, a simple check is done to ensure the file exists, and the content is delivered. MSCB messages are handled in two ways. The first is via octet-stream HTTP requests containing well-formed binary MSCB messages, which may be created via ArrayBuffers in Javascript.These messages are then passed unaltered to the MSCB Submaster for processing, and the response is sent back to the web browser. The second way an MSCB message may be handled is via an HTTP request for an MSCB node in its entirety. Every parameter in the MSCB node is packaged into JSON and returned, along with relevant node related variables. By allowing the MSCB submaster to be bypassed, we can avoid constant individual requests to the MSCB nodes and reduce our overall response latency. Waveform data requests bypass the MSCB nodes as they are not involved in the processing of the waveforms. If the web server receives a request for a waveform, it verifies the channel number is valid, and creates a binary octet-stream HTTP response containing two 16-bit unsigned integers describing the number of buffers returned and the number of samples per buffer, followed by the samples in each buffer as signed 16-bit integers. This response is then returned to the web browser.

C. *MIDAS HTTP Client*

The MIDAS HTTP client is used to handle the template system, which consists of MIDAS ODB key/value pairs contain the default settings for each detector type, and custom settings created for a specific ADC channel.

The MIDAS HTTP client polls the MIDAS mhttp server using the MIDAS AJAX functions [12], and parses the JSON encoded responses, using these responses to synchronize the local MSCB parameters with the MIDAS ODB. This synchronization occurs in both directions, if a parameter is updated locally on the GRIF-16, an appropriate request is made to update the custom settings for that ADC channel. Conversely, if a parameter is updated on the ODB for a detector type, all ADC channels using that detector type are updated unless being overridden by a custom setting.



Enabled:
Polarity: Positive [✓ Negative]
Threshold: 60
Differentiation: 32
Integration: 8

Run is Stopped

Fig. 2. Example of the Web Interface

D. *Web Interface*

The web interface is served directly to the web browser by the GRIF-16 Digitizer and consists of multiple HTML, Javascript and CSS files. Developed using the AngularJS and Angular Material frameworks, the web interface displays the parameters and waveforms available from the GRIF-16, performs input validation on the parameters, and synchronizes the data being displayed on the web browser with the data being contained on the GRIF-16.

Each parameter from the GRIF-16 being viewed is passed to the web browser via a JSON response. This response contains all of the MSCB parameters information as available per the MSCB command CMD_GET_INFO. It also contains the status of the parameters synchronization with the MIDAS ODB, by inserting it into the unused “status” field within the CMD_GET_INFO request. This status is used to graphically indicate whether the parameter is set to the default for that template, a custom setting for that channel, or not in the template system.

When a parameter is added to the web interface UI, it is wrapped with additional metadata. This metadata includes fields such as label, type, and additional descriptors. By using a custom AngularJS directive, this metadata can be used to automatically select the appropriate HTML5 input type and use that input type to force the web browser to perform validation of MSCB parameter before it is sent to the GRIF-16.

Synchronization between the web interface UI and the GRIF-16 is handled entirely within AngularJS. When a parameter is updated by the UI, a field associated with the parameter called “last_update” is set to the current time, according to the web browser. Each time the UI sends out a poll request for new MSCB parameter values, it saves the current time the request was made. When the request is returned, if the parameter value differs from what is currently in the UI, it will only be updated if the time the request was made is newer than the "last_update" field on the parameter. This check ensures that if a poll request returns while a user is updating a field, it does revert the parameter being changed by the user. Should the parameter change fail for any reason, it will get reverted the next time a poll request goes out and is returned post user update.

E. Avalon Memory-Mapped Slaves

Any communication between a NIOS II processor and the hardware registers within the FPGA must occur via the Avalon Bus. In order to map various registers into the memory address of the NIOS II, custom Avalon Memory-Mapped slaves were created. These slaves were then made into QSYS Components with C source and header files to enable seamless integration into the NIOS II environment via the Board Support Package (BSP). MSCB parameters were then added to the appropriate MSCB nodes to enable access to the hardware registers.

F. Trigger Memory

In order to capture and display waveform data, a ring buffer was created in hardware that allowed a variable amount of pre and post trigger waveform data to be stored until readout. This "trigger memory" is controlled by the NIOS II via the Avalon Bus, and is used to capture the raw waveforms, as well as signals related to the processing of the waveform, such as pulse height, hit detection, and simulated pulser input. The trigger input comes directly from the signal processing fabric, but can be set to force a waveform capture if no trigger has been received within a certain time frame, mimicking the "auto" mode on an oscilloscope.

III. LIMITATIONS OF CURRENT CONTROL SYSTEM

The current control system evolved from MSCB, which was initially developed for simple communication over slow serial bus lines to small microcontrollers. As such certain limitations have naturally arisen, such as direct addressing due to slow serial communication, and a lack of metadata and input validation due to memory concerns.

A. Routing Of Messages

As the control system was based around MSCB, it uses MSCBs addressing structure, which has no ability to pass messages between nodes, as it requires that a node be directly accessed, and that a node may not respond to a message not directed for it. Additionally, since there is no return (or sender) address specified in the message header, there is no way to respond to a specific client. Any message received is replied to blindly by the node being addressed.

This limitation of not being able to route messages did not impact the experiment at this point, as only five GRIF-16s are currently in operation, but there is a strong need for this functionality as the number of GRIF-16 digitizers increases, and it may be beneficial for cabling purposes to send messages via the GRIF-C collector cards, which each GRIF-16 is connected to in a tree-like structure.

B. Metadata and Input Validation

Metadata manually associated with each parameter via Javascript is used to properly render each parameter. Parameters are identified by types such as checkboxes, radios, numbers, lists, text, and other widgets. Depending on the type, additional metadata is added, describing valid input options for the parameter. This allows each parameter to be properly displayed and the options for it described properly. All of this information is created by hand solely within Javascript, and is not synchronized automatically by the MSCB nodes. This leads to the possibility of inconsistencies between the description of the parameter on the web browser, and its actual functionality on the MSCB node.

IV. DISCUSSION OF ALTERNATE PROTOCOLS

Due to limitations of the current control system, other protocols were evaluated to see if a suitable replacement was available that matched most or all of the following criteria:

- self-documenting variables
- open-source/free
- not dependent on reliable transport
- low resource costs
- route-able across different underlying transport protocols

After an extensive search, the list of potential replacements was narrowed down to the following four open-source protocols: ZeroMQ [13], CoAP [14], AMQP [16], MQTT [17]. While none of these protocols matched all the criteria, all have significant advantages, and are widely used, accepted protocols.

A. ZeroMQ

In terms of traffic routing and message sending ability, ZeroMQ is an excellent choice. It supports a wide variety of pattern such as pub-sub, router-deal, and push-pull. ZeroMQ is also known to be very fast [15]. The underlying ZeroMQ Transport Protocol (ZMTP) is also very straight-forward and easy to implement. However, while ZeroMQ clients exist for small embedded environments, currently all ZeroMQ routers have been implemented in POSIX/TCP environments. Also ZeroMQ has no built-in support for variables, so another layer would still need to be built on top of ZeroMQ for it to be useful for our purposes. ZeroMQ will most likely see use in future projects at TRIUMF on the linux-side, but was deemed to difficult for use in our small embedded devices.

It should be noted it is possible to build alternate transports for ZeroMQ, as the protocol itself is transport-agnostic.

B. CoAP

The Constrained Application Protocol (CoAP), is very light-weight and designed specifically for machine-to-machine (M2M) applications and embedded environments. While typically run over UDP, it is easily matched to any packet based transport, and there are forms of resource discovery available. It has a very simplistic payload design, but much like ZeroMQ, it would be possible to build upon it to support some implementation of self-documenting variables. CoAP

TABLE I
LIMITED PROTOCOL FEATURE COMPARISON

Protocol	Self Documenting	Routing	Low Resource	Transport Agnostic
ZeroMQ	No	Yes	Yes	Yes (ZMTP)
CoAP	Resources	Broker	Yes	Yes
AMQP	Yes	Broker	Client Only	Yes (Limited Implementations)
MQTT	No	Broker	Yes	Yes

does have good resource discovery, so it does have some ability to 'self document', however not with the payload data itself. CoAP was ruled out as it is designed for systems where every device is directly network-accessible, and due to its limited variable support, which was deemed insufficient.

C. AMQP

The Advanced Message Queuing Protocol (AMQP) was a very close fit, with its ability to create self-documenting messages, various routing methodologies, and clients have been ported to embedded devices. However it was determined the brokers which are used to provide the routing ability have a relatively high memory requirement, and to the authors knowledge, are only available in linux/windows operating systems. Much like ZeroMQ, it will most likely see use inside future projects at TRIUMF in systems running linux.

D. MQTT

The MQTT or "Mosquito" protocol, is another M2M protocol, centered around the use of pub/sub and brokers. With its ultra-simple protocol specifications and straight-forward embeddable C clients, MQTT was very easy to get running. However due to a reliance on brokers for message routing, and complete lack of built-in support for self-documenting variables, MQTT was determined to be a poor fit for our use cases.

V. ARCHITECTURE OF NEW PROTOCOL

Based on the previous criteria, and the unsatisfactory results of the search to find a new protocol to replace the current implementation, work has begun on a new custom messaging/-control protocol. The two key features have been deemed to be self-documenting variables and a flexible routing layer. These two features expand the utility of our existing system and allow promising possibilities, such as automatic plot creation and M2M communication.

A. Self-Documenting Variables

Similar to the MSCB protocol, each variable (or channel in MSCB parlance) is assigned a base type (such as uint8, sint16, ascii), SI unit, SI prefix, and number of elements. However what constitutes a "channel" in the protocol, is allowed to have multiple variables associated with it. Crucially, this allows a multi-dimensional array to be returned, which is key for enabling a graphically charted response. A typical channel could be described as follows:

```
[ch id][name][status][flags][# of vars]
        [var id][type][SI unit][SI prefix][# of elements]
        ...
        [var id][type][SI unit][SI prefix][# of elements]
```

Additionally, each channel may have multiple "annotations" associated with it, which are key/value pairs of the format:

```
[id][key type][key value][pair type][pair value]
```

The purpose of annotations, is to enable metadata associations with a given channel which are related to its functionality, but not critical for the channels use. Possible use cases include client-side input validation, UI widget selection, and logical grouping for display. The use of annotations is expressly designed to be open-ended and flexible.

B. Routing

The current routing implementation is loosely based on RapidIO [18] transport layer, the proposed message header format is:

```
[msgType][msgID][srcID][srcEP][dstID][dstEP][Msg Len][CRC8]
```

By leveraging RapidIOs flexible srcID/dstID approach, virtually any network topology can be used. A separate Endpoint field was added, to simplify having "virtual" sources and destinations within a device without the use of multiple device IDs. A unique msgID is used to differentiate between multiple messages on the go between the same two devices/endpoints. Basic device discovery is planned to be in done by repeatedly querying each device for its list of neighboring devices, and then querying those neighbors.

VI. CONCLUSION

While developing a real-time web display and control system for the GRIF-16 Digitizer, possible avenues of improvement were discovered. Existing alternatives were researched and it was decided that a new control system was required to fill our needs, as here is an gap in the areas of self-documenting, transport agnostic control protocols for embedded systems. This control system is still being developed, but the key features have been fleshed out, and beta implementations are currently being tested.

The new control system will allow much tighter coupling between what is displayed on the web browser, and what exists on the FPGA. It will also greatly reduce the amount of code that needs to be written, both on the FPGA, and on the client-side (Javascript). This should decrease the incident rate of bugs, speed up development, and provide an easier, more friendly experience to the end-user.

Appendix A
Javascript Metadata Example

```
{ label: 'Pole Cxn', type: 'number_float', min: 0.01, max:
     1342177.28, step: 0.001, mscb: adc_channel[adcId].data
     ['p_polec1'] },
{ label: 'Polarity', type: 'radio', desc: "Trigger Edge
     Polarity", options: [
        {label: 'Positive', value: false},
        {label: 'Negative', value: true}
], mscb: $rootScope.adc_channel[adcId].data['a_pol'] },
```

Appendix B
Javascript Template Example

```
<td ng-if="::mscbVar.type == 'number'">
<input ng-disabled='disabled' layout-fill
        style='text-align: right' type="number"
        min={{::mscbVar.min}} max={{::mscbVar.max}}
        step={{::mscbVar.step}}
        ng-model='mscbVar.mscb.d'
        ng-change='mscbVar.updateFn(
                mscbVar.nodeId, mscbVar.mscb, mscbVar.mscb.
                     d)'>
</td>
```

Acknowledgment

Bryerton Shaw would like to thank the GRIFFIN collaborators Adam Garnsworthy, Jenna Smith, and Chris Pearson for their support, feature requests, and patience during the development of the real-time web control system.

References

[1] C. E. Svensson, A. B. Garnsworthy, Hyperfine Interact. 225 (1) (2014) 127.
[2] S. Ritt and P. Amaudruz, "The MIDAS DAQ System", Proc. of the Xth. IEEE REAL TIME Conference, Beaune 97, pp 309-312. : Available at http://midas.triumf.ca
[3] Available at https://www.altera.com/products/processors/overview.html
[4] Available at https://www.cesanta.com/products/mongoose-v2
[5] Available at https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf
[6] Available at https://midas.psi.ch/mscb/
[7] Available at https://www.micrium.com
[8] Available at https://angularjs.org/
[9] Available at https://material.angularjs.org
[10] Available at https://midas.psi.ch/mscb/protocol/
[11] Available at https://www.altera.co.jp/ja_JP/pdfs/literature/hb/nios2/n2sw_nii52012.pdf
[12] Available at https://midas.triumf.ca/MidasWiki/index.php/AJAX
[13] Available at http://zeromq.org/
[14] Available at http://coap.technology/
[15] Available at http://zeromq.org/results:0mq-tests-v03
[16] Available at https://www.amqp.org/
[17] Available at http://mqtt.org/
[18] Available at http://www.rapidio.org/