ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA075

EPICS SUPPORT MODULE FOR EFFICIENT UDP COMMUNICATION WITH FPGAS*

M. Konrad†, E. Bernal, M. Davis
Facility for Rare Isotope Beams, East Lansing, USA

Abstract

The driver linac of the Facility for Rare Isotope Beams (FRIB) contains 332 cavities which are controlled by individual FPGA-based low-level RF controllers. Due to limited hardware resources the EPICS IOCs cannot be embedded in the low-level RF controllers but are running on virtual machines communicating with the devices over Ethernet. An EPICS support module communicating with the devices over UDP has been developed based on the Asyn library. It supports efficient read and write access for both scalar and array data as well as commands for triggering actions on the device. Device-related parameters like register addresses and data types are configurable in the EPICS record database making the support module independent of hardware and application. This also allows engineers to keep up with evolving firmware without recompiling the support library. The implementation of the support module leverages modern C++ features and relies on timers for periodic communication, timeouts, and detection of communication problems. This allows the communication code to be tested separately from the timers to keep the run time of the unit tests short.

INTRODUCTION

FRIB [1] is a project under cooperative agreement between the US Department of Energy and Michigan State University (MSU). It is under construction on the campus of MSU and will be a new national user facility for nuclear physics. Its driver accelerator is designed to accelerate all stable ions to energies >200 MeV/u with beam power on the target up to 400 kW [2]. Commissioning of the second linac segment is currently underway and the accelerator is planned to support routine user operations in 2022 [3].

The FRIB linac requires about 350 low-level RF controllers [4] to actively stabilize the RF field in the accelerating cavities as well as roughly 55 machine protection nodes [5] preventing damage to the machine by turning off the beam within 35 μs in case of a fault. The EPICS support module described by this publication acts as a driver enabling the EPICS Input/Output Controller (IOC) to communicate with these devices.

HARDWARE

Both the LLRF controllers as well as the MPS hardware have been developed in-house and use a low-cost pizza-box design based on a Spartan 6 FPGA. A MicroBlaze soft-core processor [6] implemented in the FPGA allows these devices to be controlled remotely. Since this processor does not provide sufficient resources for running an embedded EPICS IOC, the IOC needs to run on a remote machine instead. A simple C program running directly on the soft-core processor initializes the hardware and handles communication over Ethernet. Due to the lack of an IP stack, the devices' capabilities for handling network communication are limited to UDP (TCP is not supported).

UDP PROTOCOL

The UDP communication protocol specifies that an IOC initiates communication with a device by sending a request packet. The device generally responds with one or more UDP packets. The following commands are supported:

- Read registers
- Write registers
- Read waveform
- Read a block from persistent memory
- Erase a block from persistent memory
- Write a block to persistent memory
- Request write access

The commands for reading and writing registers can transfer multiple consecutive registers at the same time to increase efficiency. Three memory regions are defined for reading and writing registers. They correspond to read-only memory (e. g. for reading out sensor data), read/write memory (e. g. reading/writing set-points) and to "write-once" memory, respectively. The latter address range is used to implement commands which trigger some action on the device (like "start ramp").

The read waveform command transfers an array from the device to the IOC. These arrays can be very large and need to be broken into many UDP packets.

The commands for accessing the persistent memory are acting on a block of memory. They allow flash memories or EEPROMs to be read/modified remotely. The block size usually depends on the capabilities of the memory chip used. The driver automatically selects the most efficient block size if the device supports a range of block sizes.

DESIGN CONSIDERATIONS

The Asyn support module [7] is leveraged to implement the functionality for asynchronously reading/writing registers and for providing the device-support layer. However, some aspects like transfer of large waveforms or firmware in parallel to other communication are not supported by Asyn and thus are implemented by other means in the driver.

* Work supported by the U.S. Department of Energy Office of Science under Cooperative Agreement DE-SC0000661
† konrad@frib.msu.edu

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA075

Table 1: Supported Data Types for Scalar Data Transfer

Asyn Data Type	Size (bit)	Signedness
asynParamInt32	32	signed
asynParamUInt32Digital	32	unsigned
asynParamFloat64	16.16	N/A

The driver relies on the asynOctetSyncIO facilities provided by Asyn to perform network communication. This makes the driver code independent of the underlying operating system.

Dynamic Driver Configuration

The firmware of the low-level RF controllers is improved continuously resulting in registers being added or modified frequently. This also requires the IOC to be flexible. To accomplish that, register names or other information about the meaning of registers for low-level RF or MPS applications is not compiled into the driver. Instead, this information is dynamically read from the INP/OUT field of the corresponding EPICS record during IOC start. After evaluating the Asyn-specific part of these fields, Asyn passes the remaining part to the drvUserCreate function of the support module which extracts the register address and the data type from the string. This information is then used to create the parameter with Asyn Port Driver. This approach also makes the support module device agnostic, allowing it to be used with both the low-level RF controllers as well as with the MPS nodes.

DATA TRANSFER

The driver supports the data types listed in Table 1 for both scalar and waveform data transfer. While the asynParamInt32 and asynParamUInt32Digital data types can be mapped directly to the VAL/RVAL field of the corresponding EPICS records, the driver also supports conversion of fixed point values with a scaling factor of 2^{-16} (16 bit integer part, 16 bit fractional part). These values are represented as double-precision (64 bit) floating point numbers in ai/ao records.

Register Transfer

Data is read from the device periodically (by default at a rate of 10 Hz). The IOC automatically determines the memory range which needs to be read out based on the lowest and highest register address used in each address range. The required address range is generally transferred in one packet to improve efficiency. Write operations to registers on the other hand are performed immediately writing one register at a time. This ensures write packets are sent with minimum latency and in the order the data was submitted to the corresponding EPICS records.

Waveform Transfer

Array data can be read from the device by sending a single request. The device responds by transmitting the array chunk by chunk. If a chunk is lost, a timeout expires and the driver requests the missing part again until all blocks have been received. The assembled array is then passed to Asyn which updates the corresponding record.

Devices can support a command for freezing circular buffers. This can be implemented in the IOC database as a chain of records that freeze, read and unfreeze the buffer (see Fig. 1).

A separate read-out mode geared towards streaming applications is currently under development.

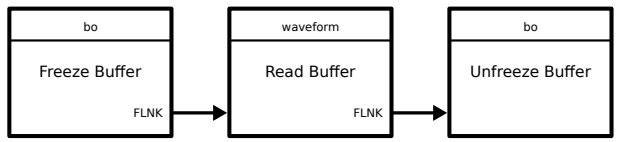

Figure 1: Chain of records that freeze, transfer and unfreeze a circular buffer.

Firmware Update

Users can update firmware by sending a firmware image through Channel Access to a waveform physical variable (PV). The driver will incrementally write the data to the controller ensuring that other communication with the device is not blocked while the firmware image is programmed. In each step a block of flash memory is erased and programmed. Once the waveform data has been written completely, the driver signals success to Asyn, which completes the asynchronous processing of the record.

Read back of the firmware image is triggered automatically (see Fig. 2). An aSub record calculates the SHA-1 hash value of the read back waveform. This value is available as an array of 40 characters. Additionally, the first 39 bytes of the array are also available as an EPICS string (by default EPICS strings are limited to 39 bytes plus one byte for the null termination). EPICS strings are simpler to display on user interfaces and are generally better suited for archiving than arrays. Archiving the hash value is an effective way of keeping a record of firmware updates. Access security on the firmware update PVs ensures only expert users can program firmware.

Optional status records can be used to monitor progress while reading or writing firmware (see Fig. 3). This is useful

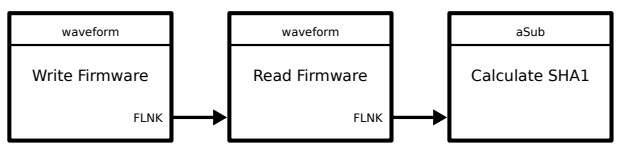

Figure 2: Mechanism for automatically reading back a firmware image after writing it.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA075

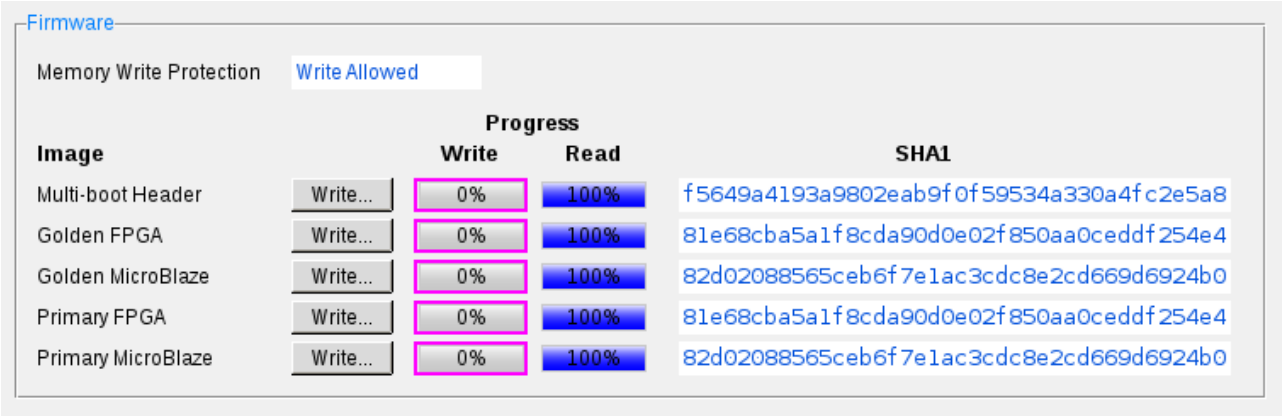

Figure 3: User interface allowing engineers to update firmware. The SHA-1 hash is calculated from the read back image.

for providing feedback to users who are working with large flash memories which might take multiple minutes to read or write.

A Python application allows firmware upgrades from the command line. Mass upgrades can be sped up by running several instances of this Python script in parallel with a tool like GNU Parallel [8], thereby enabling the upgrade of several devices at once.

TIMER-BASED ARCHITECTURE

The support module uses timers to perform the following operations:

- Periodic read-out of registers
- Writing to a register
- Incremental array transfer without blocking other communication
- Detecting communication problems (timeouts)
- Keeping write access

All timers required for these operations are managed by an instance of `epicsTimerQueue`. Each device has its own timer queue resulting in one additional thread per device. Timers used in the context of Asyn use separate timer queues.

`epicsTimerQueue` manages multiple timers with a single thread. The timers are organized in a priority queue ordered by their expiration time. The thread waits for the first timer on the queue to expire before it calls the timer's expiration handler, removes the timer from the queue and sleeps until the next timer expires. The fact that all timer-driven operations related to a device are triggered by the same thread guarantees that only one of them is performed at a time making additional locking unnecessary.

TEST AUTOMATION

The described support module is intended to be used for configuring the machine protection system. Thus malfunction might cause the machine protection system to be configured incorrectly, potentially leading to severe damage to the accelerator. To prevent this, the driver module needs to be very reliable. Unit tests verify that the driver behaves as intended. In particular, tests verify that the driver calls Asyn's `asynOctetSyncIO` facilities correctly when processing a write request. This has been accomplished by injecting either an `asynOctetSyncIO` object or a mock object into the driver's constructor. For normal operation the driver uses the `asynOctetSyncIO` object whereas the mock object is used when running the unit tests. The mock object verifies that the data passed to the object is correct thus ensuring that the driver processes data correctly.

Compared to a conventional implementation, the timer-based architecture of the driver helps to eliminate "sleep" statements from the code base making it possible to eliminate wait times when executing unit tests.

CONCLUSION

The described support module has been used successfully at FRIB for almost two years in a production environment. It supports efficient transfer of both scalar registers as well as arrays over a UDP-based protocol. FRIB's largest IOC using this driver controls 168 low-lever RF controllers with about 220 000 records.

The ability to add/modify registers without recompiling the driver speeds up the test and release process and thus facilitates agile development. Longer-running transactions like reading out large waveforms or programming a new firmware image are implemented using timers so they do not block the remaining communication. Allowing firmware to be upgraded through Channel Access enables FPGA engineers to update firmware themselves. Along with the ability to perform mass upgrades this reduces the time required to deploy new firmware considerably.

Automated tests ensure that the driver is behaving as intended making it suitable for use with FRIB's machine-protection system.

REFERENCES

[1] FRIB, `http://www.frib.msu.edu`

[2] J. Wei *et al.*, "FRIB Accelerator: Design and Construction Status", in *Proc. HIAT'15*, Yokohama, Japan, Sep. 2015, paper MOM1I02, pp. 6–10.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA075

[3] J. Wei *et al.*, "The FRIB SC-Linac - Installation and Phased Commissioning", in *Proc. SRF'19*, Dresden, Germany, Jun.–Jul. 2019. doi:10.18429/JACoW-SRF2019-MOFAA3 to be published.

[4] S. Zhao *et al.*, "The LLRF Control Design and Validation at FRIB", in *Proc.NAPAC'19*, Lansing, MI, USA, Sep. 2019. doi:10.18429/JACoW-NAPAC2019-WEPLM03 to be published.

[5] Z. Li *et al.*, "Current Status and Prospects of FRIB Machine Protection System", in *Proc. NAPAC'19*, Lansing, MI, USA, Sep. 2019. doi:10.18429/JACoW-NAPAC2019-TUPLM29 to be published.

[6] Xilinx Inc., "MicroBlaze Soft Processor Core", https://www.xilinx.com/products/design-tools/microblaze.html

[7] M. R. Kraimer, M. Rivers, and E. Norum, "EPICS Asynchronous Driver Support", in *Proc. ICALEPCS'05*, Geneva, Switzerland, Oct. 2005, paper P3_074.

[8] O. Tange, "GNU Parallel – The Command-Line Power Tool", *login: The USENIX Magazine*, vol. 36(1), pp. 42–47, Feb. 2011. http://www.gnu.org/s/parallel