ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA168

IMPROVING THROUGHPUT AND LATENCY OF D-Bus TO MEET THE REQUIREMENTS OF THE FAIR CONTROL SYSTEM

Dominic Day, Alexander Hahn, Cesar Prados, Michael Reese,
GSI Helmholtz Centre for Heavy Ion Research, Darmstadt, Germany

Abstract

In developing the control system for the FAIR accelerator complex we encountered strict latency and throughput constraints on the timely supply of data to devices controlling ramped magnets. In addition, the timing hardware that interfaces to the White Rabbit timing network may be shared by multiple processes on a single front-end computer. This paper describes the interprocess communication and resource-sharing system, and the consequences of using the D-Bus message bus. Then our experience of improving latency and throughput performance to meet the realtime requirements of the control system is discussed. Work is also presented on prioritisation techniques to allow time-critical services to share the bus with other components.

INTRODUCTION

The White Rabbit based FAIR Timing System developed at GSI [1] provides FPGA-based Timing Receiver hardware for frontend computers. The SAFTlib project (Simplified API for Timing) was designed to share the resources of the Timing Receivers and provide a stable interface that abstracts software clients from the complexity of the timing system. The design goals are to:

- Share the Timing Receiver hardware resources
- Unify different underlying hardware.
- Prevent applications creating conflicting events
- Isolate applications from failures in other clients
- Monitor hardware status

Interprocess communication between clients and the SAFTlib process (saftd) is via the d-bus shared message bus.

This paper describes the hardware and software environment in which it is used and experiences in using SAFTlib in a production environment. The primary focus is on achieving the throughput and latency necessary to operate the FAIR accelerators whilst maintaining the flexibility and compatibility of the original SAFTlib design.

FAIR ACCELERATOR ENVIRONMENT

The FAIR project will include a complex of accelerators and a new control system is under development [2]. The CRYRING low-energy storage ring is being used to test and evaluate the control system before retrofitting the existing GSI infrastructure and equipping the new FAIR accelerators.

Timing Network

The FAIR timing system uses White Rabbit to distribute high precision timing events over a dedicated Ethernet-based network. The complexities of clock synchronization, signal latencies and network topology are abstracted from the users of the timing system. The high-level applications interact with the Data Master, which maintains a schedule of events and distributes them to Timing Receivers. Low-level applications interact with the Timing Receivers located close to the equipment. Equipment that controls a logically related set of accelerator components is collected into a Timing Group.

Frontend Controllers

The standard environment for the FAIR control system is the Scalable Control Unit [3]. It provides an Intel 64-bit CPU, Linux Operating System with realtime patches, an FPGA Timing Receiver connected via PCI-express and Wishbone. The timing software is also required to run on other systems with greater processing power for Beam Diagnostics, systems interfacing to hardware using a VME Bus, and systems with USB-connected Timing Receivers. The SAFTlib software aims to provide a standard interface across multiple platforms.

Timing Receivers

The Data Master sends event messages over the White Rabbit network shortly in advance of their planned execution time. The Timing Receiver hardware matches events against a set of conditions. The Event-Condition-Action (ECA) unit is responsible for generating actions from incoming Timing Events. For equipment with hard real-time requirements, hardware actions are used. These send signals over a variety of bus interfaces directly to equipment. An example is the trigger synchronization event that is used to start waveform generators. For events that can tolerate higher latency software actions can be used. For example, sequence start events signal that a device should be prepared for a new cycle and load data for a later hardware trigger, a gap event can signal that software has a period in which it may freely read the status of equipment.

Function Generators

Central to the performance investigation is the anticipated load from Function Generator units. These generate arbitrary waveforms from a set of polynomial coefficients and after D/A conversion control a variety of magnet power supplies. The polynomials describe the waveform in segments starting as 1ms in length. This

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA168

representation was chosen over a value sequence or coordinate representation to reduce the bandwidth requirements. The requirement to support existing equipment operating via MIL-bus forces the use of a lower bandwidth representation.

The polynomial sequence is checked to ensure it will result in a valid waveform before being sent to the function generator. The function generator hardware has a limited amount of memory and requires streaming from the saftd driver during longer output ramps. The driver must load a sufficient quantity of data into the function generator and arm it before the trigger event is received. The goals during initial testing were to supply 800 coefficient sets, 12 Function Generator channels in 25 ms. This beam preparation time is acceptable for the CRYRING tests. Further development will be required to make more use of the streaming mode of operation. Pre-loading the entire segment is however a good test of the data delivery system.

SOFTWARE ENVIRONMENT

The FAIR accelerators are controlled by custom Java-based applications using the LSA (LHC Software Architecture) framework [4] to manage the settings for individual devices to fulfil the needs of experiments. This upper layer generates settings for each device and a Timing Schedule that specifies to the Data Master when settings are to be applied. Applications may also receive feedback by polling or subscribing to devices.

Software on the front-end computers is developed using the Front End Support Architecture (FESA) – a framework developed in collaboration with CERN [5] for the development of C++ software. FESA software is developed to a set of guidelines that provide a standard interface that applications use to control and monitor various types of hardware. An executable FESA binary contains instances of a FESA class representing specific devices and exposing settings and acquisition values to applications. The details of which computer the software is running on and how many devices that computer is responsible for is largely hidden.

A typical FESA application might have:

- Software Event on Beam Preparation Event to load settings
- An equipment-specific Hardware Event to trigger pre-loaded settings
- A Software Event to read acquisition values from hardware
- A Timer for status updates

A FESA software program may be responsible for equipment in different Timing Groups and responding to different Timing Events. Additionally there may be multiple FESA binaries running on a single SCU. This gives rise to the requirement that multiple processes can control the Timing Receiver and listen to Events without disturbing each other.

SAFTLIB DESIGN

The interface to the timing receiver is managed by a single process – saftd. This process contains the interrupt handler and performs all communication to the timing hardware over the PCI-express bus.

Clients connect to saftd via the d-bus Interprocess Communication (IPC) standard. All clients request conditions: Hardware Conditions for timing event to hardware action and Software Conditions for timing event to software action. Saftd checks requests do not create conflicts between processes and compiles the set of conditions to a format that may be sent to the ECA unit.

The timing receiver uses interrupts to signal incoming data. The satfd interrupt handler is responsible for sorting interrupts from different channels, requesting data and forwarding it to the clients. By having only a single process receive the interrupts and use the Etherbone interface contention is avoided. (It is still possible for another process, such as a legacy application, to use the Etherbone interface and cause a conflict.)

Driver and Proxy Client Interface

Saftd provides an object-based interface for clients. Objects exist in the saftd process and can be accessed via d-bus IPC. Clients create Proxy objects that manage the details of communication with saftd. The interfaces are specified in XML format similar to the D-Bus reference implementation but specialised to the expected use cases. A code generation tool generates C++ code for both the driver service and proxy side hiding the complexities of interacting with the d-bus argument marshalling.

Drivers are part of the saftd process. Driver developers must complete the implementation behaviour defined by the interface definition and adapt to specific hardware.

Client applications link to a library containing Proxy classes that handle all interactions on the message bus. Clients can call methods, register for callbacks and read properties. Properties are cached locally by the proxy; changes to a property are sent on the bus to all registered proxies.

SAFTLIB PERFORMANCE

For hard real-time requirements that need latencies of microseconds or lower hardware events must be used. These can reliably operate at very low latency, but dedicated hardware has a longer development time and is more costly and less flexible. Reducing the latency of software events makes them useful to a wider set of tasks and increases the flexibility of the entire control system.

The delay between a Timing Event triggering and the hardware outputting to the SCU bus is of the order of 100 ps. A software interrupt in the Linux kernel has a delay of the order of 100 µs. Figure 1 shows a distribution

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA168

of latency measurements from hardware to the FPGA softcore, Linux kernel, User-mode driver and FESA processes.

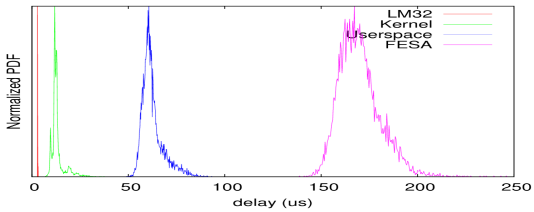

Figure 1: Latency measured by oscilloscope.

Table 1 shows the approximate 95th percentile latencies measured for the stages of a timing event notification message. Significant outliers were excluded and are discussed later in the prioritisation topic.

Table 1: Latencies (95th percentile): d-bus

Event	Time
Timing Event to Interrupt Handler	80 µs
Driver reads hardware	80 µs
D-bus (de)marshalling	200 µs
D-bus Transfer	600 µs
Proxy to FESA Eventsource	100 µs
FESA Action Scheduler	300 µs
Timing Event to FESA Action	1.5 ms

A particular performance issue is the handling of simultaneous end-of-cycle interrupts. The Function Generator unit can produce 12 simultaneous interrupts – delays whilst processing these cause the software and hardware state to disagree, which has caused faults where hardware has been incorrectly shown as busy and new data sets rejected.

The greatest gains can be achieved by optimising the d-bus IPC mechanism. Sending a d-bus message requires several context switches between user-space processes and conversion of data objects to and from the d-bus wire format. We investigated several areas looking for improvements in transfer speed and average and worst-case latency.

IMPROVMENTS INVESTIGATED

For most operations the size setting or acquisition data is so small that latency dominates. When sending the parameter sets for Function Generators, however, data throughput was found to be too slow.

D-bus File Descriptor

The d-bus standard allows processes to share file descriptors across the bus. As file descriptors are an index that only works in the context of a single process, file descriptors must be handled differently than other data types and the SAFTlib d-bus implementation had to be changed.

For the first investigation, file descriptors were used to open a pipe between saftd and the client proxy which then carried the waveform data.

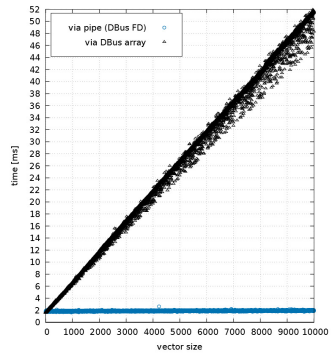

Figure 2: Data Transfer speed: d-bus vs. pipe.

As seen in Fig. 2, this significantly reduced the time taken for the data transfer, sufficiently to allow the control system to operate CRYRING successfully during early tests. However the setup time remains relatively high.

Grouping D-bus Operations

To take advantage of the higher throughput of the file descriptor and pipe mechanism, SAFTlib was extended with a Master Function Generator interface that is capable of sending commands to and aggregating replies from multiple function generator units in a single d-bus operation.

Reducing D-bus Load

The SAFTlib design includes many actions and metrics that may be used to monitor the status of the timing receiver. The Function Generator signals state transitions and data requests. Under high load it was found that registering to these signal could use enough extra CPU time to impair the performance of the critical events. Figure 3 shows the effect on performance of running an extra process that uses d-bus.

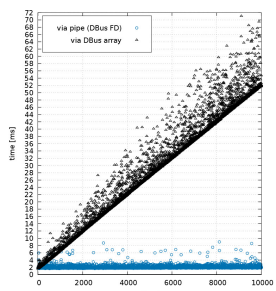

Figure 3: Data Transfer speed with concurrent d-bus users.

In the case of a 12-channel system, the Master Function Generator interface reduces the number of d-bus calls per cycle from 24 to 1 and the time taken from approx. 120 ms to under 25 ms.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUPHA168

Multi-Session File Descriptor

To avoid the overhead of transferring file descriptors via a d-bus transaction for each data exchange the d-bus mechanism can be used to create a persistent pipe. The driver and client must agree on a protocol for the data stream and more care must be taken. This makes the approach most suitable for the triggering of software events in the FESA framework. Using d-bus only for the initial negotiation phase produced the results in Table 2.

Table 2: Latencies (95th percentile): pipe

Event	Time
Timing Event to Interrupt Handler	80 µs
Driver reads hardware	80 µs
Pipe Transfer	50 µs
FESA Action Scheduler	350 µs
Timing Event to FESA Action	600 µs

Bypassing the d-bus mechanism gives significant performance gains but care must be taken to avoid conflicts. Interrupt-driven driver to client events are safe; client requests to the driver are no longer forced to be sequential by d-bus so access to hardware must be checked. Further work is planned in this area.

WORK IN PROGRESS

System Priorities

In a real-time system worst-case latency is important as late delivery of certain events is a failure. A single FESA binary may have tens of threads at many priority levels. Optimizing the relative priorities of the front-end processes will help ensure correct operation at higher CPU loads. Continuous high load needs to be avoided by limiting the tasks assigned to an SCU. Prioritisation can only mitigate the impact on critical services in the absence of a true real-time operating system.

Saftd Internal Priorities

The initial design of Saftlib has a single process and thread responsible for interrupt handling, servicing IPC requests and interacting with the hardware devices. This guarantees there is no contention on the Etherbone bus but does not deliver optimal performance. Introducing multithreading into a system introduces the complications of locks and shared resources. The first steps to a more parallel concept are the identification of critical pathways, identification of isolated components and identification of low-priority activities.

- High priority Software Action where delays cause failure.
- Generic Software Action where best-effort is acceptable.
- Idle Software Actions which may be re-scheduled until resources are available.

D-bus Prioritisation

A typical system may have time-critical actions to control hardware and less-critical actions for monitoring and diagnostics. D-bus itself does not support prioritisation of messages. However, there is a system bus and a user bus. This allows for two categories of message that can be kept separate.

Kernel D-bus

Since the d-bus daemon is a user process, a d-bus transaction will switch between kernel and user mode several times. Kernel d-bus incorporates the d-bus services into the Linux kernel and promises significant latency reductions. The requirement to backport kernel changes to the deployed Linux version has prevented further investigation at the moment.

Data Acquisition Support

Further improvements to the function generator hardware will include a Data Acquisition mode to provide feedback for the control system. It may be possible to integrate bi-directional streaming.

CONCLUSION

In this paper we have presented the current state of the SAFTlib Timing Interface, examined some of its performance characteristics and a number of ways of improving the throughput and latency characteristics. Improving the response speed of SAFTlib is required for the later stages of the FAIR project.

There are several ongoing investigations that aim to improve performance, both globally and targeted at specific use-cases, whilst retaining a flexible API that can handle multiple client processes and heterogeneous hardware.

REFERENCES

[1] D. Beck et al., “The New White Rabbit Based Timing System for the FAIR Facility”, FRIA01, Proceedings of PCaPAC (2012) Kolkata, India.

[2] R. Huhmann et al. The FAIR Control System - System Architecture and First Implementations, Proceedings of ICALEPCS2013, San Francisco, CA, USA

[3] S.Rauch et al. “Facility-wide Synchronization of Standard FAIR Equipment Controllers” Proceedings of PCaPAC 2012, Kolkata, India, WEPD48, p. 84

[4] J. Fitzek, R. Mueller, D. Ondreka, “Settings Management within the FAIR Control System based on the CERN LSA Framework”, Proceedings of PCaPAC 2010, Saskatoon, Saskatchewan, Canada, WEPL008, p. 63

[5] A. Schwinn, S. Matthies, D. Pfeiffer, M. Arruat,L. Fernandez, F. Locci “FESA3 the new Front-End Software Framework at CERN and the FAIR Facility” Proceedings of PCaPAC 2010, Saskatoon, Saskatchewan, Canada, WECOAA03, p. 22