Real-time performance issues in linux / microTCA

Kukhee Kim for microTCA support team in SLAC
SLAC National Accelerator Laboratory
Dec 12, 2013

Abstract

The microTCA platform has been selected for usage as the base platform for SLAC control systems for future designs and upgrades - along with embedded linux as the software platform. We have evaluated the microTCA and linux platform for usage in our timing, low-level RF, and BPM systems. We have found that the new platform brings challenges in interrupt handling, and scatter DMA.

Despite migration of much of the hard real-time functionality to the FPGA firmware level, the interrupt handling and its real-time performance are important factors for our control system as the software layer needs to deterministically process time critical functions driven by each interrupt. Linux has a long processing chain for the interrupt from the kernel to user space driver, and it also provides various methods of providing the interrupt notice to the user driver: signal and ioctl() with device file. Each method provides different performance. We are going to describe our experience for interrupt handling with regard to real-time performance.

In some cases DMA is also needed for applications which use fast digitizers. We have used traditional DMA for the real-time world previously, because most real-time OS(s) are based on a flat memory model. However, for linux based systems, it is not a flat memory model and we are not so lucky. We intend to use a scatter DMA engine for achieving real-time performance under linux. We have chosen the SIS8300 module from Struck for our low-level RF system and BPMs. The firmware from Struck did not support the scatter DMA, thus we had to allocate linear memory space in the kernel space for the DMA, and needed to implement a bounce buffer to copy the DMA data to the user space. This led to a lag in real-time performance. Thus, we had to implement a scatter DMA technique to avoid the bounce buffer and to allocate the DMA buffer directly into the user space. During the firmware upgrade, we learned that the following steps: configuring the DMA engine, re-arming the DMA and waiting for interrupt should be an atomic operation.

In this presentation we are going to discuss the details of our software experience with interrupt handling, and scatter DMA using the microTCA and linux platform.

Contents

- Software Environment Change: RTOS vs. Linux
- Priority Scheduling vs. Round Robin
- Signal & interrupt Handling
  - Event System
  - Asynchronous vs. Synchronous
- DMA Issues
  - LLRF System
  - Conventional DMA vs. Scatter DMA
  - Interrupt loss
- Conclusion

Software Environment Change

	RTOS (RTEMS, VxWorks)	Linux	Supplementary Work
Scheduling	Priority Base	Round Robin	Turn on RT priority for Kernel thread and User thread which are related with real-time performance
Memory Model	Flat Memory	Virtual Address	Bounce buffer in Kernel Space or Scatter DMA
Privilege	Same Privilege for Kernel and User application	Privilege for Kernel	System call, device file
I/O access	Memory mapped I/O can be accessed by user application	Kernel module can provide, - Memory mapped I/O(*1) - PCI special file in sysfs (*2) - File read/write - Ioctl()	Need kernel module, Need to use system call and device file (*1) does *not* need system call (*2) created by PCI core
Interrupt Handling	ISR can be a part of user application	ISR should be implemented in Kernel and Kernel module	Require a signal mechanism to notice to user space

Consideration for RT performance / RT priority

SLAC

- Use Linux RT preemptible patch – reduce interrupt latencies while kernel functions are executing
- EPICS base patch to use RT priority for application threads in the EPICS ioc (by Till Straumann)
- Implement “system()” command to execute shell script to set up RT priority for kernel threads

```
system("/bin/su root -c `pwd`/rtPrioritySetup.cmd")
```

```
/usr/bin/chrt -pf 95  `/bin/ps  -Leo pid,tid,rtprio,comm | /usr/bin/awk '/mrfevr/{printf $1}'`
/usr/bin/chrt -pf 90  `/bin/ps  -Leo pid,tid,rtprio,comm | /usr/bin/awk '/sis8300/{printf $1}'`
```

Adjust RT Priority / Scheduling for Kernel Thread @ linuxRT platform
with `chrt -pf <prio> <pid>`

irqHandler Thread → 1024

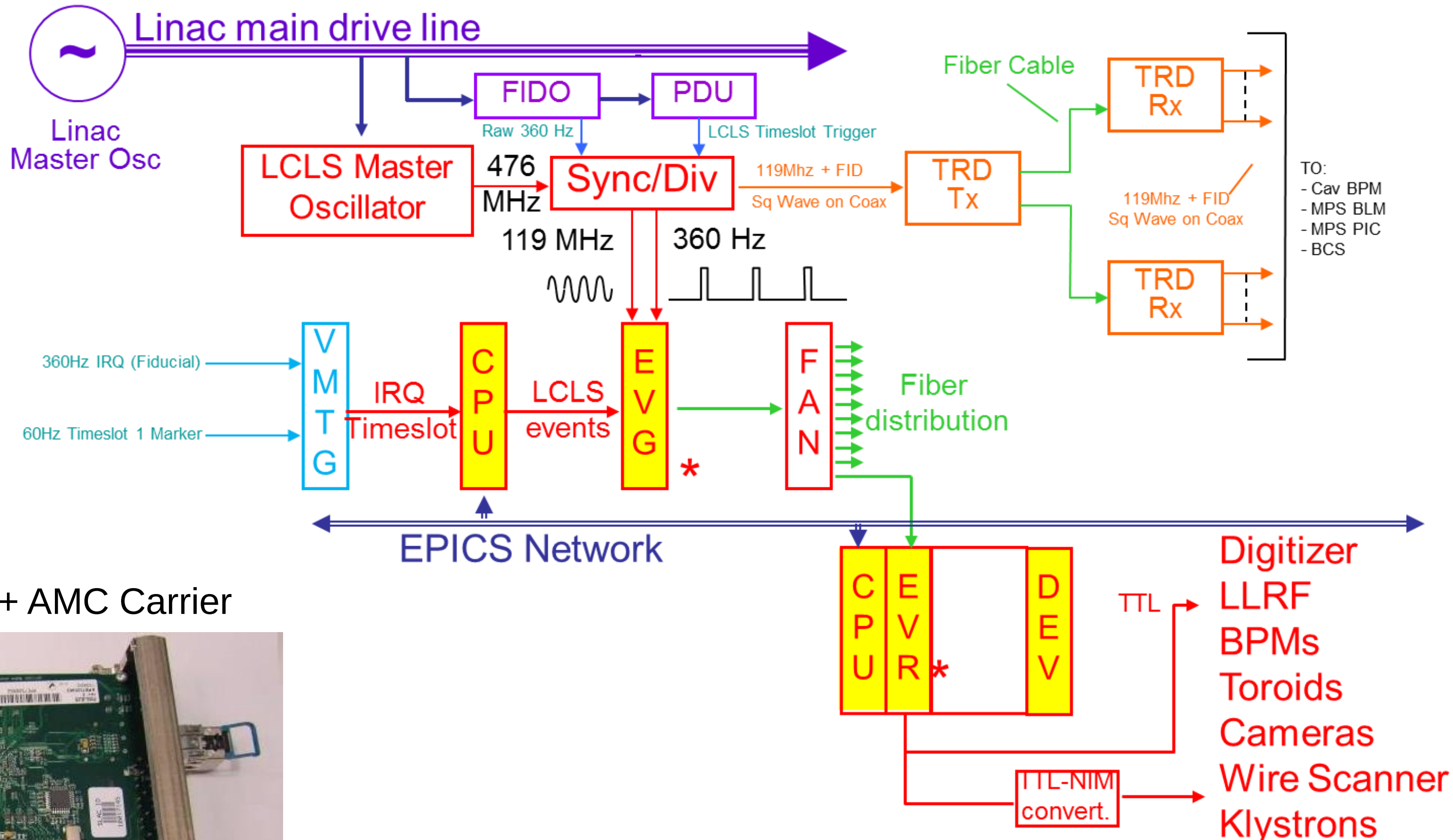
evrTask → 1031

evrRecord → 1032

Kernel Thread to handle the IRQ from EVR → 1023

```
$ ps  -Leo pid,ppid,tid,rtprio,stime,time,comm,wchan
  PID  PPID   TID RTPRIO STIME     TIME COMMAND         WCHAN
 1019     1  1019      - 14:58 00:00:00 screen          poll_schedule_timeout
 1020  1019  1020      - 14:58 00:00:00 LLRFControl     n_tty_read
 1020  1019  1022     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1024     94 14:58 00:00:39 LLRFControl     sigtimedwait
 1020  1019  1025     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1026     69 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1027     58 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1028     63 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1029     70 14:58 00:00:01 LLRFControl     futex_wait_queue_me
 1020  1019  1030     50 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1031     90 14:58 00:00:08 LLRFControl     futex_wait_queue_me
 1020  1019  1032     79 14:58 00:00:09 LLRFControl     futex_wait_queue_me
 1020  1019  1033     59 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1034     69 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1035     59 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1036     60 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1037     61 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1038     62 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1039     63 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1040     64 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1041     65 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1042     16 14:58 00:00:00 LLRFControl     inet_csk_accept
 1020  1019  1043     14 14:58 00:00:00 LLRFControl     hrtimer_nanosleep
 1020  1019  1044     12 14:58 00:00:00 LLRFControl     skb_recv_datagram
          9  1045     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
          9  1046     51 14:58 00:00:01 LLRFControl     futex_wait_queue_me
          9  1047     53 14:58 00:00:00 LLRFControl     skb_recv_datagram
          9  1049     51 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1052     18 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1053     20 14:58 00:00:00 LLRFControl     sk_wait_data
 1020  1019  1054     18 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1055     20 14:58 00:00:00 LLRFControl     sk_wait_data
 1023     2  1023     99 14:58 00:00:13 irq/19-mrfevr   irq_thread
```

Event System for MicroTCA Platform

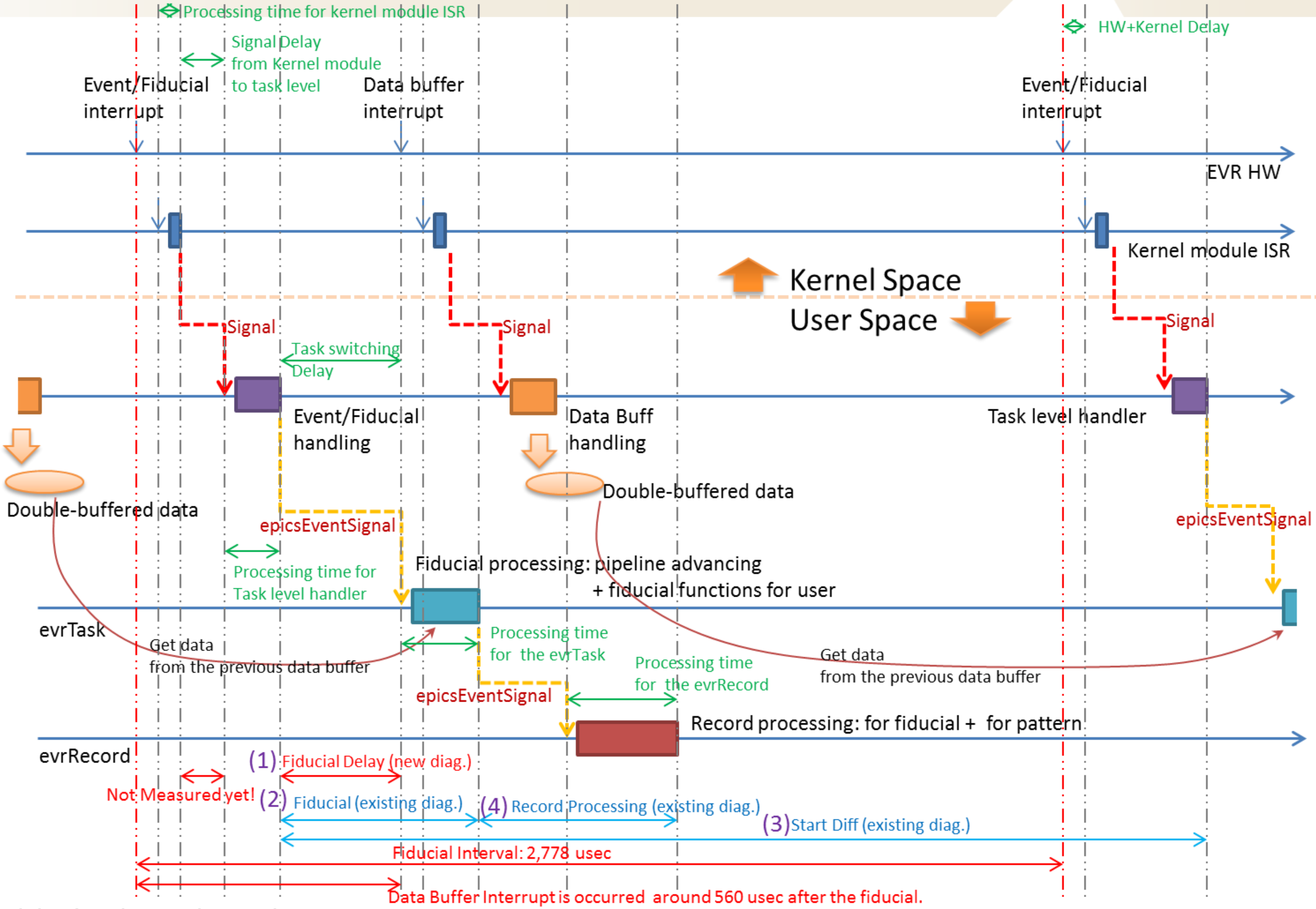


MRF EVR (PMC) + AMC Carrier

Event Receiver (EVR) and Event Module

- MRF EVR hardware and Event module (software) are a common part of applications such as LLRF, BPM, and etc
- EVR provides hardware triggers and also provides interrupts for software event (fiducial/events) and timing pattern (data interrupt)
- Fiducial/Event interrupt
  - Drive software processing for the event module
  - Generates software event
- Data interrupt
  - Provide 192 bits timing pattern, pulseID embedded timestamp
  - Essential for Beam Synchronous Acquisition (BSA) which is an entire facility wide data acquisition system
  - Make recognize which data is for which beam pulse

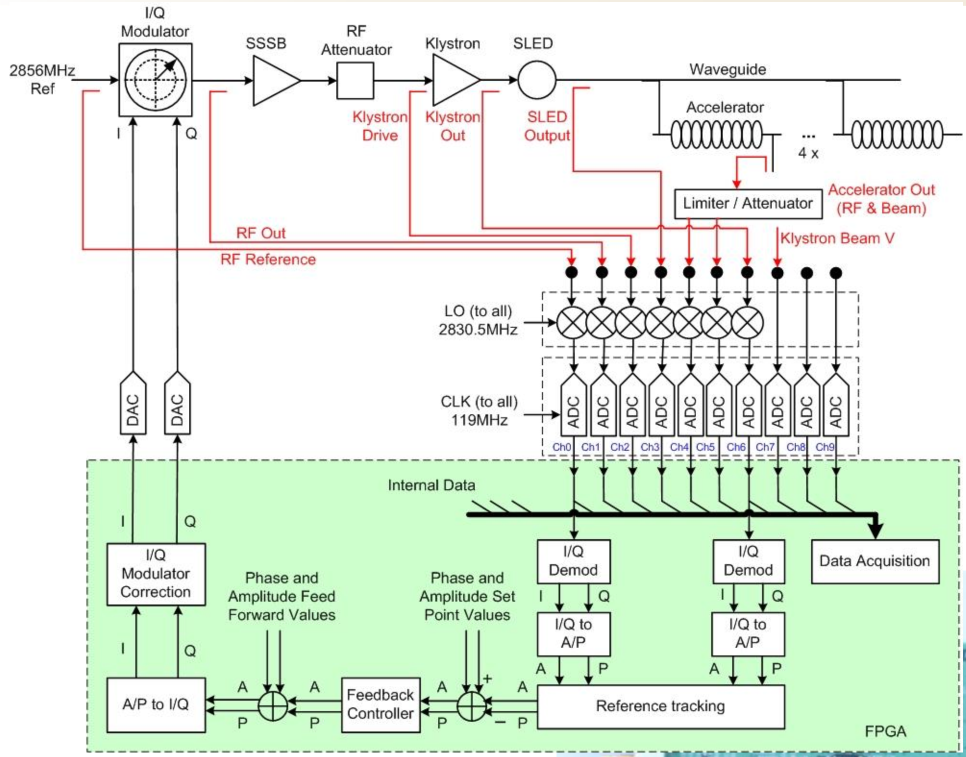
Timeline for the EVR interrupt handling



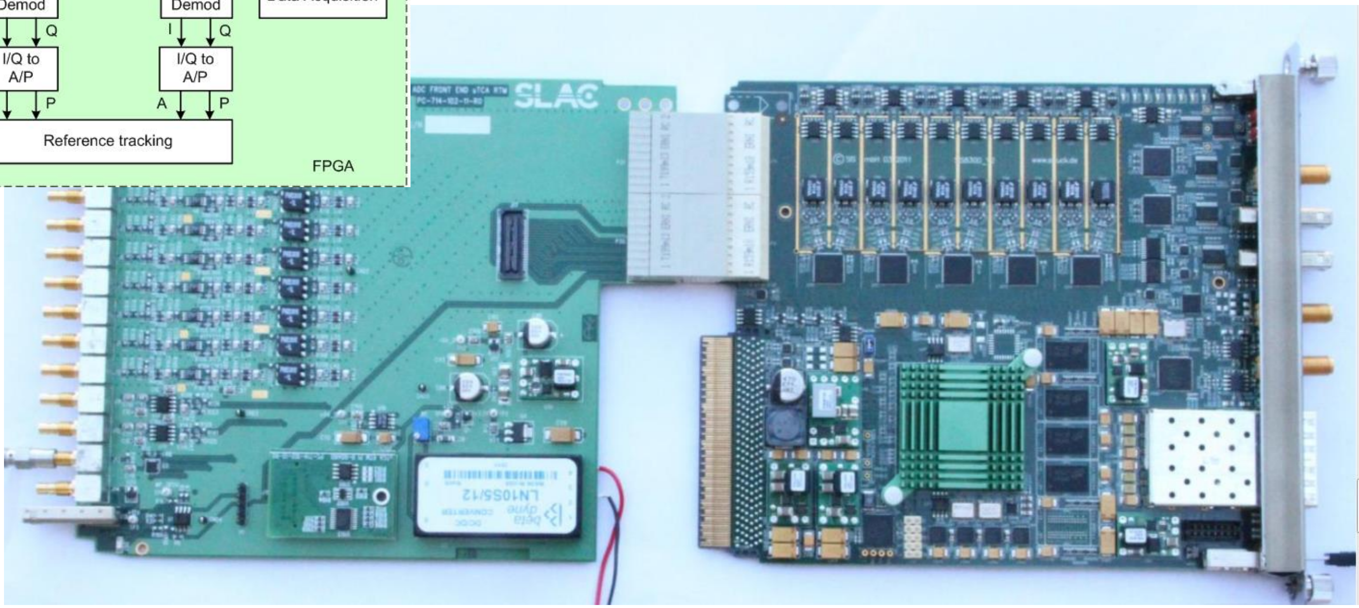
Interrupt Handling for EVR
Signal Handling: Asynchronous vs. Synchronous

- Vendor's kernel module generates POSIX signal to notify interrupt to user space driver
- User space driver in event module handles the signal with a signal handler
  - During 10 hours monitoring under normal CPU load (<50%)
    - Average interrupt delay: ~ 30 usec
    - Maximum jitter for interrupt: > 6 msec
    - RT Violation counter: > 160 times
  - Asynchronous signal handling
  - Scenario
    - The signal handler could be run on an arbitrary thread context
    - And, the context which runs the signal handler, could be preempted by higher priority thread
- Improvement for the User space driver
  - Synchronous signal handling
  - Make new thread which has higher priority than other thread in the application, and waits the signal directly
    - Average interrupt delay: ~ 13 usec
    - Maximum jitter for interrupt: ~ 30 usec
    - RT Violation counter: 0

MicroTCA LLRF

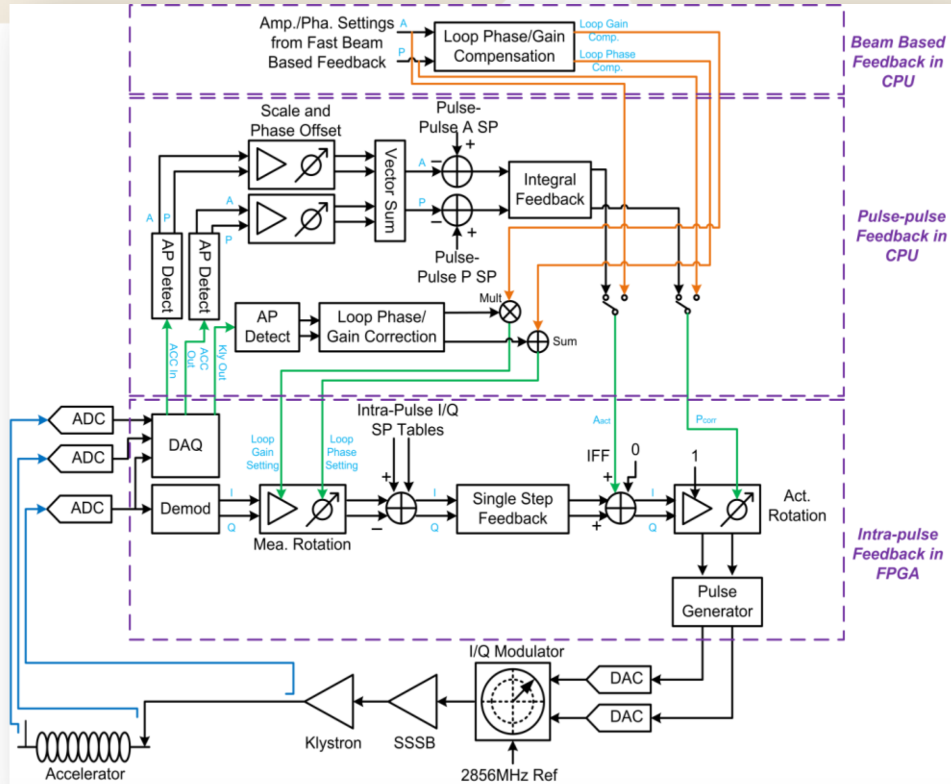


SIS8300 + RF RTM



Performance Issues on LLRF application

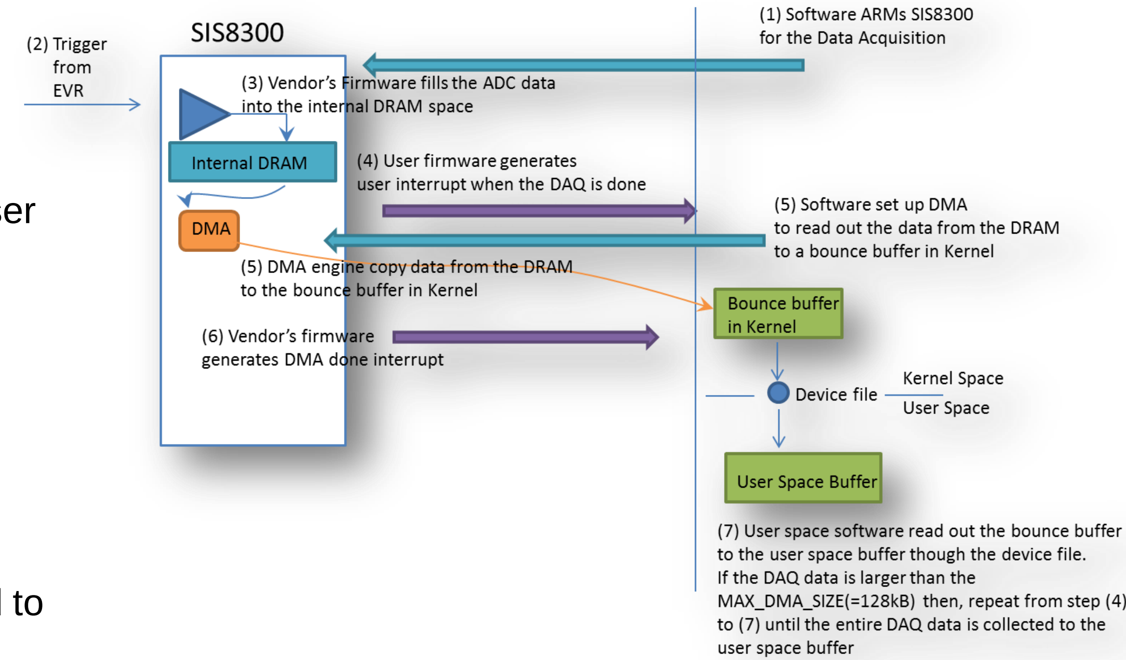
- Despite migration of much of hard real-time functionality to the FPGA firmware level, the interrupt handling and its real-time performance are still important
  - Intra-pulse feedback is implemented in the FPGA level (few usec response)
  - Pulse-Pulse feedback and Beam based feedback are implemented in software (msec response)
- Performance measurement for proto-type software
  - According to 120Hz beam rate, maximum 8.3 msec time budget, but recommended 2.7 msec (360Hz timeslot resolution) budget to consider other software activities: communication with higher level feedback, pattern awareness operation, and etc
  - Already reached budget with a single module, but we are going to put multiple modules up to 8 modules in a chassis
  - Definitely, need to improve the performance




```
epics> dbior drvPerfMeasure 3
Driver: drvPerfMeasure
Estimated Clock Speed: 1500.200532 MHz
Driver has 9 measurement point(s) now...
--------------------------------------------------------------------------------------------
   Node name      Enb        Counter   Time(usec)     Minimum      Maximum         Description
--------------------------------------------------------------------------------------------
    MAINLOOP       1        75120 1970.36458590 1715.28468702 2565.31704789 main thread loop time
         DAQ       1        75158 739.90508357 641.91418378 1056.72872805 get all DAQ data in the main
      PHCTRL       1        75157 419.01398286 411.12503752 494.66986857 phase control block in the mai
  PHCTRLDATA       1        75157 417.50818417 409.73922278 486.73892885   +data get for phase control
      NETDAQ       1        75179   2.83762064   2.55565834  18.90347283         +just net DAQ getting
      RFDEMO       1        75157 413.68669505 405.80374891 466.58762283         +RF demodulation calc
   PHCTRLCALC      1        75179   0.46793744   0.30595910  11.60844809   +calculation for the phase c
      WFDIAG       1        75175 102.58028625  16.16783855 287.57755433 diag. for waveform in the mair
     OTHDIAG       1        75179  23.05491783  20.62524265 116.01249052 diag. for others in the main t
--------------------------------------------------------------------------------------------
```

Analysis for the original implementation (SIS firmware)

- Too much handshakes to complete the data transfer
  - Arming DAQ, DAQ done interrupt
  - Initiate DMA transfer for a channel, DMA done, copy bounce buffer to user space
  - Initiate DMA again for next channel, and so on…
- Doesn't support scatter DMA
  - Requires physically continuous memory
  - Bounce buffer in Kernel space, need to copy to user space
- Too much interrupts (~1.44kHz per module)

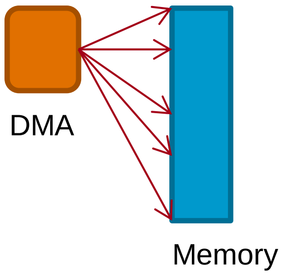


An extra read out

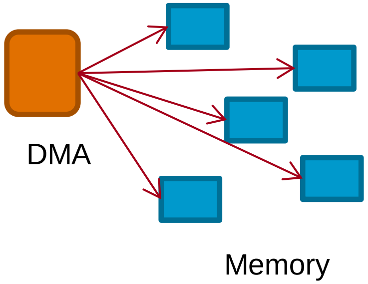
120Hz + 120Hz/Channel x (10 Channels + 1) = 1440Hz

DAQ Done Interrupt — DMA Done Interrupt — Number of Channels in a module

Conventional DMA vs. Scatter DMA



- Conventional DMA
  - requires physically contiguous memory
  - requires kernel memory such as kernel bounce buffer
  - memory copy to user space makes performance dragging
  - use mmap() to user space to reduce the performance dragging
  - but the kernel memory is expensive resource

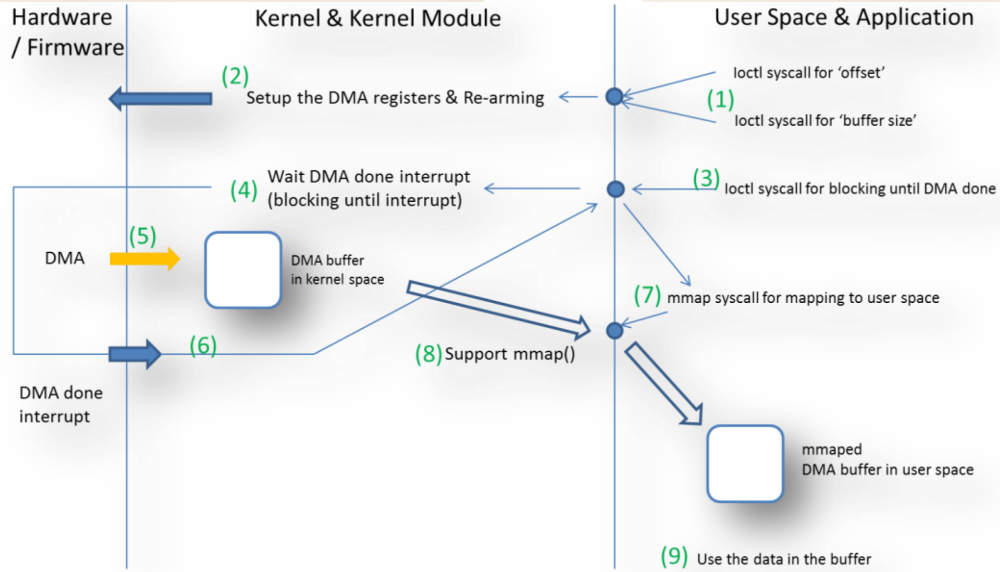


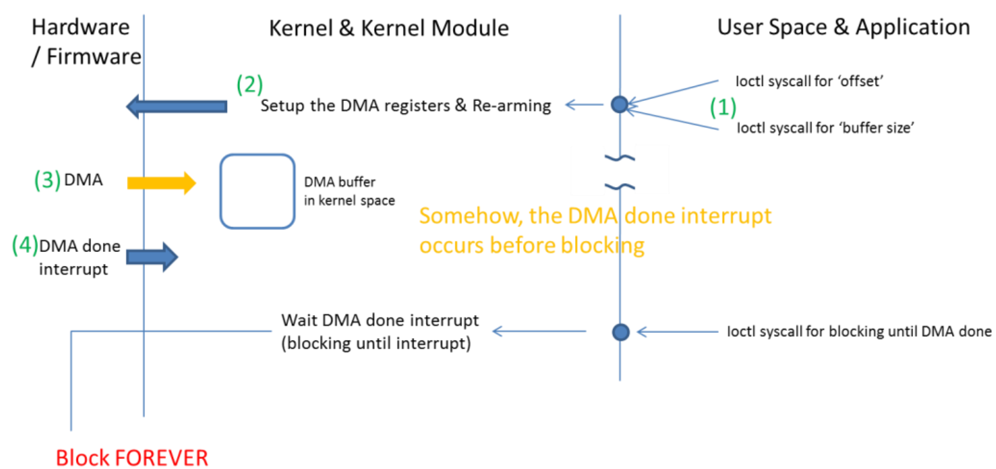
- Scatter DMA
  - Does not require contiguous memory
  - Does not require kernel space buffer
  - can make direct copy to user space memory (virtual memory)
  - need software effort to build scatter list
  - If we use pre-allocated memory buffer then,
    the effort for scatter list is just initial overhead

Improvements (eicSys firmware, current implementation)

- Improvements
  - DMA initiated by DAQ done event from the base firmware instead of software
  - DMA transfers entire data instead of single channel
- Remained issue
  - Still using kernel bounce buffer
    need to upgrade to the scatter DMA
  - Multiple system calls
    - Dragging performance
    - Missing interrupt (caused by non-atomic operation)

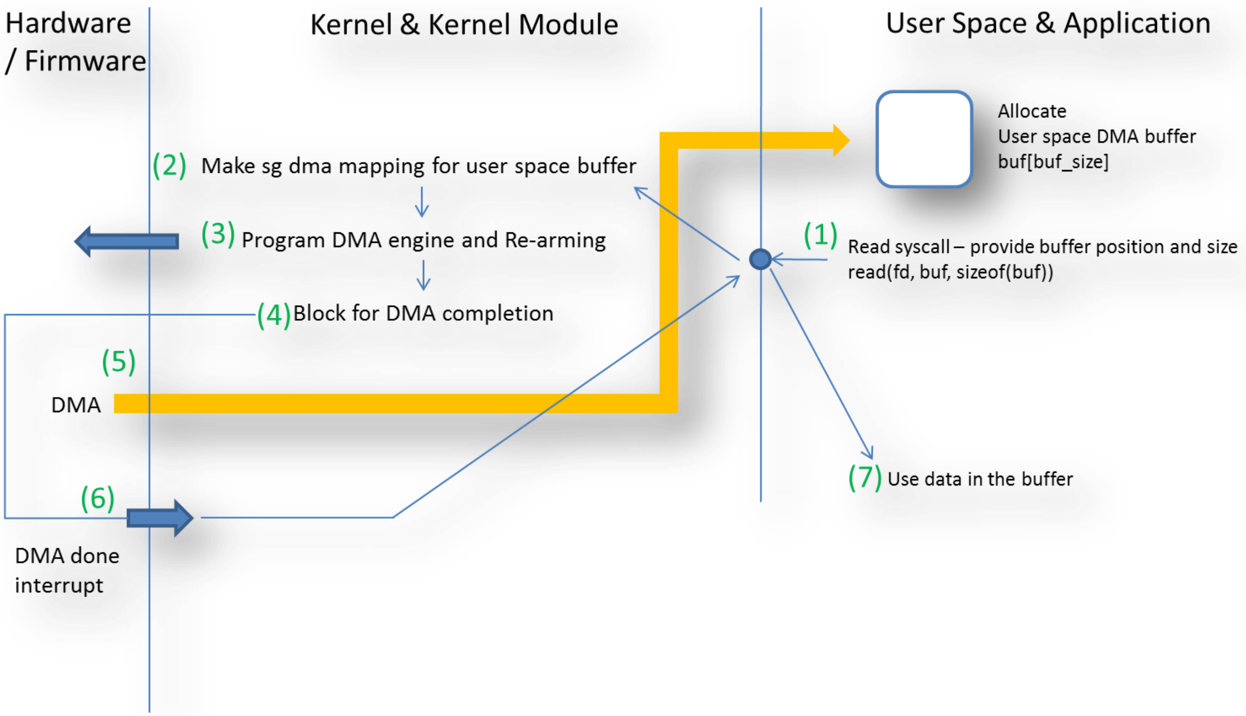
    need to make single system call





Improvement Plan (eicSys firmware)

- Atomic operation required
  - Single system call arms DAQ and waits DMA done interrupt
  - To avoid interrupt loss
  - Reduce software work load



Conclusion

- Compare RTOS vs. Linux
- Use RT priority for kernel thread and user thread which are related with the real-time processing
- Implement system escape into EPICS to adjust RT priority for kernel threads
- Use Synchronous signal handling instead of Asynchronous
- Implement performance measurement tool as an EPICS driver, it provides sub-micro second resolution delay measurements and statistics. It is almost non-invasive tool for real-time application.
- Make handshake between kernel module and user space driver as simple as possible, also simple handshake between firmware and software
- Make atomic system call for arming and waiting interrupt to avoid interrupt loss
- Use scatter DMA to avoid kernel bounce buffer overhead