Tweaking Device Drivers for Achieving Real-time Performance in Embedded Systems Using RT Linux

2023-06-29, Prague

Vaishnav Achath
Vignesh Raghavendra
Keerthy J
About us | TI Processors and Open source

Decades of contribution and collaboration

Ingrained culture to give back to the community

Upstream FIRST!

Focus on long term, sustainable and quality products

Upstream and opensource ecosystem in device architecture

Upstream FIRST mentality!
Speakers | Intro

**Vaishnav Achath**, Software Engineer at Texas Instruments India. Vaishnav works on Linux Kernel and U-Boot as part of the Texas Instruments Linux development team. Vaishnav is also a maintainer for TI platforms in Zephyr RTOS.

**Vignesh Raghavendra**, Software Engineer at Texas Instruments India. Vignesh co-maintains the TI's arm64 SoCs in mainline along with a few drivers. He has been contributing to Linux Kernel and U-Boot since 2014 as part of Texas Instruments' Linux development team.

**Keerthy J**, is a SW Application Engineer with Texas Instruments Inc., as part of this role he primarily interacts with customers regarding their use cases in automotive and industrial applications.
Overview

• Problem Statement
• RT Linux – expectations.
• Generic tweaking options in RT Linux.
• SPI Client support and Challenges.
• Subsystem-level tweaking options for the predictability.
• Profiling and improving device drivers.
  – Case study: SPI + DMA
• Results and best practices.
Problem Statement | SPI

• **DRA821** is a heterogenous multicore application processor from Texas Instruments with Dual Arm Cortex-A72 and quad Cortex-R5F.

• DRA821 consists of 11 instances of MCSPI (Multi-channel Serial Peripheral Interface), MCSPI can operate in:
  - Full duplex Host mode, Target mode.
  - DMA mode (system DMA).

Problem:

• MCSPI in Target mode on RT-Linux in DMA mode provides unreliable performance and observed packet loss issues (800 ms/128-Byte full duplex).
• MCSPI in Host mode on RT-Linux in DMA mode results in non-deterministic performance/latency spikes (5 ms).
RT Linux | User Expectations

- RT Linux is increasingly becoming popular for real-time embedded applications due to the:
  - Versatility and flexibility provided by the High-Level OS.
  - Real-time capabilities and deterministic performance.
- Demand deterministic performance, when interfacing with external peripherals.
- Predictable performance is critical in cases where the RT Linux host is acting in target mode and an external device initiates and controls the communication.
- CAN, SPI, and UART are popular media communication media with RT requirements.
Practical usage | Host and Target

- Industrial robotic application | SPI Host
  - LIDAR sensors that can be interfaced to host through SPI, in such use-cases deterministic capture of sensor data for critical for the system
  - Robotic motor control (non-safety critical)

- Device management interface | SPI target
  - External supervisor interaction.
  - Control, monitor, and power state.
  - Firmware management.
  - Watchdog-like ping-pong.
RT Linux tuning | Where to start

- cyclictest
  - measure a thread's intended wake-up time and the time at which it actually wakes up
  - Provides overheads to take into account for full RT stack

```bash
cyclictest --mlockall --smp --priority=99 --interval=200 --loops=100000000
```
RT Linux tuning | cyclic test

HW: TI AM625 SoC

RT Linux | Tools

- lmbench – DDR bandwidth and latency analysis
  - Provides bounds on HW capability and any bottlenecks in HW leading latency excursions.

- Tracing tools: ftrace, perf
  https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/start

- rtla-timerlat
  - https://docs.kernel.org/tools/rtl/rtl-timerlat.html
RT Linux | System Tuning

• RT Linux offers multiple tuning knobs:
  – Kernel config options (PM, CPUFREQ, CONFIG_DEBUG*, and any unneeded configs)
  – Real-time policies (SCHED_FIFO etc)
  – Real-time kernel thread priorities
  – Task partitioning (isolcpus, taskset, IRQ affinity, load balancing)
  – Modifying real-time priority of application programs etc. (nice, chrt)
  – Latency optimizations:
    https://wiki.linuxfoundation.org/realtime/documentation/howto/debugging/start

• Even with all the tweaking options available:
  – device drivers need optimization for deterministic performance, to benefit from the RT Linux kernel.
  – Debugging real-time issues with device drivers can be complex especially when multiple subsystems interact. (E.g. SPI + DMA + user application)
SPI Host / Target | Linux Support

• SPI is a full-duplex synchronous serial communications module. SPI provides a cost-effective way to interface serially with external peripherals.

• SPI operates in either Host mode or Target mode.
  – In host mode, the SPI controller generates the synchronous communication clock (SCLK) and initiates the transaction.
  – In target mode, the controller receives the SCLK as input and completes the transaction initiated by the external host, the client has no control over the transaction start, speed, etc.

• SPI Target mode support is available in the Linux kernel SPI subsystem from v4.13 onwards and 6 SPI controller drivers support Target mode as of v6.4.
SPI Target mode | Challenges

- Full duplex: simultaneous transmit and receive.
- Host has control: target implementation demands hard real-time.
- Target must have prepared the TX transaction before the Host starts the transfer.
- Target response cannot depend on the Host request in the same message.
- Lack of **standard HW flow control** mechanism.

![UART Flow control](image1)

![I2C Clock stretching](image2)

Courtesy: [Linux as an SPI Target](https://www.guyutech.com/), Geert Uytterhoeven
Analyzing subsystem | SPI framework

- **spidev / client driver**
- **SPI Thread (spi_pump_message())**
  - spi_message 1
  - spi_message 2
  - spi_message 3
  - spi_message 4
  - spi_message 5
- **SPI driver**
- **IRQ / Completion**
- **SPI Controller Hardware**
- **Workqueue / Thread**
- **DMA Queues (driver specific)**
- **IRQ / bottom half / DMA completion queues**
Full call stack | SPI and DMA interaction

Call Stack for SPI with DMA on DRA8x
### Kernelshark Traces | Actual trace screenshots

- RX callback within 2ms
- TX callback takes 850ms

**Table:**

<table>
<thead>
<tr>
<th>CP Time</th>
<th>Task</th>
<th>PID</th>
<th>Latency</th>
<th>Event</th>
<th>Info</th>
</tr>
</thead>
</table>
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi prepare message | omap2 mcspi prepare message
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_can_dma | omap2 mcspi_can_dma
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_set cs | omap2 mcspi_set cs
| 1 1482.691 | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_transfer one | omap2 mcspi_transfer one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_set_enable | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_setup_transfer | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_set_enable | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_setup_transfer | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_set_enable | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_set_enable | omap2 mcspi_transfer_one
| ...     | spittest | 1312 | ...0  | ftrace/function omap2 mcspi_can_dma | omap2 mcspi_can_dma
| ...     | irq/38-2140600 | 160 | ...2  | ftrace/function omap2 mcspi irq handler | omap2 mcspi_irq_handler
| ...     | irq/78-31150800 | 158 | ...5  | ftrace/function omap2 mcspi rx callback | omap2 mcspi_rx_callback
| 1 1483.542 | ksoftirqd/1 | 22  | ...5  | ftrace/function omap2 mcspi tx callback | omap2 mcspi_tx_callback

**Notes:**

- SPI in Target mode
- With DMA
Stack Latency | Trace summary

**Userspace**
- SPI_IOC_MESSAGE_IOCTL
- SPI_IOC_MESSAGE_IOCTL Complete

**Linux Kernel SPI**
- spidev_ioctl()
- spi_transfer_one_message()
- omap2_mcspi_setup()
- omap2_mcspi_transfer_one()
- omap2_mcspi_tx_dma()
- omap2_mcspi_rx_dma()
- mcspi_wait_for_completion (DMA)

**Linux Kernel DMAENGINE**
- dmaengine_config()
- dmaengine_submit()
- dmaengine_async_issue_pending()
- udma_push_to_ring
- udma_check_tx_completion

**MCSP Controller**
- Controller Setup
- Shift out TX, Shift in RX.
- Trigger DMA Read Event

**Timeline**
- 800 ms
- 200 us
- 350 us
- 800 ms
Tweaking options | SPI subsystem

- SPI Controller configuration can indicate if this controller should run the message pump thread with high (real-time) priority.

- SPI devices can also request real-time priority during transaction setup which will modify the controller message pump thread to real-time priority.

- This tweaking option might also not help in cases where the controller driver is not optimized.
SPI Controller driver | Analysis

- In the MCSP controller driver, with DMA enabled, we perform the SPI controller setup, queue the TX and RX DMA operations once the message is transferred, and wait for DMA completion.

Observations

- First transaction initiated from userspace never has data mismatch/latency issues in client mode.
- When multiple transactions are queued from userspace if the delay between each transaction is small (<100 ms), client packets RX and TX are corrupted in client mode.
- In target mode
DMA driver analysis | Finding Bottlenecks

• Typical Driver design to Queue DMA transfers
  – Submit xfer(): dmaengine_submit(), dmaengine_async_issue_pending()
  – wait_for_completion(): Wait for transfer->callback() =>
    dmaengine completion callback

• Tasklet calls driver’s completion callback
  – Per channel tasklet scheduled on completion
  – Multiple tasklets (depending on channel usage) can end up with similar priorities
  – Callback function needs to be light-weight for better response
  – Sched priority of each tasklets needs to be tuned

• DMA Subsystem on DRA8x/K3 SoCs have a Network on Chip arch
  – Centralized DMA talks to mini DMAs closer to peripherals (k3-udma.c)
  – Deferred workqueue handler (udma_check_tx_completion()) checks the remote DMA
    states to ensure the pipe is flushed before calling driver callbacks

• Above tasklet and deferred workqueue causes all the jitter in the SPI transfer RT
  path
Bottleneck | Analysis and Fixes

• Latency graphs indicated 3 issues
  – spi_pump_message
  – DMA tasklet for signaling completion
  – DMA completion / threaded IRQ handler inside DMA driver
    (udma_check_tx_completion())

• Solution
  – Setup spi_pump_message as RT (spi_controller->rt = true)
  – Set DMA tasklet as realtime and higher priority relative to other task pool
  – Convert workqueue within DMA driver to be RT priority

• This reduces the jitter and brings in determinism
  – Did not meet overall latency goal as even with tasklets, there was enough contention
    for large excursions.
Driver updates | Eliminate DMA tasklets from RT path

- McSPI controller provides a word count interrupt (End of Word), end of transfer
- Use McSPI to packetize transfers: provides interrupt after programmed number of words are clocked in/out.
- Use the same for continuing to the next transaction to avoid delays in the tasklets.
- Can be used if host and target use fixed-size packets and negotiated in advance.
- Multiple packets are needed in case of variable size.
Enhancements | GPIO-based flow control

- Bounded latency possible with RT Linux
  - During high IRQ and RT load conditions, the performance of the SPI client deteriorates
  - For non-RT systems, implement GPIO-based flow control
  - Single RDY GPIO was implemented to achieve reliability requirements

![Diagram of GPIO-based flow control mechanism]

- Userspace initiates/queues transaction
- MCSPI driver performs transaction setup and queues DMA request and then signals RDY
- Primary SPI controller waits for the rising edge of the RDY signal
- Client de-asserts RDY at the end of the transaction
After optimization | kernel shark trace

Completion within 2.9 ms
Solution
The following flow diagram shows the complete flow of operations and optimizations made with GPIO-based flow control for a full duplex transfer:

In the previous flow, the MCSPI driver would wait for a callback from UDMA to proceed further.

In the previous flow, the UDMA driver would perform the byte count calculation in deferred to understand the completion.

Round trip delay with improvements = 800us (earlier ~850 ms)
## Results before and after

### Target Mode

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>After Optimization</th>
<th>Before Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 byte (full duplex)</td>
<td>800 us per transaction</td>
<td>800 ms per transaction</td>
</tr>
<tr>
<td>160 byte (full duplex)</td>
<td>800 us per transaction</td>
<td>800 ms per transaction</td>
</tr>
<tr>
<td>128 byte (full duplex) with flow control under 99% CPU load stress</td>
<td>2 ms per transaction</td>
<td>800 ms per transaction</td>
</tr>
<tr>
<td>640 Byte (full duplex) with flow control</td>
<td>2 ms per transaction</td>
<td>800 ms per transaction</td>
</tr>
</tbody>
</table>

### Host Mode

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>After Optimization</th>
<th>Before Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>160 byte (full duplex)</td>
<td>Peak latency 400 us</td>
<td>Peak latency as high as 5 ms.</td>
</tr>
</tbody>
</table>
• Tune the system level latency first
  – cyclicstest, rtla-timerlat, lmbench with the minimal system (no drivers, features, etc).

• Keep the IRQ-off state to a minimum.

• Beware of priority inversion-like situations.
  – RT task depending on low priority workqueues, threads to proceed further.
  – Multi-level workqueues may kill the determinism.

• Complex SW state machines make it harder to avoid latency
  – Rely on HW state as much as possible.
  – Feed back to HW designers if such support is missing.
References

- **Linux as an SPI Target**, Geert Uytterhoeven, FOSSDEM 2018.
- **Adventures In Real-Time Performance Tuning**, Frank Rowand.
- [https://www.linkedin.com/pulse/checklist-real-time-applications-linux-linutronix/](https://www.linkedin.com/pulse/checklist-real-time-applications-linux-linutronix/)
  - Linuxtronix,
  
- Details and Instructions:
  - [https://github.com/vaishnavachath/elc-eoss23-rtlinux](https://github.com/vaishnavachath/elc-eoss23-rtlinux)
Credits and Acknowledgement

• Texas Instruments Inc.
• The Linux Foundation.
Q&A

• Contact Information:
  – Vaishnav Achath <vaishnav.a@ti.com>
  – Vignesh Raghavendra <vigneshr@ti.com>
  – Keerthy J <keerthy@ti.com>

• Also on IRC @ libera.chat #linux-ti

Learn more about TI products

– https://www.ti.com/linux
– http://opensource.ti.com/
– https://www.ti.com/processors
– https://www.ti.com/edgeai