Debugging ARM kernels using NMI/FIQ

HKG15-302

Presented by

Daniel Thompson
STMicroelectronics

Date

Feb 2015

Event

Linaro Connect HKG15

What this talk will cover

Look at NMI on x86 and FIQ on ARM

Review the use of NMI for kernel debugging

Discuss some practical issues

- TrustZone, ARMv8, status, kernel config

Demo!

(And a free bonus extra if there’s time)

NMI in x86

On x86 NMI has a long heritage as a debug tool

- Early PCs used it to report hardware faults such as memory parity errors
- Modern servers may have a physical NMI button on the front panel
- Watchdogs can be routed to NMI rather than reset
- Performance counters are hooked directly to local APIC

APIC allows flexible routing to/from NMI

- Hard to exploit on PCs due to unpredictable interrupt sharing

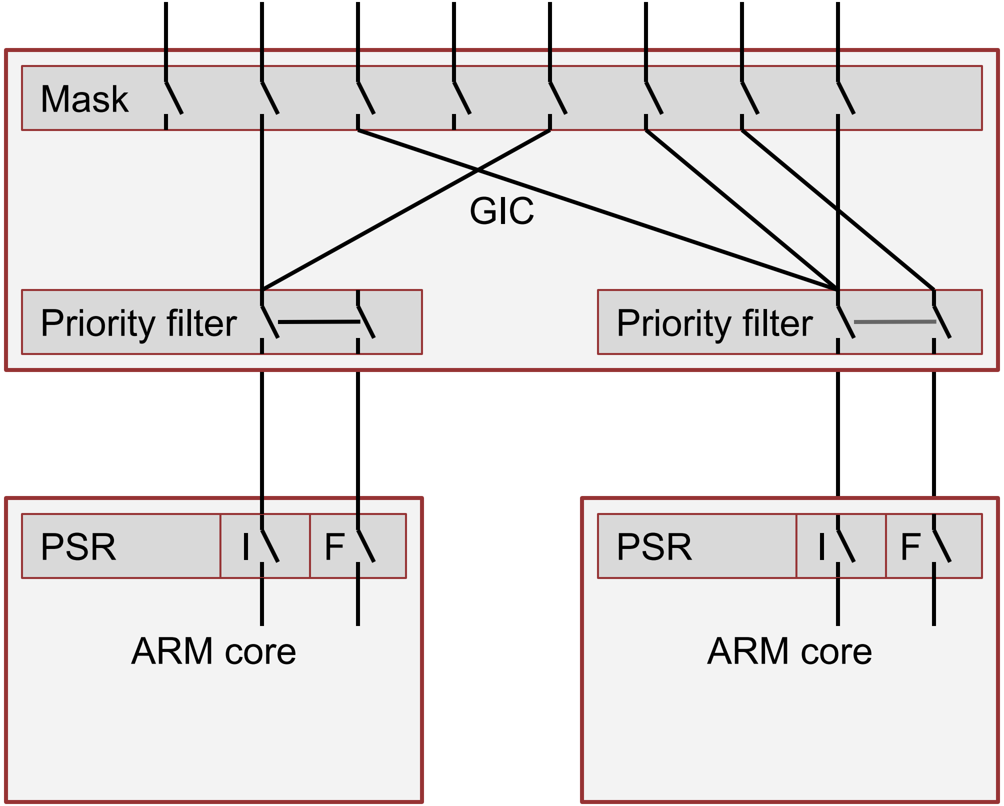
FIQ on ARM

Fast Interrupt reQuest

- A thirty year old trick to avoid putting a DMA chip into the Archimedes
- Separate mask bit in PSR
- Five extra banked registers allow stackless software DMA handlers
  - `floppydma.S, ssi-fiq.S, spi-s3c24xx-fiq.S, ams-delta-fiq-handler.S`
- socfpga devs. often want to exploit this with custom coded handlers

FIQ isn’t non-maskable but in practice is never-masked (which has the same acronym)

FIQ on ARM


Mask
GIC
Priority filter
Priority filter
PSR
I
F
PSR
I
F
ARM core
ARM core

Registers on ARM

r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8_fiq				
r9	r9_fiq				
r10	r10_fiq				
r11	f11_fiq				
r12	r12_fiq				
r13 (sp)	r13_fiq	r13_irq	r13_abt	r13_und	r13_svc
r14 (lr)	r14_fiq	r14_irq	r14_abt	r14_und	r14_svc
r15 (pc)					

FIQ in Linux

Historically used to implement software DMA

Supported by very simple API

- enable/disable
- reserve/release
- populate-banked-registers-on-calling-cpu
- please-memcpy-my-handler-into-vector-table

The new default FIQ handler for ARM

Saves register state, switches to SVC mode (for supervisor stack) and runs a C function

Intended to be the primary handler for NMI-like use cases (too heavyweight for s/ware DMA)

Almost too easy...

Gotchas

Locks are (almost) always unsafe from NMI

So...

Everything you do must be lockless
Printing to console is unsafe
Waking up threads is unsafe
Queuing tasklets is unsafe

...

Gotchas

Locks are (almost) always unsafe from NMI

So...

Everything you do must be lockless
Printing to console is unsafe
Waking up threads is unsafe
Queuing tasklets is unsafe

Top tip: `irq_queue_work()` can be used to defer work

Gremlins

```
.macro restore_user_regs, fast = 0, offset = 0
ldr    r1, [sp, #\offset + S_PSR] @ get calling cpsr
ldr    lr, [sp, #\offset + S_PC]! @ get pc
msr    spsr_cxsf, r1              @ save in spsr_svc
.if\fast
ldmdb  sp, {r1 - lr}^             @ get calling r1 - lr
.else
ldmdb  sp, {r0 - lr}^             @ get calling r0 - lr
.endif
mov    r0, r0    @ ARMv5T and earlier require a nop here
add    sp, sp, #S_FRAME_SIZE - S_PC
movs   pc, lr    @ return & move spsr_svc into cpsr
.endm
```

Gremlins everywhere

```
static void imx_poll_putchar(struct uart_port *port,
                             unsigned char c)
{
    unsigned int status;
    do {
        status = readl(port->membase + USR1);
    } while (~status & USR1_TRDY);
    writel(c, port->membase + URTX0);
    do {
        status = readl(port->membase + USR2);
    } while (~status & USR2_TXDC);
}
```

Applications

What does it do for me?

All cpu backtrace

Use an IPI to get all processors in the system to call `show_regs()`

`trigger_all[butself]_cpu_backtrace()`

Called when:

- Spinlocks take a long to acquire (*DEBUG_SPINLOCK*)
- Soft lockup detected (*softlock_all_cpu_backtrace)*
- Before a panic due to *hung_task_panic*
- SysRq-L (note that SysRq may require IRQs)

Reflections on watchdog h/ware design

Watchdogs don’t **have** to be wired directly to the reset pin

- Routing to FIQ allows us to trigger_all_cpu_backtrace() before issuing a soft reboot
- Ideally have a secondary watchdog that can perform reset (the watchdog built into the C-A9 MPCore could be coerced into doing this)

Performance monitoring

Modifying the PMU to use FIQ means we get a more accurate kernel profile

- We can instrument every part of the kernel except the FIQ handler and the big.LITTLE switcher
- spin_unlock_irqrestore() is no longer hot in the profiler

Note:
Using FIQ has no impact on userspace profiling since IRQs are always enabled in userspace anyway

Hard lockup detector

Soft lockup detector is a periodic hrtimer that checks that a high priority task gets some CPU time

Hard lockup detector (a.k.a. the NMI watchdog) uses a periodic NMI to check that the soft lockup detector is still running

- Runs on every core in the system (which is why hard lockup never called `trigger_all_cpu_backtrace`)
- Uses PMU cycle counter as source of periodic interrupt

kgdb and kdb

Linux x86 allows an NMI button to trigger k(g)db

Uses the existing polled I/O mechanism to communicate (UART, PS2 keyboard+VGA, …)

We could do better on ARM? After all the interrupt architecture usually allows us to steer the UART to FIQ

➔ Send a keystroke to the UART
➔ Debugger triggers stops whenever a byte is pending on the UART
➔ Debugger uses polled I/O to grab the character

An aside: Android FIQ debugger

Linaro’s work with FIQ was inspired by Android FIQ debugger

- A UART-based interactive debugger (similar to kdb) that can, optionally, use FIQ to process characters received from the serial port.

UART is multiplexed via the headphone jack

- Line noise? That could be bad...

kgdb and kdb

If a UART were treated like the NMI button on a server then line noise halts the system… ouch!

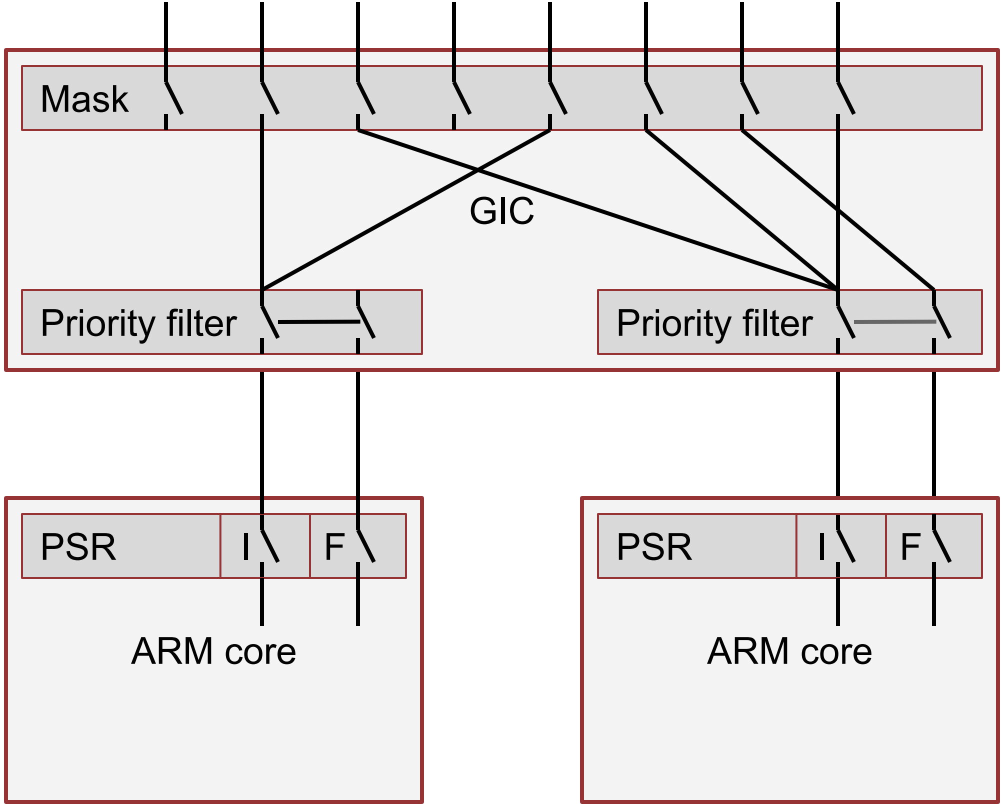
ttyNMI is a console driver that wraps the UART

- Waits for a pattern before halting the system: `$3#33`
- Provides tty services to allow console and getty to share UART with kernel debuggers

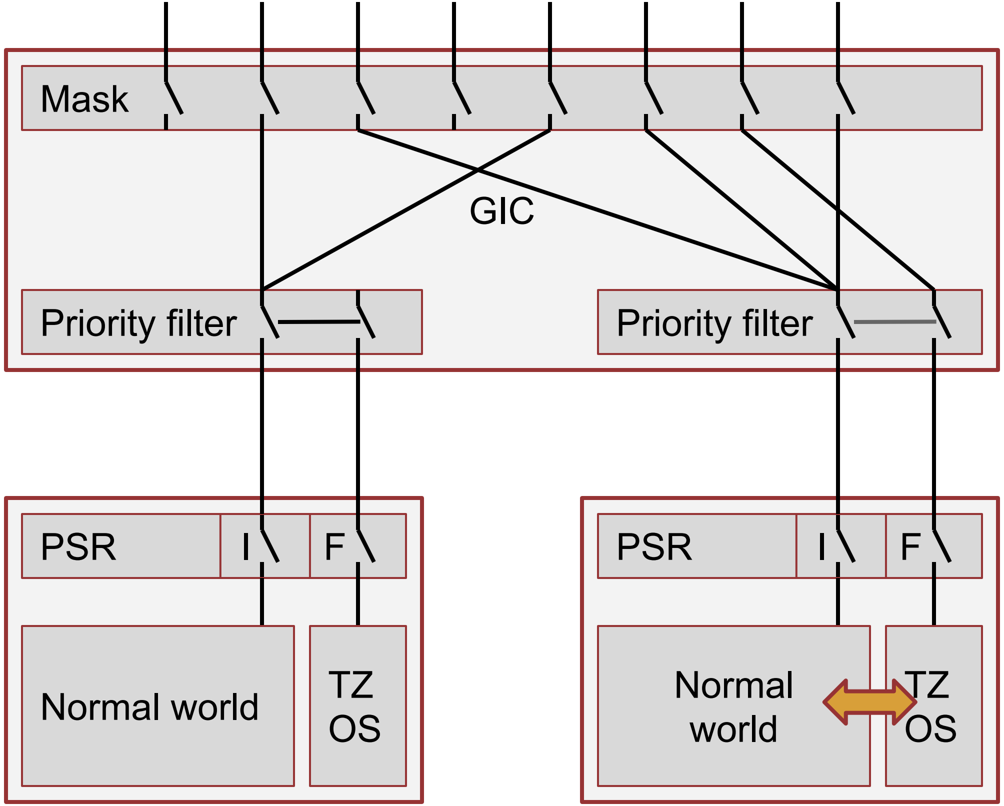
Practicalities

How do I run this stuff?

ARMv7 without TrustZone



ARMv7 with Trustzone



FIQ triggers a switch into the secure monitor and can not be observed by Linux (and any other normal world OS)

Secure monitor can disable the interrupt and alter normal world state (e.g. context switch) although this is too slow for some applications.

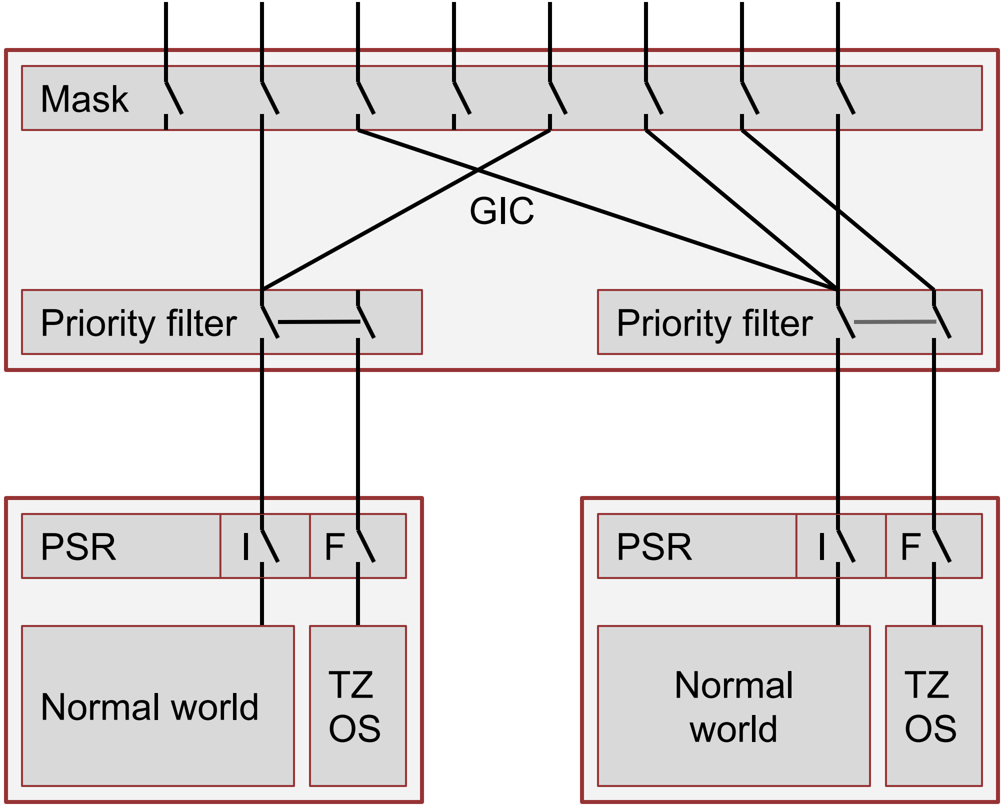
Access to secure mode

To exploit FIQ you need to be able to run Linux in secure mode

- Secure bootloaders are unlikely to be your friend
- Some “non-secure” parts have a mask programmed ROM that jumps to non-secure mode before boot

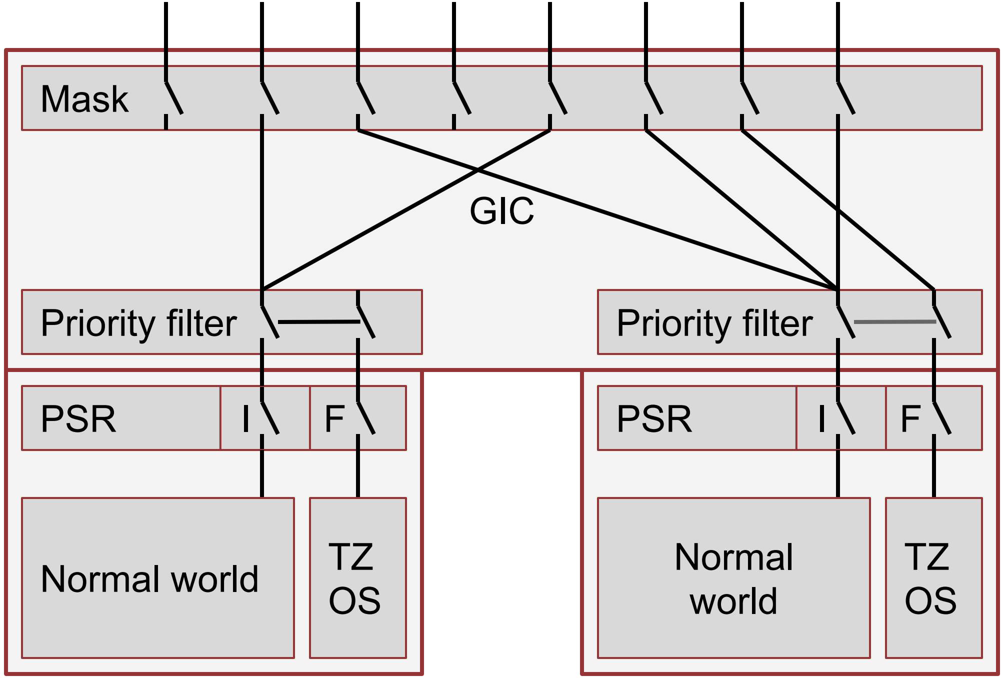
… or a TZ monitor that shares FIQ with normal world OS and a lot of spare hacking time

ARMv7 with TrustZone



ARMv8/GICv3+ with TrustZone

ARMv8 provides a co-processor interface for the GIC, making access from the CPU very fast. Fast access to the priority filter makes it possible to simulate NMI without using FIQ.



No known bugs but...

All cpu backtrace (partially upstream)

Performance monitoring (RFC)

Hard lockup detector (git only)

kgdb and kdb (git only)

ARM64 (proof-of-concept in-progress)

```
git clone https://git.linaro.org/people/daniel.thompson/linux.git -b merge/fiq
```

HOWTO - It just works

The NMI FIQ handler is installed by default so many features just work better out-of-the-box if they detect they are running on a system that support NMI/FIQ

```
    CONFIG_SPINLOCK_DEBUGGING
    CONFIG_LOCKUP_DETECTOR
    CONFIG_PMU
```

HOWTO - kgdb and kdb

Serial driver

May need porting and making NMI-safe (ttyAMA, ttyASC and ttymxc are already ported)

Kernel configuration

KGDB, KGDB_KDB, KGDB_FIQ, SERIAL_KGDB_NMI

Kernel cmdline

```
console=ttyNMI0 kgdboc=ttyAMA0,115200
```

Demo

Does it really work?

Bonus extra -
Keeping kdb turned on in production

- Another idea from Android:
  - UART is multiplexed via the headphone jack?
  - What stops an “evil” set of airline headphones stealing your data?
- kdb restricted capabilities mode
  - Work like SysRq restrictions
  - Allows debug features to be limited to passive inspection of state

Linaro

connect