## Debugging ARM kernels using NMI/FIQ

By [Daniel Thompson](http://www.linaro.org/author/daniel-thompson/)

[February 8, 2015](http://www.linaro.org/2015/02/)

In [Core Dump](http://www.linaro.org/blog/core-dump/)

[20](http://www.linaro.org/blog/core-dump/debugging-arm-kernels-using-nmifiq/)

[0](http://www.linaro.org/blog/core-dump/debugging-arm-kernels-using-nmifiq/" \l "comments)

# Debugging ARM kernels using NMI/FIQ

Daniel Thompson talks about how Linaro’s work to upstream a little known tool for Android evolved into an effort, in collaboration with other contributors, to build a framework to exploit fast interrupt requests and, as a result, port a wide variety of NMI-based diagnostic techniques to ARM.

# Introduction

For several years Linaro has, alongside several others, been working to reduce the differences between the mainline kernel and the Android (AOSP) kernel. Some of the work has involved taking code from AOSP and modifying it to be suitable for adding to the mainline kernel. On other occasions ideas flow in the other direction and AOSP is able to discard code that has been rendered obsolete by changes to the mainline kernel. This work has been successful to the extent that it is now possible to take an unmodified mainline kernel and boot Android. It will be lacking features and the graphics is not accelerated but nevertheless this is a significant achievement.

As this work has progressed, the line-of-code delta between mainline and AOSP has dropped significantly. In fact at the last audit one of the most significant contributors towards the line count turned out to be a little known tool for Android called the [FIQ debugger](https://android.googlesource.com/kernel/common.git/+/a82e9f5a7ee65687bda08d70256983fdade2d0d2/arch/arm/common/fiq_debugger.c).

The Android FIQ debugger is often shipped as part of Google’s Nexus products and is similar in concept to kdb debugger found in the mainline kernel. Both debuggers allow a developer connected via a serial port to use a simple interactive command interpreter to examine the state of the system. The FIQ debugger has a number of interesting features that did not exist within kdb, these are summarized in an article [describing our early work on the FIQ debugger](http://lwn.net/Articles/600359/).

There is a significant overlap between the two debugger so it did not seem worthwhile trying to upstream the FIQ debugger as a standalone feature, instead we sought to replicate features of the FIQ debugger in kdb.  This blog post will focus exclusively on the FIQ debugger’s signature feature: that it can be triggered by FIQ as well as IRQ.

A debugger based on FIQ are robust enough to remain functional in circumstances where other on-device debuggers fail. In particular a debugger based on regular interrupts can only be invoked when interrupts are enabled, making it very difficult to debug failures that occur within critical sections when interrupts are masked.

# An aside: What is FIQ?

FIQ stands for [Fast Interrupt reQuest](http://en.wikipedia.org/wiki/Fast_interrupt_request) and is a feature found in the majority of ARM cores, including all ARMv7-A devices. It augments regular interrupts by providing a second mechanism to asynchronously interrupt the CPU. The two interrupt signals, FIQ and IRQ, can be independently masked and Linux code seldom, if ever sets the FIQ mask bit.

*Note: On ARMv7-A devices that have security extensions (TrustZone) FIQ can only be used by the kernel if it is possible to run Linux in secure mode. It is therefore not possible to exploit FIQ for debugging and run a secure monitor simultaneously. At the end of this blog post we will discuss potential future work to mitigate this problem.*

FIQ can perhaps best be characterized as a thirty year old trick designed to eliminate the need for a DMA unit in certain low cost systems. Avoiding a DMA unit becomes possible because, in addition to the separate masking, the CPU automatically [banks some of its registers](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211h/ch02s08s01.html) when it switches to FIQ mode. These extra registers make it possible to service FIFO interrupts very quickly and without needing to use the stack. The only (data side) memory accesses needed are those required to fetch and store data from the FIFO.

Thirty years on the “fast” features of FIQ remain interesting for a few niche applications, most notably among FPGA developers, but for a debugger based on FIQ we have little interest in anything except the separate mask bit. The separate mask bit allows us to treat FIQ like the non-maskable interrupt (NMI) found on many other architectures (including x86).

# Early work

Our early work focused exclusively on extending code found in ARM’s kgdb and kdb support to allow it to be triggered using FIQ. We built just enough infrastructure within the kernel to support this use case and paid little attention to beyond getting that single job done.

The code was fully functional and allowed us to develop a good understanding of the challenges of working with NMIs. Any code that is called from an NMI handler must be carefully audited to make sure it avoids all forms of locking, including spin locks. When we start calling code from NMI for the first time we often have to make it NMI-safe by finding ways to make the code lock-less. For example, we found that several polling serial drivers used spin locks. This was an important discovery since kgdb and kdb poll the UART in order to communicate.

We regularly [shared the resulting patchset](http://thread.gmane.org/gmane.linux.ports.arm.kernel/331027) on the kernel mailing lists. The community feedback arising from these patches convinced us that we need to raise our sights beyond kgdb and build a foundation to support all of the kernels existing NMI based features. Only by building this foundation would we be able to convince the maintainers that our approach was the correct one.

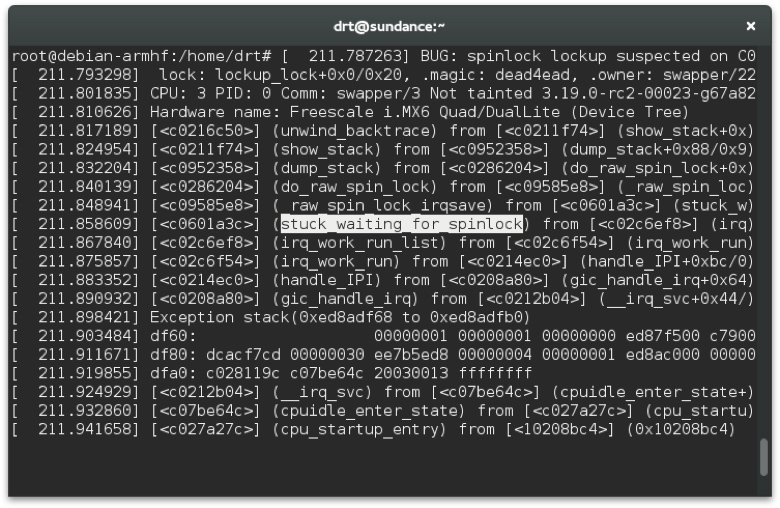
# Backtrace on all CPUs

Most [advice on upstreaming](https://www.linaro.org/blog/core-dump/working-upstream/) includes somewhere within it the idea that the way to build new kernel features is one patch at a time, piece by piece, little by little. In the context of NMI based diagnostics the question we must answer is “*what is the smallest change that can do something useful with an NMI?”*

Our answer (admittedly supplied to us in a [post from Thomas Gleixner](http://thread.gmane.org/gmane.linux.ports.arm.kernel/331027/focus=1778905)) was to implement a function called arch\_trigger\_all\_cpu\_backtrace().

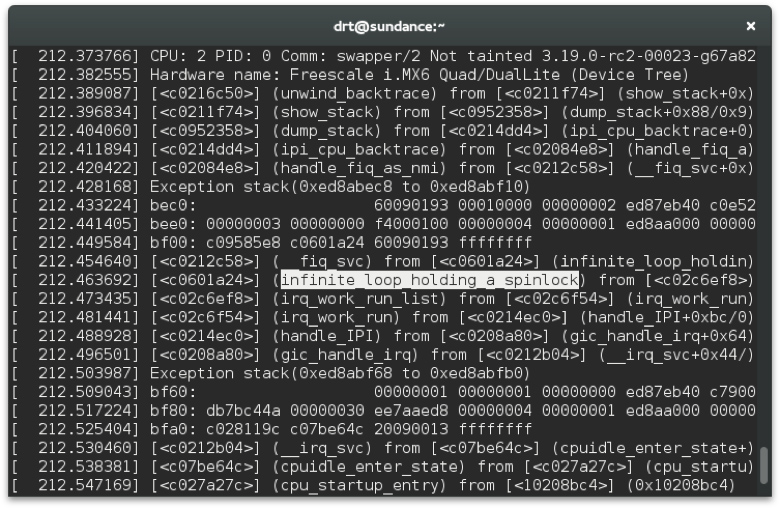
All cpu backtrace is called by the spinlock debugging code (CONFIG\_DEBUG\_SPINLOCK) when it thinks the system might have locked up. It works by sending IPIs (inter-processor interrupts) that raise FIQ on the target processes and, because it uses FIQ, these target processors respond and issue a stack trace even if they are locked up and have interrupts masked.

Normally on an ARM system, when a deadlock occurs, spinlock debugging will only show the backtrace of the CPU that’s stuck and this might not be the CPU that owns the lock. With all cpu backtrace then we get to see much more of the system hopefully allowing us to find the fault more quickly. For example the following screenshot shows what you would see the spinlock deadlock detection triggered on a typical ARM kernel (the functions highlighted were added to intentionally create a lockup warning):



Here we can see where we have locked up, but it isn’t clear why.

With all cpu backtrace enabled we would still get the above information about CPU that is stuck but we would also be able to scroll down and see this:



Which better helps us narrow down why the deadlock occurred.

This patchset is mature and no longer expected to change significantly. Some parts of it, such as the default FIQ handler (handle\_fiq\_as\_nmi) are already upstreamed. The remaining parts that are waiting to be merged include code to initialize the GIC and the ARM architecture specific code that handles the IPI.

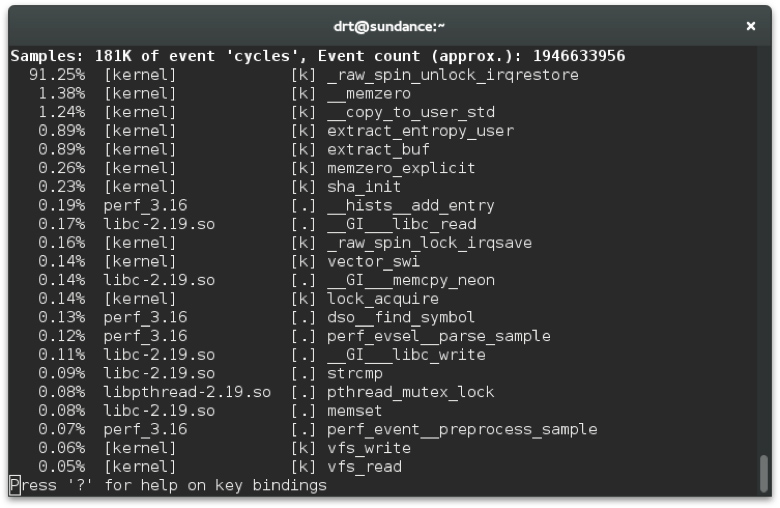
# Hardware performance monitoring

After completing the previous patch set we stop and ask again “*what is the smallest change that can do something useful?”* This time we turn our attention to the PMU (performance monitoring unit). It is an attractive target because the PMU on modern x86 Linux systems is hooked up to NMIso we can be confident of having a mature sub-system to work with and can expect  very few, if any, NMI related bug in the generic code.

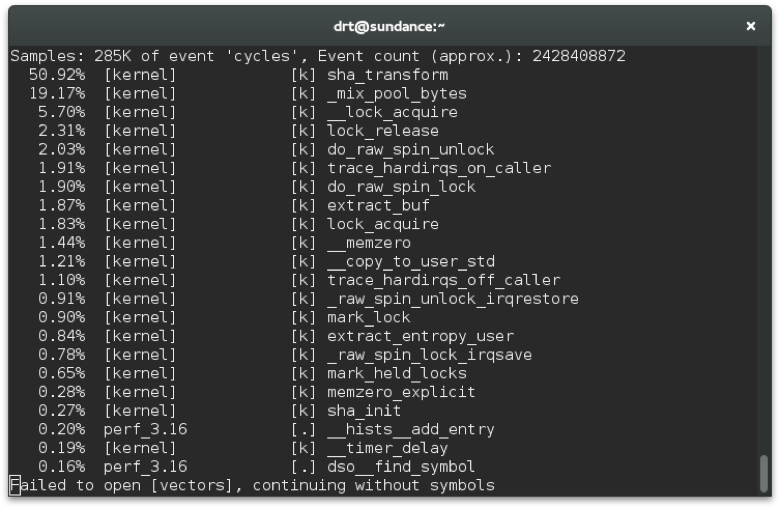
The PMU is hooked up to the kernel’s perf events framework and allows us to monitor and profile [CPU behaviour related to performance](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Bcgddibf.html) including, among many others, CPU cycles consumed, cache misses, and data load/stores. PMU events increment a counter.  For small sections of code the counts can be read before and after the code under test but this may not be practical for larger code bases. For large code bases statistical profiling is often preferred. During statistical profiling each event count is given a high watermark and when that value is reached and interrupt is generated. This allows the PMU to, for example, generate an interrupt every 20 cache misses. Statistics gathers during interrupt handling will quickly identify code that frequently misses the cache allowing it to be optimized.

The kernel already has drivers for PMU and they work well but, because they are based on normal interrupts, they do have a subtle limitation. That cannot perform statistical profiling of code that runs with interrupts masked. When we use the FIQ to handle PMU events we are able to profile the entire kernel (except for the PMU management itself) and this gives allows us see much more of the system. For example, when we use FIQ handling PMU events, it is possible for use to profile frequently called interrupt handlers or to identify a heavily contented spin\_lock\_irq().

For some workloads the difference can be striking. The workload for both examples below is the same: dd if=/dev/urandom of=/dev/null. The first screenshot perfectly illustrates the limitation of profiling from normal interrupt handler, over 90% of the CPU time is spent unlocking interrupts and the cryptographic operations that should dominate the use case are completely hidden.



When we enable the FIQ we immediately get a much deeper insight. Not only can we can see the cryptographic operations but we can also see how much impact the fact I had compiled the kernel with lockdep enabled is having on this use case.



The primary feature introduced by this patchset is to extend the irq sub-system to make it possible to route regular interrupts to FIQ. This change was not required previously because IPIs are architecture specific and do not use irq sub-system much. Once this feature was added the changes needed to the PMU driver were fairly minor.

This patch has been published as an RFC and will need further work before it is ready to merge.

# Enabling the hard lockup detector

The hard lockup detector is a watchdog built into Linux that uses a periodic NMI in order to detect if the system has become unresponsive. It is used to detect any kind of fault that can causes interrupt handling to fail. Examples include badly matched disables, spurious interrupts, and live locks inside critical sections.

*Note: The hard lockup detector is partnered by the soft lockup detector. The soft lockup detector runs from an interrupt handler and checks for faults that could prevent threads from being scheduled correctly. Interestingly the hard lockup detector doesn’t monitor interrupts directly, instead it monitors the health of the soft lockup detector. If the soft lockup detector fails to run the hard lock detector infers that interrupts have failed and reports the fault.*

The hard lockup detector was selected by the “*what is the smallest change?”* test because it uses the performance monitoring framework to configure the periodic NMI on each processor. Thus the work to enable it a tiny bit of plumbing and fits into a single patch.

At present the patch is on [Linaro’s git server](https://git.linaro.org/people/daniel.thompson/linux.git/commit/50316b4218af5b6fbe68a6478613b42258c1b491) but has not been posted on the kernel mailing lists due to its relatively trivial nature, some small issues mentioned in the commit comment and its dependence on other patches that remain at the RFC stage.

# The kernel debugger

Finally we return our attention once more to adding FIQ support for kgdb and kdb. With the infrastructure already, and with a pile of NMI-safety fixes already upstreamed as a result of our earlier work the patch set to add FIQ support comes together in just five patches.

The bulk of the work is simply the plumbing need to divert the UART interrupt from IRQ to FIQ. As a result whenever a character appears in the UART’s RX FIFO the FIQ handler runs and uses the polled UART drivers to fish out the character and decide what to do next. Also needed is a small extension to the all-cpu-backtrace IPI so it can be also be used to stop all the processors on a SMP system.

Like the hard lockup patch the kgdb patches are not yet shared on the kernel mailing lists as we are still working hard to upstream its dependencies. Nevertheless it is fully functional and available via git.

# HOWTO

A kernel containing all the NMI/FIQ work can be found here:

<https://git.linaro.org/people/daniel.thompson/linux.git>

The merge/fiq branch contains all features discuss above. Be aware that the branch is frequently rebased; at the time of writing is based on the v3.19-rc6 kernel.  
ARCH=arm make multi\_v7\_defconfig

scripts/config \

–enable DEBUG\_SPINLOCK –enable LOCKUP\_DETECTOR \

–enable DEBUG\_INFO –enable MAGIC\_SYSRQ \

–enable KGDB –enable KGDB\_KDB –enable KGDB\_SERIAL\_CONSOLE \

–enable KGDB\_FIQ –enable SERIAL\_KGDB\_NMI

ARCH=arm make olddefconfig

ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- make -j 12

If you don’t have a board capable of running a multi-platform kernel or that cannot boot into secure mode then you might prefer to test using the [TrustZone support in qemu](https://www.linaro.org/blog/core-dump/testing-qemu-arm-trustzone/).

Booting the kernel as normal will give you access to all of the features discussed above, with the exception of kgdb.

Some ideas to try out:

* <SysRq>-L (either by echo l > /proc/sysrq-trigger or by sending <Break>-L via the UART): This will show the stack trace of all CPUs. This should show the CPU requesting the backtrace running \_\_handle\_sysrq and all other CPUs responding by running handle\_fiq\_as\_nmi.
* perf top: This will show a simple statistical profile based on counting CPU cycles used. Try to run a use-case that you know involves significant interrupt locking in order to see the full benefit (or use the dd example from earlier).
* cat /proc/interrupts: The NMI field is incremented by the default FIQ handler (handle\_fiq\_as\_nmi) allowing you to quickly check FIQ is working for you.
* Set the NMI watchdog running (echo 0 > /proc/sys/kernel/nmi\_watchdog; echo 1 > /proc/sys/kernel/nmi\_watchdog) and then write a kernel module to make the kernel lockup (you could also use the one already included in the merge/fiq branch).

To experiment with kgdb/kdb you will need to modify the kernel command line to enable the NMI-based serial port wrapper. This will vary depending upon your serial port settings by as an example:

console=ttyAMA0,115200

Should be changed to:

console=**ttyNMI0 kgdboc=**ttyAMA0,115200

With this change the kernel should boot as normal but the serial port will have a wrapper applied so it can be used by the FIQ handler. To trigger kdb you must manually type the gdbserver protocols wake up command $3#33 .

# The future

There are three potential activities related to this work in the future:

1. All the patches discussed will be maintained both to nurse them until they are delivered to the upstream kernel and to ensure they continue to be supported after they are merged.
2. ARMv8-A and GICv3 introduce a new co-processor interface to the GIC (both for AArch32 and AArch64) that we hope can be exploited to simulate NMIs without using FIQ. This should allow modern ARM devices to benefit from the robustness of NMI debug features without needing to run in secure mode.
3. OP-TEE and other secure monitors could be extended to allow it to handle some FIQs on behalf of the non-secure OS and route these interrupts back into the non-secure world. This would allow an NMI to be present even where Linux cannot run in secure mode.

From the above list the first two items are being actively pursued by Linaro although our work on ARMv8-A is still in the very early stages.

Right now there are no plans at present to work on the final item, in part this is because it is more or less rendered obsolete by the switch to ARMv8-A systems. There also remain some serious technical challenges too. In particular world switching is a relatively expensive operation, making its use for performance monitoring unwise.

When we started this work our goal was to take a single feature from Android and make it more widely available. The feedback we received from the community challenged us to do more and result is a wide variety of debugging tools, all previously missing on ARM, that have been developed and can potentially be used across the eco-system, from mobile phones to large-scale servers. Interacting with the community in this way is, without doubt, one of the most exciting thing about writing open source software.

*The community is, of course, made up of individuals and among the many people I have met so far I would especially like to thank Thomas Gleixner, Russell King, John Stultz, Dirk Behme and Will Deacon who variously have helped with code reviews, advice, feedback and encouragement.*

## Correction

In the article, the section “Backtrace on all CPUs”, incorrectly implies that all work on all CPU backtrace for ARM was done by Linaro employees. In fact, Russell King provided an [initial prototype implementation](http://thread.gmane.org/gmane.linux.ports.arm.kernel/353795/) for ARM, derived from the existing x86 implementation. This patch was combined with patches from our own early work and the combined patchset evolved into the work presented in this article.

[1] Once spin\_lock\_irq() has masked interrupts it becomes invisible to the profiler no matter how long it spends spinning trying to acquire the contended lock.