Introduction

This application note provides information for software developers on how to use the (FIQ) (fast interrupt request) mechanism with Linux in the SPEAr embedded MPU family.

It is divided into four sections:

Section 1 describes the interrupt logic on SPEAr platform.

Section 2 provides a detailed introduction to the FIQ interrupt handling hardware mechanism and interrupt latency in the Linux kernel.

Section 3 describes the FIQ interrupt handler software.

Section 4 gives the results of interrupt latency measurements performed with Linux running on a SPEAr evaluation board.
1 SPEAr interrupt logic

SPEAr is a family of embedded MPUs based on the ARM core. The embedded MPU uses the PL190 vectored interrupt controller (VIC) IP. Together they provide facilities to use fast interrupts as highest priority interrupts. Fast interrupt features are supported by both the ARM core and the VIC. This VIC IP can be configured to select any hardware source to interrupt ARM core as either IRQ or FIQ.

Figure 1. SPEAr platform interrupt logic

![Diagram of SPEAr platform interrupt logic](image_url)

The order of priority for hardware interrupts (from higher to lower) is as follows:

1. Fast interrupt request (FIQ) interrupts
2. Vectored IRQ interrupts
3. Non-vectored IRQ interrupts.

FIQs are intended for fast, low-latency interrupt handling. In FIQ mode, the ARM core uses seven 32-bit banked registers, which enable the VIC to process interrupts as quickly as possible.

Multiple interrupts can be handled as FIQs, but in this case the IRQ vector needs to be loaded in the program counter (PC). In order to reduce interrupt latency, it is advised to use a single FIQ source, because the ISR can be directly executed without determining the source of the interrupt.

As the Linux kernel never disables FIQs they have ultimate priority over anything else in the system. When using an FIQ, the FIQ latency will be the time it takes the CPU to enter your FIQ function. Moreover an FIQ can preempt Linux kernel code and IRQ handlers and it can also be used for faster response to interrupts.

Note: The use of an FIQ in place of an IRQ does not require any hardware (wiring) modifications on the SPEAr board. The VIC has a 32-bit register (VICINTSELECT) which can toggle an IRQ source to raise an FIQ instead of the default mechanism of IRQ.
2 Linux kernel IRQ handling latency

In a Linux kernel based system, interrupt latency may occur in the following cases:
- Interrupt latency for top half,
- Various sections where IRQs are disabled

Figure 2. IRQ handling latency in Linux Kernel

An FIQ is a highest priority interrupt and preempts both of these sources of latency. It can preempt IRQ top half handlers as well as Linux kernel code sections where IRQs are disabled. Further to this, the FIQ handler can not be preempted by other IRQs. This ensures that the platform has predictable worst case timings for FIQ handler latency making FIQ suitable for real-time systems.

The FIQ handler can not permit tasks that require other IRQs to be active, including the timer. This also makes it suitable for real-time systems but imposes the following limitations:
- Kernel mode APIs and device driver framework can not be used inside the FIQ handler. These APIs available for Linux driver development may internally rely upon timers, schedulers, spinlocks, etc. for functioning. Linux code also needs to guard against multiple access to objects by spinlocks or other relevant mechanisms. FIQ ISRs can come in the middle of anything and have to get out quickly again too. So **unless it is a simple macro, you can not use any Linux APIs in the FIQ ISR.**
- Page faults have to be avoided inside the handler (the memory used in the handler should be already allocated and initialized). In case that a page fault occurs inside the FIQ handler, it can not be recovered by the kernel and it would lead to a kernel crash.
3 FIQ handler environment

The FIQ registration, the ISR function and the runtime environment of a FIQ handler are different from a standard IRQ handler. A working example of an FIQ handler can be found in latency measurement module in Section 4: IRQ latency measurement.

3.1 FIQ registration

For FIQ registration ‘set_fiq_handler’ API is used.

The two arguments of this API are:
- Function pointer
- Size of handler function in bytes

This handler function is copied to the FIQ vector reserved area. The FIQ vector area is last in the memory area reserved for interrupt vectors. About five hundred (500) bytes are available in this area for copying the ISR.

In the SPEAr Linux Support Package (LSP), the following API must be used to select the configuration of the IRQ source:
- `vic_unmask_irq(int number)` – enables interrupt source on VIC as IRQ
- `vic_select_fiq(int number)` – selects interrupt source on VIC as FIQ instead of IRQ

3.2 FIQ mode registers setup

When the FIQ handler runs, it utilizes FIQ mode-specific registers (r8-r14) and general mode registers (r0-r7). This set of FIQ mode registers is exclusive to FIQ mode and not available to other handlers.

To pass any information to the FIQ handler, the FIQ mode registers have to be set using the call: ‘set_fiq_regs’.

3.3 Return from FIQ mode

To return from FIQ mode on completion of the code, use the following assembly instruction:

```
subs pc, lr, #4;
```

This command restores the instruction pointer to the original location it was before FIQ arrived.

3.4 FIQ ISR in C

Writing the FIQ handler function in ARM assembly language can be cumbersome, so it is highly recommended to use C language instead. This is especially true for large functions when the handler function is larger than the maximum size available for FIQ vector (500 bytes). In such cases branching to the C function can be possible. This involves saving of the register context (r0-r12 general mode registers) and initialization of the function frame pointer.
4 IRQ latency measurement

The latency measurement module is a Linux kernel module (LKM) available as part of the STLinux SPEAr BSP release kernel code. This module can be statically linked with the kernel at the time of the build.

This module is useful for measuring the IRQ or FIQ latency in a running system. The statistics collected are:

- Maximum latency (in microseconds)
- Minimum latencies (in microseconds)
- Total number of samples captured
- Frequency distribution of IRQ handling latency in range [0-200] microseconds

**Figure 3. Linux Kernel IRQ latency measurement module**

To see above modes in the kernel source directory:

1. Run `make menuconfig`.
2. Then navigate to ‘System Type’->‘SPEAr Interrupt Latency Test’.

The source code of the latency measurement module is available in the STLinux SPEAr BSP Linux kernel source code.

The file location is: `arch/arm/plat-spear/latency_test.c`

You can use this module as a starting point for writing a new FIQ handler.

4.1 Using the latency measurement module

The latency test module exchanges configurable parameters and measured results through Linux sysfs filesystem.
This module is linked statically to the kernel image and has user mode interface at:

/sys/kernel/irq_latency

To use the module:

1. Run the following command:

   # ls /sys/kernel/irq_latency

   The command output shows four entries:
   - irq_mode – shows if IRQ latency measurement is 1 (on) or 0 (off/no capture)
   - fiq-mode – shows if FIQ latency measurement is 1 (on) or 0 (off/no capture)
   - stats – shows the running statistics of measurements as summary
   - stats_details – shows detailed log of IRQ latency counters

   During kernel booting, the statically linked module is loaded automatically. By default the module lies dormant and does not capture latency statistics.

2. To start the capture, you must activate one of the two IRQ handling modes using the command:

   - # echo 1 > irq_mode
   - # echo 1 > fiq_mode

   **irq**_**mode**: in this timer interrupts are handled with an IRQ mechanism.

   **fiq**_**mode**: in this timer interrupts are handled as FIQ interrupts. FIQ mode is designed to handle real-time scenarios.

   To get the statistics captured by the module, use the command **# cat stats** repeatedly while performing other tasks. A typical output is:

   Latency

   max (usec) : 48
   min (usec) : 6
   totalsamples : 7798

   To get detailed statistics, use the command **# cat stats_details** repeatedly while performing other tasks. A typical output is:

   Latency:    Counter (usec)

   0, 0
   1, 0
   2, 0
   3, 0
   4, 0
   5, 0
   6, 12001
   7, 70
3. To stop statistics capture, use one of the following commands (depending on the mode):
   
   – #echo 0 > irq_mode
   or
   
   – #echo 0 > fiq_mode

   Once the mode stopped, the timer is stopped and the IRQ handling is deactivated. Also the statistics are cleared.

4.2 Module internals

   The module uses a general purpose hardware timer to raise an IRQ/FIQ and capture the ticks elapsed. The timer automatically reloads itself after raising an interrupt. In this way when the IRQ handler reads the timer value, this value is the exact measurement of the system latency for handling this IRQ.

   Depending upon the choice of IRQ handling mechanism (IRQ or FIQ), the corresponding selection is made in VIC handling. FIQ registration and de-registration is part of the standard Linux kernel.

   Timer (GPT) interval: 1 millisecond
   Timer tick interval: 1/48 microsecond

   Interrupt handler latency frequency counter is captured from 0 to 9600 ticks. This corresponds to a range of 0 to 200 microseconds. This frequency range is sufficient for determining the interrupt latency performance.

4.3 Performance measurements

   The details of setup used for making the measurements are:

   ● **Kernel**: Linux kernel version 2.6.27
   ● **Board**: SPEAr Plus 600 application board
   ● **CPU**: ARM core running @333 MHz
   ● **RAM**: 128 MB DDR2 running @ 333 MHz
   ● **Flash**: 8 MB NOR
4.3.1 System load using netperf

While the timer is recording values for ISR latency, the system is loaded using high-traffic on Ethernet. For this, the netperf utility is used. Netperf is an open-source, free software used for benchmarking of Ethernet performance between two systems.

There are two components to the netperf Ethernet benchmark: server and client. When the client (netperf) is invoked, it connects to the server (netserver) and starts to send data through sockets. The total bytes transferred in 10 seconds are measured and used to calculate the network throughput.

The SPEAr board is connected directly to a Linux PC through Ethernet (cross-cable). Netperf (client) is run on the SPEAr board and netserver (server) is run on the Linux PC. The netperf utility sends Ethernet packets at the maximum possible rate to the netserver which sends them back.

4.4 Results

This experiment was done both for IRQ mode and FIQ mode. The values from the timer module and the netperf throughput are recorded in the following table.

In all three tests, the overall system throughput was measured as peak network performance.

Table 1. Timer ISR latency measurements (minimum/maximum)

<table>
<thead>
<tr>
<th>Interrupt Latency (µs)</th>
<th>IRQ</th>
<th>FIQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under Network load</td>
<td>Test1</td>
<td>Test2</td>
</tr>
<tr>
<td>Maximum (µs)</td>
<td>175</td>
<td>177</td>
</tr>
<tr>
<td>Minimum (µs)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>94</td>
<td>94</td>
</tr>
</tbody>
</table>

1. Throughput was calculated using the netperf client data transfer rate.

According to the table, we can infer that:

- The maximum latency for IRQ shoots up to 177 µs
- The maximum latency for FIQ is close to 8 µs
- The overall throughput of system remains the same in both cases.
5 Conclusion

The fast interrupt mechanism ensures a bounded performance with the Linux kernel without any decrease in the overall system throughput. Its worst case is less than ten microseconds (<10usec). However this deterministic behavior comes at the cost of some limitations in ISR handler development, which can not utilize Linux kernel API and device driver framework.

The FIQ interrupt handling mechanism used, as described in this application note, is well-suited for critical real time applications with SPEAr.
## References

### Table 2. References

<table>
<thead>
<tr>
<th>Title</th>
<th>Revision</th>
<th>Path/Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimeCell® Vectored Interrupt Controller (PL190)</td>
<td>r1p2</td>
<td>ARM website</td>
</tr>
<tr>
<td>ARM Architecture Reference Manual</td>
<td>2005</td>
<td>ARM website</td>
</tr>
<tr>
<td>SPEAr300 User Manual</td>
<td>1.2</td>
<td>STLinux SPEAr BSP</td>
</tr>
<tr>
<td>SPEAr600 User Manual</td>
<td>1.3</td>
<td>STLinux SPEAr BSP</td>
</tr>
</tbody>
</table>
## Revision history

Table 3. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-Jan-2010</td>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>