# Using the Realtime Preemption Patch on ARM CPUs

**Jan Altenberg**
Linutronix GmbH
Auf dem Berg 3, 88690 Uhldingen-Muehlhofen, Germany
jan.altenberg@linutronix.de

**Abstract**

During the last few years, Linux has established itself as the fastest growing platform in the embedded sector. This arose from the fact that there is no other Operating System which supports such a big variety of different hardware. Also the ongoing rapid development of the Realtime Preemption Patch made Linux a perfect choice for automation and control applications which require hard-realtime capabilities.

ARM CPUs are widely used in embedded designs. There is a huge variety of cheap ARM9 and powerful ARM11 and Cortex CPUs which are predestined for use in embedded realtime controllers (because of the low power consumption and the growing number of features).

This paper demonstrates the behavior of the Realtime Preemption Patch on different ARM based embedded designs. The specific measurement environments will be explained in detail and the basics on how to get the Realtime Preemption Patch running on a specific ARM platform (and the major pitfalls) will be shown. Also the differences among ARM9 / ARM11 and Cortex CPU designs and how those differences influence the realtime behavior will be examined. As a result, the paper provides a decision guide for choosing the best fitting ARM design for a Realtime Linux based embedded platform.

## 1 What is Realtime?

### 1.1 Terms and definitions

Before looking into different ARM CPU designs and their behaviour in a Realtime Linux environment, I'd like to introduce some terms and definitions. First of all we should clarify the meaning of the word "Realtime". Misleadingly realtime systems are often considered to be as fast as possible. But realtime systems should be "as fast as specified rather than as fast as possible" (Doug Niehaus). Realtime systems have to meet specific timing constraints, which means they have to behave in a completely deterministic way. Calculations have to be done at a specific point in time and the system has to respond to external events in a given timeframe. The time duration between the occurancy of an external event and the correct reaction to that event is termed as latency. The variation of the latency is called jitter. Realtime systems are evaluated on the basis of their latency and their jitter (are those values deterministic and do they meet the needed timing constraints?).

## 2 ARM designs

### 2.1 ARM9

ARM9 is a 32 bit RISC CPU design. With this generation of CPUs ARM has moved from a "von Neumann" architecture to a Harvard architecture (which means the CPU can read an instruction and perform a data memory access at the same time). There are two ARM9 subfamilies:

- the ARM9T family (based on the ARMv4T architecture)
- the ARM9E family (based on the ARMv5TE architecture, which implements longer pipelines. It also has a slightly enhanced instruction set)

The major disadvantage of the ARMv4 / ARMv5 designs is their cache implementation. The cache is organized as virtual indexed and virtual tagged, which means that both the index and the tag are based on virtual addresses. On Linux each process has its own virtual address space. This implies that a physical address might be mapped to different virtual addresses at a time. So this cache implementation will lead to a more frequent cache flushing. The cache has to be invalidated after each context switch. The cost of the cache flushing is about 1k - 18k CPU cycles. The "indirect cost" (influenced by the side operations for filling up the cache lines and the TLB) can take up to 50k CPU cycles. This means 200us for a 250MHz CPU! There are two different approaches for addressing this problem:

- Using a flat address space (uClinux)
- Using the F ast C ontext S witch E xtension (FCSE)

On ARM9 FCSE offers a 7 bit PID register, which can be combined with 25 bit virtual addresses. This will lead to 128 independent address spaces. Richard Cochran recently posted a patch on the Xenomai-Devel list, which implements FCSE for the Intel IXP425 architecture. Gilles Chanteperdrix posted an enhanced version on the ARM kernel list, which has been tested on a AT91RM9200 CPU. The advantage of this approach is the combination of memory protection and a good cache performance. The disadvantage is the limitation of 128 processes and 32MB address space (although this might be sufficient for most of the embedded systems).

## 2.2 Xscale

Xscale is Marvell's implementation of the ARMv5 architecture. The Xscale family consists of five different families:

- IXP (network processors)
- IXC (control plane processors)
- IOP (i/o processors)
- PXA (application processors)
- CE (consumer electronix processors)

There are several powerful Xscale CPUs, which are designed to run at high CPU frequencies. But since the design is also based on the ARMv5 architecture, Xscale also suffers the problem with virtually tagged caches.

## 2.3 ARM11

ARM11 is a 32 bit RISC design which implements the ARMv6 instruction set. There are 4 subfamilies of the ARM11 architecture:

- ARM1176 (targeted for consumer electronics)
- ARM1156 (targeted for automotive, embedded control, imaging, ...)
- ARM1136 (targeted for network infrastructure, consumer electronics and infotainment)
- ARM11 MPcore, which is a multiprocessor design

The complete family implements the Jazelle technology for efficient Java execution (on ARM9 this is only available for the ARM926-EJS cores). ARM1136JF-S, ARM1176JZF-S and ARM1156T2F-S cores also have a powerful Vector Floating Point Unit. The main difference to the ARM9 family is the cache design. Unlike the ARM9 implementation, the ARM11 cache implementation is virtually indexed and physically tagged. In combination with its high performance memory system this speeds up the context switching time by several orders of magnitude (in comparison to the ARM9 family). Furthermore, the ARM11 family is designed to run at higher CPU frequencies.

## 2.4 Cortex

Cortex constitutes the new generation of ARM CPUs. The Cortex family is splitted into three families for different applications:

- Cortex A: Application (variable caches, MMU / MPU)
- Cortex M: Microcontroller (no cache, MPU optional)
- Cortex R: Realtime (variable cache, MPU optional)

Those families are evolved from the applications which are adressed by the most popular ARM designs on the market: ARM7, ARM9 and ARM11. This doesn't mean that Cortex implents those designs! The Cortex families are just designed to meet the requirements of the target applications. The Cortex M series adresses the Microcontroller market, so it can be considered as a modern successor of the ARM7 design. The M series combines a simple programming concept with the the performance of modern CPUs. The Cortex A series are designed for complex applications and for modern operating systems. The Cortex R series are designed for Realtime applications (Low Level RTOSses, ...). In view of Realtime Linux the Cortex A series is quite interesting, because it can be considered as the successor of the ARM11 design. The Cortex A series also offers a multicore design (Cortex A9 MPcore).

## 3 Realtime Preemption Patch

The Realtime Preemption Patch was started by Ingo Molnar and Thomas Gleixner. Unlike other approaches Preempt RT doesn't introduce a Microkernel. Preempt RT brings hard realtime capabilities directly into the Linux kernel. One of the basic features which have been introduced by the Realtime Preemption Patch, are Threaded Interrupt Handlers. IRQ Handlers are moved into their own kernel thread, which means they can be scheduled like any other process in the system. The basic feature of Preempt RT is the replacement of the locking primitives. Spinlocks are relaced by sleeping mutexes (that's why Preempt RT is sometimes called the "sleeping spinlocks patch"). Lots of the Preempt RT features already made it into Mainline: PI Mutexes, High Resolution Timer, Preemptive RCU (Read Copy Update), IRQ Threads.

### 3.1 The Realtime Preemption Patch on ARM

The great thing about the Realtime Preemption Patch is, that everything is implemented in a portable and generic way. Which means that if a platform is supported there's nothing special to care about. So the usage on an ARM CPU isn't that different to the usage on a generic x86 based system. Recent versions of Preempt RT have full ARM support. Table 2 shows the available features and the kernel version since when they are available for the ARM platform.

| Feature | Kernel |
|---|---|
| Deterministic scheduler | vanilla |
| Preemption Support | vanilla |
| PI Mutexes | vanilla |
| High-Resolution Timer | vanilla (since 2.6.24) |
| Preemptive Read-Copy Update | vanilla (2.6.25) |
| IRQ Threads | vanilla (since 2.6.30) |
| Full Realtime Preemption Support | Preempt RT |

**TABLE 1:** *Preempt RT features on ARM*

In addition to the kernel features, there are also some prerequisites for the userspace environment. The C Library needs a proper threading implementation, PI futex support and clock handling. For GLIBC at least version 2.5 is needed. Be careful: Currently uClibc can't be used for Realtime applications. The most recent release has still no support for priority inheritance pthread mutexes.

## 4 Measurements

### 4.1 Suitable test scenarios

#### 4.1.1 Load scenarios

As already mentioned in 1.1 the behaviour of a Realtime system must be deterministic. A Realtime system is characterized by its latencies and by its jitter. When choosing a Realtime platform, one has to prove, that this platform meets the timing requirements of the application under all circumstances. Benchmarking a Realtime system means to examine the latencies and the jitter in a worst-case scenario. An Operating System has to manage the system resources (CPU, memory, ...) and to coordinate the activities like external events and user applications. So, a worst-case scenario for an OS means a high utilization of the system resources and a large number of applications and external events (like interrupts). For a Linux system such a scenario could be easily achieved with:

- hackbench: Hackbench has been designed as a scheduler benchmark. It creates n groups of 20 servers and 20 clients. The servers and the clients talk to each other via sockets. The number of groups is given by a commandline argument (Example: hackbench 10 would create 400 processes: 10 * 20 * 20). This scenario generates a high CPU load and high memory utilization.

- Floodping from another system: Adding the -f options to the ping command causes ping not to wait after sending an ICMP message. It'll just send the packets as fast as possible. "Ping flooding" the target system from a different machine will cause a huge number of network interrupts. Be careful: The Floodping will cause the IRQ Handler of the network interface to run very often. It'll also increase the runtime of the Soft IRQ Handlers for the RX and TX handling. All of them are Kernel Threads with Realtime priority. So, on PREEMPT_RT please check /proc/sys/kernel/sched_rt_runtime_ns. This defines a threshold for the runtime of Realtime Tasks (to prevent a starvation of the low priority tasks). The default value is 950ms, which means if the rt runtime exceeds 950ms, the rt tasks won't be scheduled up to the full second. You can disable this behaviour by writing -1 to sched_rt_runtime_ns.

### 4.1.2 Test tools

In addition to an appropriate load scenario, suitable test applications are needed. Those test applications should prove the determinism of our Realtime System and they should give us an idea about the worst-case latencies and the jitter. Therefore the following parameters are of interest:

- The responsiveness to external events
- The accuracy of tasks which are scheduled for a specific time

The response times for external events can easily be determined by a simple UIO driver. The UIO driver just implements an interrupt handler, which:

- responds to an interrupt, which might be generated by a GPIO input and
- toggles a pin

The response time can be determined with a scope.

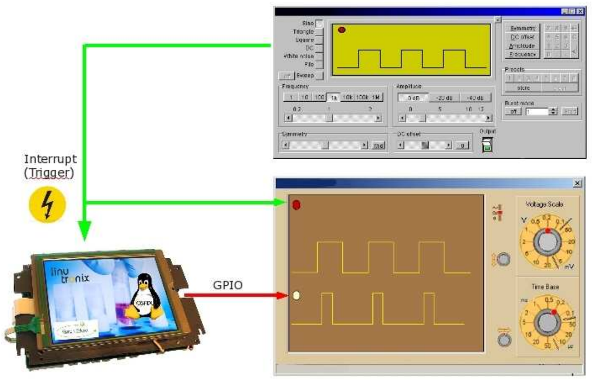
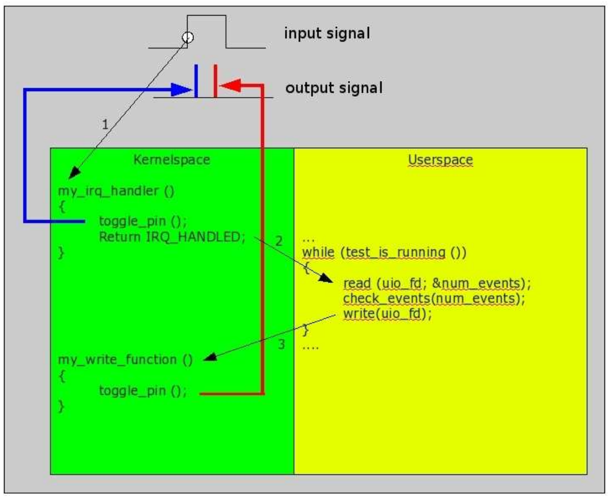


**FIGURE 1:** *Measuring the IRQ response time*

Figure 1 shows this test scenario: A function generator is used as an interrupt source, which is connected to a general purpose pin of the target. The UIO driver resonds to this interrupt and toggles a pin. Both, the interrupt source and the "response pin" are connected to different channels of a scope. So, the time difference between the slopes on both channels shows the response time of the interrupt. Since we implement the interrupt handling with an UIO driver, we can wait for the interrupt occuracy with a userspace task. The userspace task can toggle the "response pin" one more time. So, the time difference between two peeks of the "response pin" can give us an idea about the context switching time.



```
my_irq_handler ()
{
        toggle_pin ();
        Return IRQ_HANDLED;
}

my_write_function ()
{
        toggle_pin ();
}
```

```
...
while (test_is_running ())
{
        read (uio_fd; &num_events);
        check_events(num_events);
        write(uio_fd);
}
....
```

**FIGURE 2:** *Measuring the context switching time*

As shown in figure 2 the userspace task does a blocking read on the UIO driver. After an interrupt event has occured, the IRQ handler will be executed and the UIO core will wake up all waiters on the UIO device. The userspace task itself ”abuses” the UIO write() function to toggle the response pin. This will cause two context switches between two peaks of the ”response pin”: Interrupt occurs - Userspace task will be woken up - Userspace Task calls back into kernel to toggle the pin. Be careful: We can’t force the UIO interrupt handler to run in hard interrupt context. The UIO core uses wakeup_interruptible (which is not allowed in hard interrupt context). Since the read() call to the UIO driver will return the number of interrupt events, we use this to check, if we have missed an interrupt event. The UIO driver can be implemented as a UIO platform driver (uio_pdrv), which can be registered in the board support. The following listings show, how this mechanism can be implemented:

```

static irqreturn_t uio_latency_handler
        (int irq,
         struct uio_info *dev_info)
{
 at91_set_gpio_value(AT91_PIN_PB21, 1);
 udelay(2);
 at91_set_gpio_value(AT91_PIN_PB21, 0);

 /* edge triggered interrupt
  * so just returning IRQ_HANDLED is ok
  */
 return IRQ_HANDLED;
}

int uio_write_gpio
        (struct uio_info *info,
         s32 irq_on)
{
 at91_set_gpio_value(AT91_PIN_PB21, 1);
 udelay(2);
 at91_set_gpio_value(AT91_PIN_PB21, 0);
 return 0;
}

struct uio_info uioinfo_latency = {
 .name = "at91_latency_measurement",
 .version = "0.0.1",
 .handler = uio_latency_handler,
 .irqcontrol = uio_write_gpio,
};

static struct platform_device
                uio_latency = {
 .name = "uio_pdrv",
 .id = -1,
 .dev.platform_data = &uioinfo_latency,
};

void __init
      at91_add_device_uio_latency(void)
{
 uioinfo_latency.irq =
        gpio_to_irq(AT91_PIN_PB19);
 /* configuring the gpio pins */
 at91_set_gpio_input(AT91_PIN_PB19, 0);
 at91_set_deglitch(AT91_PIN_PB19, 1);
 at91_set_gpio_output(AT91_PIN_PB21, 0);

 platform_device_register(&uio_latency);
}
```

The userspace task for the test environment looks like:

```
int32_t num_events = 0, old_events = 0;
int32_t irq_on = 1;
...
fd = open("/dev/uio0", O_RDWR);
...
while(1) {
  old_events = num_events;
  read(fd, &num_events,
        sizeof(num_events));
  if((num_events - old_events) > 1
                && !first_run)
          printf("Missed␣events\n");
  write(fd, &irq_on, sizeof(irq_on));
  if(first_run)
        first_run = 0;
}
```

The accuracy of a timer task can be benchmarked with cyclictest. Cyclictest is a high resolution timer test software, which has been initially written by Thomas Gleixner. It’s part of the rt test tools (maintained by Clark Williams). Cyclictest is highly configurable and also includes some tracing options for ftrace and other kernel tracers. Since cyclictest covers several codepaths (we set up a timer - the systems programs the timer chip - the timer interrupt occurs - the system has to wake up the task) it gives a real good overview of the realtime performance of a system.

### 4.1.3 Test scenarios used for the measurements

Based on the scenarios shown in Chapter 4.1.1 and Chapter 4.1.2 we define the following test environment:

1. External events: A function generator will be set up to generate events with a frequency of 1000 Hz. A UIO driver and a userspace task will respond to those events. The interrupt response time and the time which is needed for two context switches will be measured. The Interrupt handler will be priorized with SCHED_FIFO 99 and the userspace task with SCHED_FIFO 98.

2. Timer accuracy: Cyclictest will be set up to create a timer task based on clock_nanosleep. The timer interval is 1ms, the priority is SCHED_FIFO 80.

For both test cases load will be generated with hackbench and with a "Ping Flooding" from a different machine.

## 4.2 ARM9

### 4.2.1 Target hardware

The measurements were taken on an Atmel AT91SAM9263 evaluation kit. Technical data:

- CPU: Atmel AT91SAM9263 (ARM926-EJS) @ 180MHz
- RAM: 64MB SD-RAM

### 4.2.2 Results

Figure 3 shows the result for the interrupt latency. The falling edge of the upper channel shows the interrupt event. The rising edge of the lower channel is triggered by the IRQ handler. The worst-case interrupt latency in a long-time run was at 248us.

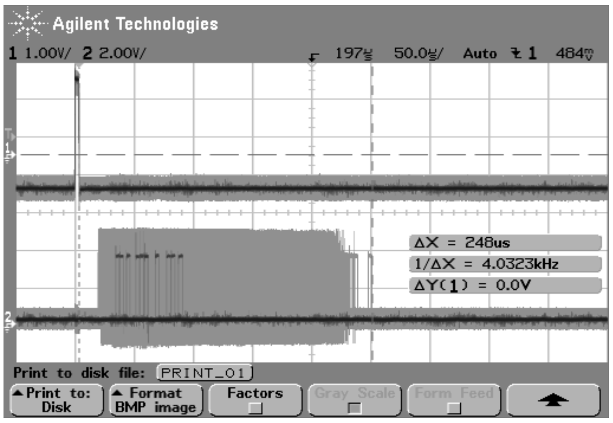


**FIGURE 3:** *IRQ latency ARM9*

Figure 4 combines the interrupt latency and the context switching time. The worst-case latency for waking up a userspace task and calling back into the kernel was 374us. The userspace task didn't miss any interrupt events.

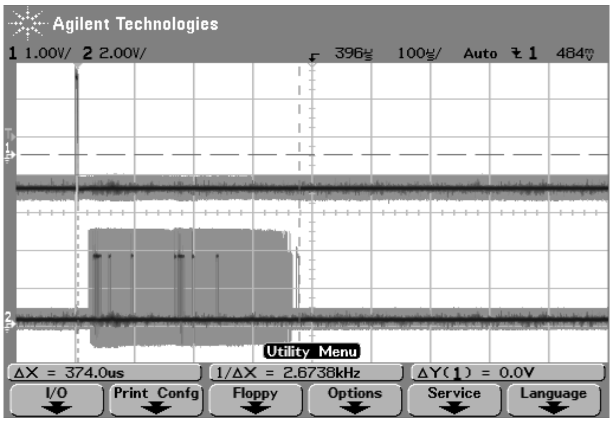


**FIGURE 4:** *context switching time on ARM9*

Figure 5 shows the histogram of the cyclictest run. The average latency is at about 85us, the worst-case latency is 312us. In chapter 4.1.2 I mentioned, that cyclictest gives a good overview of the realtime behaviour of a system. This cyclictest run proves that theory once again, since the test run shows the same worst-case latencies as the interrupt latency measurements.

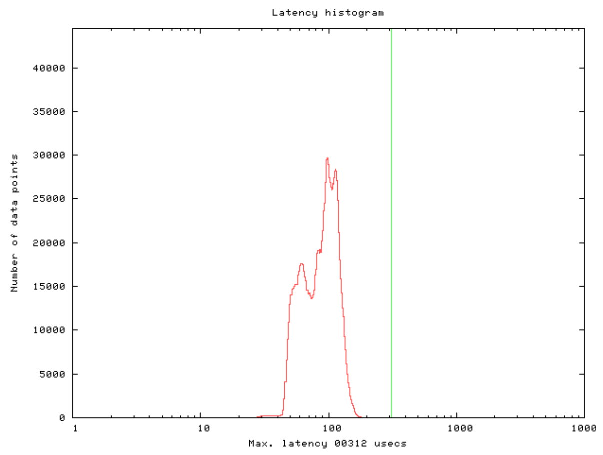


**FIGURE 5:** *High resolution timer on ARM9*

## 4.3 ARM11

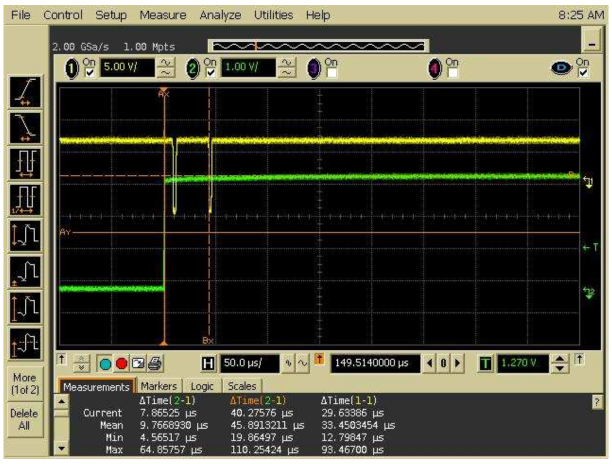
### 4.3.1 Target hardware

These measurements were done on a Garz & Fricke Adelaide module, which is based on a Freescale i.mx31 CPU. Technical data:

- CPU: Freescale i.MX31 @ 532 Mhz
- RAM: 64 MB DDR RAM

### 4.3.2 Results

Figure 6 shows the results for the interrupt latency and the context switching time. The rising edge of the lower channel shows the occurancy of the irq event. The first peek on the upper channel shows the responce of the UIO handler. The second peek on the upper channel shows the response which is generated by the write() call of the userspace task. A long-time test run showed a worst-case latency for the UIO handler of about 65us. The worst-case latency for the UIO handler + 2 context switches was about 110us. The application didn't miss an interrupt event.



```
         ΔTime(2-1)      ΔTime(2-1)       ΔTime(1-1)
Current  7.86525 µs      40.27576 µs      29.63386 µs
   Mean  9.7668930 µs    45.8913211 µs    33.4503454 µs
    Min  4.56517 µs      19.86497 µs      12.79847 µs
    Max  64.85757 µs     110.25424 µs     93.46700 µs
```

**FIGURE 6:** *IRQ latency and context switching time on ARM11*

Figure 7 shows the result for the cyclictest run. It shows an average latency at 35us and a worst-case latency at 72us.

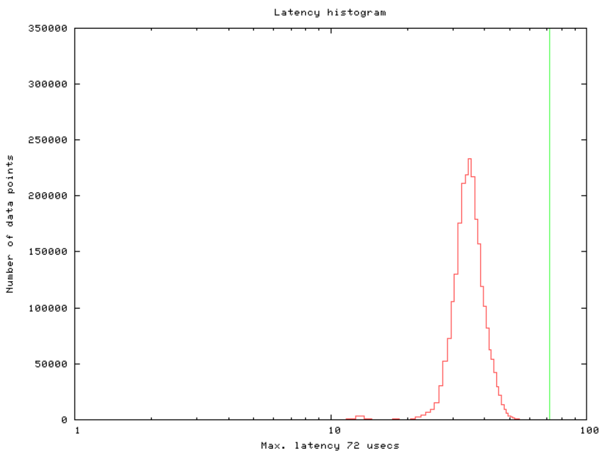


**FIGURE 7:** *High resolution timer on ARM11*

Since the results of those test were quite impressing, one more test scenario has been added for the ARM11 case. The Interrupt frequency has been increased until the system became unstable. In this scenario the system kept beeing responsive up to a interrupt frequency of 150kHz! After reducing the frequency the system startet to be responsive again! So the system didn't get into an unstable state. It was just busy servicing interrupts.

## 5 Conclusion

### 5.1 Analyzing the results

The benchmarking results on both platforms have proven a 100% deterministic behaviour of Preempt RT on ARM. So, Preempt RT on ARM CPUs can meet hard realtime requirements. Table 2 contrasts the benchmarking results of ARM9 with the results on the ARM11 platform.

| | ARM9 (AT91SAM9263) | ARM11 (i.mx31) |
|---|---|---|
| IRQ latency | 248us | 68us |
| UIO wakeup | 374us | 110us |
| cyclictest | 312us | 72us |

**TABLE 2:** *Comparison of the worst-case latencies on ARM9 and ARM11*

It's quite obvious, that ARM11 can achieve much better latencies than ARM9. This is mainly related to the caching issue mentioned in chapter 2.1. The worst-case latencies on ARM11 are about 200us lower.

### 5.2 Decision guide

ARM9 based systems can achieve worst-case latencies of about 250us. If we are running a cyclic task, the worst-case latencies shouldn't exceed 25% of our cycletime. So, the lowest cycletime which can be achieved on ARM9 platforms is about 1ms. The worst-case latencies on ARM11 are much better. The worst-case latency is at about 100us. This should be sufficient for cycle times down to 400/500us. So, when worst-case latencients lower than 250us and cycletimes lower than 1ms are needed the system should be based on an ARM11 or a Cortex A8 design (which is designed to meet the requirements of the ARM11 platmorm). Figure 8 gives an overview of the boundary conditions for choosing an appropriate ARM design.

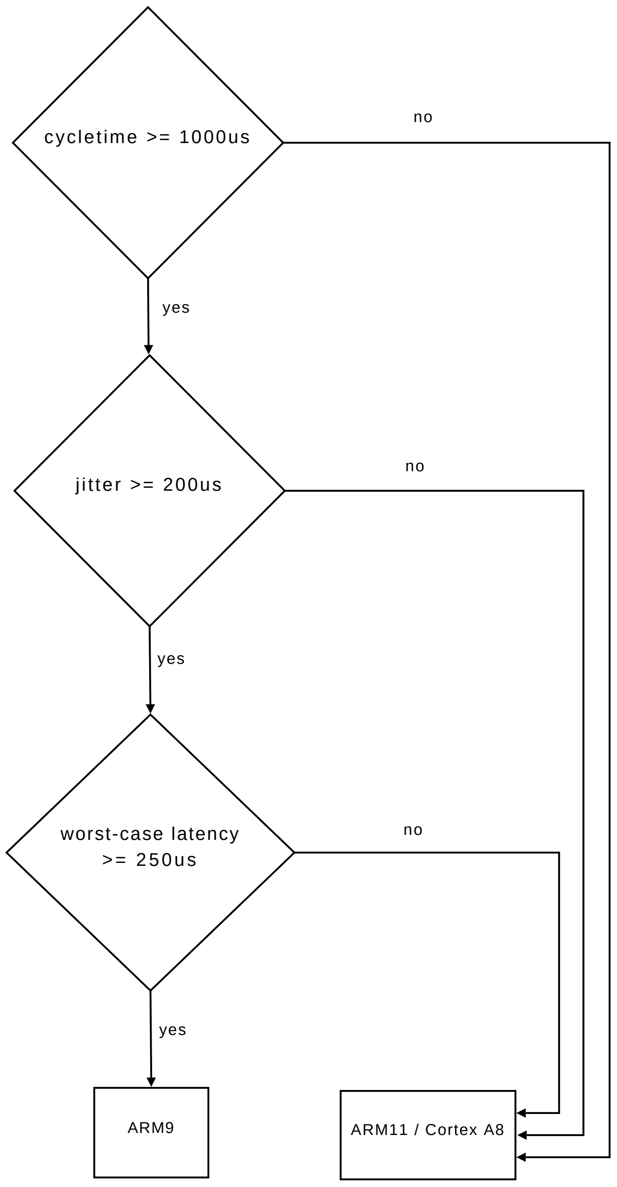


**FIGURE 8:** *Chosing an appropriate ARM CPU for Preempt RT*

## References

[1] The RT Wiki: http://rt.wiki.kernel.org/-index.php/Main_Page

[2] The OSADL Web Page: http://www.osadl.org

[3] Richard Cochrans FCSE Patches: http://www.opensubscriber.com/message/-xenomai-core@gna.org/7615242.html

[4] Gilles Chanteperdrix' enhanced FCSE Patch: http://article.gmane.org/-gmane.linux.ports.arm.kernel/47943

[5] Building Embedded Linux Systems (2nd Edition) Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Phillipe Gerum, Steven Rostedt. Published by O'Reilly

[6] The rt test tools: git://git.kernel.org/pub/scm/linux/kernel/git/-clrkwllms/rt-tests.git