RTAI

freescale™ semiconductor

This document describes setting up and usage of RTAI for MCF54455 installed into Linux embedded OS, and the changes in RTAI and Linux kernel source code, which allow using RTAI with MCF54455.

This document targets Linux software developers using the MCF54455 processor.

[1] MCF54455 Reference Manual Rev. 0

[2] RTAI 3.4 User Manual Rev 0.3

[3] Advanced Linux Programming. M. Mitchel, J. Oldham, A. Samuel

The following list defines the acronyms and abbreviations used in this document.

ADEOS	Adaptive Domain Environment for Operating Systems, a nanokernel used by RTAI
FEC	ColdFire Fast Ethernet Controller
FIFO	First Input First Output
HAL	Hardware Abstraction Layer
I-Pipe	Interrupt Pipeline
LED	Light-Emitting Diode
OS	Operating System
RDTSC	Read Time Stamp Counter – the function returning number of ticks from the system start.
RTAI	Real Time Application Interface
RTC	Real Time Clock
SRQ	System Request
UART	Universal Asynchronous Receiver/Transmitter
BSP	Board Support Package

LTIB	Linux Target Image Builder

This document describes the Real Time Application Interface (RTAI), ported to the MCF54455. RTAI is a Linux kernel extension, that allows preemption of the Linux kernel at any time in order to perform real time operations with interrupt latencies in the microseconds range. The standard Linux kernel can have latencies of several milliseconds.

The document is divided logically into five parts.

The first part contains a general overview of RTAI.

The second part contains the description of its installation and usage.

The third part describes the changes, which were made in the Linux kernel 2.6.23 and Linux drivers. Mainly it contains information about modifications of the interrupt handling routines and timer routines.

The fourth part is a description of the changes made in the RTAI source code during porting.

The fifth part gives a short manual of creation of RTAI applications.

- Correct execution of the real time tasks in periodic mode with the frequencies 3 kHz and less (In this mode real time task period have to be a multiple of the timer period).

- Correct execution of the real time tasks in oneshot mode with the frequencies 5 kHz and less (In this mode real time task period have to be a variable value based on the timer clock frequency).

- RTAI services are provided by 14 kernel modules, which allow hard real time, fully preemptive scheduling. These modules are: , , , , , , , , , , , , . Note that 15-th module, , contains no code, so there is no reason to use it.

This section briefly describes RTAI’s real time services. They are provided via kernel modules, which can be loaded and unloaded using the standard Linux and commands. Although the and (or) modules are required every time any real time service is needed, all other modules are necessary only when their associated real time services are desired.

rtai_hal

It's the RTAI hardware abstraction layer used by other RTAI modules. It offers interrupt handling and timing functions.

rtai_lxrt

It's a real time, preemptive, priority-based scheduler, modified to work on MCF54455. It's simply a GNU/Linux co-scheduler. This means that it supports hard real time for all Linux schedulable objects like processes/threads/kthreads.

rtai_sched

It's a real time, preemptive, priority-based scheduler, modified to work on MCF54455. The ***rtai_sched*** instead supports not only hard real time for all Linux schedulable objects, like processes/threads/kthreads, but also for RTAI own kernel tasks, which are very light kernel space only schedulable objects proper to RTAI.

rtai_fifos

Real time FIFOs are included into this module.

rtai_wd

It's a watchdog module intended for controlling RTAI tasks for overruns that is able to perform some actions, like killing those tasks.

rtai_msg

It's RTAI message handling and rpc functions.

rtai_bits

It's RTAI event flags functions.

rtai_mq

It's POSIX-like message queues.

rtai_sem

It's RTAI semaphore functions.

rtai_netrpc

It's a module for network real time communications.

It's RTAI message queues.

It's RTAI mailbox functions.

rtai_tasklets

It's an RTAI's implementation of tasklets. RTAI tasklets are used when functions are needed to be called from user- and kernel-space.

rtai_shm

It's RTAI shared memory functions.

The following files are relevant to RTAI:

- rtai-3.8.tar.bz2 – RTAI 3.8 original package.

This chapter describes how to install and patch Linux BSP and RTAI and deploy it on M54455 evaluation board. It also contains a general overview of RTAI and information about RTAI installation.

True multi-tasking operating systems, such as Linux, are adopted for use in increasingly complex systems, where the need for hard real time often becomes apparent. “Hard real time” can be found in the systems, which are dependent from guaranteed system responses of thousandths or millionths of a second. Since these control deadlines can never be missed, a hard real time system cannot use average case performance to compensate for worst-case performance.

There are four primary variants of hard real time Linux available: RTLinux, Xenomai and RTAI.

RTLinux was developed at the New Mexico Institute of Technology by Michael Barabanov under the direction of Professor Victor Yodaiken. Real Time Application Interface (RTAI) was developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza. One of the main advantages of RTAI is the support of periodic mode scheduling and its performance. Xenomai, that was launched in 2001 provides slightly worser performance comparing to RTAI.

For the real time Linux scheduler the Linux OS kernel is an idle task. Therefore Linux executes only when the real time tasks aren’t running and the real time kernel isn’t active. RTAI 3.8 uses ADEOS nanokernel for managing interrupts. ADEOS provides Interrupt Pipeline (called I-Pipe), that delivers interrupts to domains. One of domains is Linux, the second – RTAI. In hard real time mode (when there is a real time task running) Linux domain is in “stalled” state, which means it doesn't receive interrupts. So Linux kernel doesn't schedule, because timer interrupt never occurs. In hard real time interrupts are delivered to RTAI scheduler, which manages RTAI tasks.

To install the Linux BSP for MCF54455 refer to the “BSP Targeting the Freescale ColdFire M54455EVB” User's Guide from the Linux BSP distribution.

To install RTAI, the following steps must be performed:

1. Add RTAI to Linux BSP

   Create RTAI archive md5 sum:

   ```
   $ md5sum rtai-3.8.tar.bz2 > rtai-3.8.tar.bz2.md5
   ```

   Copy the following files to the Local Package Pool. (*/opt/freescale/pkgs* directory):

   ```
   2. rtai-3.8.tar.bz2
      rtai-3.8.tar.bz2.md5
   ```

   Extract RTAI archive. Copy the following files from *<rtaidir>/base/arch/m68k/patches/MCF54455/BSP20071214/patches* to the Local Package Pool. (*/opt/freescale/pkgs* directory):

   ```
   lin2-6-23-m5445x-0049-I-Pipe.patch
   lin2-6-23-m5445x-0049-I-Pipe.patch.md5
   ```

   Copy file '*ltib-rtai-3.8-mcf54455.patch*' from *<rtaidir>/base/arch/m68k/patches/MCF54455/BSP20071214/patches* to the directory where BSP is installed.

   Go to BSP dir and apply this patch:

   ```
   $ patch -p1 < ltib-rtai-3.8-mcf54455.patch
   ```

   Go back to *<rtaidir>/base/arch/m68k/patches/MCF54455/BSP20071214/patches* and apply glibc patch (must be done under *root* user):

   ```
   # patch -d / -p1 < glibc-fix.patch
   ```

2. Build target image

After the path is applied, run LTIB configuration script:

```
$ ./ltib --configure
```

Afterwards do the following selections:

```
Configure the kernel = y
Package list->rtai = y
```

Selecting rtai will automatically select `"Leave sources after building"` option.

If you want to configure RTAI, set the following item:

```
Package List->RTAI->Configure RTAI at build time = y
```

Then exit with saving options.

In the kernel configuration window select `"Platform dependent setup->Interrupt pipeline"`

Exit from kernel configuration with saving options.

If you have selected `"Configure RTAI at build time"`, then RTAI configuration dialog will appear. Make your changes and exit.

After this command completes, RTAI modules and testsuite will be deployed in the rootfs.

RTAI creates device files for RTAI FIFOs, so the following command must be run to add those files to rootfs:

```
$ ./ltib -p dev -f
```

3. Now it is possible to load and run Linux on the target board as described in the “BSP Targeting the Freescale ColdFire M54455EVB” User's Guide.

   Now load RTAI modules and get information through `/proc` filesystem:

```
# cd /usr/realtime/modules

# insmod rtai_hal.ko

# insmod rtai_sched.ko

# insmod rtai_fifos.ko
```

... (and so on, all RTAI modules you need)

To launch RTAI testsuite , some additional steps must be performed:

1. Insert required modules:

```
# insmod rtai_hal.ko
# insmod rtai_sched.ko
# insmod rtai_sem.ko
# insmod rtai_fifos.ko
# insmod rtai_mbx.ko
# insmod rtai_msg.ko
```

2. Launch RTAI tests:

Kernel-space *latency* test in oneshot mode:

```
# insmod latency_rt.ko
# ../testsuite/kern/latency/display
```

test will start displaying its results, until you press Ctrl+C.

```
# rmmod latency_rt.ko
```

Kernel-space *latency* test in periodic mode:

```
# insmod latency_rt.ko timer_mode=1
# ../testsuite/kern/latency/display
```

test will start displaying its results, until you press Ctrl+C.

```
# rmmod latency_rt.ko
```

Kernel-space *preempt* test:

```
# insmod preempt_rt.ko
# ../testsuite/kern/preempt/display
```

test will start displaying its results, until you press Ctrl+C.

```
# rmmod preempt_rt.ko
```

Kernel-space *switches* test:

```
# insmod switches_rt.ko
```

test will display its results in a few seconds.

```
# rmmod switches_rt.ko
```

User-space *latency* test in oneshot mode:

```
# cd /usr/realtime/testsuite/user/latency

# ./latency&

# ./display
```

test will start displaying its results, until you press `ENTER`.

User-space *preempt* test:

```
# cd /usr/realtime/testsuite/user/preempt

# ./preempt&

# ./display
```

test will start displaying its results, until you press Ctrl+C. Also you will need to type

```
# killall preempt
```

in order to stop user-space preempt test.

User-space *switches* test:

```
# cd /usr/realtime/testsuite/user/switches

# ./switches
```

test will display its results in a few seconds.

The description for each test can be found in README file in the test directory.

Notice 1:

You should enable I-Pipe in your kernel only if you want to use RTAI. I-Pipe slows down interrupt processing in Linux, that will be visible when working with drivers that use lots of interrupts for data transfers instead of DMA. If both I-Pipe and fast interrupt processing is required, then either use RTAI drivers (that should be written by you) or use hacks in I-Pipe to pass some interrupts directly to drivers (not via I-Pipe). The second way must be used very carefully and with full understanding of what you are doing.

Notice 2:

If RTAI is active then Linux interrupts are not called immediately. Instead, each captured interrupt is and then hardware interrupts will be enabled. Linux handler (if it exists) will be called when all RTAI tasks will become inactive. The interrupt will be immediately after the Linux handler. This will not interfere with existing Linux BSP drivers, because all of them use kernel functions for masking/unmasking interrupts instead of working directly with interrupt controller registers. These kernel functions are modified by I-Pipe patch to make I-Pipe masking/unmasking described earlier transparent to drivers. If you are writing your own driver you must use these kernel functions (and) for masking and unmasking interrupts instead of direct access to interrupt controller registers.

Both Linux kernel and I-Pipe source code has been changed during porting. Changes affect architecture-dependent part. I-Pipe was ported to the m68k architecture.

1. The `return` function from was modified to call `EMULATE_ROOT_IRET` macro. It was made to process syscalls correctly with the I-Pipe;
2. The `system_call` function from was modified to call `CATCH_ROOT_SYSCALL` macro. It was made to deliver syscall to the I-Pipe;
3. The `trap` function from was modified to call `__ipipe_handle_exception()` function. It was made to deliver exceptions to the I-Pipe;
4. The `buserr` function from was modified to call `__ipipe_handle_buserr()` function. It was made to handle TLB misses in the I-Pipe and deliver page faults to the I-Pipe;
5. The `inthandler` function from was modified to disable interrupts at the beginning of the execution;
6. The `inthandler` function from was modified to call `ipipe_irq_handler()` function instead of Linux interrupt handler. It was made to deliver interrupts to the I-Pipe;
7. All references to the `current` variable in the `inthandler` function from were removed. It was made because this variable is not available in interrupt handling routine if interrupt occurs during RTAI task execution;
8. All interrupt enabling/disabling routines from were modified to use the I-Pipe stall/unstall domain functions instead of hardware interrupt managing. The functions with `_hw` suffix which implement hardware interrupts enabling/disabling were added;
9. The `read_timer_cnt()` function was added to This function calculates the number of timer ticks

passed from the timer initialization It is used to implement the *`rdtsc()`* function in the RTAI part;

10. The *`ack_linux_disable()`* function was added to . It was made to perform temporary masking of interrupts of different devices in the I-Pipe;
11. *`_ipipe_post_irq`* array was added to . This array contains pointers to the functions for each interrupt that must be called after interrupt handling. This allows to enable interrupt disabled in *`ack_linux_disable()`* function after it is properly handled;
12. The *`coldfire_sched_init()`* function from was modified to initialize timer to work with higher timer precision;
13. The *`ack_linux_tmr()`* function was added to to perform correct timer interrupt acknowledgment in I-Pipe;
14. The *`ipipe_mask_irq_hw()`* and *`ipipe_unmask_irq_hw()`* routines were added to *`arch/m68k/coldfire/ipipe_masks.c`* to add a possibility to perform interrupt masking and unmasking (see 10 and 11) transparently for Linux drivers. Also, all Linux masking unmasking routines ( ) were modified to not unmask interrupts if they are already masked by I-Pipe.
15. The routine from was modified to initialize data structures required by masking/unmasking code.
16. Unmasking code was also replaced in routine from and in from .