RTAI Port to MCF5329

Developer's manual

M5329/RTAI3.6.2

Rev. 0.2 11/2008

CONTENTS

About This Document

This document describes setting up and usage of RTAI for MCF5329 installed into μClinux embedded OS, and the changes in RTAI and Linux kernel source code, which allow using RTAI with MCF5329.

Audience

This document targets μClinux software developers using the MCF5329 processor.

Suggested Reading

[1] MCF5329 Reference Manual Rev. 0

[2] RTAI 3.4 User Manual Rev 0.3

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

Definitions Acronyms and Abbreviation

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
μClinux	Micro-Controller version of Linux OS for Embedded Applications

1. Introduction

This document describes the Real Time Application Interface (RTAI), ported to the MCF5329. 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 µClinux kernel 2.6.25 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.

1.1. RTAI Features for MCF5329

- 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: *rtai_hal*, *rtai_sched*, *rtai_lxrt*, *rtai_fifos*, *rtai_wd*, *rtai_msg*, *rtai_bits*, *rtai_mq*, *rtai_sem*, *rtai_netrpc*, *rtai_tbx*, *rtai_mbx*, *rtai_signal*, *rtai_tasklets*. Note that 15-th module, *rtai_usi*, contains no code, so there is no reason to use it.

1.2. RTAI's Services Overview

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 *insmod* and *rmmod* commands. Although the *rtai_hal* and *rtai_sched*(or *rtai_lxrt*) 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.

1.2.1. Module *rtai_hal*

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

1.2.2. Module *rtai_lxrt*

It's a real time, preemptive, priority-based scheduler, modified to work on MCF5329. 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.

1.2.3. Module *rtai_sched*

It's a real time, preemptive, priority-based scheduler, modified to work on MCF5329. 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.

1.2.4. Module *rtai_fifos*

Real time FIFOs are included into this module.

1.2.5. 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.

1.2.6. Module *rtai_msg*

It's RTAI message handling and rpc functions.

1.2.7. Module *rtai_bits*

It's RTAI event flags functions.

1.2.8. Module *rtai_mq*

It's POSIX-like message queues.

1.2.9. Module *rtai_sem*

It's RTAI semaphore functions.

1.2.10. Module *rtai_netrpc*

It's a module for network real time communications.

1.2.11. Module rtai_tbx

It's RTAI message queues.

1.2.12. Module rtai_mbx

It's RTAI mailbox functions.

1.2.13. Module *rtai_signal*

It's RTAI signal services.

1.2.14. Module *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.

1.3. Related files

The following files are relevant to RTAI:

- uClinux-dist-20080808.tar.bz2 – source code of the μClinux.
- rtai-3.6.2.tar.bz2 – RTAI 3.6.2 original package.

- m68k-uclinux-tools-20061214.sh - m68k-uclinux tool chain for μClinux and RTAI compilation.
- rtai3.6.2-mcf5329.patch – patch for RTAI to support MCF5329.
- uClinux-rtai-mcf5329_2.6.25.patch – patch for μClinux 2.6.25 kernel containing I-Pipe.

2. RTAI Installation and Usage

This chapter describes how to install and patch μClinux and RTAI, download image with μClinux and it's real time extension for MCF5329. It also contains a general overview of RTAI and information about RTAI installation.

2.1. RTAI General Overview

2.1.1. Hard real time

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.

2.1.2. RTAI and other real time projects

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.

2.1.3. RTAI implementation

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 3.6.2 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.

2.2. m68k-elf Tool Chain Setup

To install the m68k-uclinux tool chain for correct µClinux and RTAI compilation, the following steps must be accomplished:

1. Login as root
2. Run *m68k-uclinux-tools-20061214.sh*

2.3. µClinux and RTAI installation. Patching and Compilation

To install and patch µClinux and RTAI, the following steps must be performed:

1. Copy µClinux archive (*uClinux-dist-20080808.tar.bz2*) into the */opt/rtai* directory (it developer wants to have uClinux-dist directory with µClinux in the other directory, the sequence of actions will be the same – only change of paths is needed).
2. Extract its content:

```
$ tar xf uClinux-dist-20080808.tar.bz2
```

3. Copy RTAI archive (*rtai-3.6.2.tar.bz2*) into the */opt/rtai* directory (location also can be changed)
4. Extract its content:

```
$ tar xf rtai-3.6.2.tar.bz2
```

5. If the patches are stored in the */opt/rtai/patches* directory, execute the following commands to patch µClinux and RTAI code:

```
$ cd /opt/rtai/rtai-3.6.2

$ patch -p1 < /opt/rtai/patches/rtai3.6.2-mcf5329.patch

$ cd /opt/rtai/uClinux-dist

$ patch -p1 < /opt/rtai/patches/uClinux-rtai-mcf5329_2.6.25.patch
```

6. Configure the Linux kernel:

```
$ cd /opt/rtai/uClinux-dist
$ make menuconfig
```

It's possible to use `xconfig` and `config` if the developer prefers such way.

Afterwards do the following selections:

```
Vendor/Product Selection->Vendor = Freescale

Vendor/Product Selection->Freescale Products = M5329EVB

Kernel/Library/Defaults Selection->Kernel Version = linux-2.6.x

Kernel/Library/Defaults Selection->libc Version = uClibc

Kernel/Library/Defaults Selection->Customize Kernel Settings = y

Kernel/Library/Defaults Selection->Customize Application/Library
Settings = y
```

Then save and exit. The second window “Kernel Configuration” will be shown. Here the following should be selected:

```
Enable loadable module support = y

Enable loadable module support/Module unloading = y

Processor type and features->Interrupt Pipeline = y
```

Then save and exit. The third window “uClinux Distribution Configuration” will be shown. Here the following should be selected:

```
Busybox/Busybox = y

Busybox/Busybox/Linux Module Utilites/insmod = y

Busybox/Busybox/Linux Module Utilites/lsmod = y

Busybox/Busybox/Linux Module Utilites/rmmod = y

Busybox/Busybox/Linux Module Utilites/Support version 2.6.x Linux
kernels = y
```

Then save and exit.

POSIX threading support is required to compile some RTAI testsuite files. To enable POSIX threading support configure uClibc:

```
$ cd uClibc
$ make menuconfig
```

In the configuration window set:

```
General Library Settings/POSIX Threading Support = y
```

Then save and exit.

Go back to μClinux root directory:

```
$ cd /opt/rtai/uClinux-dist
```

7. Build μClinux:

```
$ make
```

8. Configure RTAI:

```
$ cd /opt/rtai/rtai-3.6.2

$ make ARCH=m68knommu CROSS_COMPILE=m68k-uclinux- menuconfig
```

In the configuration window set:

```
General->Linux Source Tree = /opt/rtai/uClinux-dist/linux-2.6.x
```

9. Build RTAI and copy modules to the target file system:

```
$ make

$ cd base

$ install -d /opt/rtai/uClinux-dist/romfs/lib/modules/rtai

$ find -name *.ko -exec cp '{}' /opt/rtai/uClinux-
dist/romfs/lib/modules/rtai \;
```

10. Then compile μClinux again:

```
$ cd /opt/rtai/uClinux-dist

$ make
```

11. Now it is possible to load and run μClinux using the `dBUG` monitor of the built-in ROMs. Use the `dn` command to load the image. And then type `go 40020000` to run it.

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

```
# insmod rtai_hal.ko

# insmod rtai_sched.ko

# insmod rtai_fifos.ko
```

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

Go to `/proc`:

```
# cd /proc/rtai

# cat hal

** RTAI/m68knommu:



** Real-time IRQs used by RTAI: none


** RTAI extension traps:


    SYSREQ=0x2b


** RTAI SYSREQs in use: #1 #2

# cat scheduler

RTAI LXRT Real Time Task Scheduler.


    Calibrated CPU Frequency: 240000000 Hz

    Calibrated interrupt to scheduler latency: 81991 ns

     Calibrated oneshot timer setup_to_firing time: 8008
ns
```

```
Number of RT CPUs in system: 1 (sized for 1)


Real time kthreads in resorvoir (cpu/#): (0/1)


Number of forced hard/soft/hard transitions: traps 0,
syscalls 0


Priority  Period(ns)  FPU  Sig  State  CPU  Task  HD/SF
PID  RT_TASK *  TIME
------------------------------------------------------------------
---------------------

TIMED


READY
```

2.4. RTAI testsuite installation

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

1. `General->Build RTAI testsuite` option must be selected when configuring RTAI.
2. After building testsuite files must be copied to the target filesystem:

```
$ UC_RT_TEST=/opt/rtai/uClinux-dist/romfs/rtai-testsuite

$ install -d $UC_RT_TEST/kern/latency

$ install -d $UC_RT_TEST/kern/preempt

$ install -d $UC_RT_TEST/kern/switches

$ install -d $UC_RT_TEST/user/latency

$ install -d $UC_RT_TEST/user/preempt
$ install -d $UC_RT_TEST/user/switches

$ cd /opt/rtai/rtai-3.6.2/testsuite

$ cp kern/latency/latency_rt.ko $UC_RT_TEST/kern/latency/

$ cp kern/latency/display $UC_RT_TEST/kern/latency/

$ cp kern/preempt/preempt_rt.ko $UC_RT_TEST/kern/preempt/

$ cp kern/preempt/display $UC_RT_TEST/kern/preempt/

$ cp kern/switches/switches_rt.ko $UC_RT_TEST/kern/switches/

$ cp user/latency/latency $UC_RT_TEST/user/latency/

$ cp user/latency/display $UC_RT_TEST/user/latency/

$ cp user/preempt/preempt $UC_RT_TEST/user/preempt/

$ cp user/preempt/display $UC_RT_TEST/user/preempt/

$ cp user/switches/switches $UC_RT_TEST/user/switches/
```

3. Testsuite uses RTAI FIFOs, so FIFO device files should be created:

```
$ touch /opt/rtai/uClinux-dist/romfs/dev/@rtf0,c,150,0

$ touch /opt/rtai/uClinux-dist/romfs/dev/@rtf1,c,150,1

$ touch /opt/rtai/uClinux-dist/romfs/dev/@rtf2,c,150,2

$ touch /opt/rtai/uClinux-dist/romfs/dev/@rtf3,c,150,3
```

4. Rebuild µClinux

```
$ cd /opt/rtai/uClinux-dist

$ make
```

5. Now it is possible to load and run µClinux using the `dBUG` monitor of the built-in ROMs. Use the `dn` command to load the image. And then type `go 40020000` to run it.

6. When µClinux is launched on the board, 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

# cd /rtai-testsuite
```

7. Launch RTAI tests:

Kernel-space *latency* test in oneshot mode:

```
# cd kern/latency

# insmod latency_rt.ko

# ./display
```

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

```
# rmmod latency_rt.ko

# cd ../..
```

Kernel-space *latency* test in periodic mode:

```
# cd kern/latency

# insmod latency_rt.ko timer_mode=1

# ./display
```

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

```
# rmmod latency_rt.ko

# cd ../..
```

Kernel-space *preempt* test:

```
# cd kern/preempt

# insmod preempt_rt.ko

# ./display
```

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

```
# rmmod preempt_rt.ko
```

```
# cd ../..
```

Kernel-space *switches* test:

```
# cd kern/switches
```

```
# insmod switches_rt.ko
```

Switches test will be displaying its results after few seconds.

```
# rmmod switches_rt.ko
```

```
# cd ../..
```

User-space *latency* test in oneshot mode:

```
# cd user/latency
```

```
# ./latency&
```

```
# ./display
```

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

```
# cd ../..
```

User-space *latency* test in periodic mode:

You will need to modify latency test, rebuild RTAI and μClinux and launch μClinux on the board again.

First, in the following file

```
/opt/rtai/rtai-3.6.2/testsuite/user/latency/latency.c
```

change line 38 from

```
#define TIMER_MODE  0
```

to

```
#define TIMER_MODE  1
```

After it rebuild RTAI:

```
$ cd /opt/rtai/rtai-3.6.2

$ make
```

Then copy updated `latency` executable to the `romfs`:

```
$ UC_RT_TEST=/opt/rtai/uClinux-dist/romfs/rtai-testsuite

$ cp /opt/rtai/rtai-3.6.2/testsuite/user/latency/latency
/opt/rtai/uClinux-dist/romfs/rtai-testsuite/user/latency/
```

And rebuild µClinux:

```
$ cd /opt/rtai/uClinux-dist

$ make
```

Then load and run µClinux using the `dBUG` monitor of the built-in ROMs. Use the `dn` command to load the image. And then type `go 40020000` to run it.

Insert required modules again:

```
# insmod rtai_hal.ko

# insmod rtai_sched.ko

# insmod rtai_sem.ko

# insmod rtai_fifos.ko

# insmod rtai_mbx.ko

# insmod rtai_msg.ko

# cd /rtai-testsuite
```

And finally launch user-space latency test in periodic mode:

```
# cd user/latency

# ./latency&

# ./display
```

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

```
# cd ../..
```

User-space *preempt* test:

```
# cd user/preempt
# ./preempt&
# ./display
```

Preempt test will be displaying its results, until you press Ctrl+C twice.

```
# cd ../..
```

User-space *switches* test:

```
# cd user/switches
# ./switches
```

Switches test will be displaying its results after few seconds.

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

3. Changes in the µClinux and I-Pipe Source Code

Both µClinux and I-Pipe source code has been changed during porting. Changes affect architecture-dependent part. I-Pipe was ported to the m68knommu architecture.

1. The `return` function from *arch/m68knommu/platform/coldfire/entry.S* 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 *arch/m68knommu/platform/coldfire/entry.S* was modified to call `CATCH_ROOT_SYSCALL` macro. It was made to deliver syscall to the I-Pipe;

3. The `trap` function from *arch/m68knommu/kernel/entry.S* was modified to call `__ipipe_handle_exception()` function. It was made to deliver exceptions to the I-Pipe;

4. The `inthandler` function from *arch/m68knommu/platform/5307/entry.S* was modified to call `ipipe_irq_handler()` function instead of Linux interrupt handler. It was made to deliver interrupts to the I-Pipe;

5. All interrupt enabling/disabling routines from *include/asm-m68knommu/system.h* 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;

6. The `read_timer_cnt()` function was added to *arch/m68knommu/platform/coldfire/timers.c*. This function calculates the number of timer ticks passed from the timer initialization. It is used to implement the `rdtsc()` function in the *m68knommu* RTAI part;

7. The `ack_linux_icr0()` function was added to *arch/m68knommu/kernel/ipipe.c*. It was made to perform correct acknowledgment of the FEC and UART interrupt routines in the I-Pipe;

8. The `FREQ` macro definition was modified in *arch/m68knommu/platform/coldfire/timers.c* to equal to `MCF_BUSCLK`. It was made because of another timer frequency;

9. The `mcftmr_tick()` function from *arch/m68knommu/platform/coldfire/timers.c* was modified by deleting timer acknowledgment, because now it is performed from the I-Pipe;

10. The `mcftmr_read_clk()` function from *arch/m68knommu/platform/ coldfire/ timers.c* was modified to use `read_timer_cnt()` function;

11. The `hw_timer_init()` function from *arch/m68knommu/platform/coldfire/ timers.c* was modified to initialize timer to work with better timer precision;

12. The `ack_linux_tmr()` function was added to *arch/m68knommu/platform/ coldfire/timers.c* to perform correct timer interrupt acknowledgment in I-Pipe;

13. The `mcf_interrupt()` function from *drivers/serial/mfc.c* was modified to enable UART interrupt. It was made to perform correct UART interrupt acknowledgment;

14. The `fec_enet_interrupt()` function from *drivers/net/fec.c* was modified to enable FEC interrupt. It was made to perform correct FEC interrupt acknowledgment.

4. Changes in RTAI Source Code

RTAI source code has been changed during porting. Changes affect both architecture-independent and architecture-dependent parts.

Main changes in RTAI code are described here.

4.1. RTAI HAL Changes

In RTAI HAL (file */base/arch/m68knommu/hal.c*) the following changes were made:

1. Timer code was changed to work with ColdFire timer. First system timer is used by both RTAI for scheduling real time processes and Linux when RTAI scheduler is inactive.

2. `rdtsc()` function implementation is now based on `read_timer_cnt()` kernel function. Thus RDTSC functionality is emulated by ColdFire timer.

   ```
   long long rdtsc()
   {
        return  read_timer_cnt() * (tuned.cpu_freq / TIMER_FREQ);
   }
   ```

3. RTAI SRQ dispatcher code was adopted to MCF5329. SRQ dispatcher is a function that is called as a trap handler from user space to access RTAI functionality. The result of a syscall is always a 64-bit value, but it's meaning depends on a specific syscall.

   ```
   //We have: d0 - srq, d1 - args, d2 - retval
   asmlinkage  int  rtai_syscall_dispatcher  (__volatile  struct
   pt_regs pt)
   {
        int cpuid;
        //unsigned long lsr = pt.sr;
        long long result;
        //IF_IS_A_USI_SRQ_CALL_IT(pt.d0, pt.d1, (long long*)pt.d2,
   lsr, 0);
        if (usi_SRQ_call(pt.d0, pt.d1, &result, pt.sr))
             return 0;
        result      =      pt.d0      >      RTAI_NR_SRQS      ?
   rtai_lxrt_dispatcher(pt.d0,    pt.d1,    (void    *)&pt)    :
   rtai_usrq_dispatcher(pt.d0, pt.d1);
   ```

```
        pt.d2 = result & 0xFFFFFFFF;
        pt.d3 = (result >> 32);
        if (!in_hrt_mode(cpuid = rtai_cpuid())) {
             hal_test_and_fast_flush_pipeline(cpuid);
             return 1;
        }
        return 0;
}

#define SAVE_REG \
        "move     #0x2700,%sr\n\t"         /* disable intrs */ \
        "btst     #5,%sp@(2)\n\t"               /* from user? */ \
        "bnes     6f\n\t"                  /* no, skip */ \
        "movel    %sp,sw_usp\n\t"               /*   save   user   sp
*/ \
        "addql    #8,sw_usp\n\t"           /* remove exception */ \
        "movel    sw_ksp,%sp\n\t"               /* kernel sp */ \
        "subql    #8,%sp\n\t"              /* room for exception */
\
        "clrl     %sp@-\n\t"               /* stkadj */ \
        "movel    %d0,%sp@-\n\t"           /* orig d0 */ \
        "movel    %d0,%sp@-\n\t"           /* d0 */ \
        "lea %sp@(-32),%sp\n\t"       /* space for 8 regs */ \
        "moveml   %d1-%d5/%a0-%a2,%sp@\n\t" \
        "movel    sw_usp,%a0\n\t"               /* get usp */ \
        "movel    %a0@-,%sp@(48)\n\t"      /*     copy     exception
program counter (PT_PC=48)*/ \
        "movel    %a0@-,%sp@(44)\n\t" /*       copy       exception
format/vector/sr (PT_FORMATVEC=44)*/ \
        "bra 7f\n\t" \
        "6:\n\t" \
        "clrl     %sp@-\n\t"               /* stkadj */ \
        "movel    %d0,%sp@-\n\t"           /* orig d0 */ \
        "movel    %d0,%sp@-\n\t"           /* d0 */ \
        "lea %sp@(-32),%sp\n\t"       /* space for 8 regs */ \
        "moveml   %d1-%d5/%a0-%a2,%sp@\n\t" \
        "7:\n\t" \
        "move     #0x2000,%sr\n\t"

#define RSTR_REG \
        "btst     #5,%sp@(46)\n\t"         /*      going      user?
(PT_SR=46)*/ \
        "bnes     8f\n\t"                  /* no, skip */ \
        "move     #0x2700,%sr\n\t"         /* disable intrs */ \
        "movel    sw_usp,%a0\n\t"               /* get usp */ \
        "movel    %sp@(48),%a0@-\n\t"      /*     copy     exception
program counter (PT_PC=48)*/ \
        "movel    %sp@(44),%a0@-\n\t" /*       copy       exception
format/vector/sr (PT_FORMATVEC=44)*/ \
        "moveml   %sp@,%d1-%d5/%a0-%a2\n\t" \
        "lea %sp@(32),%sp\n\t"       /* space for 8 regs */ \
        "movel    %sp@+,%d0\n\t" \
        "addql    #4,%sp\n\t"             /* orig d0 */ \
        "addl     %sp@+,%sp\n\t"          /* stkadj */ \
        "addql    #8,%sp\n\t"             /* remove exception */ \
        "movel    %sp,sw_ksp\n\t"              /* save ksp */ \
        "subql    #8,sw_usp\n\t"          /* set exception */ \
        "movel    sw_usp,%sp\n\t"              /* restore usp */ \
        "rte\n\t" \
        "8:\n\t" \
        "moveml   %sp@,%d1-%d5/%a0-%a2\n\t" \
        "lea %sp@(32),%sp\n\t"       /* space for 8 regs */ \
        "movel    %sp@+,%d0\n\t" \
        "addql    #4,%sp\n\t"             /* orig d0 */ \
        "addl     %sp@+,%sp\n\t"          /* stkadj */ \
        "rte"

#define DEFINE_VECTORED_ISR(name, fun) \
        __asm__ ( \
             SYMBOL_NAME_STR(name) ":\n\t" \
             SAVE_REG \
             "jsr "SYMBOL_NAME_STR(fun)"\n\t" \
             RSTR_REG);

void rtai_uvec_handler(void);
DEFINE_VECTORED_ISR(rtai_uvec_handler,
rtai_syscall_dispatcher);
```

This code switches stack from user-space stack to kernel-space when RTAI syscall occurs and switches back at syscall return. Stack switching code is similar to Linux syscalls implementation, because this RTAI code is based on it.

4. Two functions from *rtai_atomic.h* (`atomic_xchg()` and `atomic_cmpxchg()`) were changed to use RTAI traps. The following trap code was added to the HAL:

```
void rtai_cmpxchg_trap_handler(void);
__asm__ ( \
     "rtai_cmpxchg_trap_handler:\n\t" \
     "move   #0x2700,%sr\n\t" \
     "movel    %a1@, %d0\n\t" \
     "cmpl     %d0,%d2\n\t" \
     "bnes     1f\n\t" \
     "movel    %d3,%a1@\n\t" \
        "1:\n\t" \
        "rte");

    void rtai_xchg_trap_handler(void);
    __asm__ ( \
        "rtai_xchg_trap_handler:\n\t" \
        "move   #0x2700,%sr\n\t" \
        "movel    %a1@, %d0\n\t" \
        "movel    %d2,%a1@\n\t" \
        "rte");

    ...
    rtai_xchg_trap_vec =
    rtai_set_gate_vector(RTAI_XCHG_TRAP_SYS_VECTOR, 15, 3,
    &rtai_xchg_trap_handler);rtai_cmpxchg_trap_vec =
    rtai_set_gate_vector(RTAI_CMPXCHG_TRAP_SYS_VECTOR, 15, 3,
    &rtai_cmpxchg_trap_handler);
```

5. `rt_set_timer_delay()` function from *rtai_hal.h* was changed to work with ColdFire timer and was made cache-aware to avoid timer TCN overruns (i.e. when TCN is further then TRR).

4.2. RTC Removal

RTAI port for MCF5329 doesn't allow to use an RTC as a time source, so all related code was removed.

4.3. RTAI LEDs Support

rtai_leds module hasn't been ported, because LED functionality doesn't depend on RTAI. If you need to use LEDs for debugging you will need to write a code that is specific to your LEDs and hardware it is based on.

5. Programming with RTAI.

This section gives some information about creation of RTAI applications.

The following acronyms were used in this section:

<uClinux> - µClinux root directory

<rtai> - RTAI root directory

<filename> - name code of file that should be built (without extension).

This section describes building of written RTAI application or module, it doesn't cover any programming information.

For help in RTAI API please refer to RTAI Doxygen documentation or RTAI programming manuals.

5.1 Creating user-space RTAI programs

To compile ***<filename>***`.c` in user-space that uses some exportations from `liblxrt` library:

1. Change the current directory to the directory with ***<filename>***`.c`;
2. Compile ***<filename>***`.c` to the object file ***<filename>***`.o`:

   ```
   $ m68k-uclinux-gcc -DHAVE_CONFIG_H -I. -
   I<uClinux>/linux-2.6.x/include -DCONFIG_UCLINUX -D__IN_RTAI__
   -I<rtai>/base/include -I<rtai> -m5307 -Wa,-m5307 -c -o
   <filename>.o <filename>.c
   ```

3. Link ***<filename>***`.o` to the executable ***<filename>***:

   ```
   $ m68k-uclinux-gcc -m5307 -Wa,-m5307 -Wl,-elf2flt -o
   <filename> <filename>.o  <rtai>/base/sched/liblxrt/liblxrt.a
   -lpthread
   ```

4. If there were no errors, then in the current folder ***<filename>*** executable will appear. Then it can be added to the `romfs` of µClinux.

5.2 Creating kernel-space RTAI modules

1. To compile kernel-space RTAI-based module ***<filename>**.c* for MCF5329, the next steps should be done:

2. Change the current directory to the directory with ***<filename>**.c*;

3. Create *Makefile* file and put the next lines into it:

   ```
   rtai_srctree:=<rtai>

   EXTRA_CFLAGS+=-I$(rtai_srctree) -I$
   (rtai_srctree)/base/include

   obj-m := <filename>.o
   ```

4. Launch the make utility:

   ```
   $ make -C <uClinux> SUBDIRS=$PWD modules
   ```

5. If there were no errors, ***<filename>**.ko* kernel module will appear in the current folder. Then it can be added to the *romfs* of µClinux.