RTAI Port to MCF5407 Developer's Manual

M5407/RTAI
Rev. 0.1 02/2003

Digital DNA from Motorola

m

CONTENTS

Paragraph Title Page

About This Document

This document describes setting up and usage of RTAI for MCF5407 installed into ?Clinux embedded OS, and the changes in RTAI source code, which allow using it with MCF5407.

Audience

This document targets ?Clinux software developers using the MCF5407 processor.

Suggested Reading

[1] MCF5407 ColdFire Integrated Microprocessor. User's manual
[2] DIAPM RTAI Programming Guide 1.0
[3] Advanced Linux Programming. M Mitchell, J Oldham, A Samuel

Definitions, Acronyms and Abbreviations

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

Table i. Acronyms and Abbreviated Terms

FIFO	First In First Out Buffer
IRQ	Interrupt Request
ISR	Interrupt Service Routine
OS	Operating System
RTAI	Real Time Application Interface
RTHAL	Real Time Hardware Abstract Layer
RTOS	Real Time Operating System
SRQ?	System Request
?Clinux	Micro-Controller version of Linux Operating System for Embedded Applications

1. Introduction

This document describes the Real Time Application Interface (RTAI), ported to the MCF5407 processor. 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 four parts. The first part contains a general overview of RTAI, the description of its installation and usage.

The second part (Chapter 3) describes the changes, which were made in the ?Clinux kernel 2.4.x. and Linux drivers. Mainly it contains detailed information about modification of the interrupt handling routines and timer routines.

The third part (Chapter 4) is a description of the RTAI services for the MCF5407 processor.

The fourth part (Chapter 5) includes the results of tests from RTAI standard test suite (with the description of changes in their source code) and the description of specific tests, which were developed for MCF5407.

1.1. RTAI Features for MCF5407

- ?? Correct execution of **periodic real time tasks** with the frequencies **40 kHz** and less (for the periodic mode of the timer. This mode consists of a frequency timing of the real time tasks in multiple of the period).
- ?? Correct execution of **real time tasks in oneshot mode** with the frequencies **10 kHz** and less (the oneshot mode of the timer consists in a variable timing based on the CPU clock frequency).
- ?? The usual jitter measured on a 50 MHz MCF5407 processor is **about 8 Microseconds**.
- ?? RTAI's services are provided by four kernel modules, which allow hard real time, fully preemptive scheduling. These four modules are: *rtai, rtai_sched, rtai_fifos and rtai_shm* **(**MCF5407 has no Memory Management Unit, so the implementation of shared memory is simple)**.**

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* module is 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*

It is the basic RTAI framework, plus interrupts dispatching and timer support.

1.2.2. Module *rtai_sched*

It is the real time, preemptive, priority-based Uni-Processor (UP) scheduler, chosen according to the hardware of MCF5407.

1.2.3. Module *rtai_fifos*

Real time FIFOs and semaphores are included into this module.

1.2.4. Module *rtai_shm*

In a similar way to FIFOs, shared memory provides a way to easily transfer data between real time and user space tasks in which a portion of physical memory is set aside for sharing between Linux processes and real time Linux tasks. But unlike FIFOs, shared memory is able to easily pass large amounts of data in one step. Since MCF5407 hasn't Memory Management Unit, the implementation of shared memory is simple; only functions *rtai_malloc, rtai_free, rtai_kmalloc* and *rtai_kfree* have to be created.

1.3. Related Files

The following files are relevant to RTAI:

- ?? *uClinux-dist-20020927.tar.gz* – source code of ?Clinux embedded OS.
- ?? *rtai-24.1.8.tgz* – RTAI 24.1.8 original package.
- ?? *m68k-elf-tools-20011219.tar.gz* – m68k-elf tool chain for ?Clinux and RTAI compilation.
- ?? *specs* – specifications for m68k-elf tool chain.
- ?? *sched.h* – definition for the sched_setscheduler() system call (this file is also a part of m68k-elf tool chain).
- ?? *uClinux_20020927_RTAI_MCF5407.patch.bz2* – patch for ?Clinux kernel.
- ?? *rtai-24.1.8_MCF5407.patch.bz2* - patch for RTAI.
- ?? *RTAI_5407_examples.tar.bz2* – specific tests developed for MCF5407.

2. RTAI Installation and Usage

This chapter describes how to install and patch ?Clinux and RTAI, download image with ?Clinux and its real time extension to M5407C3 board. It also contains a general overview of RTAI and information about installation and changes in **m68k-elf tool chain** in order to compile ?Clinux and RTAI.

2.1. RTAI General Overview

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 two primary variants of hard real time Linux available: RTLinux 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.

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. Unlike the RTAIs for x86, PowerPC and MIPS, the ColdFire version uses the classic capability of the MC68000 architecture to prioritize the interrupts in hardware by the interrupt controller. In the implementation of RTAI for the MCF5407 Linux interrupts are assigned up to interrupt priority level 4, while RTAI interrupts can have interrupt priority levels 5 and 6 (level 7 is a non-maskable interrupt and haven’t to be used for normal operation).

Hence when Linux is doing *cli()*, all interrupts with the priority level 4 or less are disabled; at the same time real time interrupts can preempt the kernel at any time.

For the realization of these features a small RTHAL was created. Its structure is provided below:

```
struct rt_hal {
     void *ret_from_intr;
     void *__switch_to;
     struct desc_struct *idt_table;
     void (*disint)(void);
     void (*enint)(void);
     unsigned int (*getflags)(void);
```

```
    void (*setflags)(unsigned int flags);
    unsigned int (*getflags_and_cli)(void);
    void *irq_desc;
    int *irq_vector;
    unsigned long *irq_affinity;
    void (*smp_invalidate_interrupt)(void);
    void (*ack_8259_irq)(unsigned int);
    int *idle_weight;
    void (*lxrt_global_cli)(void);
    void (*switch_mem)(struct task_struct *,struct task_struct *,int);
    struct task_struct **init_tasks;
};
```

Only marked fields are used by RTAI and ?Clinux for the MCF5407 (the other fields are kept only for compatibility).

After loading *rtai* kernel module the initial state of RTHAL (corresponding to the Linux state) is stored, and the structure's procedures are replaced by real time Linux procedures. When *rtai* kernel module is unloaded, the reverse substitution occurs. Hence the real time extension can be completely removed. It can be useful in different debugging situations. The module-based architecture yields a system that can be easily expanded and customized according to the services required by a developer. E.g. if the developer wants to communicate with standard Linux processes using FIFOS, *rtai_fifos* kernel module can be loaded; *rtai_shm* module provides shared memory etc. The exception is *rtai* module – it is used by all other RTAI modules and real time applications created by a developer.

2.2. m68k-elf Tool Chain Setup

To install the m68k-elf tool chain for correct ?Clinux and RTAI compilation, the following steps must be accomplished:

1. Copy m68k-elf tool chain archive (*m68k-elf-tools-20011219.tar.gz*) into the root directory (it's possible to install tool chain into another directory, but the sequence of actions will be the same – only change of paths is needed).

2. Dearchive it:

```
tar xvzf m68k-elf-tools-20011219.tar.gz
```

3. Copy two files (*specs, sched.h*) in order to compile ?Clinux and RTAI:

```
cp specs /usr/local/lib/gcc-lib/m68k-elf/2.95.3
cp sched.h /usr/local/m68k-elf/sys-include
```

2.3. ?Clinux and RTAI Installation, Patching and Compilation

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

1. Copy ?Clinux archive (*uClinux-dist-20020927.tar.gz*) into the root directory (if developer wants to have *uClinux-dist* directory with ?Clinux at the non-root directory, the sequence of actions will be the same – only change of paths is needed).

2. Dearchive it:

```
tar xvzf uClinux-dist-20020927.tar.gz
```

3. Copy RTAI archive (*rtai-24.1.8.tgz*) into the directory */uClinux-dist* (if ?Clinux was dearchived from the root directory).

4. Dearchive it:

```
tar xvzf rtai-24.1.8.tgz
```

5. If the patches are stored e.g. in directory */usr/patches*, execute the following commands to patch ?Clinux and RTAI:

```
cd /uClinux-dist/linux-2.4.x

bunzip2 –cd /usr/patches/uClinux_20020927_RTAI_MCF5407.patch.bz2 \
| patch –p1

cd ../rtai-24.1.8

bunzip2 –cd /usr/patches/rtai-24.1.8_MCF5407.patch.bz2 | patch –p1
```

6. Configure the Linux kernel:

```
UC_DIST=/uClinux-dist
cd $UC_DIST
make xconfig
```

It's possible to use menuconfig and config, if the developer isn't running X Window systems on the development machine.
Afterwards do the following selections:

Target platform: `Motorola/M5407C3`
`Kernel Version: linux-2.4.x`
`Libc Version: uC-libc`
`Default all settings: N`

```
Customize kernel settings: Y
Customize vendor/user settings: Y
```

Then save and exit. The second window “Kernel Configuration” will be shown. Here the loadable module support should be enabled.

Loadable module support => Enable loadable module support: *Y*

Then save and exit. The third window “?Clinux Application Configuration” will be shown. The busybox should be configured with the loadable module support (*insmod, lsmod* and *rmmod*).

BusyBox => BusyBox: *Y*
BusyBox => insmod: *Y*
BusyBox => lsmod: *Y*
BusyBox => rmmod: *Y*

Save and exit. Of course developer may change another features needed from ?Clinux; the changes above are required for the RTAI usage.

7. Build the dependencies and ?Clinux image:

```
make dep
make
```

8. Configure RTAI:

```
cd rtai-24.1.8
make ARCH=m68knommu CROSS_COMPILE=m68k-elf- \
LINUXDIR=”$UC_DIST/linux-2.4.x” menuconfig
```

Then press <enter> when being asked about the location of Linux source tree. In configuration tool mark with <M> the items listed below:

- UP scheduler
- POSIX API support
- Fifos
- Compile tests

Next mark the fields “RT memory manager” and “Compile examples” with <*>, save the configuration and exit. All other fields should be empty.

Depending on Linux installed at the developer’s machine, *menuconfig* can’t be used; if make fails, use *config* instead. In that case <M> corresponds to “m”, <*> corresponds to “y” and < > corresponds to “n”.

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

```
make dep
```

```
make

install –d $UC_DIST/romfs/lib/modules/rtai
cp modules/rtai.o $UC_DIST/romfs/lib/modules/rtai
cp modules/rtai_sched.o $UC_DIST/romfs/lib/modules/rtai
cp modules/rtai_fifos.o $UC_DIST/romfs/lib/modules/rtai
cp modules/rtai_shm.o $UC_DIST/romfs/lib/modules/rtai
m68k-elf-strip -g $UC_DIST/romfs/lib/modules/rtai/*.o
```

Then compile ?Clinux again:

```
cd $UC_DIST
make
```

Now it's possible to load and run ?Clinux using the *dBUG* monitor of the built-in ROMs. Use its *dn* command to load the image. And then type *go 20000* to run it. The variant of information at console is provided below:

```
dBUG> go 20000
Linux version 2.4.19-uc1 (root@OLEG) (gcc version 2.95.3 20010315
(release)(ColdFire patches - 20010318 from
http://fiddes.net/coldfire/)(-msep-data patches)) #26 Tue Jan 14
17:20:48 EET 2003

uClinux/COLDFIRE(m5407)
COLDFIRE port done by Greg Ungerer, gerg@snapgear.com
Flat model support (C) 1998,1999 Kenneth Albanowski, D. Jeff Dionne
On node 0 totalpages: 8192
zone(0): 0 pages.
zone(1): 8192 pages.
zone(2): 0 pages.
Kernel command line:
Calibrating delay loop... 149.50 BogoMIPS
Memory available: 30356k/32768k RAM, 0k/0k ROM (739k kernel code, 211k
data)
kmem_create: Forcing size word alignment - vm_area_struct
kmem_create: Forcing size word alignment - mm_struct
kmem_create: Forcing size word alignment - filp
Dentry cache hash table entries: 4096 (order: 3, 32768 bytes)
Inode cache hash table entries: 2048 (order: 2, 16384 bytes)
kmem_create: Forcing size word alignment - inode_cache
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
kmem_create: Forcing size word alignment - bdev_cache
kmem_create: Forcing size word alignment - cdev_cache
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
kmem_create: Forcing size word alignment - sock
Initializing RT netlink socket
Starting kswapd
kmem_create: Forcing size word alignment - file lock cache
ColdFire internal UART serial driver version 1.00
```

```
ttyS0 at 0x100001c0 (irq = 73) is a builtin ColdFire UART
ttyS1 at 0x10000200 (irq = 74) is a builtin ColdFire UART
kmem_create: Forcing size word alignment - blkdev_requests
ne.c:v1.10 9/23/94 Donald Becker (becker@scyld.com)
Last modified Nov 1, 2000 by Paul Gortmaker
NE*000 ethercard probe at 0x40000300: 00 cf 54 07 c3 01
eth0: NE2000 found at 0x40000300, using IRQ 27.
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
CSLIP: code copyright 1989 Regents of the University of California.
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 10DA6C-1FF26B [VIRTUAL 10DA6C-1FF26B] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
PPP generic driver version 2.4.2
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 2048)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (romfs filesystem) readonly.
Freeing unused kernel memory: 24k freed (0xf4000 - 0xf9000)
Shell invoked to run file: /etc/rc Command: hostname uClinux
Command: /bin/expand /etc/ramfs.img /dev/ram0
Command: mount -t proc proc /proc
Command: mount -t ext2 /dev/ram0 /var
Command: mkdir /var/tmp
Command: mkdir /var/tmp
Command: mkdir /var/run
Command: mkdir /var/lock
Command: ifconfig lo 127.0.0.1
Command: route add -net 127.0.0.0 netmask 255.0.0.0 lo
Command: dhcpcd -p -a eth0 &
[11]
Command: cat /etc/motd
Welcome to
          ____ _  _
         /  __| ||_|
    _   _| |  | | _ ____  _   _  _  _
   | | | | |  | || |  _ \| | | |\ \/ /
   | |_| | |__| || | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting

Sash command shell (version 1.1.1)
/>
```

Now load RTAI modules and get information through *proc* filesystem:

```
insmod rtai
insmod rtai_sched
insmod rtai_fifos
insmod rtai_shm
lsmod
cd proc/rtai
cat rtai
cat scheduler
cat memory_manager
cat fifos
cat shmem
```

The variant of information at console is provided below:

```
/> insmod rtai
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/rtai.o
rtai: using clock_freq 1000000
/> insmod rtai_sched
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/rtai_sched.o

==== RT memory manager v1.3 Loaded. ====

***** STARTING THE UP REAL TIME SCHEDULER WITH LINUX *****
***** FP SUPPORT AND READY FOR A PERIODIC TIMER *****
***<> LINUX TICK AT 100 (HZ) <>***
***<> CALIBRATED CPU FREQUENCY 1000000 (HZ) <>***
***<>  CALIBRATED  8254-TIMER-INTERRUPT-TO-SCHEDULER  LATENCY  27000  (ns)
<>***
***<> CALIBRATED ONE SHOT SETUP TIME 0 (ns) <>***

/> insmod rtai_fifos
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/rtai_fifos.o
/> insmod rtai_shm
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/rtai_shm.o
/> lsmod
Module                  Size  Used by
rtai_shm                1484   0 (unused)
rtai_fifos             22056   0 (unused)
rtai_sched             34708   0 (unused)
rtai                   27528   0 [rtai_shm rtai_fifos rtai_sched]
/> cd proc/rtai
/proc/rtai> cat rtai

RTAI Real Time Kernel, Version: 2.3.xx

    RTAI mount count: 1

RTAI IRQs

RTAI sysreqs in use:
```

```
1 2 3 4 5

/proc/rtai> cat scheduler

RTAI Uniprocessor Real Time Task Scheduler.

    Calibrated CPU Frequency: 1000000 Hz
    Calibrated 8254 interrupt to scheduler latency: 27000 ns
    Calibrated one shot setup time: 0 ns

Number of RT CPUs in system: 1

Priority  Period(ns)  FPU  Sig  State  Task  RT_TASK *
-----------------------------------------------------
TIMED

READY

/proc/rtai> cat memory_manager

RTAI Dynamic Memory Management Status.
--------------------------------------
Chunk Size  Address    1st free block  Block size
-------------------------------------------------
0     65536 0x1ff0000  0x1ff0010         65504
1     65536 0x1fe0000  0x1fe0010         65504
2     65536 0x1fd0000  0x1fd0010         65504
3     65536 0x1fc0000  0x1fc0010         65504
/proc/rtai> cat fifos
RTAI Real Time fifos status.

fifo No  Open Cnt  Buff Size  handler  malloc type
---------------------------------------
/proc/rtai> cat shmem

RTAI Shared Memory Manager for uClinux.

Name        Address     Size    Used
-----------------------------------
```

2.4. Installation of Specific MCF5407 Test Examples

To install specific MCF5407 test examples for ?Clinux and RTAI, the following steps must be accomplished:

1. Copy archive with test examples (*RTAI_5407_examples.tar.bz2*) into the RTAI directory (it is *$UC_DIST/rtai-24.1.8*).

2. Dearchive it:

```
tar –xjf RTAI_5407_examples.tar.bz2
```

3. Compile them and copy to ?Clinux romfs (If ?Clinux was installed into non-root directory, makefile for test examples should be modified):

```
cd 5407_examples
make
make standard
```

Afterwards all test examples from standard test suite, which were compiled correctly for the MCF5407, and specific tests will be placed at the `romfs`.

4. Create FIFO device nodes in `/dev` file system in order to provide the correct execution of test examples, which use RTAI FIFOs:

```
cd $UC_DIST/romfs/dev
for i in 0 1 2 3 4 5 6 7; do touch "@rtf$i,c,150,$i"; done
```

5. Recompile ?Clinux:

```
cd $UC_DIST
make
```

3. Changes in ?Clinux Source Code

The main changes in ?Clinux source code were made in the interrupt handling and timer routines. Also all Linux drivers were modified so that their interrupts don't occur at levels that are intended for real time operation. Levels 1 - 4 are assigned for Linux interrupts and levels 5 and 6 are assigned for the real time operation. Hence all Linux drivers, which cause interrupts level 5 or 6 should be modified. Interrupts of the 7th level are non-maskable interrupts, which shouldn't be used for normal operation.

The patch for the ?Clinux is in the file *uClinux_20020927_RTAI_MCF5407.patch.bz2.* It can be used for older versions of ?Clinux.

3.1. Changes in Timer Routines

In the file *linux-2.4.x/arch/m68knommu/platform/5407/config.c* several important changes were made.

In the function `coldfire_timer_init` the value of Timer1 reference register TRR1 is equal:

`CLOCK_TICK_RATE/`*`HZ = 1000000/100 = 10000.`*

NOTE:

Hereafter all calculations are made for 50 MHz M5407C3 board. According to ?Clinux definitions Timer0 and Timer1 are called Timer1 and Timer2 respectively. The time-out period remains the same (10 msec).

In original source code it was:

(MCF_CLK/16)/HZ = ((50000000/16)/100)=31250.

It was made because after each call of the function *coldfire_tick* the value of TRR1 is changed by the formula below:

```
TRR1 = rdtsc() + CLOCK_TICK_RATE/HZ = rdtsc() + 10000
```

This addition was made to keep the emulated Time Stamp Counter up to date.

In the function `coldfire_timer_init` Timer1 autovectored interrupt moved from Level 6 to Level 1. Therefore the first parameter in the call of function *request_irq* was changed:

```
*icrp = MCFSIM_ICR_AUTOVEC | MCFSIM_ICR_LEVEL1 | MCFSIM_ICR_PRI3
request_irq(25, handler, SA_INTERRUPT, "ColdFire Timer", NULL);
```

In the file *linux-2.4.x/arch/m68knommu/platform/5307/ints.c* the function rdtsc() is located. Moreover It's used by RTAI modules. Its source code is provided below:

```
long long rt_time_rdtsc = 0;
```

```
long long rdtsc(void) {
  volatile unsigned short *timerp;
  long flags;
  unsigned short diff;
  hard_save_flags_and_cli(flags);
  timerp = (volatile unsigned short *) (MCF_MBAR + MCFTIMER_BASE1);
  diff = timerp[MCFTIMER_TCN] - rt_time_rdtsc;
  rt_time_rdtsc += diff;
  hard_restore_flags(flags);
  return rt_time_rdtsc;
};
```

In the header file *linux-2.4.x/include/asm-m68k/timex.h* the value of the constant *CLOCK_TICK_RATE* is changed. Now it's equal 1000000 instead of 1193180, because the timer settings were changed.

3.2. Changes in Interrupt Handling Routines

The most important modification was the usage of another structure for table, which stores the address information about the handlers for all IRQs. It is *irq_node_t* instead of *irq_handler_t*. The definitions of these structures are shown below (both are defined in the file *linux-2.4.x/include/asm-m68knommu/irq.h*):

```
typedef struct irq_node {
      void        (*handler)(int, void *, struct pt_regs *);
      unsigned long     flags;
      void        *dev_id;
      const char  *devname;
      struct irq_node *next;
} irq_node_t;

typedef struct irq_handler {
      void        (*handler)(int, void *, struct pt_regs *);
      unsigned long     flags;
      void        *dev_id;
      const char  *devname;
} irq_handler_t;
```

In the file *linux-2.4.x/arch/m68knommu/platform/5307/ints.c* the definition of this table is changed:

```
irq_node_t *int_irq_list[SYS_IRQS];
```

instead of

```
irq_handler_t irq_list[SYS_IRQS];
```

This file contains general interrupt handling code. The main changes, which were made in it, are discussed below.

To reserve the IRQ node the following function was created:

```
irq_node_t* new_irq_node(void) {
  int i;
  for(i=0;i<NUM_IRQ_NODES;i++) {
    if(nodes[i].handler==0) return &nodes[i];
  };
  return 0;
};
```

To delete the IRQ node the following routine was written:

```
void delete_irq_node(irq_node_t* node) {
  node->handler=0;
};
```

The code of the IRQ initialization procedure was shortened:

```
void init_IRQ(void)
{
      if (mach_init_IRQ)
            mach_init_IRQ ();
      mach_kstat_irqs = &kstat.irqs[0][0];
};
```

The routine, which processes interrupt handlers for the specified IRQ vector was rewritten:

```
asmlinkage void process_int(unsigned long vec, struct pt_regs *fp)
{
  irq_node_t *irqnode = int_irq_list[vec];
  while(irqnode) {
    irqnode->handler(vec, irqnode->dev_id, fp);
    irqnode = irqnode->next;
  };
};
```

It's called from the generic interrupt handler *inthandler* and fast interrupt handler *fasthandler* (file *linux-2.4.x/arch/m68knommu/platform/5307/entry.s*). Earlier the algorithm of the interrupt handling was:

1. Calculate the real vector number.
2. Calculate the offset in the array *irq_list* and the value of the pointer to the array element.
3. Get the address of the vector handler.
4. Call the vector handler.

After changes, which were made in this file, the algorithm is:

1. Get the address of the *process_int* routine.

2. Call this routine.

Through */proc* interface it is possible to get the IRQ list, which is formed by a function *get_irq_list*. Its synopsis is:

```
int get_irq_list(char *buf);
```

This function returns the length of the buffer *buf*, which contains the information about the requested interrupts. This list can be shown using the command:

```
cat /proc/interrupts
```

Due to the changes mentioned above this function was rewritten. Its source code is provided below:

```
int get_irq_list(char *buf)
{
      int i, len = 0;
      irq_node_t *node;

      len += sprintf(buf+len, "Internal MCF5407 interrupts\n");

      for (i = 0; i < NR_IRQS; i++) {
          int start=1,any=0;
          irq_node_t *nextirq = int_irq_list[i];
          while(nextirq) {
            if(nextirq->handler) {
              if(start) {
                start=0;
                len += sprintf(buf+len, " %2d: %10u    %s", i,
                 i ? kstat.irqs[0][i] : num_spurious,nextirq->devname);
              }
               else {
                  len += sprintf(buf+len, ", %s", nextirq->devname);
              };
              any=1;
            };
            nextirq = nextirq->next;
          };
          if(any) len += sprintf(buf+len, "\n");
      }
      return len;
}
```

The functions *request_irq* and *free_irq* were modified to provide a correct placement and freeing of IRQ handlers in the table of structures (as each structure has the new field *next*). In the routine *request_irq* one condition check was added to provide shared interrupts support on PCI. The modified source code of these functions is given below:

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct
     pt_regs *), unsigned long flags, const char *devname, void *dev_id)
```

```
{
      irq_node_t *newirq;

#if DAVIDM
      if ((irq & IRQ_MACHSPEC) && mach_request_irq) {
            return mach_request_irq(IRQ_IDX(irq), handler, flags,
                  devname, dev_id);
      }
#endif

      if (irq < 0 || irq >= NR_IRQS) {
            printk("%s: Incorrect IRQ %d from %s\n", __FUNCTION__,
                  irq, devname);
            return -ENXIO;
      }

  /*
   * Sanity-check: shared interrupts should REALLY pass in
   * a real dev-ID, otherwise we'll have trouble later trying
   * to figure out which interrupt is which (messes up the
   * interrupt freeing logic etc).
   */
  if (flags & SA_SHIRQ) {
    if (!dev_id)
      printk("Caution: %s called us without a dev_id!\n", devname);
  }

  if(int_irq_list[irq]) {
    if(!(int_irq_list[irq]->flags & flags & SA_SHIRQ)) {
      return -EBUSY;
    }
    else {
      irq_node_t *lastirq = int_irq_list[irq];
      newirq = new_irq_node();
      while(lastirq->next) lastirq = lastirq->next;
      lastirq->next = newirq;
    };
  }
  else {
    int_irq_list[irq] = newirq = new_irq_node();
  };

      if (flags & IRQ_FLG_FAST) {
            extern asmlinkage void fasthandler(void);
            extern void set_evector(int vecnum, void (*handler)(void));
            set_evector(irq, fasthandler);
      }

      newirq->handler = handler;
      newirq->flags   = flags;
      newirq->dev_id  = dev_id;
      newirq->devname = devname;
      newirq->next    = 0;
      return 0;

}
```

```
void free_irq(unsigned int irq, void *dev_id)
{
      irq_node_t *nextirq,*previrq;

#if DAVIDM
      if (irq & IRQ_MACHSPEC) {
            mach_free_irq(IRQ_IDX(irq), dev_id);
            return;
      }
#endif

      if (irq < 0 || irq >= NR_IRQS) {
            printk("%s: Incorrect IRQ %d\n", __FUNCTION__, irq);
            return;
      }

  nextirq = int_irq_list[irq];
  previrq = 0;
  if(!nextirq) return;
  while(nextirq) {
    if(nextirq->dev_id == dev_id) {
      if(previrq) previrq->next=nextirq->next;
      else int_irq_list[irq]=nextirq->next;
      nextirq->handler = 0;
      nextirq->flags   = IRQ_FLG_STD;
      nextirq->dev_id  = NULL;
      nextirq->devname = NULL;
    }
    else {
      previrq = nextirq;
    };
    nextirq = nextirq->next;
  };

}
```

The most important change, which was made in kernel to make it compatible with RTAI is the integration of a small RTHAL. Its structure was discussed in Section 2.1. RTHAL for ?Clinux (after loading *rtai* module ?Clinux RTHAL is replaced by RTAI RTHAL) and related routines are given below:

```
struct rt_hal rthal = {
  0, // &ret_from_intr,
  0, // __switch_to,
  0, // idt_table,
  linux_cli,
  linux_sti,
  linux_save_flags,
  linux_restore_flags,
  linux_save_flags_and_cli,
  0, // irq_desc,
  0, // irq_vector,
  0, // irq_affinity,
```

```
  0, // smp_invalidate_interrupt,
  0, // ack_8259_irq,
  0, // &idle_weight,
  0,
  0, // switch_mem,
  0, // init_tasks
};

#define soft_cli()  __asm__ __volatile__ ( \
                        "move %0,%/sr\n" \
                        : /* no output */ \
                        : "i" (LINUX_IRQS_DISABLE) \
                        : "cc", "memory")

#define soft_sti()  __asm__ __volatile__ ( \
                        "move #0x2000,%/sr\n" \
                        : /* no output */ \
                        : /* no input */ \
                        : "cc", "memory")

static void linux_sti(void) {
  soft_sti();
};

static void linux_cli(void) {
  soft_cli();
};

static unsigned int linux_save_flags(void) {
  int x;
  hard_save_flags(x);
  return x;
}

static void linux_restore_flags(unsigned int x) {
  hard_restore_flags(x);
};

static unsigned int linux_save_flags_and_cli(void) {
  int x;
  hard_save_flags(x);
  soft_cli();
  return x;
};
```

In the header file *linux-2.4.x/include/asm-m68knommu/rt.h* several macros were defined (in the file *linux-2.4.x/include/asm-m68knommu/system.h* the similar macros were deleted, except *cli()* and *sti()*). They are associated with RTHAL's routines (*linux_cli*, *linux_sti* etc.). Here are their definitions:

```
#define hard_cli() asm volatile ("movew #0x2700,%%sr": : : "memory")
#define hard_sti() asm volatile ("movew #0x2000,%%sr": : : "memory")
#define hard_save_flags(x) asm volatile ("movew %%sr,%0":"=d" (x) : :
"memory")
```

```
#define hard_restore_flags(x) asm volatile ("movew %0,%%sr": :"d" (x) :
"memory")
#define  hard_save_flags_and_cli(flags)  do  {  hard_save_flags(flags);
hard_cli(); } while(0)

#define __cli()                (rthal.disint())
#define __sti()                (rthal.enint())
#define __save_flags(x)        ((x) = rthal.getflags())
#define __restore_flags(x)     (rthal.setflags(x))
#define __save_and_cli(x)      ((x) = rthal.getflags_and_cli())
#define local_irq_disable()    (rthal.disint())
#define local_irq_enable()     (rthal.enint())
#define local_irq_save(x)      ((x) = rthal.getflags_and_cli())
#define local_irq_restore(x)   (rthal.setflags(x))
```

Example: Linux is doing `local_irq_save(x)`. As defined above:

```
x = rthal.getflags_and_cli()
```

In Linux RTHAL `rthal` function `unsigned int (*getfalgs_and_cli(void))` is initialized with routine `linux_save_flags_and_cli()`, which finally will be executed. In RTAI RTHAL `rtai_rthal` another routines will be executed (see Chapter 4).

3.3. Miscellaneous Changes

In the file `linux-2.4.x/drivers/char/mcfserial.h` the values written into UARTs' interrupt control registers were changed, because earlier UARTs caused interrupts level 6 (priorities 1 and 2 for UART0 and UART1 correspondingly). This level is reserved for the real time operation. Therefore the interrupt level for UARTs was changed:

```
 icrp = (volatile unsigned char *) (MCF_MBAR + MCFSIM_UART1ICR);
*icrp = /*MCFSIM_ICR_AUTOVEC |*/ MCFSIM_ICR_LEVEL1 | MCFSIM_ICR_PRI1;

 icrp = (volatile unsigned char *) (MCF_MBAR + MCFSIM_UART2ICR);
*icrp = /*MCFSIM_ICR_AUTOVEC |*/ MCFSIM_ICR_LEVEL1 | MCFSIM_ICR_PRI2;
```

In the file `linux-2.4.x/include/asm-m68knommu/bitops.h` the sequence of operations

```
save_flags(flags); cli();
```

is substituted by an instruction:

```
hard_save_flags_and_cli(flags);
```

And `restore_flags(flags)` is changed to `hard_restore_flags(flags)`.

4. RTAI services for MCF5407

RTAI API functions are described [2]. RTAI was created for i386, PowerPC and MIPS architectures. The description of non-documented (i.e. specific for ColdFire RTAI version) data structures and functions for each RTAI module is given in this chapter. For the other functions the location of the basic code and changes in it are indicated. In the modules *rtai_fifos* and *rtai_sched* only several modifications were made.

4.1. Module *rtai*

This is the basic RTAI module. All other RTAI modules depend from it. It's possible to split this module into the following parts:

1. Functions for setting and resetting exception vectors.
2. System request handling.
3. Pending Linux interrupts handling.
4. Definition of RTAI RTHAL.
5. RTAI IRQ handling.
6. The RTAI */proc* interface.
7. Module initialization and cleanup.
8. Spinlocks functions.
9. Timer functions.
10. RTAI version of *printk* - *rt_printk*.

4.1.1. Variables, Definitions, Macros and Data Types

```
struct {
   int pending_srqs,
   int active_srqs,
   int pending_irqs,
   int active_irqs,
   int cpu_in_sti,
   int used_by_linux,
   int locked_cpus,
   int current_priority;
} global = {0,0,0,0,0,1,0,-1};
```

Defined in *rtai-24.1.8/arch/m68knommu/rtai.c*. This structure is responsible for the current status of RTAI as a whole. The similar structure was defined for i386 architecture (in the file *rtai-24.1.8/include/asm-i386/rtai-24.h*). The field *current_priority* is not used.

```
#define RT_NUM_SRQS 32
static struct sysrq_t {
        unsigned int label;
        void (*rtai_handler)(void);
```

```
        long long (*user_handler)(unsigned int whatever);
} sysrq[RT_NUM_SRQS];
```

The same structure for RTAI system requests was defined in the file *rtai-24.1.8/arch/i386/rtai.c.* Its number is limited by a quantity of bits in an unsigned integer.

```
#define LX_NUM_PEND_IRQS (32)
static int lx_pend_irq[LX_NUM_PEND_IRQS];
static int lx_pend_mask[NR_IRQS];
static int lx_pend_inuse[NR_IRQS];
static unsigned long irq_action_flags[NR_IRQS];
static int chained_to_linux[NR_IRQS];
static int rtai_irq_inuse[NR_IRQS];
```

These arrays indicate the current state of pending Linux IRQs. See Section 4.1.4 for more details.

NOTE:

The number of pendable interrupts is limited to 32. It should be enough for most applications.

```
static struct rt_irq_t {
  void (*handler)(void);
  unsigned int enabled;
  unsigned int flags;
  void (*ack)(void);
} rt_irq[NR_IRQS];
```

Each element of the structure is responsible for one RTAI IRQ. This structure is specific for RTAI for ColdFire. During IRQ dispatching before the handler's startup the routine *ack* can be executed (if it is initialized for the particular IRQ number). See Section 4.1.6 for more details.

```
struct desc_struct {
      void* a;
};
```

This structure's definition is used for the exception vectors setting and resetting.

```
struct desc_struct rt_linux_irqvec[NR_IRQS];
```

In this array the values of the exception vectors are stored (during setting of the exception vector the old value is stored in it; during resetting this value should be written backwards). See Section 4.1.6 for more information.

```
struct rt_times rt_times;
```

The definition of the structure *rt_times* is given below (from the file *rtai-24.1.8/include/rtai_types.h*):

```
typedef long long RTIME;

struct rt_times {
      int linux_tick;
      int periodic_tick;
      RTIME tick_time;
      RTIME linux_time;
      RTIME intr_time;
};
```

rt_times is a global variable. The settings of the timer are stored during RTAI timer request to provide the correct functioning of RTAI UP-scheduler. See Section 4.1.10 for more details.

```
struct desc_struct trap_srsave_vector;
struct desc_struct trap_sys_vector;
```

These variables store the addresses of the routines, which are executed when the traps #10 and #11 correspondingly occur. After loading *rtai* module the information about them is stored in these variables to make an opportunity to restore the original Linux handlers after removing RTAI.

```
static int CpuFreq;
```

For RTAI for ColdFire this variable equals *CLOCK_TICK_RATE* (i.e. 1000000).

```
struct proc_dir_entry *rtai_proc_root;
```

After loading *rtai* module the directory */proc/rtai* will be created. In this directory the */proc* entries for all RTAI modules will be located. For these needs the structure, which is given above is used.

```
#define SAVE_SR_TRAP __asm__ __volatile__ ( \
        "movew  #0x2700,%sr  ;" \
        "movel  %d0,%sp@-    ;" \
        "movew  %sp@(6),%d0  ;" \
        "movew  %d0,sr_saved ;" \
        "movew  #0x2700,%d0  ;" \
        "movew  %d0,%sp@(6)  ;" \
        "movel  %sp@+,%d0    ;" \
        "rte                 ;" )
```

This macro saves the value of SR, which was written by an exception into the stack, into variable *sr_saved*. The first instruction disables all interrupts and always executes. This macro is a body of ISR *save_sr_trap* (see Section 4.1.6.1).

```
#define SAVE_REG __asm__ __volatile__ ( \
      "trap   #10       ;" \
      "btst   #5,%sp@(2)      ;" \
      "bnes 1f          ;" \
      "                 ;" \
      "link %a0,#12     ;" \
      "movel      %sp,sw_usp  ;" \
      "movel      sw_ksp,%sp  ;" \
      "movel      %a0@(8),%sp@-     ;" \
      "movel      %a0@(4),%sp@-     ;" \
      "movel      %a0@,%a0    ;" \
"1:                   ;" \
      "clrl %sp@-       ;" \
      "pea    -1:w            ;" \
      "movel      %d0,%sp@-   ;" \
      "movew  sr_saved,%d0    ;" \
      "movew      %d0,%sr          ;" \
      "lea  %sp@(-32),%sp     ;" \
      "moveml     %d1-%d5/%a0-%a2,%sp@;" \
      "                       ; "\
      "movel   %sp@(44),%d0   ; "\
      "movel   %sp,%sp@-      ; "\
      "movel   %d0,%sp@-      ; "\
      "swap    %d0            ; "\
      "andl    #0x03fc,%d0    ; "\
      "lsrl    #2,%d0         ; "\
      "movel   %d0,%sp@-      ; ")

#define RSTR_REG __asm__ __volatile__ ( \
      "lea    %sp@(12),%sp    ;" \
      "tstl   %d0             ;" \
      "bnes   2f              ;" \
      "                       ;" \
      "moveml     %sp@,%d1-%d5/%a0-%a2;" \
      "lea  %sp@(32),%sp      ;" \
      "movel      %sp@+,%d0   ;" \
      "addql      #4,%sp            ;" \
      "addl %sp@+,%sp   ;" \
      "                 ;" \
      "move #0x2700,%sr ;" \
      "btst #5,%sp@(2)  ;" \
      "bnes 1f          ;" \
      "                 ;" \
      "link %a0,#12           ;" \
      "movel      %sp,sw_ksp  ;" \
      "movel      sw_usp,%sp  ;" \
      "movel      %a0@(8),%sp@-     ;" \
      "movel      %a0@(4),%sp@-     ;" \
      "movel      %a0@,%a0    ;" \
"1:                   ;" \
      "rte              ;" \
"2:                           ;" \
      "addql  #1,local_irq_count;" \
      "jmp ret_from_interrupt ;" )
```

These macros save and restore registers actually like the Linux kernel does it. They are a part of the ISR *rt_rtai_irqvec* (see Section 4.1.6.2).

4.1.2. Functions for Setting and Resetting Exception Vectors

These functions use the global *_ramvec*, which is defined in the file *linux-2.4.x/arch/m68knommu/platform/5407/MOTOROLA/crt0_ram.S*. Its value is the address of the exception vector table (in the idem file it equals *0x00000000*). They are used by the functions *rt_request_global_irq*, *rt_free_global_irq* and *init_module* of *rtai* module.

4.1.2.1. rt_set_full_intr_vect

SYNOPSIS
```
struct desc_struct rt_set_full_intr_vect(unsigned int vector, int type,
int dpl, void (*handler)(void));
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
Sets the interrupt handler *handler* for the interrupt vector with number *vector* at the exception vector table. For RTAI implementation for ColdFire *type* and *dpl* are ignored.

RETURN VALUES
The structure, which contains the address of the old interrupt handler is returned.

4.1.2.2. rt_reset_full_intr_vect

SYNOPSIS
```
void rt_reset_full_intr_vect(unsigned int vector, struct desc_struct
idt_element);
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
Saves *idt_element*, which contains the address of the handler, in exception vector table for the particular *vector*. The information about the old handler is not stored.

RETURN VALUES
None

4.1.3. System Request Handling

4.1.3.1. rt_srq_init

SYNOPSIS
```
static void rt_srq_init(void);
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
Initializes RTAI system requests (SRQs) during *rtai* startup. See [2] for more details.

RETURN VALUES
None

4.1.3.2. rt_request_srq

SYNOPSIS
```
int rt_request_srq(unsigned int label, void (*rtai_handler)(void), long long (*user_handler)(unsigned int whatever));
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*). A few changes were made.

DESCRIPTION
Installs a two way RTAI system request by assigning *user_handler*, a function to be used when a developer calls SRQ from user space, and *rtai_handler*, the function to be called in kernel space.

RETURN VALUES
On success the number of appointed *srq* is returned.
-EBUSY, when all 32 SRQs are busy.
-EINVAL, when the second parameter is NULL.

4.1.3.3. rt_free_srq

SYNOPSIS
```
int rt_free_srq(unsigned int srq);
```

BASIS

Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*). A few changes were made.

DESCRIPTION
Frees the SRQ number *srq*. All fields of the structure for the particular *srq* are nulled.

RETURN VALUES
On success 0 is returned.
-EINVAL, when the *srq* number is illegal or *sysrq[srq].rtai_handler* is NULL.

4.1.3.4. rt_pend_linux_srq

SYNOPSIS
```
void rt_pend_linux_srq(unsigned int srq);
```

BASIS
The function was copied from RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
Appends a system call request *srq* to be used as a service request to the Linux kernel by setting the bit number *srq* in *global.pending_srqs*.

RETURN VALUES
None

4.1.3.5. srqisr

SYNOPSIS
```
static void srqisr(void);
```

BASIS
The algorithm, which is realized by this function is similar to the algorithm in RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
The source code of this function is provided below:

```
static void srqisr(void) {
  __asm__ __volatile__ ("movew %0,%%sr\n\t"
                "lea %%sp@(-36),%%sp\n\t"
                "moveml %%d0-%%d5/%%a0-%%a2,%%sp@\n\t"
                "jsr dispatch_srq\n\t"
                "addql #8,%%sp\n\t"
                "moveml %%sp@,%%d2-%%d5/%%a0-%%a2\n\t"
                "lea %%sp@(28),%%sp\n\t"
                "rte\n\t" : : "i" (LINUX_IRQS_DISABLE));
```

```
}
```

The value of *LINUX_IRQS_DISABLE* (0x2400) is written into status register (SR) to disable interrupts level 4 and lower. Then after storing the registers in stack the function *dispatch_srq* is called. Afterwards the registers are restored from the stack and *rte* instruction (return from exception) is executed. This ISR is called when trap #11 occurs.

RETURN VALUES
None

4.1.3.6. dispatch_srq

SYNOPSIS
```
long long dispatch_srq(unsigned int srq, unsigned int whatever);
```

BASIS
The function was copied from RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*). Only the constant *RT_NUM_SRQS* is used instead of *NR_GLOBAL_IRQS*.

DESCRIPTION
This function is called from *srqisr* ISR. It dispatches SRQ number *srq* by processing its user handler if it is initialized.

RETURN VALUES
The value returned by user handler is returned if the *srq* number is correct and user handler for the particular *srq* is initialized.

In other case the value of *srq* can be returned if the parameter *whatever* is equal to the label of one of the SRQs. Otherwise 0 is returned.

4.1.3.7. rtai_srq

SYNOPSIS
```
static inline long long rtai_srq(unsigned long srq, unsigned long
whatever);
```

BASIS
The operations, which are realized by this function are similar to the operations in RTAI implementation for i386 architecture (file *rtai-24.1.8/include/asm-i386/rtai_srq.h*).

DESCRIPTION
The source code of this function is given below (file *rtai-24.1.8/include/asm-m68knommu/rtai_srq.h*):

```
static inline long long rtai_srq(unsigned long srq, unsigned long
whatever) {
      long long retval;
      register unsigned long __srq __asm__ ("%d0") = srq;
      register unsigned long __whatever __asm__ ("%d1") = whatever;

      __asm__ __volatile__ ("trap #11\n\t"
                        : "=X" (retval)
                        : "d" (__srq), "d" (__whatever)
                        : "memory" );
      return retval;
}
```

After the call of this function trap #11 occurs. It means that ISR *srqisr* (see Section 4.1.3.5) is called. The values of parameters *srq* and *whatever* are stored in *d0* and *d1* correspondingly. They are used by the function *dispatch_srq* (see Section 4.1.3.6).

RETURN VALUES
This function returns the value given from the function *dispatch_srq*.

4.1.4. Pending Linux Interrupts Handling

4.1.4.1. lx_irq_init

SYNOPSIS
```
static void lx_irq_init(void);
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
Initializes data structures, which are used for pending Linux interrupts handling. Called from the function *init_module* of *rtai* module during *rtai* startup. See [2] for more details.

RETURN VALUES
None

4.1.4.2. lx_irq_findfree

SYNOPSIS
```
static int lx_irq_findfree(void);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
For ColdFire the maximal quantity of pendable interrupts is 32. This routine searches in the vector *lx_pend_irq* the first element, which equals −1 (this value indicates that in this element it's possible to store the IRQ number of another pending Linux interrupt).

RETURN VALUES
The number of the first element in the vector *lx_pend_irq* , which equals −1, is returned.

4.1.4.3. lx_irq_pend_inuse_inc

SYNOPSIS
```
static void lx_irq_pend_inuse_inc(unsigned int irq);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This function increments the value of the irq^{th} element in vector *rtai_irq_inuse*. If as a result its value is 1, in vector *lx_pend_irq* with the help of the function *lx_irq_findfree* the value of *irq* is stored in the first "free element", and in vector *lx_pend_mask* its number is saved in the element with the index *irq*.

RETURN VALUES
None

4.1.4.4. lx_irq_pend_inuse_dec

SYNOPSIS
```
static void lx_irq_pend_inuse_dec(unsigned int irq);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This function decrements the value of the irq^{th} element in vector *rtai_irq_inuse*. If as a result its value is 0, the value −1 is written into the element of vector *lx_pend_mask* with the index *irq*. This routine is opposite to the function *lx_irq_pend_inuse_inc*.

RETURN VALUES
None

4.1.4.5. rt_request_linux_irq

SYNOPSIS

```
int rt_request_linux_irq(unsigned int irq,
    void (*linux_handler)(int irq, void *dev_id, struct pt_regs *regs),
    char *linux_handler_id, void *dev_id);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*). A few changes were made.

DESCRIPTION
This routine installs function *handler* as a standard Linux ISR. The handler is appended to any already existing Linux handler for the same *irq* and is run by Linux IRQ as any of its handlers. See [2], where this function is well-described.

RETURN VALUES
On success 0 is returned.
-EINVAL, if *irq* number is not correct or *handler* is NULL.

4.1.4.6. rt_free_linux_irq

SYNOPSIS
```
int rt_free_linux_irq(unsigned int irq, void *dev_id);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*). A few changes were made.

DESCRIPTION
This routine uninstalls Linux ISR. See [2] for more details.

RETURN VALUES
On success 0 is returned.
-EINVAL, if *irq* number is not correct or the value of the element with index *irq* in vector *chained_to_linux* is 0.

4.1.4.7. rt_pend_linux_irq

SYNOPSIS
```
void rt_pend_linux_irq(unsigned int srq);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This function appends a Linux interrupt *irq* by setting the bit in *global.pending_irqs* (the number of bit is stored in vector *lx_pend_mask* for the particular *irq*).

RETURN VALUES
None

4.1.4.8. linux_irq_check

SYNOPSIS
```
static void linux_irq_check(void);
```

BASIS
The function was mainly copied from RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*, routine *linux_sti*).

DESCRIPTION
This function dispatches all pending IRQs and all pending SRQs. It can be called when Linux is doing *sti()* or *local_irq_restore(x)* (see Section 4.1.5 for more details).

RETURN VALUES
None

4.1.5. Definition of RTAI RTHAL

After loading *rtai* module ?Clinux RTHAL is replaced by RTAI RTHAL. The definition of RTAI RTHAL (structure *rtai_rthal*) is the copy of the definition given in Section 3.2. But two fields of this structure (*linux_sti* and *linux_restore_flags*) were modified. Other routines were copied from the file *linux-2.4.x/arch/m68knommu /platform/5307/ints.c.* The source code of these two functions is provided below:

```
static void linux_sti(void) {
  if(global.pending_irqs || global.pending_srqs) {
    soft_cli();
    linux_irq_check();
  };
  soft_sti();
};

static void linux_restore_flags(unsigned int x) {
  if(!(x&0x700)) {
    if(global.pending_irqs || global.pending_srqs) {
      soft_cli();
      linux_irq_check();
    };
    soft_sti();
  }
  else {
    soft_cli();
  };
};
```

When Linux is doing *sti()*, RTHAL checks for pending SRQs and IRQs, does *soft_cli()* (the value 0x2400 is written into SR) and dispatches all pending IRQs and SRQs (if there are any IRQs or SRQs to dispatch) and finally executes *soft_sti()* (the value *0x2000* is written into SR, which means that all interrupts are enabled).

When Linux is doing *local_irq_restore(x)* and *x* is unequal *0x2700*, RTHAL does the same as during Linux *sti()*. If *x* equals *0x2700*, *soft_cli()* occurs (the value *0x2400* is written into SR, because the ideology of RTAI for ColdFire is to give to the interrupts level 5 and 6 an ability to preempt Linux kernel at any time).

To store the Linux RTHAL RTAI saves its state in the temporary variable *linux_rthal*. The part of *init_module* function of the *rtai* module is provided below:

```
linux_rthal = rthal;
rthal = rtai_rthal;
```

And in the function *cleanup_module* of the *rtai* module the reverse substitution occurs:

```
rthal = linux_rthal;
```

Also two functions were remained for compatibility. They simply duplicate another RTHAL functions:

```
unsigned int linux_save_flags_and_cli_cpuid(int cpuid) {
  return linux_save_flags_and_cli();
};

void rtai_just_copy_back(unsigned long flags, int cpuid) {
  linux_restore_flags(flags);
};
```

The macros *soft_cli* and *soft_sti* were placed in the file *rtai-24.1.8/include/asm-m68knommu/rtai.h*. They are exact copies of macros from the file *linux-2.4.x/arch/m68knommu/platform/5307/ints.c*

4.1.6. RTAI IRQ handling

4.1.6.1. save_sr_trap

SYNOPSIS
```
static void save_sr_trap(void);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION

This is an ISR, which is executed, when trap #10 occurs. This exception occurs, when one of the RTAI interrupts should be executed (before RTAI interrupts dispatching the macro SAVE_REG is executed and the first instruction of this macro is trap #10).

RETURN VALUES
None

4.1.6.2. rt_rtai_irqvec

SYNOPSIS
```
static void rt_rtai_irqvec(void);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This is an ISR, which is executed, when RTAI interrupt occurs. Its source code is provided below:

```
static void rt_rtai_irqvec(void) {
        SAVE_REG;
        __asm__ __volatile__ ("jsr rt_dispatch_irqvec\n\t");
        RSTR_REG;
}
```

The first macro SAVE_REG restores the value of SR from the variable `sr_saved`, saves registers in the stack and prepares information for the routine `rt_dispatch_irqvec`. The second macro RSTR_REG can jump to `ret_from_interrupt` (if pended interrupts took place) or restore registers and jump to `rte` (return from exception).

RETURN VALUES
None

4.1.6.3. rt_dispatch_irqvec

SYNOPSIS
```
int rt_dispatch_irqvec(int vec,int sr,struct rt_regs* regs);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This is an ISR, which is executed, when RTAI interrupt occurs. Its source code is given below:

```
int rt_dispatch_irqvec(int vec,int sr,struct rt_regs* regs) {
```

```
  if(rt_irq[vec].enabled) {
    if(rt_irq[vec].ack) rt_irq[vec].ack();
    rt_irq[vec].handler();
  };

  if(global.pending_irqs || global.pending_srqs) {
    if(!(sr & 0x0700)) {
      soft_cli();
      linux_irq_check();
      return 1;
    };
  };
  return 0;
};
```

First this function checks if RTAI interrupt with index *vec* is enabled and executes routines *ack* (if this routine was initialized) and *handler* for it. Then if *sr* isn't *0x2700* and there are pending IRQs or SRQs, the value *0x2400* will be written into SR and procedure *linux_irq_check* will be executed.

RETURN VALUES
1, if there were pending IRQs or SRQs and *sr* isn't *0x2700*. Otherwise 0 is returned.

4.1.6.4. rt_irq_init

SYNOPSIS
```
static void rt_irq_init(void);
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
Initializes data structure, which is used for RTAI IRQ handling. Called from the function *init_module* of *rtai* module during *rtai* startup.

RETURN VALUES
None

4.1.6.5. rt_request_global_irq

SYNOPSIS
```
int rt_request_global_irq(unsigned int i,void (*handler)(void));
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This routine installs function *handler* as RTAI ISR for IRQ vector number *i*. The information about the original Linux ISR is saved in the vector *rt_linux_irqvec* to enable its restoring after uninstalling RTAI ISR.

RETURN VALUES
On success 0 is returned.
-1, if *i* is not correct or this interrupt is already enabled.

4.1.6.6. rt_free_global_irq

SYNOPSIS
```
int rt_free_global_irq(unsigned int i);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This routine uninstalls RTAI IRQ. The original Linux ISR for the particular interrupt vector *i* is restored. See [2] for more details.

RETURN VALUES
On success 0 is returned.
-1, if *i* is not correct or this interrupt is already disabled.

4.1.7. The RTAI */proc* Interface

4.1.7.1. rtai_read_rtai

SYNOPSIS
```
static int rtai_read_rtai(char *page, char **start, off_t off,
                          int  count, int *eof, void *data);
```

BASIS
The function is based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This function is executed when the developer extracts the information about RTAI from */proc* file system using the command:

```
cat /proc/rtai/rtai
```

The information about RTAI version, IRQs and SRQs will be displayed. The output might look like:

```
RTAI Real Time Kernel, Version: 2.3.xx

    RTAI mount count: 1

RTAI IRQs
74
RTAI sysreqs in use:
1 2 3 4 5
```

RETURN VALUES
None

4.1.7.2. rtai_proc_register

SYNOPSIS
```
static int rtai_proc_register(void);
```

BASIS
The function was copied from RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*) without any changes.

DESCRIPTION
This function creates two entries in */proc* file system: the entry for directory */proc/rtai*, where all entries of RTAI modules are created during their startup, and the entry for *rtai* module */proc/rtai/rtai*.

RETURN VALUES
-1, if an error occurred during initialization. On success 0 is returned.

4.1.7.3. rtai_proc_unregister

SYNOPSIS
```
static void rtai_proc_unregister(void);
```

BASIS
The function was copied from RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*) without any changes.

DESCRIPTION
This function removes two entries in */proc* file system: the entry for directory */proc/rtai* and the entry for *rtai* module */proc/rtai/rtai*.

RETURN VALUES
None.

4.1.8. Module Initialization and Cleanup

4.1.8.1. init_module

SYNOPSIS
```
int init_module(void);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This routine is a usual entry point for *rtai* module. It makes the following initializations:

1. Initialization of the data structures for pending Linux interrupts.
2. RTAI IRQs initialization.
3. RTAI SRQs initialization.
4. Substitution of Linux RTHAL by RTAI RTHAL.
5. Initialization of the *rt_printk* RTAI SRQ handler (see Section 4.1.11 for more details).
6. SRQ's ISR *srqisr* installation (see Section 4.1.3 for more details).
7. Installation of the ISR *save_sr_trap* (see Section 4.1.6 for more details).
8. RTAI registration in */proc* file system.

RETURN VALUES
On success 0 is returned.

4.1.8.2. cleanup_module

SYNOPSIS
```
void cleanup_module(void);
```

BASIS
Based on RTAI implementation for MIPS architecture (file *rtai-24.1.8/arch/mips/rtai.c*).

DESCRIPTION
This routine is a usual exit point for *rtai* module. It removes RTAI entries in */proc* file system, restores the initial value of RTHAL (viz Linux RTHAL, which was stored in the structure *linux_rthal*) and handlers, which Linux executed (before loading RTAI), when trap #10 and trap #11 occurred.

RETURN VALUES
None

4.1.9. Spinlocks Functions

The spinlock calling convention for IRQ save and restore is slightly different from the one used in Linux. They are done on purpose to get an error if the developer uses Linux spinlocks in real time applications, as they don't guaranty any protection because of the soft IRQ disable.

All spinlocks functions and macros are in the file *rtai-24.1.8/include/asm-m68knommu/rtai.h*.

Hence the definition of basic spinlocks functions is very simple or even empty:

```
#define rt_spin_lock(whatever)
#define rt_spin_unlock(whatever)

#define rt_get_global_lock()  hard_cli()
#define rt_release_global_lock()
```

The source code of all functions, which use these definitions, is shown below:

```
static inline void rt_spin_lock_irq(spinlock_t *lock)
{
      hard_cli();
      rt_spin_lock(lock);
}

static inline void rt_spin_unlock_irq(spinlock_t *lock)
{
      rt_spin_unlock(lock);
      hard_sti();
}

static inline unsigned int rt_spin_lock_irqsave(spinlock_t *lock)
{
      unsigned long flags;
      hard_save_flags_and_cli(flags);
      rt_spin_lock(lock);
      return flags;
}

static inline void rt_spin_unlock_irqrestore(unsigned long flags,
spinlock_t *lock)
{
      rt_spin_unlock(lock);
      hard_restore_flags(flags);
}

static inline void rt_global_cli(void)
{
      rt_get_global_lock();
}
```

```
static inline void rt_global_sti(void)
{
      int flags=0x2500;
      rt_release_global_lock();
      hard_restore_flags(flags);
}

static inline int rt_global_save_flags_and_cli(void)
{
      unsigned long flags;
      hard_save_flags_and_cli(flags);
      return flags;
}

static inline void rt_global_save_flags(unsigned long *flags)
{
      unsigned long rflags;
      hard_save_flags(rflags);
      *flags = rflags;
}

static inline void rt_global_restore_flags(unsigned long flags)
{
      hard_restore_flags(flags);
}
```

4.1.10. Timer Functions

4.1.10.1. rt_set_timer_delay

SYNOPSIS
```
void rt_set_timer_delay(unsigned short delay);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This routine changes the value of TRR1 for Timer1. If the timer is in the periodic mode, the value of `rt_times.intr_time` is written; otherwise the result of summation of the emulated time stamp counter and `delay` is written into TRR1.

RETURN VALUES
None

4.1.10.2. rt_ack_tmr

SYNOPSIS
```
static void rt_ack_tmr(void);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This function resets the ColdFire timer (just like Linux kernel does it) and keeps emulated time stamp counter up to date.

RETURN VALUES
None

4.1.10.3. rt_request_timer

SYNOPSIS
```
void rt_request_timer(void (*handler)(void), unsigned int tick, int unused);
```

BASIS
Based on RTAI implementation for i386 architecture (file *rtai-24.1.8/arch/i386/rtai.c*).

DESCRIPTION
This function is called from the function *start_rt_timer* (module *rtai_sched*). Depending from the timer mode (oneshot or periodic) it sets the parameters of RTAI timer. The fields of the structure *rt_times* are initialized with the following values:

tick_time = rdtsc()
linux_tick = 10000 (the period of Linux tick is 10 msec)
linux_time = tick_time + linux_tick (i.e. *rdtsc() + 10000*)

For oneshot timer mode (when parameter *tick* is 0):

intr_time = tick_time + linux_tick
periodic_tick = linux_tick (i.e.10000)

For periodic timer mode (when parameter *tick* is non-zero):

intr_time = tick_time + tick (i.e. *rdtsc() + tick*)
periodic_tick = tick

NOTE:
If the timer mode is periodic and a developer passes the parameter *period*, which is more than 10000 to the function *start_rt_timer*, the parameter *tick* would be 10000.

Then this routine saves the initial value of *mach_tick* (it is routine *coldfire_tick*) in *lx_mach_tick* and nulls it. It frees RTAI timer IRQ and forms new RTAI timer IRQ:

```
  if(mach_tick) {
    lx_mach_tick = mach_tick;
    mach_tick = 0;
  };
  rt_free_global_irq(RT_TIMER_IRQ);
  rt_request_global_irq(RT_TIMER_IRQ,handler);
  rt_irq[RT_TIMER_IRQ].ack = rt_ack_tmr;
  rt_irq[RT_TIMER_IRQ].flags = 0;
```

Afterwards Timer1 registers initialization occurs. RTAI timer is Level 6 and priority 3 (Linux timer invokes Level 1 and priority 3 interrupts). TRR1 is initialized with *rt_times.intr_time*. TMR1 is initialized in the same way as Linux does it:

```
  timerp = (volatile unsigned short *) (MCF_MBAR + MCFTIMER_BASE1);
  timerp[MCFTIMER_TMR] = MCFTIMER_TMR_DISABLE;

#ifdef CONFIG_MOTOROLA
  do {
    volatile unsigned char  *icrp;
    icrp = (volatile unsigned char *) (MCF_MBAR + MCFSIM_TIMER1ICR);
    *icrp = MCFSIM_ICR_AUTOVEC | MCFSIM_ICR_LEVEL6 | MCFSIM_ICR_PRI3;
  } while(0);
#endif

  timerp[MCFTIMER_TRR] = rt_times.intr_time;
  timerp[MCFTIMER_TMR] = MCFTIMER_TMR_ENORI | MCFTIMER_TMR_CLK1 |
    MCFTIMER_TMR_ENABLE | ( ((MCF_CLK/CLOCK_TICK_RATE)-1) << 8 );
```

RETURN VALUES
None

4.1.10.4. rt_free_timer

SYNOPSIS
```
void rt_free_timer(void);
```

BASIS
This function is specific for RTAI implementation for ColdFire.

DESCRIPTION
This function is called from the function *stop_rt_timer* (module *rtai_sched*). It restores the original value of *mach_tick* (i.e. *coldfire_tick*), frees RTAI timer IRQ and restores the original values of Timer1 registers:

```
  timerp = (volatile unsigned short *) (MCF_MBAR + MCFTIMER_BASE1);
  timerp[MCFTIMER_TMR] = MCFTIMER_TMR_DISABLE;

#ifdef CONFIG_MOTOROLA
    do {
```

```
      volatile unsigned char *icrp;
      icrp = (volatile unsigned char *) (MCF_MBAR + MCFSIM_TIMER1ICR);
      *icrp = MCFSIM_ICR_AUTOVEC | MCFSIM_ICR_LEVEL1 | MCFSIM_ICR_PRI3;
    } while(0);
#endif

  timerp[MCFTIMER_TRR] = rdtsc() + (CLOCK_TICK_RATE / HZ);
  timerp[MCFTIMER_TMR] = MCFTIMER_TMR_ENORI | MCFTIMER_TMR_CLK1 |
    MCFTIMER_TMR_ENABLE | ( ((MCF_CLK/CLOCK_TICK_RATE)-1) << 8 );
```

So Timer1 interrupts become Level 1 and priority 3 again.

RETURN VALUES
None

4.1.11. RTAI Version of *printk* - *rt_printk*

These functions in RTAI implementations for ColdFire and MIPS are absolutely identical. Their brief overview is provided below.

4.1.11.1. rt_printk

SYNOPSIS
```
int rt_printk(const char *fmt, ...);
```

DESCRIPTION
This routine fills the buffer `rt_printk_buf` with the information, which should be printed and appends a system call request with number 1 (because `sysrq[1].rtai_handler` is initialized with `rt_printk_sysreq_handler` during RTAI startup). It offers safe `printk` functionality to real time modules.

RETURN VALUES
This function returns the length of the constant char string, which should be printed.

4.1.11.2. rt_printk_sysreq_handler

SYNOPSIS
```
static void rt_printk_sysreq_handler(void);
```

DESCRIPTION
This routine is a handler for Linux SRQ with number 1. Hence it prints the contents of `rt_printk_buf` (information about its front and back is stored in the variables `buf_front` and `buf_back`) using `printk`.

RETURN VALUES
None

4.1.11.3. rtai_print_to_screen

SYNOPSIS
```
int rtai_print_to_screen(const char *format, ...);
```

DESCRIPTION
This routine uses the *console_drivers* list to display information. Unlike *rt_printk* it doesn't use *printk* function and RTAI SRQ number 1.

RETURN VALUES
This function returns the length of the constant char string, which should be printed.

4.2. Modules *rtai_sched* and *rtai_fifos*

Actually these modules are hardware independent. Only minimal changes were made in its source code.

In the module *rtai_fifos* (file *rtai-24.1.8/fifos/rtai_fifos.c*) only two changes were made in order to increase the speed of mailbox service. These changes were made in static inline functions *mbx_put* and *mbx_get*. The part of RTAI patch is given below:

```
diff   -urN   rtai-24.1.8/fifos/rtai_fifos.c   rtai-24.1.8_MCF5407/fifos
/rtai_fifos.c
--- rtai-24.1.8/fifos/rtai_fifos.c  Mon Jan  7 19:26:37 2002
+++ rtai-24.1.8_MCF5407/fifos/rtai_fifos.c      Wed Dec  4 15:09:02 2002
@@ -370,12 +370,12 @@
                    memcpy(mbx->bufadr + mbx->lbyte, *msg, tocpy);
             }
             rtf_spin_lock_irqsave(flags, mbx->buflock);
+            mbx->lbyte = MOD_SIZE(mbx->lbyte + tocpy);
             mbx->frbs -= tocpy;
             mbx->avbs += tocpy;
             rtf_spin_unlock_irqrestore(flags, mbx->buflock);
             msg_size -= tocpy;
             *msg     += tocpy;
-            mbx->lbyte = MOD_SIZE(mbx->lbyte + tocpy);
       }
       return msg_size;
 }
@@ -398,12 +398,12 @@
                    memcpy(*msg, mbx->bufadr + mbx->fbyte, tocpy);
             }
             rtf_spin_lock_irqsave(flags, mbx->buflock);
+            mbx->fbyte = MOD_SIZE(mbx->fbyte + tocpy);
             mbx->frbs += tocpy;
             mbx->avbs -= tocpy;
             rtf_spin_unlock_irqrestore(flags, mbx->buflock);
             msg_size  -= tocpy;
             *msg      += tocpy;
```

```
-           mbx->fbyte = MOD_SIZE(mbx->fbyte + tocpy);
      }
      return msg_size;
 }
```

The main purpose of these changes is to gather all operations with the fields of the structure *mbx* between *rtf_spin_lock_irqsave* and *rtf_spin_unlock_irqrestore*.

A small routine, which provides switching from one real time task to another, was written in assembler in order to increase the scheduler's speed. Its code is given below (from file *rtai-24.1.8/arch/m68knommu/rtai.c*):

```
void up_task_sw(void *current_task_ref, void *new_task) {
  __asm__ __volatile__ ("lea %%sp@(-60),%%sp\n\t"
                  "movem.l %%d0-%%d7/%%a0-%%a6,%%sp@\n\t"
                  "pea %%pc@(1f)\n\t"
                  "move.l (%0),%%a1\n\t"
                  "move.l %%sp,(%%a1)\n\t"
                  "move.l %1,(%0)\n\t"
                  "move.l %1,%%a1\n\t"
                  "move.l (%%a1),%%sp\n\t"
                  "rts\n\t"
                  "1:  movem.l %%sp@,%%d0-%%d7/%%a0-%%a6\n\t"
                  "lea %%sp@(60),%%sp\n\t"
                  : /* no output */
                  : "a" (current_task_ref) , "d" (new_task)
                  : "%a1", "memory");
};
```

A simple macros *rt_switch_to* and *rt_exchange_tasks* based on this function are provided below (file *rtai-24.1.8/include/asm-m68knommu/rtai_sched.h*):

```
#define rt_switch_to(new_task) up_task_sw(&rt_current, (new_task));

#define   rt_exchange_tasks(oldtask,   newtask)   up_task_sw(&(oldtask),
(new_task));
```

Another macro *init_arch_stack* was copied from file *rtai-24.1.8/include/asm-i386/rtai_sched.h*:

```
#define init_arch_stack() \
do { \
      *--(task->stack) = data;            \
      *--(task->stack) = (int) rt_thread; \
      *--(task->stack) = 0;               \
      *--(task->stack) = (int) rt_startup;      \
} while(0)
```

The source code of a small inline function *get_stack_pointer* is given below (file *rtai-24.1.8/include/asm-m68knommu/rtai_sched.h*):

```
static inline void *get_stack_pointer(void)
{
      void *sp;
      asm volatile ("movel %%sp,%0" : "=d" (sp));
      return sp;
}
```

In the file *rtai-24.1.8/upscheduler/rtai_sched.c.ml* several modifications were made in disabling/enabling interrupts operations. These changes are related with the necessity to store or restore the contents of SR before disabling/enabling interrupts.

4.3. Module *rtai_shm*

The functionality of this RTAI service is fully described in [2]. Since MCF5407 has no memory management unit, the implementation of shared memory is rather simple and specific in RTAI implementation for ColdFire.

It's possible to split this module into the following parts:

1. RTAI shared memory access functions.
2. Auxiliary functions.
3. RTAI shared memory system request handling routines.
4. The RTAI shared memory module */proc* interface.
5. Module initialization and cleanup.

All RTAI shared memory functions operate with the structure, which definition is given below:

```
struct rtai_kmalloc_node {
  struct rtai_kmalloc_node *next;
  struct rtai_kmalloc_node *prev;
  int name;
  int addr;
  int size;
  int used;
  char alignfill[8];
};
```

The definition of the root node, which exists after loading *rtai_shm* module, is shown below:

```
struct rtai_kmalloc_node rtai_kmalloc_node_root =
 { &rtai_kmalloc_node_root, &rtai_kmalloc_node_root, 0, 0, 0 };
```

4.3.1. RTAI Shared Memory Access Functions

The algorithm of the functions *rtai_malloc* and *rtai_free* execution is:

1. Both these functions call routine *rtai_shmrq*. The first passed parameter is the constant CMD_RTAI_KFREE or CMD_RTAI_KMALLOC, which defines the operation. The second parameter is the information about memory area.
2. The execution of the routine *rtai_shmrq* causes trap #12.
3. In the *init_module* of RTAI shared memory module the handler for trap #12 is installed. It is *rtai_shm_handler*.
4. This handler calls function *shm_handler*, where the process of memory area allocation/freeing takes place.

4.3.1.1. rtai_malloc

SYNOPSIS
```
static inline void *rtai_malloc(unsigned long name, int size);
```

DESCRIPTION
Allocates *size* bytes and brings this memory area to conformity with the identifier *name*. It is used in order to access shared memory from the tasks running in **user** space. See [2] for more details.

RETURN VALUES
The return parameter is the base address of the allocated area of shared memory.

4.3.1.2. rtai_free

SYNOPSIS
```
static inline void rtai_free(unsigned long name, void *addr);
```

DESCRIPTION
Frees memory area with base address *addr* and identified as *name*. Like *rtai_malloc* it is used in order to access shared memory from the tasks running in **user** space. See [2] for more details.

RETURN VALUES
None.

4.3.1.3. rtai_kmalloc

SYNOPSIS
```
void *rtai_kmalloc(int name, int size);
```

DESCRIPTION
Allocates *size* bytes and brings this memory area to conformity with the identifier *name*. It is used in order to access shared memory from the real time tasks running in **kernel** space. See [2] for more details.

RETURN VALUES
The return parameter is the base address of the allocated area of shared memory.

4.3.1.4. rtai_kfree

SYNOPSIS
```
void rtai_kfree(int name);
```

DESCRIPTION

Frees memory area, which is identified as *name*. Like *rtai_kmalloc* it is used in order to access shared memory from the tasks running in **kernel** space. See [2] for more details.

RETURN VALUES
None.

4.3.2. Auxiliary Functions

4.3.2.1. rtai_kmalloc_node_find

SYNOPSIS
```
struct rtai_kmalloc_node* rtai_kmalloc_node_find(int name);
```

DESCRIPTION

This function searches the memory area in shared memory, which is identified as *name*.

RETURN VALUES
If the search was successful, a pointer to the shared memory node would be returned. Otherwise 0 is returned.

4.3.2.2. rtai_kmalloc_node_new

SYNOPSIS
```
struct rtai_kmalloc_node* rtai_kmalloc_node_new(int name, int size);
```

DESCRIPTION
This function creates and initializes a new node in shared memory named *name* with the size *size*. It uses Linux *kmalloc* function.

RETURN VALUES
A pointer to the shared memory node is returned.

4.3.2.3. rtai_kmalloc_node_delete

SYNOPSIS
```
void rtai_kmalloc_node_delete(struct rtai_kmalloc_node* node);
```

DESCRIPTION
This function destroys the node *node* in shared memory. It uses Linux *kfree* function.

RETURN VALUES
None.

4.3.3. RTAI Shared Memory System Request Handling Routines

4.3.3.1. rtai_shmrq

SYNOPSIS
```
static inline int rtai_shmrq(int srq, unsigned int whatever);
```

DESCRIPTION
This function is used by *rtai_malloc* and *rtai_free*. Its source code is provided below:

```
static inline int rtai_shmrq(int srq, unsigned int whatever) {
  register long __res __asm__ ("%d0") = srq;
  register long __whatever __asm__ ("%d1") = whatever;

  __asm__ __volatile__ ("trap #12\n\t"
                : "=d" (__res)
                : "d" (__res), "d" (__whatever));
  return __res;
}
```

After the call of this function trap #12 occurs. It means that ISR *rtai_shm_handler* (see Section 4.3.3.2) is called. The values of parameters *srq* and *whatever* are stored in *d0* and *d1* correspondingly. They are used by the function *shm_handler* (see Section 4.3.3.3).

RETURN VALUES
None.

4.3.3.2. rtai_shm_handler

SYNOPSIS
```
static void rtai_shm_handler(void);
```

DESCRIPTION
The source code of this function is provided below:

```
#define DEFINE_SHM_HANDLER \
```

```
static void rtai_shm_handler(void) \
{ \
      __asm__ __volatile__ ("lea %sp@(-36),%sp\n\t" \
                            "moveml %d0-%d5/%a0-%a2,%sp@\n\t" \
                            "jsr shm_handler\n\t" \
                            "addql #4,%sp\n\t" \
                            "moveml %sp@,%d1-%d5/%a0-%a2\n\t" \
                            "lea %sp@(32),%sp\n\t" \
                            "rte\n\t"); \
}
```

This function is an ISR, which is executed when trap #12 occurs. It calls the function *shm_handler*.

RETURN VALUES
None.

4.3.3.3. shm_handler

SYNOPSIS
```
void* shm_handler(unsigned int srq, unsigned long arg);
```

DESCRIPTION
This function is the final step of memory area freeing/allocation in shared memory by tasks running in **user** space. Depending from the value of *srq* it calls *rtai_kmalloc* or *rtai_kfree*.

RETURN VALUES
0, if *srq* is CMD_RTAI_KFREE (i.e. 2). If *srq* is CMD_RTAI_KMALLOC (i.e. 1), the base address of the allocated area of shared memory is returned. Otherwise this function finishes with the return value -EINVAL.

4.3.4. The RTAI Shared Memory Module */proc* Interface

4.3.4.1. rtai_read_shm

SYNOPSIS
```
static int rtai_read_shm(char *page, char **start, off_t off,
                         int  count, int *eof, void *data);
```

DESCRIPTION
This function is executed when the developer extracts the information about RTAI shared memory from */proc* file system using the command:

```
cat /proc/rtai/shmem
```

All information about allocated areas in shared memory would be displayed.

RETURN VALUES
None

4.3.4.2. rtai_proc_shm_register

SYNOPSIS
```
static int rtai_proc_shm_register(void);
```

DESCRIPTION
This function creates the entry for *rtai_shm* module */proc/rtai/shmem*.

RETURN VALUES
On success 0 is returned.
-1, if an error occurred during initialization.

4.3.4.3. rtai_proc_shm_unregister

SYNOPSIS
```
static void rtai_proc_shm_unregister(void);
```

DESCRIPTION
This function removes the entry for *rtai* module */proc/rtai/shmem*.

RETURN VALUES
None.

4.3.5. Module Initialization and Cleanup.

4.3.5.1. init_module

SYNOPSIS
```
int init_module(void);
```

DESCRIPTION
This routine is a usual entry point for *rtai_shm* module. It registers module in */proc* file system and sets up the exception handler, which is executed when trap #12 occurs.

RETURN VALUES
On success 0 is returned.

4.3.5.2. cleanup_module

SYNOPSIS
```
void cleanup_module(void);
```

DESCRIPTION
This routine is a usual exit point for *rtai_shm* module. It removes RTAI shared memory entry in */proc* file system, restores the handler, which Linux executed (before loading RTAI shared memory module), when trap #12 occurs.

RETURN VALUES
None

5. RTAI Test Suite

The chapter contains the results of some tests from RTAI standard test suite (with the description of changes in their source code) and the description of specific tests, which were developed for MCF5407.

It's assumed that all test examples are in *romfs* file system.

5.1. Tests from RTAI Standard Test Suite

5.1.1. edf

LOCATION
rtai-24.1.8/examples/edf

DESCRIPTION
This example features NTASKS real time tasks running periodically in EDF mode. The task are given a priority ordered in such a way that low numbered tasks have the lowest priority. However the tasks execute for duration proportional to their number so that, under EDF, the lowest priority tasks run first. During the first loop real time tasks run according to their priorities, but then the routine *rt_task_set_resume_end_time* changes their priorities to 0 and changes resume and end times:

```
rt_task_set_resume_end_times(-NTASKS*tick_period, -(t + 1)*tick_period);
```

Therefore the tasks should appear increasingly ordered on the screen.

CHANGES
No changes were made

INSTRUCTIONS TO RUN/STOP
```
insmod ex_edf
…
rmmod ex_edf
```

OUTPUT

```
/> insmod ex_edf
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_edf.o
/> TASK 15 0


TASK 14 1
TASK 13 2
TASK 12 3
TASK 11 4
TASK 10 5
TASK 9 6
```

```
TASK 8 7
TASK 7 8
TASK 6 9
TASK 5 10
TASK 4 11
TASK 3 12
TASK 2 13
TASK 1 14
TASK 0 15
TASK 0 0
TASK 1 0
TASK 2 0
TASK 3 0
TASK 4 0
TASK 5 0
TASK 6 0
TASK 7 0
TASK 8 0
TASK 9 0
TASK 10 0
TASK 11 0
TASK 12 0
TASK 13 0
TASK 14 0
TASK 15 0


TASK 0 0
TASK 1 0
TASK 2 0
TASK 3 0
TASK 4 0
TASK 5 0
TASK 6 0
TASK 7 0
TASK 8 0
TASK 9 0
TASK 10 0
TASK 11 0
TASK 12 0
TASK 13 0
TASK 14 0
TASK 15 0


TASK 0 0
TASK 1 0
TASK 2 0
…
…
/> rmmod ex_edf


CPU USE SUMMARY
# 0 -> 1253
END OF CPU USE SUMMARY

/>
```

5.1.2. fastick

LOCATION
rtai-24.1.8/examples/fastick

DESCRIPTION
These examples show how the developer can use the module *rtai* to install either a real time task (ex_fastick_1) or an interrupt handler (ex_fastick_2) to simulate a higher tick for a task communicating with them through RTAI FIFO.

CHANGES
No changes were made

INSTRUCTIONS TO RUN/STOP
```
insmod ex_fastick_1
checkfast
…
rmmod ex_fastick_1
insmod ex_fastick_2
checkfast
…
rmmod ex_fastick_2
```

OUTPUT
For ex_fastick_2 output should be the same. The number's printing can be stopped using "Ctrl"+"C".

```
/> insmod ex_fastick_1
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_fastick_1.o
/> checkfast
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
…
…
/> rmmod ex_fastick_1


CPU USE SUMMARY
# 0 -> 446917
END OF CPU USE SUMMARY

/>
```

5.1.3. jepplin

LOCATION
rtai-24.1.8/examples/jepplin

DESCRIPTION
This example demonstrates the use of semaphores and message queues. See file *rtai-24.1.8/examples/jepplin/README* for more details.

CHANGES
No changes were made

INSTRUCTIONS TO RUN/STOP
```
insmod ex_jepplin
…
rmmod ex_jepplin
```

OUTPUT

```
/mnt> insmod ex_jepplin
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_jepplin.o
Joey    Johnny  Dee Dee Marky
Joey    Johnny  Dee Dee Marky
Joey    Johnny  Dee Dee Marky
Joey    Johnny  Dee Dee Marky
```

```
Joey    Johnny  Dee Dee Marky
sync
Marky   Dee Dee Johnny  Joey
/> sem timeout, task 0, Joey
sem timeout, task 1, Johnny
sem timeout, task 2, Dee Dee
sem timeout, task 3, Marky
sync
testing message queues

received by task 0 Joey
received by task 3 Johnny
received by task 2 Dee Dee
received by task 1 Marky
received from mbx_out: Joey
received from mbx_out: Marky
received from mbx_out: Dee Dee
received from mbx_out: Johnny

init task complete
mbx timeout, task 0, Joey

task 0 complete
mbx timeout, task 1, Johnny

task 1 complete
mbx timeout, task 2, Dee Dee

task 2 complete
mbx timeout, task 3, Marky

task 3 complete

/>rmmod ex_jepplin
```

5.1.4. preempt

LOCATION
rtai-24.1.8/examples/preempt

DESCRIPTION
This test verifies that a fast high priority task preempts a long lasting lower priority one. These tasks send to RTAI FIFO messages, which contain information about tasks (F – fast task with period 40 msec, S – slow task with period 240 msec). At the interval between two messages from the slow task the fast task should send 6 messages to RTAI FIFO. See file *rtai-24.1.8/examples/preempt/README* for more details.

CHANGES
Several minor changes were made in the files *rtai-24.1.8/examples/preempt /rt_process.c* and *rtai-24.1.8/examples/preempt/check.c*.

INSTRUCTIONS TO RUN/STOP
insmod ex_preempt
check
…

```
rmmod ex_preempt
```

OUTPUT
Printing of information by *check* can be stopped using “Ctrl”+”C”.

```
/> insmod ex_preempt
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_preempt.o

/> check
> F s 0 199832
> F r 0 199834
> F s 0 199836
> F r 0 199838
> F s 0 199840
> S r 0 199840
> F r 0 199842
> F s 0 199844
> F r 0 199846
> F s 0 199848
> F r 0 199850
> F s 0 199852
> S s 0 199852
> F r 0 199854
> F s 0 199856
> F r 0 199858
> F s 0 199860
> F r 0 199862
> F s 0 199864
> S r 0 199864
> F r 0 199866
> F s 0 199868
> F r 0 199870
> F s 0 199872
> F r 0 199874
> F s 0 199876
> S s 0 199876
> F r 0 199878
> F s 0 199880
> F r 0 199882
> F s 0 199884
> F r 0 199886
> F s 0 199888
> S r 0 199888
> F r 0 199890
> F s 0 199892
> F r 0 199894
> F s 0 199896
> F r 0 199898
> F s 0 199900
> S s 0 199900
> F r 0 199902
> F s 0 199904
> F r 0 199906
> F s 0 199908
> F r 0 199910
> F s 0 199912
> S r 0 199912
> F r 0 199914
> F s 0 199916
```

```
> F r 0 199918
> F s 0 199920
> F r 0 199922
> F s 0 199924
> S s 0 199924
> F r 0 199926
> F s 0 199928
> F r 0 199930
> F s 0 199932
> F r 0 199934
> F s 0 199936
> S r 0 199936
> F r 0 199938
> F s 0 199940
> F r 0 199942
> F s 0 199944
> F r 0 199946
> F s 0 199948
> S s 0 199948
> F r 0 199950

/> rmmod ex_preempt


CPU USE SUMMARY
# 0 -> 753
END OF CPU USE SUMMARY

/>
```

5.1.5. stress

LOCATION
rtai-24.1.8/examples/stress

DESCRIPTION
This test verifies that the scheduler works well under intense load. It combines the latency calibration with the preemption example (see Section 5.1.4) in such a way to have two level of preemption nesting on an odd number of tasks. To verify what is happening the developer can launch either *checkl* for the latency output or *checkp* for the preemption output.

CHANGES
Just one modification was made – constant *TICK_TIME* in the file *rtai-24.1.8/examples/stress/rt_process.c* is 1000000 instead of 500000.

INSTRUCTIONS TO RUN/STOP
```
insmod ex_stress
checkl
…
checkp
…
rmmod ex_stress
```

OUTPUT

Printing of information by *checkl* and *checkp* can be stopped using “Ctrl”+”C”.

```
/> insmod ex_stress
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_stress.o
/> checkl
*** min:     6000, max:    17000 average: 8281 flags: 0***
*** min:     6000, max:    17000 average: 8305 flags: 0***
*** min:     6000, max:    30000 average: 8598 flags: 0***
*** min:     6000, max:    30000 average: 8288 flags: 0***
*** min:     6000, max:    30000 average: 8256 flags: 0***
*** min:     6000, max:    33000 average: 8588 flags: 0***
*** min:     6000, max:    33000 average: 8330 flags: 0***
*** min:     6000, max:    33000 average: 8293 flags: 0***
*** min:     6000, max:    33000 average: 8298 flags: 0***
*** min:     6000, max:    33000 average: 8541 flags: 0***
*** min:     6000, max:    33000 average: 8281 flags: 0***
*** min:     6000, max:    33000 average: 8317 flags: 0***
*** min:     6000, max:    33000 average: 8312 flags: 0***
*** min:     6000, max:    33000 average: 8275 flags: 0***
*** min:     6000, max:    33000 average: 8551 flags: 0***
*** min:     6000, max:    33000 average: 8268 flags: 0***
*** min:     6000, max:    33000 average: 8296 flags: 0***
*** min:     6000, max:    33000 average: 8358 flags: 0***
*** min:     6000, max:    33000 average: 8246 flags: 0***
*** min:     6000, max:    33000 average: 8227 flags: 0***
*** min:     6000, max:    33000 average: 8303 flags: 0***
*** min:     6000, max:    33000 average: 8313 flags: 0***
*** min:     6000, max:    33000 average: 8670 flags: 0***
*** min:     6000, max:    33000 average: 8286 flags: 0***
*** min:     6000, max:    33000 average: 8327 flags: 0***
*** min:     6000, max:    33000 average: 8232 flags: 0***
*** min:     6000, max:    33000 average: 8320 flags: 0***
*** min:     6000, max:    33000 average: 8556 flags: 0***
*** min:     6000, max:    33000 average: 8286 flags: 0***
*** min:     6000, max:    33000 average: 8532 flags: 0***
*** min:     6000, max:    33000 average: 8295 flags: 0***
*** min:     6000, max:    33000 average: 8292 flags: 0***
/> checkp
> F r 0 341608
> F s 0 341610
> S r 0 341610
> F r 0 341612
> F s 0 341614
> F r 0 341616
> F s 0 341618
> F r 0 341620
> F s 0 341622
> S s 0 341622
> F r 0 341624
> F s 0 341626
> F r 0 341628
> F s 0 341630
> F r 0 341632
> F s 0 341634
> S r 0 341634
> F r 0 341636
> F s 0 341638
> F r 0 341640
> F s 0 341642
```

```
> F r 0 341644
> F s 0 341646
> S s 0 341646
> F r 0 341648
> F s 0 341650
> F r 0 341652
> F s 0 341654
> F r 0 341656
> F s 0 341658
> S r 0 341658
> F r 0 341660
> F s 0 341662
> F r 0 341664
> F s 0 341666
> F r 0 341668
> F s 0 341670
> S s 0 341670
> F r 0 341672
> F s 0 341674
> F r 0 341676
> F s 0 341678
> F r 0 341680
> F s 0 341682
> S r 0 341682
> F r 0 341684
> F s 0 341686
> F r 0 341688
> F s 0 341690
> F r 0 341692
> F s 0 341694
> S s 0 341694
> F r 0 341696
/> rmmod ex_stress


CPU USE SUMMARY
# 0 -> 336566
END OF CPU USE SUMMARY

/>
```

5.2. MCF5407 Specific Tests

5.2.1. mbx_example

LOCATION
rtai-24.1.8/5407_examples/mbx_example

DESCRIPTION
This example demonstrates the use of RTAI mailboxes. There are eight tasks, which receive messages from the mailbox *mbx* and one task, which deletes mailbox. In *init_module* after the start of tasks-receivers two messages are sent to the mailbox. When the task gets messages, it unblocks and prints the information, which was received from the mailbox. Therefore tasks *TASK0* and *TASK1* get the correct information. Afterwards the task, which deletes the mailbox, should start. All other tasks should be

unblocked after removing the mailbox and inform about the quantity of bytes, which weren't received.

INSTRUCTIONS TO RUN/STOP

```
insmod mbx_example
…
rmmod mbx_example
```

OUTPUT

```
/> insmod mbx_example
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/mbx_example.o

TASK 0 BLOCKS RECEIVING ON MBX
TASK 1 BLOCKS TIMED RECEIVING ON MBX
TASK 2 BLOCKS RECEIVING ON MBX
TASK 3 BLOCKS TIMED RECEIVING ON MBX
TASK 4 BLOCKS RECEIVING ON MBX
TASK 5 BLOCKS TIMED RECEIVING ON MBX
TASK 6 BLOCKS RECEIVING ON MBX
TASK 7 BLOCKS TIMED RECEIVING ON MBX

SENDING 0x31323334 TO MBX...

TASK 0 UNBLOCKS. BYTES NOT RECEIVED: 0
TASK 0 GOT MESSAGE: 31323334

SENDING 0x35363738 TO MBX...

TASK 1 UNBLOCKS. BYTES NOT RECEIVED: 0
TASK 1 GOT MESSAGE: 35363738


ANOTHER TASK DELETES THE MBX AND ...


TASK 3 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 3 GOT MESSAGE: 328114

TASK 4 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 4 GOT MESSAGE: 328114

TASK 5 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 5 GOT MESSAGE: 328114

TASK 6 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 6 GOT MESSAGE: 328114

TASK 7 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 7 GOT MESSAGE: 328114

TASK 2 UNBLOCKS. BYTES NOT RECEIVED: 4
TASK 2 GOT MESSAGE: 328114

/> rmmod mbx_example
```

5.2.2. periodic_two_tasks

LOCATION
rtai-24.1.8/5407_examples/periodic_two_tasks

DESCRIPTION
This example provides the visual representation of the periodic task switching. The RTAI timer runs in periodic mode with the period 1 sec. LED D8 blinks each 1 sec and LED D8 – each 4 secs. Two signal functions print LED's current state. Note that the task, which changes the state of LED D8, has the highest priority.

INSTRUCTIONS TO RUN/STOP
```
insmod rt_p2t
…
rmmod rt_p2t
```

OUTPUT

```
/> insmod rt_p2t
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/rt_p2t.o
LOOK AT LEDs D6 & D8

LED D8 off
LED D8 on
LED D8 off
LED D8 on
LED       D6 off
LED D8 off
LED D8 on
LED D8 off
LED D8 on
LED       D6 on
LED D8 off
LED D8 on
LED D8 off
LED D8 on
LED       D6 off
LED D8 off
LED D8 on
LED D8 off
LED D8 on
LED       D6 on
LED D8 off
LED D8 on
LED D8 off
LED D8 on
LED       D6 off
LED D8 off
LED D8 on

/> rmmod rt_p2t
```

5.2.3. rtai_oneshot

LOCATION
rtai-24.1.8/5407_examples/rtai_oneshot

DESCRIPTION
In this example RTAI real time task produces the rectangular wave on the parallel port. The RTAI timer runs in oneshot mode with the period 100 usec.

INSTRUCTIONS TO RUN/STOP
insmod rtai_oneshot
…
rmmod rtai_oneshot

OUTPUT
The 10 KHz signal can be captured from the expansion connector J2, pin 26 on M5407C3 board.

5.2.4. rtai_periodic

LOCATION
rtai-24.1.8/5407_examples/rtai_periodic

DESCRIPTION
In this example RTAI real time task produces the rectangular wave on the parallel port. The RTAI timer runs in periodic mode with the period 50 usec.

INSTRUCTIONS TO RUN/STOP
insmod rtai_periodic
…
rmmod rtai_periodic

OUTPUT
The 20 KHz signal can be captured from the expansion connector J2, pin 26 on M5407C3 board.

5.2.5. RPC_example

LOCATION
rtai-24.1.8/5407_examples/RPC_example

DESCRIPTION
This example demonstrates the use of RTAI real time tasks communication using remote procedure calls and message handling functions. It checks the correct execution of the functions *rt_rpc, rt_isrpc, rt_send, rt_receive* and *rt_return*.

INSTRUCTIONS TO RUN/STOP

```
insmod ex_rpc
…
rmmod ex_rpc
```

OUTPUT

```
/> insmod ex_rpc
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_rpc.o

TASK1: MAKE RPC TO TASK3; SEND 1
TASK3: RECEIVED FROM TASK1: 1
TASK3: SEND 3 TO TASK1
TASK1: RECEIVED FROM TASK3 3
TASK1: MAKE RPC TO TASK2; SEND 11
TASK2: RECEIVED FROM TASK1: 11
TASK2: SEND 2 TO TASK1
TASK1: RECEIVED FROM TASK2 2
TASK1: --- FINISHED ---
TASK2: MAKE RPC TO TASK3; SEND 2
TASK3: RECEIVED FROM TASK2: 2
TASK3: SEND 3 TO TASK2
TASK2: RECEIVED FROM TASK3 3
TASK2: SEND WITH rt_send 22 TO TASK3
TASK2: TASK3 WAS READY. ---  FINISHED ---
TASK3: RECEIVED INFORMATION: 22
TASK3: --- FINISHED ---

/> rmmod ex_rpc
```

5.2.6. rtf_sem_example

LOCATION

rtai-24.1.8/5407_examples/rtf_sem_example

DESCRIPTION

This example demonstrates the use of RTAI FIFO semaphores. It checks the correct execution of the functions *rtf_sem_init, rtf_sem_post, rtf_sem_trywait* and *rtf_sem_destroy*.

INSTRUCTIONS TO RUN/STOP

```
insmod ex_rtf_sem
checksem
…
rmmod ex_rtf_sem
```

OUTPUT

```
/> insmod ex_rtf_sem
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_rtf_sem.o
/> checksem

READY TO MAKE rtf_sem_post
*** TIME: min:     7000, max:    32000 average: 7632 ***
*** TIME: min:     7000, max:    11000 average: 7706 ***
```

```
*** TIME: min:     6000, max:    11000 average: 7576 ***
*** TIME: min:     6000, max:    11000 average: 7707 ***
*** TIME: min:     6000, max:    11000 average: 7675 ***

LET'S TRY TO READ INFO FROM FIFO WITHOUT rtf_sem_post
TIMED OUT READ TIMED OUT 0

/> rmmod ex_rtf_sem
```

5.2.7. shared_memory

LOCATION
rtai-24.1.8/5407_examples/shared_memory

DESCRIPTION
This example demonstrates the use of RTAI shared memory module. It checks the correct execution of the functions *rtai_malloc* and *rtai_free* for the tasks running in **user** space and *rtai_kmalloc* and *rtai_kfree* for the tasks running in **kernel** space.

INSTRUCTIONS TO RUN/STOP
insmod ex_kmalloc
malloctest
…
rmmod ex_kmalloc

OUTPUT

```
/> insmod ex_kmalloc
insmod: /lib/modules/2.4.19-uc1: No such file or directory
Using /lib/modules/rtai/ex_kmalloc.o

---ex_kmalloc test---
ALLOCATING aaaa AND bbbb IN CURRENT PROCESS
NOW THE FIRST 4 BYTES ARE 6f64756c AND 2bf20

WE CHANGE THEM TO 11223344
NOW IN MEMORY WE HAVE: 11223344 AND 11223344
/> malloctest

--- malloctest ---
ALLOCATING cccc AND dddd IN CURRENT PROCESS
NOW THE FIRST 4 BYTES ARE 0 AND 197fd

WE CHANGE THEM TO 55667788
NOW IN MEMORY WE HAVE: 55667788 AND 55667788

 BEFORE FREEING LET'S LOOK AT 4 FIRST BYTES:
THEY ARE: 55667788 AND 55667788
NOW FREEING cccc AND dddd

/> rmmod ex_kmalloc

 BEFORE FREEING LET'S LOOK AT 4 FIRST BYTES:
THEY ARE: 11223344 AND 11223344
NOW FREEING aaaa AND bbbb
/>
```