Networked control system using Linux Real Time Application Interface

TOMÁŠ MURGAŠ*, PETER FODREK*, ĽUDOVÍT FARKAS**

*RT Systems s.r.o.
Kopčianska 14, 851 01 Bratislava
SLOVAK REPUBLIC

**Institute of Control and Industrial Informatics
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
Ilkovičova 3, 812 19 Bratislava
SLOVAK REPUBLIC
tomas.murgas@rtsystems.sk, peter.fodrek@rtsystems.sk, ludovit.farkas@stuba.sk,

Abstract: - This paper will report our results of Real Time Application Interface hard real time performance measurement. We will start with basics of RTAI installation and Linux kernel patching principle. Next we will to describe our Control Area Network library driver for Adlink based CAN controller. We have been specially used PCI-7841 controller card. It contains two SJA1000 controller chips. This library driver will be tested using real time application of RTAI and third party measurement tools. We anticipated that real time capabilities of the RTAI will not be negated by our library. Practical results show that our anticipation was right. We achieved hard real time performance for whole system.

Key-Words: - Real Time Application Interface, Distributed control system, Control Area Network, Performance measurement, Hard real time OS.

1 Introduction

The clear trend in industrial automation today is the development, deployment and use of Networked Control Systems (NCS). Its idea is based on digital system for controlling actuators and data collection process using sensors. All elements of NCS are connected via industrial communication network. It is therefore an appropriate set of hardware and software capable of communicating according to a defined industry standard. Most manufactures provide standard communication protocol which enable us to create a driver for connecting all NCS elements. Industrial automation process requires real time control, so it is necessary to choose an appropriate operating system (OS). OS is a system program that provides an interface between applications and computer hardware. In most cases it is designed for mainstream business use. Exact timing is considered critical for proper functionality and quality management for NCS. It is necessary to choose the operating system to ensure precise sampling rate and the fastest reactions to events. In case we are interested to create a universal platform NCS it is necessary to choose the OS that allow us its further modifications. In this case it is advisable to choose the freely redistributable operating system using General Public License (GPL) from GNU.

2 Real time operating system

An Real Time Operating Systems (RTOS) is an operating system specialized for real time operations. It is possible to divide into two main categories:

- soft real-time OS
- hard real-time OS

A hard real-time operating system has less jitter than a soft real-time operating system. Response time jitter is one of the most sneaky source of problems when designing a real time system. It means deviation between true periodicity of an assumed periodic signal, often in relation to a reference clock source. An hard real time system is a system where the programmed reaction to a stimulus is guaranteed to be completed within a known finite time. An important parameter when evaluating an RTOS is its response time. An efficient RTOS only adds a small overhead to the system theoretical minimal response time. Typical parameters falling in this category are:

- interrupt latency
- threads fly-back time
- context switch time

2.1 Real Time Application Interface

Real Time Application Interface (RTAI) is an extension of standard Linux kernel which allows us to change a default Linux operating system to hard
RTOS. RTAI principle is in installation of RTAI patch (text file). With this patch we change several hundred lines program code of basic Linux kernel. The main change is full preemption and dual kernel architecture. Real time tasks use additional installed microkernel. In general there are functions and data structures necessary to handle hardware interrupts, software interrupts handlers, system calls and timers. Since the Linux kernel is modular and allows insert and remove kernel modules at run time, RTAI use this feature. After inserting of specific modules into the kernel, RTAI takes all the necessary functions, but provides operation of the rest Linux operating system, including the Linux desktop applications.

2.2 RTAI installation

It is possible to use any Linux distribution as the base OS, but it is advised to use conservative distribution like Slackware. It is also often recommended as a distribution on less powerful computers, since the startup does not run as many services as in the case of "desktop"-oriented distribution. Latest stable release of the RTAI contains the certified patches for the GNU/Linux kernel. Any RTAI release comes with several patches, but all of them are for a specific vanilla GNU/Linux kernel. This means that RTAI is designed to run on a standard Linux kernel and not on the kernel's source code that comes with the most common distributions. We are able to import configuration file of the actual running kernel for RTAI kernel. We need to pay attention to enable all the features your hardware requires to work properly. This includes the file system, the network card and other so in the end you will not be able to use the system without restrictions. The first step is to patch and build a vanilla Linux kernel. We need to chose patch suitable for our Linux distribution and computer architectures. Once we chose one patch that match our kernel version we can apply selected patch. After patch kernel source we need to configure and compile it. The configuration depends mainly on specific needs, hardware specification and the devices which are on the system. It's a good idea to start the configuration by importing the existing .config file. Beside all the specific options we may enable or disable. There are several of them which must be configured to allow RTAI to work properly. After compiling RTAI patched kernel we have to configure RTAI alone by

$ make menuconfig

After configuration we have to compile RTAI with command

$ make

After that we have to proceed the installation command

$ make install

Everything needed to use the RTAI distribution will be copied to the default installation directory.

3 RTAI performance test

After successful installation we have a suitable real time OS. Within the help of 3 tests, which are part of each RTAI installation we will test our system for real time operations.

3.1 Latency test

We used standard latency test that is part of RTAI installation and we can found in /usr/realtime/testsuite/ to demonstrate the key features of hard realtime systems. It verifies the architectural latency and jitter, up to the scheduling of a task. We conducted four tests, two of them are running in user mode, and two are in kernel mode. We measured the delay (latency) between planned and actual start of the thread. All data are displayed in nanoseconds.

Fig. 1 show results of first test run in user mode without additional load. The results can be compared with the second run of the same test, but with the difference that this time we tried to load the system with running glxgears program, lunch several periodic network pinging command in background while compressing large system files. Nevertheless, as can be seen in Fig. 2 test results are very similar and average latency increased for a few hundred nanoseconds.

<table>
<thead>
<tr>
<th>Test</th>
<th>Lat min</th>
<th>Lat max</th>
<th>Lat avg</th>
<th>Over runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>1.2 ms</td>
<td>5.8 ms</td>
<td>3.4 ms</td>
<td>1.1 ms</td>
</tr>
<tr>
<td>User</td>
<td>1.2 ms</td>
<td>5.8 ms</td>
<td>3.4 ms</td>
<td>1.1 ms</td>
</tr>
<tr>
<td>Kernel</td>
<td>1.2 ms</td>
<td>5.8 ms</td>
<td>3.4 ms</td>
<td>1.1 ms</td>
</tr>
<tr>
<td>Kernel</td>
<td>1.2 ms</td>
<td>5.8 ms</td>
<td>3.4 ms</td>
<td>1.1 ms</td>
</tr>
</tbody>
</table>

Fig. 1: RTAI latency test without load.

Fig. 2: RTAI latency test without load.

Another pair of tests was made in kernel mode. They are same conditions as in user space test. On the results we can see low difference between this two tests. The difference from the previous pair is lower average latency of approximately 0.5 microseconds. This test confirms our installation of RTAI as a suitable system for real time operations.
3.2 Switch test

There is another important test tool, which is placed in the testsuite directory, that can provide us some informations about the maximum time RTAI needs to disable the interrupts. The software is under the testsuite/kern/switches (or user/switches) and checks for the task context switching timings. If the values are too high we should verify if there is any “latency killer” present. “Latency killer” cause unpredictable timing results and that is incompatible with the concept of real-time. It could be:

- heavy DMA activities (like the hard-drive or other PCI devices)
- the use of an accelerated x-server
- USB legacy support
- power management (both APM and ACPI)
- CPU frequency scaling
- many others.

4 Control Area Network driver using RTAI

Control Area Network (CAN) bus was originally designed for the automotive industry. In this paper we will not describe CAN protocol which can be found in many papers. We will explain you our software driver made specially for using CAN controller in real time operating system RTAI. We used Adlink PCI-7841 controller. It is very difficult to describe the entire driver, so we try to focus only on the most important parts.

4.1 CAN driver for Adlink PCI-7841

Our driver is programmed in C programming language as a library to be compiled with kernel module application. First step to run application made in RTAI is to compile whole application as a kernel module. Basic part of code that is to be run when we insert kernel module in to RTAI kernel is called init_module.

```c
int init_module(void)
{
    timer=start_rt_timer(nano2count(4700*28));
    rtf_create(MSG_PROCESS,1000);
    pc7841_init_card();
    return(0);
}
```

Init module start up real time timer and create to FIFOs necessary for time synchronization between sending and receiving data. Our self made function pc7841_init_card() starts up Adlink PCI-7841.

```c
int pc7841_init_card(void)
{
    if(dev_pci7841=pci_find_device(SJA_CAN_PCI_VENDOR_ID,
                                SJA_CAN_PCI_DEVICE_ID, dev_pci7841))
        return RET_ERROR;
    if(sja1000_initboard(dev_pci7841))
        return RET_ERROR;
    if((errno=rt_request_global_irq((unsigned int)
                              (dev_pci7841->irq),pci7841_irq_handler)))
    {
        if(errno==EINVAL)
            rt_printk("Bad IRQ
");
        if(errno==EBUSY)
            rt_printk("Used IRQ
");
        rt_startup_irq((unsigned int)(dev_pci7841->irq));
        rt_enable_irq(dev_pci7841->irq);
        return RET_OK;
    }
    return 0;
}
```

This function start PCI-7841 card as pci device and assign an IRQ fro the card to the interrupt handler function. It is called each time we received or send any CAN data. To ensure memory space for our IO operations we call function

```c
int sja1000_initboard(struct pci_dev *dev)
{
    if(check_region(PLX9050_BASE, 127))
        return RET_ERROR;
    if(check_region(SJA_BASE[0], 255))
        return RET_ERROR;
    request_region(PLX9050_BASE, -(dev->resource[1].start)+
                   dev->resource[1].end, "PCI7841/PLX9050");
    request_region(SJA_BASE[0], -(dev->resource[2].start)+
                   dev->resource[2].end, "PCI7841/SJA1000(1)";
    request_region(SJA_BASE[1], -(dev->resource[3].start)+
                   dev->resource[3].end, "PCI7841/SJA1000(2)";
    SJA_BASE[1] = SJA_BASE[0]+128;
    return 0;
}
```
After communication start up we are to prepare interrupt handler to ensure card interrupts generated upon receiving a data from the bus.

```c
void message_handler(int port, DataReceive *data)
{
    rtf_put(MSG_PROCESS, data, sizeof(DataReceive));
    rt_sem_signal(&sem_msg_processing);
}
```

This simple handler just put received data and send him to prepared FIFO named MSG_PROCESS. Releasing the semaphore we synchronize another thread where we read data from this FIFO and where we can do any logic.

```c
void thread_msg_processing(long t)
{
    while(1)
    {
        rt_sem_wait(&sem_msg_processing);
        if(rtf_get(MSG_PROCESS, &data, sizeof(DataReceive)) == sizeof(DataReceive))
        {
            ....
            "any logic algorithms with received data"
            ....
        }
    }
}
```

To send data to network we prepare another thread

```c
void thread_msg_send_port(long t)
{
    while(1)
    {
        rt_sem_wait(&sem_msg_transmit_port);
        pci7841_send_transmit(0, &g_sDataToTransmit);
        rt_sleep(nano2count(450000));
        rt_sem_signal(&sem_msg_send_port);
    }
}
```

In this thread we send prepared packet of data store in variable g_sDataToTransmit. Semaphores are used to synchronize sending data. If we would like to send an data in our application we use function

```c
void fnc_msg_send(DataTransmit dataT)
{
    rt_sem_wait(&sem_msg_send_port);
    g_sDataToTransmit = dataT;
    rt_sem_signal(&sem_msg_transmit_port);
}
```

Our time synchronization using semaphores observe sending data in time delay 450 microseconds between two CAN messages should be sent. This time must be respected due to various devices connected to the bus to recognize several messages coming immediately.

```c
int pci7841_send_transmit(int port, DataTransmit *data)
{
    rt_disable_irq((unsigned int)(dev_pci7841->irq));
    sja1000_sendframe(port, &data->par.msg);
    rt_enable_irq((unsigned int)(dev_pci7841->irq));
}
```

This function disable card IRQ and send CAN data to the network.

```c
Message* sja1000_sendframe(int bus_nb, Message *mes)
{
    int dlc, i;
    unsigned int SJA = SJA_BASE[bus_nb];
    dlc = mes->len & 0x0f;
    outb((mes->cob_id.b.b1 & 0x07) << 5 | ((mes->cob_id.b.b0 & 0x08) >> 3), SJA_TRANSMIT_ID_0);
    outb((mes->cob_id.b.b0 & 0x07) << 5 | ((mes->rtr & 0x01)<< 4) | dlc, SJA_TRANSMIT_ID_1);
    for(i=0; i < dlc; i++) {
        outb(mes->data[i], SJA_TRANSMIT_DATA+i);
    }
    outb(SJA_TRANS_REQUEST, SJA_COMMAND);
    return mes;
}
```

This is function is low level function which sends prepared data to corresponding bytes in memory space of CAN card selected by communication protocol of used chipset (SJA1000).

5. System testing

For testing communication driver we used physical model of metal materials cutting machine. It's an laboratory model, which is used to simulate cutting technology of metal materials. The actual cutting is performed moving the technological cutting head over the material in a well defined constant rate, after a pre-programmed path. It is motion control system with 3 servos provide motion using 3 axes. To achieve best precision we need to send data as fast as possible. Due to hardware limits we can't send our data faster then every 450 microseconds. In case of sending data immediately, our hardware components fail to recognize sent packets. Given limitations, we have to set sampling time to 2 milliseconds. Figure 4 shows complete system.

Fig. 4: Physical laboratory model of cutting machine

Fig. 5: 3 servo systems connected by CAN bus.
5.1 Testing program

In our test we will send 10000 samples with sampling period 2ms. Each servo systems send us back received value and also actual speed and position of system. Send and received data will be stored in our control system and also by hardware diagnostic tool. To demonstrate functionality and stability our NCS using RTAI and CAN controller we prepared program that consist several threads and our CAN function library.

During test we will send 10000 samples with sampling period 2ms by CANbus network to 3 Lenze servos. Each packet will be sent true Process Data Channel (PDO). CANopen protocol defines packet identifier as basic identifier + device address. In our test we set address of Lenze servo systems for 11,12,13 so identifier for each servo system will be:

- \[512 + 11 = 523_{\text{dec}} = 20B_{\text{hex}}\]
- \[512 + 12 = 524_{\text{dec}} = 20C_{\text{hex}}\]
- \[512 + 13 = 525_{\text{dec}} = 20D_{\text{hex}}\]

Packet transmitted by device will be again set with CANopen protocol as:

- \[384 + 11 = 395_{\text{dec}} = 18B_{\text{hex}}\]
- \[385 + 12 = 396_{\text{dec}} = 18C_{\text{hex}}\]
- \[386 + 13 = 397_{\text{dec}} = 18D_{\text{hex}}\]

Each packet will contain 8 data bytes, but we use only first 2 bytes where will be order number of sent packet. At the end of the test there should by number 10000dec = 2710hex. After successful compilation we start our testing program by inserting RTAI modules in to the kernel by:

```
$ /sbin/insmod /bin/rtai_hal
$ /sbin/insmod /bin/rtai_sched
$ /sbin/insmod /bin/rtai_sem
$ /sbin/insmod /bin/rtai_sem
```
and then we are to insert our project modules in to the kernel by:

```
$ /sbin/insmod /bin/adlink.ko
$ /sbin/insmod /bin/can.ko
$ /sbin/insmod /bin/canopen.ko
$ /sbin/insmod /bin/application.ko
```

6 Testing results

Fig. 7 shows results for first and last samples from standard kernel print by typing command $ dmesg. Figure show that we start transmitting data at time of 103.470578s. After send packets for all tree servo systems we need to confirm transmit by sending SYNC packet (0x80) and all devices send us their actual values. At the end of transmitting we can see time of last sent sample for first servo systems 123.470369s. Timer difference between start and end of transmit is then 19,999791s. Compared to the desired time of 20s it is different of 209 microseconds. In received data we can see delay of whole NCS because as we send our last data packet with value 10000dec (1027hex), in the time but value of received data is only 9998, so there is 2 samples delay. Last value 1027hex we received in time 123.474414 seconds so difference between last send and received value is 123. 474414 – 123.471158 = 3,256 ms.

```
[103.470578] send> id: 523 port: 0 len: 8 data: 01 00 00 00 00 00 00 00
[103.470943] send> id: 524 port: 0 len: 8 data: 01 00 00 00 00 00 00 00
[103.471345] send> id: 525 port: 0 len: 8 data: 01 00 00 00 00 00 00 00

[123.469947] RECEIVE PACKET No. 29995
recv > id: 395 rtr: 0 len: 8 data: 0e 27 ff 00 00 00 00 00
[123.470191] RECEIVE PACKET No. 29996
recv > id: 396 rtr: 0 len: 8 data: 0e 27 02 00 00 00 00 00
[123.470369] receive id:523 port:0 len:8 data:10 27 00 00 00 00 00 00

[123.470432] RECEIVE PACKET No. 29997
recv > id: 397 rtr: 0 len: 8 data: 0e 27 ff 00 00 00 00 00
[123.470672] send> id: 524 port: 0 len: 8 data: 10 27 00 00 00 00 00 00
[123.471158] send> id: 525 port: 0 len: 8 data: 10 27 00 00 00 00 00 00
[123.471545] send> id: 128 port: 0 len: 0 data:
[123.471919] RECEIVE PACKET No. 29998
recv > id: 395 rtr: 0 len: 8 data: 0f 27 f7 ff 00 00 00 00
[123.472164] RECEIVE PACKET No. 29999
recv > id: 396 rtr: 0 len: 8 data: 0f 27 01 00 00 00 00 00
[123.472408] RECEIVE PACKET No. 30000
recv > id: 397 rtr: 0 len: 8 data: 0f 27 06 00 00 00 00 00
[123.473923] RECEIVE PACKET No. 29998
recv > id: 395 rtr: 0 len: 8 data: 10 27 77 ff 00 00 00 00
[123.474172] RECEIVE PACKET No. 29999
recv > id: 396 rtr: 0 len: 8 data: 10 27 01 00 00 00 00 00
[123.474414] RECEIVE PACKET No. 30000
recv > id: 397 rtr: 0 len: 8 data: 10 27 06 00 00 00 00 00
[123.503010] TEST >>>>> Test finish up
```

We where to confirm given results by special measurement tool from Peak System. This tool contains data buffer directly connected to the CAN bus. For each packed tool stores time-stamp as well. Figures 8 and 9 shows results from this tool for first and last sent packed. This results confirms previous result.
Conclusion

We have been designed application to test real time capabilities for CANbus network using RTAI based scheduling.

Results from performance measurement were confirmed by special tool from another vendor.

Comparing results from both measurement tools we have been found that our measurement method and tool are suitable to measure real time capabilities of the system.

We also have been found out that RTAI is hard real time operating system with desired accuracy of the real time task scheduling.

8 Acknowledgments

This work was supported by the Slovak Research and Development Agency under contract No. VMSP-II-0034-09. It has been supported by the project Req-00048-001 too.

References:

