The Real-Time Driver Model and First Applications

J. Kiszka
University of Hannover
Appelstrasse 9A, 30167 Hannover, Germany
kiszka@rts.uni-hannover.de

Abstract

The Real-Time Driver Model (RTDM) is an approach to unify the interfaces for developing device drivers and associated applications under real-time Linux. The device models supported by RTDM will be introduced in this paper. An overview of the low and high level APIs will be given and the concept of device profiles for defining generic device interfaces will be presented. Moreover, the integration of RTDM in Xenomai will be explained and a survey of already available applications of this concept will be provided.

1 Introduction

Since the introduction of so-called dual-kernel hard real-time Linux extensions like RTLinux [1] and RTAI [2], a large number of drivers have been developed as well. Just to name a few, there are both vendor-independent projects like Comedi [3], rtcan [4], or rt_com [5] as well as vendor-provided driver packages [7, 6].

Although many of those drivers address similar hardware, only few effort has been spent so far on unified application programming interfaces. Most drivers define their own library-like API which hinders easy replacements of hardware, as the application software has to be adapted to the driver of the new hardware.

Comedi provides a useful abstraction for data acquisition devices but is limited to a subset of control applications, excluding scenarios where direct access to low-level devices like fieldbuses is required. Also, Comedi does not include intermediate interface specifications for more complex device stacks, e.g. when USB-attached acquisition devices shall be used over a reusable real-time USB stack.

Some RTLinux/GPL drivers [8, 9] map their services on the POSIX I/O interface this variant provides. This mapping aims in principle into the right direction, but the basic POSIX I/O interface is too restrictive for message-oriented devices. Under common OSes, this type of devices is typically addressed via protocol stacks which are mapped on the well-established socket programming model. In contrast, POSIX-I/O-based devices that handle discrete messages either have to misuse the stream-oriented read/write interface or define specific IOCTLs for message transmission and reception.

Moreover, drivers are still being developed separately for the currently predominant free real-time Linux variants. Partly, this seems to happen due to personal favours of the driver maintainers. But there are also obvious technical reasons. As no common API exists, every driver would have to come with its own abstraction layer for all target variants. The required effort makes it understandable that portable drivers are very few these days.

In order to overcome this deficits technically and organisationally, the Real-Time Driver Model (RTDM) has been specified and first implemented on top of Xenomai [10], the successor of RTAI/fusion. Also several reference drivers have been ported or newly developed to demonstrate the concept and proof its feasibility. Another aim of this effort is to create a platform for research on secure hard real-time operating systems and driver [11, 12], but the latter aspect goes beyond the scope of this paper.

In the following section, the RTDM core concept is introduced. Section 3 presents the implementation for Xenomai, and Section 4 gives an overview of already realised or upcoming applications. The paper concludes with a summary and an outlook on future goals.

2 RTDM

The Real-Time Driver Model is first of all intended to act as a mediator between the application requesting a service from a certain device and the device driver offering it. Figure 1 shows its relation to other subsystem layers.

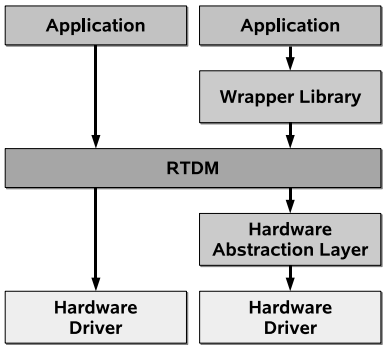

FIGURE 1: *RTDM and Related Layers*

The depicted libraries on its top and the hardware abstraction layers on the bottom are not within the scope of RTDM. Rather, they are optional indirection layers which may be added where further abstraction is desired. Libraries may be introduced to simplify the usage of the upper RTDM API for a specific device class. HALs are recommended to reuse code passages common to several drivers, e.g. a protocol stack attached on top of different low-level communication adapter drivers. Consequently, RTDM supports and encourages driver stacking, see Figure 2, but the driver developer remains free to define different inter-driver layers where appropriate. An example is the NIC adapter interface of RTnet which is derived from standard Linux in order to simplify the porting of non-real-time drivers to RTnet [13].

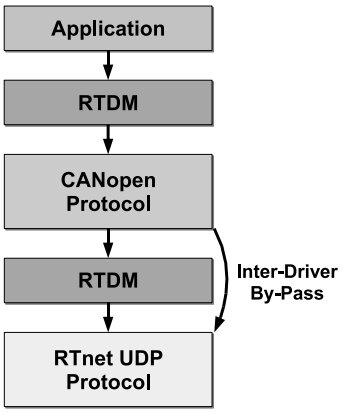

FIGURE 2: *Stacking RTDM Devices*

While RTDM's high-level API follows the POSIX socket and I/O model [14], its low-level interface is designed to provide a small RTOS abstraction layer for building portable drivers upon.

2.1 Device Models

Two different types of devices are supported by RTDM. They have been selected based on the characteristics of currently available drivers for real-time Linux.

- **Protocol Devices**
  All message-oriented devices fall in this group. Protocol devices are registered using two identifiers, the protocol family and the socket type. They are addressed according to the POSIX socket model, i.e. by invoking `socket()` for creation and `close()` for destruction. At the least, they have to provide support for sending and receiving messages: `sendmsg()` and `recvmsg()` on which `send()`/`sendto()` and accordingly `recv()`/`recvfrom()` are mapped internally. Moreover, drivers of protocol devices may handle requests issued via `ioctl()`. This interface is also used to pass the remaining socket calls `bind()`, `connect()`, `listen()`, `accept()`, `shutdown()`, `getsockopt()`, `setsockopt()`, `getsockname()`, and `getpeername()` to the driver. This mapping has been chosen to avoid creating a significant amout of entry points in the RTDM layer for only infrequently used functions.

- **Named Devices**
  These devices are registered with the real-time subsystem under a unique clear-text name and can then be instantiated using the `open()` function. RTDM does not maintain a specific naming hierarchy or file system. Basically, drivers are free in choosing device names, but regular schemes are specified for common classes in device profiles. Named devices can be subdivided into those which support stream-oriented I/O (`read()`/`write()`) and those that solely offer functionality via the `ioctl()` interface. A typical example for the former group are UART devices. The latter group catches all devices which do not match both the message and stream oriented model.

RTDM does not include a specific model for real-time block devices or even file systems. Such class of devices is currently considered minor important because most hard real-time applications can be realised with non-real-time file systems in combination with appropriately sized intermediate buffers. However, future extensions are possible in case demanding scenarios emerge.

2.2 Device Registration and Invocation

A RTDM device is registered by passing a device description to `rtdm_dev_register()`. Table 1 gives an overview of central parameters that have to be specified for this purpose (informational fields were left out).

Field Name	Description
`device_flags`	Defines the device type (named/protocol) and if the device can be instantiated only once at the same time (exclusive device) or multiple times.
`device_name`	Name to be registered (named devices only).
`protocol_family` `socket_type`	Protocol family (`PF_xxx`) and socket type (`SOCK_xxx`) to be registered (protocol devices only).
`context_size`	Size of the driver defined appendix to the structure describing a device instance (device context).
`open_rt` `open_nrt`	Handler for named device instance creation (real-time and non-real-time invocation).
`socket_rt` `socket_nrt`	Handler for Protocol socket creation (real-time and non-real-time invocation).
`ops`	Contains the operation handlers for newly opened device instances. The operations are close, ioctl, read, write, recvmsg, and sendmsg. Depending on the device type, not every handler has to be provided. Again, different handlers can be specified for real-time and non-real-time invocations.
`device_class` `device_sub_class`	Categorisation of the device, see Section 2.4.

TABLE 1: *Device Structure (digest)*

As different synchronisation mechanisms and resource allocation strategies have to be used in real-time contexts in contrast to non-real-time environments, a clear differentiation of service invocation contexts is essential. In order to let the driver decide how to handle different contexts, separate handlers can be installed for each entry type. If, e.g., the open procedure of a device shall only be executed in non-real-time context, the driver simply registers no real-time handler for open, and vice versa. In case a handler is context-agnostic, it is also possible to register the same handler for both entry points. Moreover, the RTDM implementation can provide intelligent context switching upon service invocations in case the underlying RTOS supports such a feature.

When an application or another driver creates a device instance, RTDM sets up a new context structure for this usage and redirects the call to the related driver. The context structure keeps general information about the instance like the file descriptor or a usage counter. It also allows to overwrite the operation handlers to be used with this particular instance. Attached to the general part is a private storage for the driver. The size of the storage can be specified during device registration.

Closing a device instance is sensitive to the correct context. If the instance has been created in non-real-time context, it cannot be closed within a real-time task. This is due to the fact that resource allocations (foremost memory) of both the RTDM layer and the drivers will prefer non-real-time pools when accessible to save typically more limited resources of real-time pools. The RTDM layer has to take care of this rule and catch potential violations.

Drivers are allowed to create device instances as well. An equivalence of the user API is available to them. Alternatively, drivers can also call operation handlers of other drivers directly. For this purpose, the related device context first has to be resolved via `rtdm_context_get()` after instantiating the device as normal. Then this context can be used to obtain the desired handler reference and to invoke the operation without any indirections of RTDM. The context is locked against closure and has to be explicitly released via `rtdm_context_unlock()` before closing the device.

2.3 RTOS Abstraction Layer

To increase the portability of drivers, RTDM provides a generic API of elementary RTOS services that is independent of the underlying system. The API is designed to offer only a minimum set of services typical real-time drivers require. This helps to keep the RTDM layer small and also improves its own portability over other real-time Linux variants. The following groups of services are available:

- **Clock Services**
  RTDM offers a single clock source which can be queried by `rtdm_clock_read()`. Time is expressed by 64 bit values in nanoseconds.

- **Task Services**
  This group of functions allows drivers to create their own real-time tasks, suspend the execution of user and driver tasks, or manipulate their characteristics (priority and periodicity).

- **Synchronisation Services**
  Various elementary synchronisation services are provided by RTDM. First of all, spinlocks can be used for protecting small critical paths, also when they are located in interrupt handlers or run in non-real-time contexts. Classic mutexes and semaphores for synchronising real-time tasks are available as well. As an alternative to semaphores, events can be applied. They do not count beyond 1 and can also be instructed to only wake up currently waiting tasks without storing the event. Mutexes, semaphores, and events support so-called timeout sequences which will be introduced later.

- **Interrupt Management Services**
  An essential service for most hardware drivers are interrupts. Handlers for real-time interrupt lines can be registered with RTDM, and the lines can be enabled and disabled explicitly.

- **Non-Real-Time Signalling Services**
  To propagate events from the real-time to the non-real-time domain, a special signalling service can be requested from the RTDM layer. Triggering such a signal is safe from any context. The registered handler will then be executed in non-real-time context as soon as no more time-critical tasks are pending.

- **Utility Services**
  This group of services includes real-time memory allocation, safe access to user space memory areas, real-time-safe kernel console output, and, as an alternative to separate service entry points, a check if the current context is a real-time task.

Timers are not explicitly provided. Instead, drivers are given the powerful mechanism of timeout sequences for handling service requests. Timeout sequences can be used to apply a single, continuous timeout value while calling blocking synchronisation functions multiple times. Stream devices, e.g., require to wait iteratively on incoming data while maintaining an overall timeout. The code fragment below illustrates such a service handler and the usage of timeout sequences.

```
int device_service_routine(...)
{
    rtdm_toseq_t timeout_seq;
    ...

    rtdm_toseq_init(&timeout_seq, timeout);
    ...
    while (received < requested) {
        ret = rtdm_event_timedwait(
                &data_available,
                timeout,
                &timeout_seq);
        if (ret < 0)    // including -ETIMEDOUT
            break;

        // receive some data
        ...
    }
    ...
}
```

By convention, timeouts should be passed in nanoseconds as signed 64-bit values when requesting device services. The timeout value 0 is automatically interpreted by synchronisation services as infinite delay, any negative value as no delay at all (non-blocking request).

Time-triggered events outside service requests can be executed within driver tasks. This allows to assign specific priorities to time-triggered actions instead of just executing them at the level of all timers, i.e. typically above any user task or even in interrupt context.

Detailed information on all RTDM services is available through the Xenomai API documentation which can be found on the project homepage [10].

2.4 Device Profiles

Beyond simplifying the implementation of device drivers for real-time Linux and improving their portability, another aim of RTDM is to establish device abstractions for common hardware classes. Generic device interfaces allows to develop applications that are less dependent on specific hardware and their interfaces than it is now the case with real-time Linux. Moreover, the learning curve for writing real-time applications becomes steeper with a compacter set of device interfaces.

RTDM organises device abstractions in so-called device profiles. Every profile defines for a specific device class how conforming drivers have to be realised. More in details, the following aspects are covered:

- **Device Characteristics**
  This category defines the device class and its representation in the device structure (see Table 1), the device type (i.e. the `device_flags` field), and the naming scheme for named devices, respectively the protocol family and socket type for protocol devices.

- **Supported Operations**
  All major operations that have to be implemented by conforming drivers are listed here together with the required and optional invocation contexts (real-time/non-real-time) and the meaning of specific return values. Moreover, the IOCTLs and socket options devices have to provide are described in details, i.e. their arguments, return codes, and invocation contexts.

- **Types and Constants**
  Structures and other data types typically introduced with device class specific IOCTLs and socket options have to be described in details, as well as any constants to be used in this context.

Non-common extensions to device classes are intended to be organised in subclasses. Thereby, features that are only available with certain hardware e.g. can be specified as optional to the superior class.

A significant number of device profile specifications is currently under development or is pending to be derived from existing drivers, Section 4 will give an overview.

3 RTDM in Xenomai

While the history of RTDM lasts back to the releases 0.6 (RTOS abstraction layer) and 0.7 (device models) of the real-time networking stack RTnet, Xenomai was the first real-time Linux extension to support the new RTDM revision as presented in this paper.

Xenomai allows to attach various user APIs, called skins, to its scheduler nucleus. These APIs are available to kernel space and user space real-time applications. Most skins may even be loaded in parallel, thus offering interfaces for different types of real-time applications at the same time. Most popular skins are the native skin, an improved version of the classic RTAI API, and a POSIX-conforming skin. There are also interfaces for commercial RTOSes available, e.g. VxWorks or pSOS+.

Early ideas to include RTDM in Xenomai's native skin were quickly dropped in favour of the final concept to create a separate skin. This skin can now be loaded together with the preferred main interface and is thus able to provide RTDM services independently. The RTDM user API is addressable both from kernel and user space. The function names of RTDM's POSIX interface carry a `rt_dev_` prefix in order to distinguish them from non-real-time variants. A library, `librtdm`, is used to redirect API calls from user space to the kernel services. The RTDM skin makes use of Xenomai's ability to switch user space real-time threads automatically between hard real-time (primary) and Linux (secondary) operation mode. In case no handler for a device service request is available in the current mode, RTDM switches over to the supported one. If handlers for both contexts are present, RTDM always preserves the operation mode so that the user is able to control in which context a device service shall be executed.

The integration of RTDM into the POSIX skin is realised differently. Here, the user space interface is directly included in the wrapper library `libpthread_rt`. RTDM devices can be opened under the POSIX skin by using the normal function names without any prefixes. As the name spaces of RTDM and standard Linux now overlap, the following scheme is applied when handling instantiation requests: first, the RTDM subsystem is consulted to open the device or create the socket. If this fails due to an unknown device name or protocol, the request is forwarded to the non-real-time subsystem as usual. To differentiate between file descriptors created by standard Linux on the one side and RTDM on the other, the maximum value of a non-real-time descriptor is limited to `FD_SETSIZE`-1, reduced by the typically small maximum number of real-time descriptors. Every descriptor value above this limit is handled by RTDM, everything below or equal is forwarded to Linux. Keeping real-time descriptors within the range of file descriptor sets allows future extensions of RTDM with poll/select functionalities.

4 Applications

The RTDM revision described here was first tested extensively with RTnet and a UART device driver under Xenomai. Both scenarios and the related device profiles are shortly introduced below. Meanwhile, further applications emerged that are currently being developed or planned for RTDM. On overview is given in the following as well.

4.1 RTDM Devices of RTnet

The real-time networking stack RTnet comes with three different device profiles for RTDM. First of all, there are two protocol devices, real-time UDP and the packet socket interface. Both profiles basically

conform to the interfaces available with Linux and other POSIX implementations. Additionally, several IOCTLs are defined to control RTnet-specific transmission parameters (priority and channel/slot), set a reception timeout according to the RTDM format, or manipulate the number of packet buffers per socket.

Moreover, the TDMA discipline for RTmac maintains a special device for each real-time NIC it is attached to. This device provides an IOCTL to retrieve the offset of the local clock relative to the global TDMA clock. Another IOCTL is available to synchronise the calling real-time task on the TDMA cycle of the related NIC.

Support for recent RTDM can be found in RTnet since release 0.9.0.

4.2 Serial Device

As an example for named stream devices, a serial device profile has been worked out. This profile provides access to a serial device via read/write. It also defines IOCTLs to manipulate the output status lines, retrieve the input lines, wait on device events, and configure the line characteristics, timeouts, and events of a serial device. As a specifically real-time-oriented feature, a timestamp history for every character in the input queue can optionally be maintained. Such precise timestamping is typically required when synchronising multiple sensors and actuators attached to different I/O interfaces. The code excerpt below shows the related IOCTL and data structure to obtain timestamps:

```
#define RTSER_RTIOC_WAIT_EVENT      \
    _IOR(RTIOC_TYPE_SERIAL, 0x05,   \
         struct rtser_event)

typedef struct rtser_event {
    /* signalled events */
    int     events;
    /* number of pending input characters */
    int     rx_pending;
    /* last interrupt timestamp */
    __u64   last_timestamp;
    /* reception timestamp of oldest character
       in input queue */
    __u64   rxpend_timestamp;
} rtser_event_t;
```

Xenomai comes with a reference driver for UART 16550A chips conforming to the serial profile. The latest profile specification is available with the Xenomai API documentation.

4.3 Process Image Device

In automation scenarios, distributed I/O points are typically collected via various fieldbuses or other interfaces and mapped into one or more process images. All communication to collect and update the I/O points are hidden behind this abstraction. The process image is updated either automatically or explicitly triggered by the control application.

As a first approach to map this model in a generic way on RTDM, the process image device profile has been developed. It consists of a named RTDM device which solely supports two elementary IOCTLs. One is used to update the process image. The image is split into an input and an output memory block that can further be limited to a continuous block within the full process image, see following code excerpt:

```
#define RTPI_RTIOC_UPDATE           \
    _IOWR(RTIOC_TYPE_PROCIMG, 0x01, \
          struct rtprocimg)

struct rtprocimg {
    off_t       procdata_out_offs;
    size_t      procdata_out_size;
    void        *procdata_out_buf;

    off_t       procdata_in_offs;
    size_t      procdata_in_size;
    void        *procdata_in_buf;
};
```

The update may happen synchronously or asynchronously, depending on the device configuration and hardware abilities. Moreover, several instances referring to the same physical or logical process image device can be created. In this case, one instance has to be elected to become the update master which ultimately decides about the update time. The slaves have to provide their output changes before the master starts a new cycle. This scheme allows to distribute the processing across several threads or programs while still keeping the image consistent.

The second IOCTL provides an interface to configure the device, but it only defines mechanisms to pass and retrieve unspecified configuration data and to set the update timeout. The precise configuration data format depends on the subclass and is highly hardware-dependent.

To prove the profile's applicability, a driver for a Hilscher InterBus master adapter has already been implemented [15] and required configuration data has been specified for this subclass.

4.4 CAN Protocol Stack

With the aim to develop a compact socket-based programming model for the CAN fieldbus, a driver for a SJA1000 CAN extension card is currently being implemented at our institute. This effort is tightly coordinated with parallel work on a CAN framework

for standard Linux. The goal is to have a common user interface for both approaches, while the implementation details will differ: the Linux variant will re-use the networking subsystem, the RTDM version has to implement a hard real-time capable infrastructure. The second revision of the device profile was under development at the time of writing in order to reflect practical experiences gained through the ongoing implementation efforts and to include beneficial ideas of a similar approach [16].

With a first concept for mapping CANopen on the socket model [17] being available now, the development of a RTDM CANopen profile is in reach as well. Drivers implementing it may then either attach to CAN devices or even to other communication interfaces like RTnet. In best case, the application will not notice significant differences between such stacks.

4.5 Tiny Messaging Service (TiMS)

As a revision and update of the real-time communication framework [18] used for mobile robotics at our institute, the Tiny Messaging Service has been defined and implemented for RTDM. While a detailed description would exceed the scope of this paper, a short overview of improvements compared to the original concept is given in the following.

The basic idea of TiMS remains to exchange messages between real-time components both locally and remotely in a transparent but efficient and deterministic way. Addressing is based on 32 bit message IDs that are unique across the whole system. Unlike the old concept which came with a set of special library functions, TiMS provides its services as a RTDM protocol device, thus based on the socket model. A message receiver opens a `PF_TIMS` raw socket and binds it to a specific ID. The sender opens a socket as well and passes the raw message header followed by the payload to TiMS e.g. via `send()`. In case a destination ID is not locally available, routing is currently still based on static information for real-time links and dynamic lists for non-real-time traffic. By making use of RTnet for hard real-time communication between distributed users, TiMS is also an example for stacked RTDM drivers.

TiMS is part of the new Robotics Application Construction Kit (RACK) that has been developed over Xenomai and is currently in an intensive test phase. The RACK core is going to be released under Open Source licenses afterwards. The goal is to provide a mature framework for distributed real-time computing in user space both for academic and industrial use cases.

4.6 Planned Profiles

Beyond the work listed above, at least two further projects plan to make use of RTDM soon. The RT-FireWire project [19], a hard real-time implementation of an IEEE 1394 stack, already ported its core over the RTOS abstraction layer of RTDM. Future plans include to export a raw FireWire interface as a RTDM device.

Also the USB4RT project [20] which realises a hard real-time USB stack aims at a full RTDM support for its next releases. Again, the goal is to provide the lower USB interface, on which high level drivers for I/O devices, cameras, joysticks, etc. can be built, via an RTDM device.

Beyond these plans, mapping Comedi on RTDM is considered feasible as well. Comedi already uses a POSIX I/O interface between its core and the user API, which would be mappable directly on named RTDM devices. Porting Comedi's real-time core over the RTDM RTOS layer would furthermore simplify the maintenance effort for this project regarding supported real-time platforms.

5 Summary and Future Development

This paper presented the Real-Time Driver Model. It aims at unifying the interfaces against which real-time device drivers and the applications using them can be developed. RTDM supports the two models of named devices for stream and miscellaneous use cases on the one side and protocol devices for message-oriented use cases on the other. An abstraction layer for RTOS services, specifically addressing driver development, allows portable device drivers for any Linux platform that implements RTDM. Device profiles are being defined in order to create generic interfaces for common classes of devices, thus making applications less dependent on specific hardware and simplifying its replacement.

RTDM is fully implemented under Xenomai and serves as the reference model for driver development in this environment. At the time of writing, an effort to port it over the RTAI development branch (upcoming release 3.3) has been started as well. Implementing RTDM on other real-time Linux variants is considered to be feasible, too. As soon as the currently emerging effort around the PREEMT_RT patch [21] to add native real-time support directly to the kernel matured and became common, a compatibility layer for RTDM is also imaginable and would increase the usability of existing hard real-time drivers even more.

Future work on RTDM itself will deal with interfaces and mechanisms to provide poll/select semantics also under hard real-time constraints. The design of such services is challenging because dynamic allocation of data structures for managing file descriptor groups that are monitored by poll/select has to be avoided. Moreover, a common signalling mechanism has to be defined so that all combinations of RTDM device types will be usable. Poll/select support will likely introduce a certain overhead with respect to code complexity and data structure sizes. Therefore, it is planned to specify it as a configuration option both for the RTDM core as well as conforming drivers. As many real-time applications work fine without these semantics, they should not suffer from overhead in the future without gaining advantages.

References

[1] M. Barabanov, 1997, *A Linux-based Real-Time Operating System*, Master's thesis, New Mexico Institute of Mining and Technology.

[2] P. Mantegazza, E. Bianchi, et al., 2000, *RTAI: Real-Time Application Interface*, Linux Journal #72, www.linuxjournal.com/article/3838

[3] Comedi, *Control and Measurement Interface*, www.comedi.org

[4] rtcan, *Realtime CAN for Linux/RTAI*, www.esfnet.co.uk/index.php?page=rtcan

[5] rt_com, *Real-Time Linux driver for the serial port*, rt-com.sourceforge.net

[6] Hilscher GmbH, 2004, *RTL CIF Device Driver V2.000*, System Software CD.

[7] ADDI-DATA GmbH, *Company Homepage*, driver download section, www.addidata.com

[8] S. Pez, J. Vila, I. Ripoll, 2003, *Building Ethernet Drivers on RTLinux-GPL*, 5th Real-Time Linux Workshop, Valencia (Spain).

[9] OCERA, *Linux CAN Driver (LinCAN)*, www.ocera.org/download/components/WP7

[10] Xenomai, *Project Homepage*, www.xenomai.org

[11] J. Kiszka, B. Wagner, 2003, *Domain and Type Enforcement for Real-Time Operating Systems*, 9th IEEE International Conference on Emerging Technologies and Factory Automation, Lisbon (Portugal).

[12] J. Kiszka, B. Wagner, 2004, *Securing Software-Based Hard Real-Time Ethernet*, 2nd IEEE International Conference on Industrial Informatics, Berlin (Germany).

[13] J. Kiszka, B. Wagner, Y. Zhang, J. Broenink, 2005, *RTnet A Flexible Hard Real-Time Networking Framework*, 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania (Italy).

[14] IEEE, 2004, *IEEE Std 1003.1*, 2004 Edition.

[15] rt_cifibm, 2005, *CIF InterBus Master Driver*, www.rts.uni-hannover.de/mitarbeiter/kiszka/rtaddon

[16] I. Bertolotti, G. Cena, A. Valenzano, 2005, *A Socket-based Interface to CAN*, 10th international CAN Conference, Rome (Italy).

[17] G. Cena, I. Bertolotti, A. Valenzano, 2005, *Modelling CANopen Communications According to the Socket Paradigm*, 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania (Italy).

[18] O. Wulf, J. Kiszka, B. Wagner, 2003, *A Compact Software Framework for Distributed Real-Time Computing*, 5th Real-Time Linux Workshop, Valencia, Spain.

[19] Y. Zhang, B. Orlic, P. Visser, J. Broenink, 2005, *Hard Real-Time Networking on FireWire using Linux/RTAI*, 7th Real-Time Linux Workshop, Lille (France).

[20] USB4RT, *USB for Real-Time*, developer.berlios.de/projects/usb4rt

[21] PREEMPT_RT, *Realtime Preemption Patch*, redhat.com/~mingo/realtime-preempt