Implementation of a CAN Controller and Monitor Application on the Rapid Prototyping Platform REAR*

Franz Fischer Thomas Hopfner Thomas Kolloch
Annette Muth Stefan Petters
Stefan Rudolph Georg Färber

Laboratory for Process Control and Real–Time Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München
D–80290 München, Germany
Phone: +49–89–2 89–2 35 50, Fax: +49–89–2 89–2 35 55
firstname.lastname@lpr.e-technik.tu-muenchen.de

Abstract. Rapid Prototyping is used in embedded systems design as a means to reduce development time and costs. At an early stage in the development cycle, the specification is implemented in a working protoype, which can be used to test the specification and, in real-time systems, also the timing constraints. The REAR Rapid Prototyping Environment was built as an adaptable target platform for embedded real-time systems. It supports both the proof that the system meets all its deadlines, and the automated translation of a system specification into an executable prototype. This paper presents a CAN controller and monitor application, which was implemented and evaluated on REAR as a first non–trivial real–world application with a wide range of timing and coordination requirements towards the target architecture.
Keywords: rapid prototyping, real–time, CAN, hardware software codesign, PCI, FPGA, Statemate

1 Introduction

Embedded hard real–time control systems show growing functional complexity as well as increasing demand for short response times and high computing performance. REAR was built as a target system architecture suitable for implementing a working prototype of such a system at an early stage of development. Using a HW/SW–codesign methodology, the rapid prototyping design process starts with a implementation independent specification of the embedded system, followed by a classification and partitioning step. Including a schedulability proof, the design flow closes with HW/SW code generation. In order to test REAR, a CAN controller and monitor application — an example of a complex application with high real–time demands — was implemented on the REAR rapid prototyping environment.

Our target architecture **REAR** (Rapid Prototyping Environment for Advanced Real-Time Systems) was built based on the multiprocessor architecture framework presented in [4]. In this approach, real–time systems are analyzed and partitioned according to a task classification model. Each class of tasks corresponds to a type of processor best suited in terms of performance and deterministic execution times. The resulting target architecture framework is a tightly coupled heterogenous multiprocessor system consisting of the following processing units:

High Performance Units (HPUs) are based on standard computer architectures to benefit from the technological advances regarding processing performance. In the actual configuration the HPU is a PCI slot CPU with Intel pentium processor, large L2–cache and main memory. The impact of interrupts and context switches on predictability is limited by software means. **Real–Time Units (RTUs)** are optimized for small tasks with short response times. They use a limited amount of high speed memory to enhance predictability. The RTU was built using a MIPS R4600 based single board computer with PCI interface. **Special Purpose Units (SPUs)** are based on processing elements optimized for special classes of tasks. Examples include DSP–based SPUs for digital signal and image processing algorithms or FPGA–based units for processing fast input and output tasks. Currently, REAR possesses one SPU: A Configurable I/O Processor (CIOP), consisting of one Xilinx FPGA and additional dual ported RAM. It serves two dedicated functions: It acts as a separate application specific processing unit for tasks with deadlines too short to be met in software and it provides a flexible way of linking the prototyping architecture to the embedding process.

* This work was supported with funds of the *Deutsche Forschungsgemeinschaft* under reference number Fa 109/11-1 within the priority program "Rapid Prototyping for Embedded Hard Real–Time Systems."

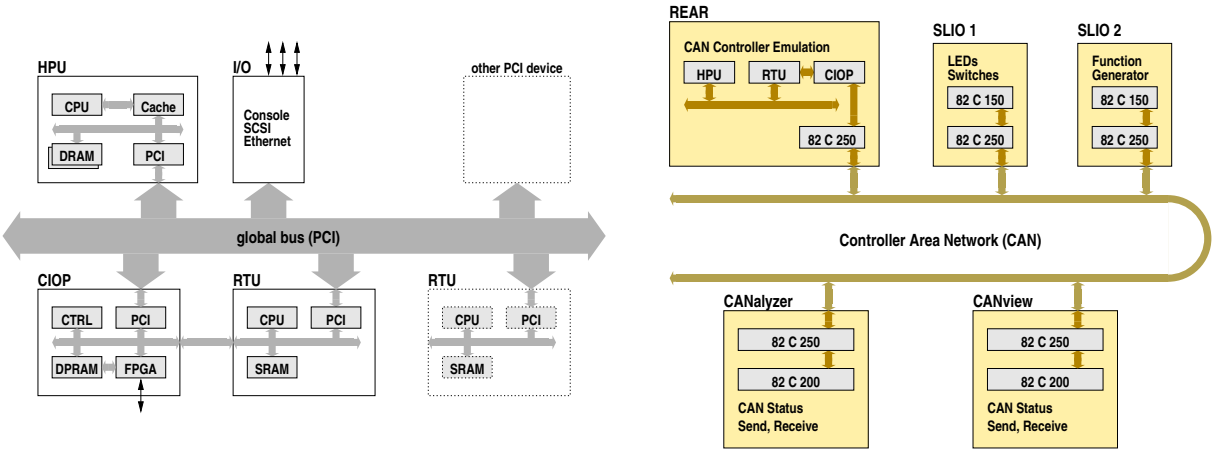

Fig. 1. (a) REAR hardware architecture and (b) CAN Bus Layout

The nodes are tightly coupled by a global PCI–bus, which offers high throughput and low latency. Figure 1(a) and [2] give an overview of the target system architecture, which is mostly built from off–the–shelf components.

CAN [1] is a serial field bus which was developed for use in automobiles. The CAN bus runs a multi-master, message oriented bus protocol in CSMA/CA (Carrier-Sense Multiple Access/Collision Avoidance) mode. Transmitting prioritized messages to all nodes, the bitwise arbitration is done, using the individual message IDs. Based on the message identifiers, the receivers decide, whether they should process the message or not. Several cooperating error detection mechanisms guarantee fast system wide error detection and recovery. CSMA/CA bus access, in combination with message priorities, the short data block length (max. 8 Byte) and data rates up to 1 Mbit/s lead to very short message latencies, thus posing challenging demands on the response times of the system, controlling and serving the CAN bus.

The following section describes the CAN controller and monitor application developed for testing the REAR rapid prototyping environment. Section 3 provides details of the implementation. Results and experiences are presented in Section 4, followed by conclusions in Section 5.

2 CAN Controller and Monitor Application

2.1 Application Environment

The CAN bus environment as depicted in Figure 1(b) was built as a test bed for the application to be implemented on REAR. Typically, a CAN bus application consists of several sensor and actor units, which are connected to one or more control units via CAN. The cheapest and therefore often used components to connect sensors and actors to a CAN bus are Serial Linked I/O devices (SLIOs). SLIOs do not contain the expensive high quality oscillators necessary to synchronize themselves to the bus frequency. Instead, they need to be continuously synchronized via regular synchronization messages by at least one "intelligent" bus node. These synchronization messages have to be received every 3800 to 8000 bit times, i.e. 30 ms to 64 ms at a bus frequency of 125 kHz, the highest one the SLIOs allow.

For our application environment two exemplary SLIO–based CAN participants with digital and analog inputs and outputs were built. Both cards are based on the Philips 82C150 SLIO device. As a working CAN bus needs at least two fully functional CAN devices (one sending messages and the other one responding with acknowledgments), two commercial PC based CAN participants with monitor and analyzing software were connected to the bus additionally. Details on the realized CAN environment can be found in [8].

2.2 Application Functions, Classification and Partitioning

Controller and monitor functions From the user's point of view, the application performs two functions: The CAN bus monitor allows the user to send, receive and filter CAN messages, to monitor activity on the CAN bus and additionally to control the other parts of the application (start, stop, initialization,...).

The SLIO controller provides an interface to the SLIO cards. This includes monitoring the state of the "sensors" as well as the possibility to set new "actor signals". For both user level functions, the lower levels of the application have to route the CAN messages to and from the CAN monitor and the SLIO controller and have to fulfill all functions of a complete CAN bus participant [6].

Task Classification The individual tasks to be processed can be classified according to the task classification model presented in [4]. In this model, the attributes *deadline of the task* and *complexity of the function to be performed* are used to allocate the tasks to the best suitable type of processing unit (here: HPU, RTU and CIOP). At *message level*, the complexity of the tasks — message identification and message frame generation, CRC checksum generation, error protocol functions, data handling — is medium to high. The timing constraint here is identical with the length of one CAN message, $44-108$ μs (44 control and up to 64 data bits, at an assumed data rate of 1 Mbit/s). The controller tasks at *bit level* — transmission of the message bits, CRC checksum error detection, bit stuffing and destuffing — need to be finished in the worst case before the start of the next message bit. This results in a timing constraint of 1 μs. The complexity of these tasks is medium. Bitwise arbitration — i.e. transmission is stopped before the next message bit if a station sending a message with higher priority ID is detected on the bus — and synchronization of the sample points while receiving the message bit stream (bit timing) are tasks with timing constraints *below bit level*. The complexity of these tasks is low to very low.

Task Allocation The tasks at and below bit level, with timing constraints below 1 μs, can only be implemented in hardware, on the Configurable I/O Processor. Tasks at message level (deadlines below 1 ms) can be alternatively implemented on the RTU. In a first approach, the entire CAN controller (data link layer and physical layer) was implemented on the CIOP. The message routing functions and the interface to the CAN controller on the CIOP were implemented on the RTU, as well as the generation of the periodic SLIO calibration messages (one every 30 ms). SLIO controller and CAN monitor, which included also graphical user interfaces, were implemented on the HPU (see Figure 2(a)).

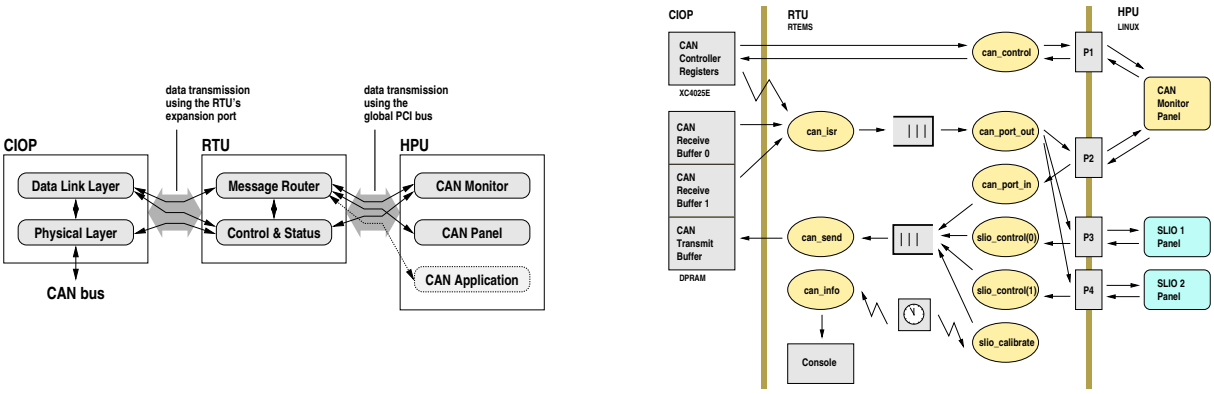

Fig. 2. (a) Task allocation and communication; (b) CAN monitor tasks

In a second version, some CAN controller functions from the message level were moved to the RTU, while maintaining the ability to run the bus at 1 MHz. With a reduction of the bus frequency, the timing constraints on the application can be scaled. This makes it possible to further explore the HW/SW–boundary.

3 Implementation on REAR

After mapping the tasks identified above on REAR's different processing units, they were implemented using a CASE tool chain for the hardware part, i.e. the tasks mapped on the CIOP, and handcoded C programs for the software parts, i.e. the tasks to be run on the RTU and the HPU. In this first approach the main goal was to gain experience in implementing and debugging a distributed application derived from a single specification on REAR.

3.1 CIOP

As mentioned above, two versions of the CAN controller were realized and tested on the CIOP. The first design includes all the basic CAN functions (transmission and reception of complete message frames, message frame handling, error handling, CRC check, acceptance check) and is similar to the 82C200 stand–alone basic CAN controller [6] regarding functionality as well as programming model. Additionally a limited design was developed to further explore the hardware/software design space. In the latter case CRC check and generation as well as message frame generation are moved to software. Both implementations are tested successfully on a CAN bus at the maximum bit rate of 1 Mbit/s.

In the automated design process for the HW–part the CAN controller was specified and simulated using Statemate. This tool also generated the VHDL–Code, which was synthesized into Xilinx netlists using Synopsis and then fitted into the target technology FPGA using Xilinx XAct.

CAN Controller Structure The CAN controller was designed in accordance with the "Basic CAN" specification. Hence, it provides one send and two receive buffers, in combination with acceptance checks for message filtering. The main functions of the implemented CAN controller are distributed in three layers.

Physical Layer The physical layer essentially fulfills the bit timing, synchronization and bit stuffing. Using the bit timing parameters and the clock dividing coefficient, the length of a bit interval and therefore also the transmission frequency of the CAN bus is defined. This layer is controlled by the data transmission layer via several control flags. The message data is transferred bitwise (serially) from the data transmission layer to the physical layer. Finally the physical layer generates three error flags which inform the data transmission layer that a synchronization, stuffing or configuration error has occurred.

Data Transmission Layer The modules of the data transmission layer are the central control unit, calculation of the position in the message frame, error handling and CRC check. The central control unit switches the state according to each of the transmission conditions - i.e. send, receive, idle and error. It performs the arbitration, the format conversion, i.e. the handling of frame bits and the CRC sequence in the bit stream, and controls the physical and the communication layer. As the data transmission layer filters all information not relevant for the host controller, the exchange of data to the upper layer is reduced to the message identification, the remote request bit, the data length code and the data.

Communication Layer This layer implements the user and the message interface. The user interface uses four registers for command and status information and for configuration data. In the message interface the data is read from the message buffers, which are located in the DPRAM, serialized, sent to the data transmission layer and vice versa. Additionally a control unit sets the message related flags accordingly and also performs acceptance filtering.

Reduced CAN Controller In the limited design of the reduced CAN controller, the data transmission layer does not insert nor remove frame bits, contains no CRC handling, indicates no errors and does not cope with the acknowledgment of messages. Frame bit and CRC handling were transferred to software, the rest of the missing functions are neglected by now.

3.2 RTU

The tasks to be implemented on the RTU were message routing, programming the CAN controller on the CIOP, and sending the SLIO calibration messages. The RTU runs the real-time operating system RTEMS. The tasks were hand coded in C as threads running on top of RTEMS, communicating via message queues. The task structure implemented on the RTU is shown in Figure 2(b).

The CAN controller signals the arrival of a new message with an interrupt, which is caught by the interrupt service routine *can_isr*. The *can_isr* reads the message from the DPRAM, frees the corresponding message buffer and writes the message to the received messages queue. The task *can_port_out* receives the message at this message queue and passes it to the ports P2, P3 and P4, using the IPC–functions described in 3.4. Messages to be sent over the CAN bus are buffered in a second queue, placed there by several tasks: The task *slio_calibrate*, which is periodically activated by the RTOS, sends the SLIO calibration messages. The tasks *can_port_in* and *slio_control{0,1}* receive their messages to be sent at ports P2 resp. P3 and P4 of the IPC. The task *can_control* starts, configures and resets the CAN controller using the control registers of the FPGA, while the task *can_info* regularly outputs information from the CAN controller's status registers on the RTU's console.

3.3 HPU

The CAN monitor application and the SLIO control panels are implemented on the HPU. As there are no hard real–time requirements to be met for displaying data, in this case no real–time operating system is necessary and Linux was used as operating system. Both applications read and write messages from and to the already mentioned IPC ports. In the *CAN monitor* GUI (Fig. 3), the entire CAN application can be initialized, started and stopped. All received CAN messages can be displayed or filtered, if the monitor is configured correspondingly, and CAN messages can be sent. The *SLIO control panels* display and control the contents of all SLIO registers, as well as the state of the switches and LEDs.

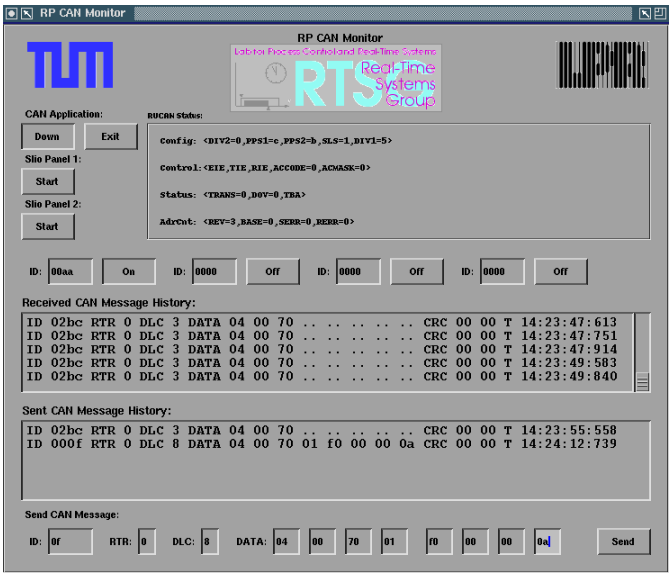

Fig. 3. CAN Monitor GUI

3.4 Interface and Communication

A uniform IPC layer providing support for inter and intra unit communication is currently being developed [3] in order to simplify partitioning and distribution of application threads to different processing units.

The basic idea for the implementation of inter unit message queues is to use a shared memory area for a message buffer pool and the send and receive queues. In order to send a message, the application task allocates a message buffer, prepares the message and enqueues the message buffer in the receivers receive queue (Fig. 5(a)). The receiving task in turn then processes the message and afterwards deallocates the buffer, which then is available for allocation again. This communication scheme is based on the following properties of the processing units (PU) and the global bus system of the REAR target architecture:

- Most units can be master on the global bus in addition to their function as slaves (targets).
- On the global bus all units share a common physical address space.
- At least a portion of a processing unit's memory is accessible to other bus masters.
- A PU can generate interrupts on a remote unit by accessing predefined addresses on the global bus.
- If the global bus or some units on it do not support an atomic "test–and–set" operation (which is usually the case), at least one unit[1] should provide an efficient spinlock or semaphore mechanism to avoid excessive synchronization effort when accessing shared communication data structures.

In this example, inter node communication between RTEMS threads and Linux processes on the HPU was based on a simple IPC module as presented below. For local IPC on the RTU RTEMS message queues were used, while communication with the CIOP threads was realized by reading and writing registers and regions of the dual ported RAM as described in the following subsection.

Communicating with the CIOP For this application the CIOP's interface to the RTU's 32 bit wide local bus was used (Fig. 1(a)). It allows word access to a maximum of 32 registers implemented within the FPGA and to one side of the dual ported RAM, organized as 8K × 32bit. The other side is accessible by the FPGA as 16K × 16bit.

The CAN controller's communication layer implements four registers for configuration, control and status information. To receive a message from the CAN bus, the software side has to wait for one of the *receive message flags* to be set by the controller either by polling or by use of interrupts. Afterwards the message can be copied from the according receive buffer (within the dual ported RAM), which is freed by setting the respective *release buffer flag*. When sending a message, the completion of a previous transmission has to be awaited before the *transmit request flag* can be reset. After the controller has granted access to the transmit buffer, the new message can be copied to the transmit buffer and the transmission request can be set.

[1] on the Real–Time Unit a CPLD implements 8 hardware spinlocks with a single read access being equivalent to the "test–and–set" and a write access clearing (freeing) the spinlock again

Communication between the HPU and the RTU For communication between RTEMS threads on the RTU and Linux processes on the HPU, a simple module for message based IPC provides services to establish a communication *port* and to send messages to and receive messages from the port.

The port encapsulates information concerning the sender and receiver threads or processes and pointers to a pair of FIFO queues for the messages. The queues are located within the RTU's DRAM, which is accessible by the HPU via PCI bus. A queue consists of the *in* and *out* indices and a configurable fixed number of message buffers.

```
int port_sndmsg(port_t *p, msg_t *m)
{
    while (!queue_putmsg(p->sq, m))
        PORT_WAIT(p, NOT_FULL);
    PORT_SIGNAL(p, NOT_EMPTY);
    return(SUCCESS);
}
```

```
int port_rcvmsg(port_t *p, msg_t *m)
{
    while (!queue_getmsg(p->rq, m))
        PORT_WAIT(p, NOT_EMPTY);
    PORT_SIGNAL(p, NOT_FULL);
    return(SUCCESS);
}
```

Fig. 4. Simple port send and receive functions

The basic algorithm for receiving messages is as follows: `queue_getmsg()` checks whether a message is available and if so, copies the message from the queue's to the thread's message buffer and returns `TRUE`. A receiving thread will wait for a message to arrive and then notify the remote side of the receive queue being not full by triggering an interrupt. `queue_putmsg()` performs the same function vice versa. On the RTU, the queue ISR handles the notifications from the HPU side and propagates them as RTEMS *events* to any waiting thread, which in turn is unblocked and re–checks its receive or send queue. On the HPU, access to the queue memory area as well as triggering and handling notification interrupts is supported by a UNIX device driver. Results from a first performance evaluation are discussed in Section 4.2.

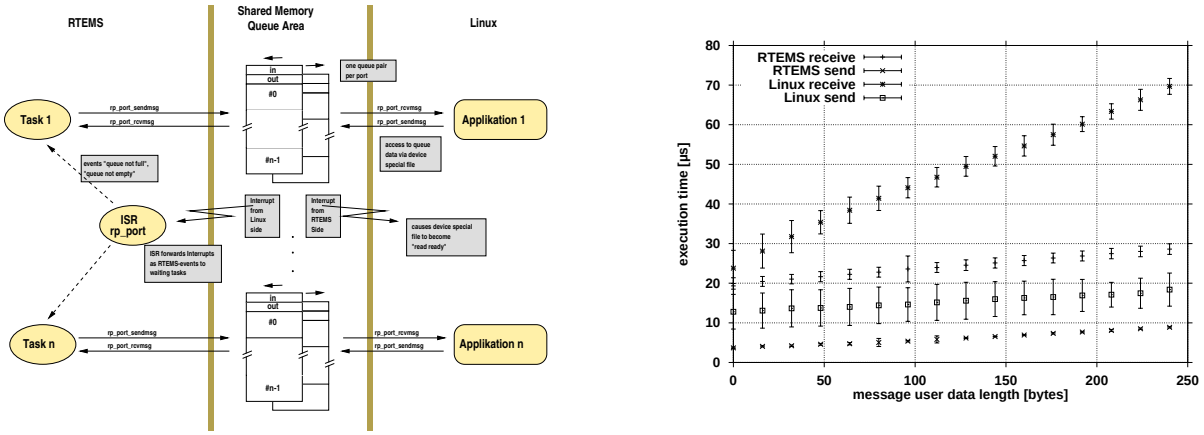

Fig. 5. (a) simple message queue implementation and (b) performance of implementation

4 Results

4.1 Resource Usage

To give an impression of the complexity of the CAN application, this section describes the characteristics of the CIOP utilization and code and data sizes of the RTU software implementation.

As mentioned above, two different solutions of the HW part of the CAN application were implemented. The smaller one, called "RULIM" (because of its limited functionality) reached a FPGA chip utilization of about 61% (631 of 1024 maximum available XC4025E CLBs). It was possible to implement "RULIM"

using the smaller XC4013E FPGA chip, with a maximum capacity of only 576 Configurable Logic Blocks (CLBs).

The fully featured CAN controller implementation (called "RUCAN") reached an overall chip utilization of about 88% of the CLB resources (909 of 1024 available CLBs), although only 479 of the 2048 available CLB flip flops (23%) were used. The reason for this is the coarse structure of the can controller system model. Implementing the physical, data link and communication layer in only three main activities, Statemate is forced to synthesize three main VHDL processes. These huge state machines are resulting in very complex state transition conditions, which were mapped to the function generators in the CLBs. A further drawback of this solution is, that most of the CLB resources were needed to route the connections between the three main VHDL entities.

These numbers indicate, that a more fine granular system structure (activity chart structure in Statemate) would be easier to place and route and therefore would free some chip resources for further application features. Further overhead is caused by the inefficient tri–state register implementation. A hand optimization of often reused model components would be worth while, because these system components could be reinstantiated in future designs. Nevertheless, comparing the Xilinx XC4000 FPGA series with the newer XC4000E, the chip utilization reached is near to the optimum — achievable with an automated translation of a system specification to a FPGA configuration file.

The current CAN controller implementation uses only 3×16 Bytes of the DPRAM for the message buffers. However, the DPRAM could be used to store additional CAN bus debugging information, or to implement more elaborate, table based message filtering. The application threads on the RTU compiled to 14 KByte program code (text segment) and less than 4 KByte initialized and uninitialized data (data and bss segments). Linked with the necessary RTEMS modules (52 KByte text) and the C library (43 KByte)[2], the executable file included 109 KByte text, 13 + 11 KByte data and used 224 KByte RTEMS workspace during execution for RTEMS objects, thread stacks (8 KByte each) and the heap (64 KByte).

Taking into account the size of the RTU's SRAM (512 KByte), this means that already this hand coded, medium size application almost fills the available fast memory. The planned automated code generation usually results in even larger code and data sizes. Consequently, a resource optimization of RTEMS has to be considered and/or the size of the SRAM has to be increased.

4.2 IPC performance

For a first evaluation of IPC performance, the port functions described above were instrumented to write time stamps to a memory buffer.[3] The timing test application included one Linux process to send a message to a RTEMS thread, which after being unblocked and receiving the message, sent the unmodified message back to the now blocked Linux process. I.e. the receive operations on both sides were blocking and involved processing an interrupt and a context switch, while the send could be performed without the sender being blocked. Figure 5(b) shows the measured (average) execution times of the `port_sndmsg()` and `port_rcvmsg()` calls on RTEMS and Linux, respectively.

The graph indicates clearly that the nonblocking send call on RTEMS includes nearly no overhead except for the copy operation, while the send on the Linux side involves even in the nonblocking case one *ioctl* system call. The system call takes approximately 16 μs. Due to interrupt and context switch overhead the blocking receive functions take much more time than the nonblocking sends. The time from the notification interrupt to the end of the receive call involves in both cases interrupt processing, unblocking the waiting thread (by `rtems_event_send()`) or process (`wake_up_interruptible()` within the rapid driver) and the context switch to the receiving thread or process.

The next implementation will take into account these results. E.g. one system call could be saved in the receive operation on Linux by moving IPC functionality into the driver. This is also necessary for mutual exclusion if more than one process is allowed to access a port.

---

[2] this included functions like `printf()`, which were used only for debugging purposes

[3] The time stamps were taken from the RTU's MIPS Orion Processor (R4600), which includes a counter register incrementing at half the pipeline clock frequency (50 MHz in our case); the overhead of writing the time stamp to DRAM is below 0.2 μs (10 system clock cycles).

5 Conclusions

With the CAN controller and monitor application it was shown that it is feasible to implement, in a short span of time, a working prototype of a complex real–time application on the target architecture REAR. The implemented system met all the requirements posed by the planned CAN controller and monitor application. Of particular interest in the context of real–time systems are guaranteeable response times: The CAN application on REAR met all the deadlines of the CAN protocol and was therefore able to communicate with the commercial CAN bus participants at bus frequencies up to the CAN bus maximum of 1 MHz without message loss. It was possible to integrate the SLIO components and to keep them active by periodic calibration messages at the SLIO maximum frequency of 125 kHz. The RTU running RTEMS proved its fitness to guarantee deterministic sending intervals of these messages, in this case exactly 30 ms.

Future work will concentrate on the one hand on improving the actual design, to further explore the characteristics and limitations of the different hardware/software layers, e.g. adding more message buffers, using a DPRAM table for message filtering, providing access to the physical layer and a bit time counter for more detailed monitoring functions.

On the other hand it proved to be a time consuming task to model the CAN controller's registers as well as to code the software connecting the higher level application tasks to that programming interface. This overhead results from the completely separated development of hardware and software and will be inherently avoided in the planned rapid prototyping development process, which starts from one specification for the entire system. The fundamental prerequisite for this to work will be the availability of efficient reusable IPC components realizing also the communication between tasks implemented in hardware and in software.

Acknowledgements

The authors want to thank the graduates and students Christian Mühlbauer [5], Robert Pinzinger [7], Gunnar Larisch and Andreas Michael for their great personal effort. We greatly appreciate the support of *Softing GmbH*, München, in contributing the CANalyzer soft– and hardware.

References

1. Konrad Etschberger et al. *CAN Controller–Area–Network, Grundlagen, Protokolle, Bausteine, Anwendungen.* Hanser Verlag, 1994.
2. Franz Fischer, Thomas Kolloch, Annette Muth, and Georg Färber. A configurable target architecture for rapid prototyping high performance control systems. In Hamid R. Arabnia et al., editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.
3. Franz Fischer, Annette Muth, and Georg Färber. Towards interprocess communication and interface synthesis for a heterogeneous real–time rapid prototyping environment. In *Proceedings of the 6th International Workshop on Hardware/Software Co–Design — Codes/CASHE '98*, pages 35–39, Seattle, Washington, USA, 15–18 March 1998. IEEE, IEEE Computer Society Press.
4. Georg Färber, Franz Fischer, Thomas Kolloch, and Annette Muth. Improving processor utilization with a task classification model based application specific hard real–time architecture. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, October 27–29 1997.
5. Christian Mühlbauer. Konzeption und Implementierung einer Schnittstellenkarte mit programmierbaren Logikbausteinen zur Erweiterung einer Rapid–Prototyping Plattform, 1996. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.
6. Philips Semiconductors, Eindhoven, The Netherlands. *PCA82C200, Stand–alone CAN Controller, Product Specification*, 1992.
7. Robert Pinzinger. Speichersubsystem und flexible Prozeßanbindung für einen Rechnerknoten eines Rapid–Prototyping–Systems, 1997. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.
8. Stefan Rudolph. Modellierung und Realisierung eines CAN–Bus Controllers als Testszenario der Rapid Prototyping Umgebung *REAR*, 1997. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.