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

Technical Report

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

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. This application represented a wide range of timing and coordination requirements towards the target architecture. The fact that it was possible to implement it successfully in reasonably short time on REAR is a proof of the soundness of the concept behind the REAR rapid prototyping architecture.

Keywords: rapid prototyping, real–time, CAN, hardware software codesign, PCI, FPGA, Statemate

[*] 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."

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. In order to exercise and test REAR, a CAN controller and monitor application — an example of a complex application with high real–time demands — was developed and implemented on the REAR rapid prototyping environment.

Our target architecture **REAR** (Rapid Prototyping Environment for Advanced Real-Time Systems) was built after the multiprocessor architecture framework presented in [3]. 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. The impact of interrupts and context switches on predictability is limited by software means.

In the actual configuration the HPU is a PCI slot CPU with Intel pentium processor, large L2–cache and main memory, satisfying the very high demands for comput-

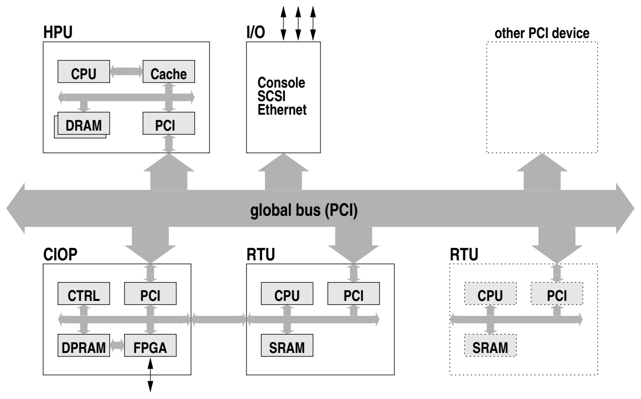


Figure 1. REAR hardware architecture

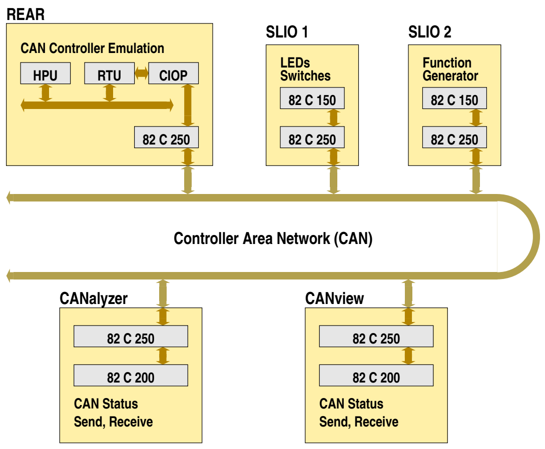


Figure 2. CAN Bus Layout

ing performance and memory requirements.

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. To narrow the memory bandwidth gap between the CPU and the DRAM, a fast SRAM module was added to the processor board.

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.

The nodes are tightly coupled by a global PCI–bus, which offers high throughput and low latency. Figure 1 gives an overview of the target system architecture, which is mostly built from off–the–shelf components. A more detailed overview of REAR is given in [2].

CAN [1] is a serial field bus which was originally developed for use in automobiles, but is increasingly being used in industrial as well as building automation. It is standardized in the ISO 11898 international standard. The CAN bus runs a multi-master, message oriented bus protocol in CSMA/CA (Carrier-Sense Multiple Access/Collision Avoidance) mode. In CAN networks there is no addressing of nodes in the conventional sense, but instead, prioritized messages are transmitted, i.e. bus access is granted to each participant by bitwise arbitration using the individual message IDs. A transmitter sends a message to all nodes on the bus and each decides on the basis of the identifier received, whether they should process the message or not. Several cooperating error detection mechanisms guarantee fast system wide error detection and error 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. At the same time, this poses very challeging demands on the response times of the systems controlling and serving the CAN bus.

The following section describes the CAN controller and monitor application we developed for testing the REAR rapid prototyping environment. Section 3 gives details of the realized implementation on REAR. The results and experiences gained are presented in Section 4, followed by conclusions and an outlook on future work in Section 5.

2. CAN Controller and Monitor Application

2.1. Application Environment

The CAN bus environment as depicted in Figure 2 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 were built. The first SLIO–card contains LEDs and switches to simulate simple digital sensor/actor functions. In order to emulate more complex sensor functions, the second SLIO–card contains a function generator which is programmable via CAN. 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. Both use the Philips 82C200 stand–alone CAN controller.

Using the SLIOs we were able to clock the bus at the maximum frequency for SLIOs of 125 kHz. Without them it was possible to run the CAN bus at the maximum CAN bus frequency of 1 MHz. Details on the realized CAN environment can be found in [7].

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" (state of the switches and current function values of the function generator) as well as the possibility to set new "actor signals" (switching on and off the LEDs and controlling the function generator).

For both user level functions, the lower levels of the application have to provide the distribution of CAN messages to and from the CAN monitor and the SLIO controller (message routing) and all functions of a fully functional CAN bus participant [5].

Task Classification The individual tasks to be performed can be classified according to the task classification model presented in [3]. 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). For the CAN bus participation (from now on called CAN controller function) and the message routing this classification is shown in Table 1.

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, $44 - 108\,\mu s$ (44 control and up to 64 data bits, at an assumed data rate of 1 Mbit/s).

Table 1. CAN controller functions

Function	Deadline	Complexity
Message Level:		
Message Identification	44–108 μs	medium
Message Routing	44–108 μs	medium
Message Frame Generation	44–108 μs	medium
CRC Checksum Generation	44–108 μs	high
Error Logic	44–108 μs	high
Data Handling	44–108 μs	medium
Bit Level:		
Message Transmission	1 μs	medium
CRC Error Detection	1–3 μs	medium
Bit Stuffing and Destuffing	1 μs	medium
Below Bit Level:		
Bit Timing	270 ns	low
Bitwise Arbitration	60 ns	very low

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. That 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.

The CAN monitor and SLIO control functions are not mentioned explicitly in Table 1. Their deadlines correspond to the deadlines at message level or higher, their complexity is medium to high.

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 hardest time constraints on the CAN controller are imposed by the bit timing and bitwise arbitration tasks, with deadlines of 60 ns and 270 ns respectively. The execution times of the arbitration mechanism and the bit synchronization were found to be 1 and 2 clock cycles, respectively. On a FPGA board clocked with 25 MHz the execution times then amount to 40 ns and 80 ns. It was therefore certain that the deadlines of these two functions would be met on the CIOP.

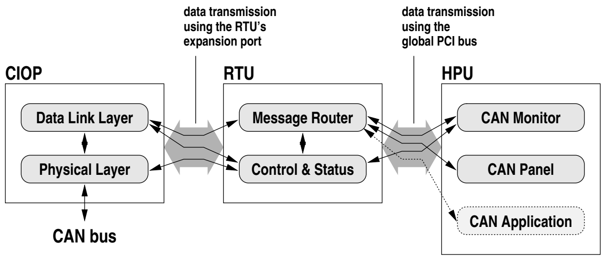


Figure 3. Task allocation and communication of the CAN application

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 3).

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 [5] 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.

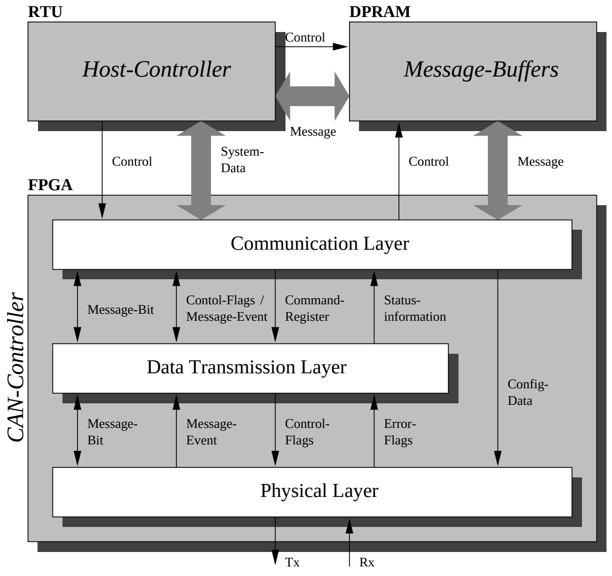


Figure 4. CAN controller

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.

Connection to the Physical Bus The CIOP is connected to the physical CAN bus using the configurable I/O Pins of the FPGA. A Philips 82C250 transceiver converts the logic signals of the implemented CAN controller to the bus level required by the CAN high speed specification and vice versa. This interface device is also used in commercial designs which contain the 82C200 CAN controller.

Structure of the full CAN Controller The present 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 communication is restricted to the standard CAN 1.0 format.

The main functions of the implemented CAN controller are distributed in three layers. The lowest layer, the *physical layer*, performs tasks on the bit level. The *data transmission layer* is responsible for tasks concerning the message frame. Finally the *communication layer* (CL) organizes the data and information exchange with the host controller, which is implemented in software on the RTU.

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 bit timing function defines the length of a bit interval, i.e. also the transmission frequency of the CAN bus, and keeps track of several significant timing events during the bit interval. These include the sampling point and the start of a new bit interval, where the transmission of a new bit can be started or the CAN controller synchronizes itself to the start of a new message frame (hard synchronization). The soft synchronization adjusts the sampling time by modifying the length of the phase buffer segments according to the detected shift of a signal edge on the bus. The bit stuffing ensures that signal edges appear after a maximum number of equal signal levels on the bus by inserting stuff bits in the send bit sequence. Analogously these stuff bits are deleted from the received bit sequence by this function.

The physical layer is controlled by the data transmission layer via several control flags: The stuffing mechanism can be turned on and off, the start of an idle phase indicated (important for the hard synchronization) and the transmission of a message bit can be enabled. The message data is transferred bit wise (serially) from the data transmission layer to the physical layer. When a message is received, the physical layer notifies the data transmission layer of the time to sample the message bit. 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 Submodules of the data transmission layer are a central control unit, a unit for the calculation of the position in the message frame, a module for error handling and a CRC-unit. The central control unit assigns a specific response program to each of the transmission conditions - i.e. sending, receiving, idle and the sending of an error message. The following functions are performed:

- arbitration,
- format conversion, i.e. the insertion respectively the deletion of frame bits and the CRC sequence in the bit stream to be sent or to be received,
- control of the physical layer and the communication layer

An essential information for the central control of the data transmission layer is the position within the message frame currently being received or transmitted. Since there is a fixed number of frame bits in a CAN message but the number of data bits is variable the frame position is being kept track of by two separate counters, a frame bit and a data bit counter.

All error symptoms are fed into the error handling unit (including the three error flags of the physical layer, the CRC error flag of the CRC unit and ACK flag of the central control unit). Further error flags are generated in this module: a difference error, if the the sending level deviates from the received level and a frame error, if a frame bit does not correspond to the message frame format of the CAN protocol. With this error symptoms and the present position in the message frame the error handling unit decides whether a significant error is present. As a consequence of a significant error the central control unit is requested to send an error message. Additionally this module updates the error counters. The CRC unit checks the received code and generates the CRC sequence for sending.

As in the case of the interface from the physical layer, the exchange of the message data between the data transmission layer and the communication layer is serial. At this interface, however, the number of data bits to be transferred is much lower than at the interface to the physical layer, because the data transmission layer filters all information not relevant for the host controller. Thus, the exchange of data is reduced to the message identification, the remote request bit, the data length code and the data.

Communication Layer The communication layer consists of the three main modules user interface, message interface and control. The user interface provides an interface allowing the host controller to access the CAN controller memory mapped over a 32 bit bus. Four registers are implemented in the user interface serving the exchange the system informations — commands, status informations and configuration data — with the user. The CAN controller can access the DPRAM which contains the message buffers over a 16 bit bus. In the message interface data is converted from serial to parallel and vice versa. The control module sets the message related flags: the information when a message was received; in which buffer it is waiting; when the message was successfully sent; whether the user is currently granted write access to the send buffer. Additionally the control unit performas acceptance filtering.

Reduced CAN Controller In the limited design of the reduced CAN controller, the data transmission layer is simplified as follows:

- Frame bits are neither inserted in the send data stream nor deleted from the receive data stream.
- No CRC check is made and no CRC sequence is generated.
- The only errors which lead to an abortion of the transmission are the stuff error flag (only during dominant bus level) and the synchronization error flag.
- No error states are indicated.

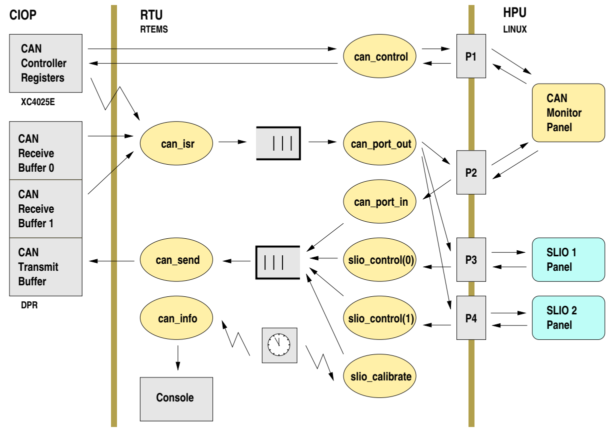


Figure 5. CAN monitor tasks

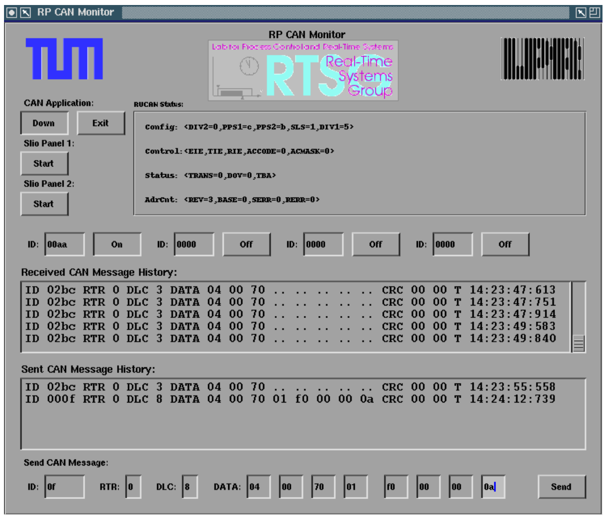
- An acknowledgment is neither send nor a received one checked.

The functions according to the first two items were transfered to software while the last three functions were neglected in this implementation.

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 5.

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 3.4.2. 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.



RUCAN Status:

```
Config: <DIV2=0,PPS1=c,PPS2=b,SLS=1,DIV1=5>

Control:<EIE,TIE,RIE,ACCODE=0,ACMASK=0>

Status: <TRANS=0,DOV=0,TBA>

AdrCnt: <REV=3,BASE=0,SERR=0,RERR=0>
```

ID: 00aa ID: 0000 ID: 0000 ID: 0000

Received CAN Message History:

```
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:47:613
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:47:751
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:47:914
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:49:583
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:49:840
```

Sent CAN Message History:

```
ID 02bc RTR 0 DLC 3 DATA 04 00 70 .. .. .. .. .. CRC 00 00 T 14:23:55:558
ID 000f RTR 0 DLC 8 DATA 04 00 70 01 f0 00 00 0a CRC 00 00 T 14:24:12:739
```

Send CAN Message:

ID: 0f RTR: 0 DLC: 8 DATA: 04 00 70 01 f0 00 00 0a

Figure 6. CAN monitor GUI

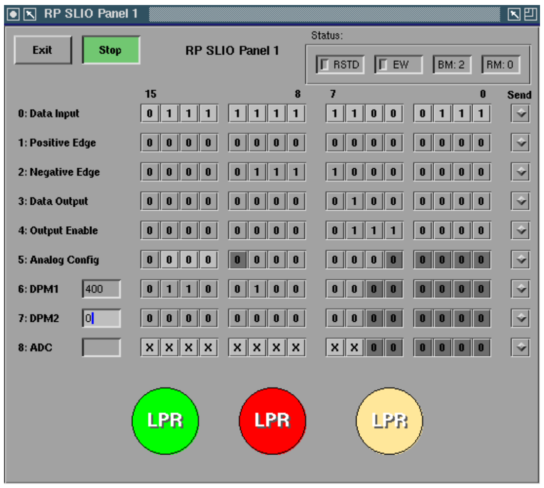
3.3. HPU

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. 6), 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* (Fig. 7) display the contents of all SLIO registers, as well as the state of the switches and LEDs. By pushing buttons on the panel, the LEDs can be switches on and off, and the function generator on the second SLIO card can be programmed and started. The values from the function generator, which are sampled and sent over the CAN bus, are displayed in a graph over the time of their arrival.

3.4. Interface and Communication

In the future, a uniform interprocess communication layer that provides support for inter and intra unit IPC — including interfacing with the FPGA–based CIOP — will be developed in order to simplify partitioning and distribution of application threads to different processing units.

The basic idea for the implementation fo e.g. 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. The receiving task in



		15							8	7							0	Send
0: Data Input		0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	
1: Positive Edge		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2: Negative Edge		0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	
3: Data Output		0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	
4: Output Enable		0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	
5: Analog Config		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6: DPM1	400	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	
7: DPM2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
8: ADC		X	X	X	X	X	X	X	X	X	X	0	0	0	0	0	0	

Figure 7. SLIO controller GUI

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 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 processing unit can generate an interrupt on a remote unit by accessing certain 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.

For this example, however, 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. For local IPC on the RTU RTEMS message queues were used while inter node communication between RTEMS threads and Linux processes (on the HPU) was based on a simple IPC module as presented in subsection 3.4.2.

[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

3.4.1 Communicating with the CIOP

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

The CAN controller's communication layer implements four registers for configuration, control and status information as shown in Figure 8 and Tables 2 and 3.

Table 2. General System Data

Name	Bit	Function
Configuration Register (0)		
CDC	0-15	clock division coefficient
PTS	16-19	propagation time segment
PBS1	20-23	phase buffer segment1
PBS2	24-27	phase buffer segment2
Control Register (1)		
Name	**Bit**	**Function**
AM	0-7	acceptance mask
AC	8-15	acceptance code
RB0	16	release buffer0
RB1	17	release buffer1
TA	18	transmission abort
TR	19	transmit request
RR	20	reset request
IER	24	receive (RMx) interrupt enable
IET	25	transmit (TCS) interrupt enable
IEE	26	error (BOS) interrupt enable
IP	27	interrupt pending
Status Register (2)		
Name	**Bit**	**Function**
RM0	0	received message in buffer0
RM1	1	received message in buffer1
TBA	2	transmit buffer access
TCS	3	transmission complete status
DOV	4-7	data overrun (counter)
WLS	24	warning level status
EPS	25	error passive status
BOS	26	bus off status
Address/Count Register (3)		
Name	**Bit**	**Function**
BA	16-23	base address (of the transmission buffers in the DPRAM)
RMN	24-27	revision id minor (version)
RMJ	28-31	revision id major (type)

The *configuration register* (0) holds the bit timing parameters (fields PTS, PBS1, PBS2) and a 16 bit clock dividing coefficient (CDC). The latter is used to derive the bittiming clock from the master clock of 25 or 12.5 MHz. This register is read-only during operation and writable only if the CAN controller is in the reset state.

The read/write *control register* (1) implements separate enable bits for bus–off, send and receive interrupts (IEB,

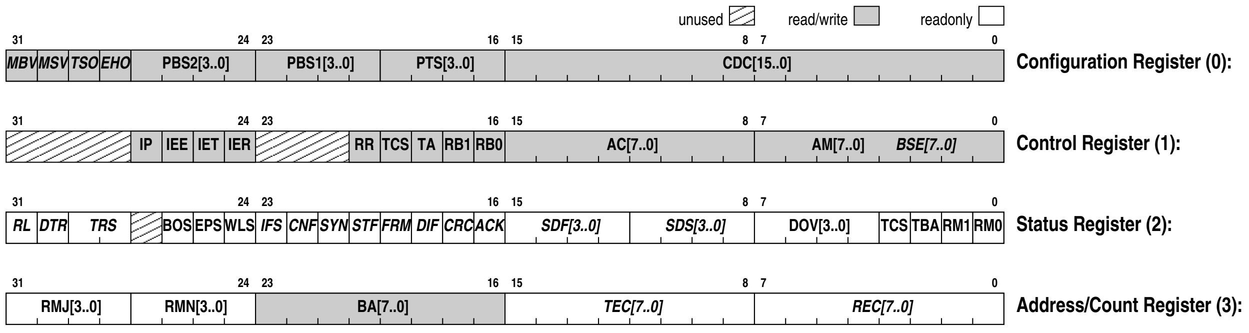


Figure 8. CAN Controller Registers

```
can_send(message)
{
    /* wait for previous
       transmission to complete */
    while (TR && !TCS);
    TR = 0;
    while (!TBA);
    /* copy message to transmit buffer */
    msgbuf[2] = message;
    TR = 1;
    return;
}

can_receive()
{
    /* wait for message to arrive */
    while (!(RM0 || RM1));
    /* copy message from receive buffer */
    if (RM0) {
        message = msgbuf[0];
        RB0 = 1;
    } else {
        message = msgbuf[1];
        RB1 = 1;
    }
    return(message);
}
```

Figure 9. CAN send and receive with polling

IET, IER), the reset request bit (RR) as well as control bits for transmitting and receiving messages (TA, TR, RBx). The acceptance code and mask fields (AC, AM) allow simple message filtering according to the basic CAN specification.

In addition to status information on received and transmitted messages (RMx, TBA, TCS) the read-only *status register* (2) provides error information (BOS, EPS, WLS, DOV). together with the error counters within The read-only revision id (fields RMJ and RMN) within the *addr_cnt register* (3) is a unique identifier for the CAN controller model loaded into the FPGA (to be checked by the controlling software) while the 8 bit base address (BA) allows to relocate the CAN message buffers within the dual ported RAM.

To receive a message from the CAN bus, the software side has to wait for one of the receive message flags (RMx) to be set by the controller either by polling (cf. Fig.9) or by use of interrupts and and ISR. 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 RBx. 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 (TBA), the new message can be copied to the transmit buffer and TR set.

The additional control and status bits listed in Table 3 are implemented for debugging purposes:

If the control register bit MSV is set, the last 4 transmission states of the CAN controller can be read from bits 16–23 of register 3 instead of the base address. With MBV set to one, the bit position of the last synchronization error is reported in bits 0–7 on a read access to the status register instead of the acceptance mask. EHO and THO (control register) turn of error handling and synchronization during transmission, respectively. In register 2 additional transmission states (TRS, RL), the error symptoms occurred since the last reset (ACK, CRC, DIF, FRM, STF, SYN) and the maximum synchronization drift (SDS, SDF) are provided. The value of the error counters can be read from the addr_cnt register (fields REC and TEC).

3.4.2 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

Table 3. Debug Informations

Name	Bit	Function
Configuration Register (0)		
Name	**Bit**	**Function**
EHO	28	error handling off
TSO	29	transmit synchronization omit
MSV	30	make states visible
MBV	31	make bit position visible (if a synchronization error occurres)
Control Register (1)		
Name	**Bit**	**Function**
BSE	0-7	bit position (at the last) synchronization error
Status Register (2)		
Name	**Bit**	**Function**
SDS	8-11	synchronization drift (too fast)
SDF	12-15	synchronization drift (too slow)
ACK	16	acknowledge error
CRC	17	CRC error
DIF	18	Rx $\neq$ Tx level
FRM	19	frame error
STF	20	stuffing error
SYN	21	synchronization error
CNF	22	configuration error
TRS	28-29	actual transmission state
RL	31	reset lock
Address/Count Register (3)		
Name	**Bit**	**Function**
REC	0-7	receive error count
TEC	8-15	transmit error count
L4S	16-23	last 4 transmission states

```
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);
}
```

Figure 10. Simple port send and receive functions

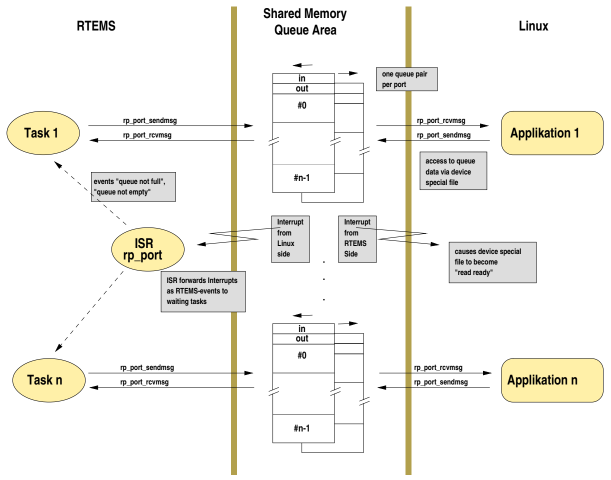

RTEMS
Shared Memory Queue Area
Linux
in
out
#0
#n-1
one queue pair per port
Task 1
rp_port_sendmsg
rp_port_rcvmsg
rp_port_rcvmsg
rp_port_sendmsg
Applikation 1
access to queue data via device special file
events "queue not full", "queue not empty"
ISR rp_port
Interrupt from Linux side
Interrupt from RTEMS Side
causes device special file to become "read ready"
ISR forwards Interrupts as RTEMS-events to waiting tasks
Task n
rp_port_sendmsg
rp_port_rcvmsg
rp_port_rcvmsg
rp_port_sendmsg
Applikation n

Figure 11. simple message queue implementation

via PCI bus. A queue consists of the *in* and *out* indices and a configurable fixed number of message buffers.

The basic algorithm for sending and receiving messages can be seen in Fig. 10: `queue_getmsg()` (`queue_putmsg()`) check whether a message (a FIFO location) is available and if so, copy the message from the queue's (thread's) to the thread's (queue's) message buffer and return `TRUE`. A receiving (sending) thread will wait for a message to arrive (a FIFO location to become available) and then notify the remote side of the receive (send) queue being not full (not empty) by triggering an interrupt.[2]

On the RTU, the port 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 (Fig. 11).

On the HPU, access to the queue memory area as well as triggering and handling notification interrupts is supported by our "rapid" device driver. The queue memory area is mapped into the user processes address space (*mmap(2)*) to speedup and simplify access to the queue indices and message buffers. Triggering the notification interrupt is accomplished by an appropriate *ioctl(2)*, while the driver's ISR services the remote interrupts and unblocks processes waiting in the *select(2)* system call.

This implementation was successfully used in the CAN monitor application example. However, for general use within our rapid prototyping architecture, it lacks flexibility and some functionality:

- Missing mutual exclusion of access to port and queue data reduces overhead but allows only one thread or process per port side.
- Maximum message size and queue length are configurable but fixed and identically for all ports.
- The maximum number of ports is fixed and no flexible naming scheme is provided.

[2]The RTU's system controller chip (Galileo GT 64010) supports CPU (RTU) to PCI (HPU) as well as PCI to CPU side interrupts by simply writing the respective bits in the `int_cause` register.

- Waiting for messages on more than one port is not supported, except for a nonblocking version of the `port_rcvmsg()` call.

These limitations will be addressed in a next revision. Results from a first performance evaluation are discussed in Section 4.2.

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). The fact that it was possible to implement “RULIM” using the smaller XC4013E FPGA chip, with a maximum capacity of only 576 Configurable Logic Blocks (CLBs), shows the optimization power of the Partition Place and Route (PPR) tool of Xilinx XAct.

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.

In the moment, none of the IOB flip flops were used, because of the generated global set and reset signal in the complete VHDL model. To make use of these resources as well, the generated code or the code generator itself would have to be modified.

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.

For the application threads on the RTU approximately 2500 lines of code (including application specific header files and comments) were written, which 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)[3], 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.[4] 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. Fig. 12 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

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

[4]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 $0.2\ \mu$s (10 system clock cycles).

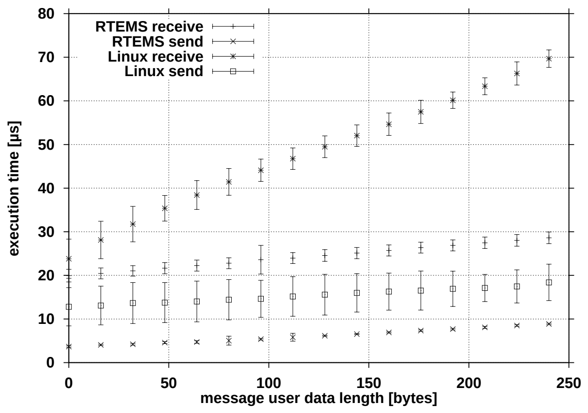


Figure 12. performance of the simple message queue implementation

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.

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 maximum 1 MHz without message loss. At a bus frequency of 125 kHz it was also possible to integrate the SLIO components and to keep them active by periodic calibration messages. 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 physical layer and 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.

Acknowledgments

The authors want to thank the graduates and students Christian Mühlbauer [4], Robert Pinzinger [6], 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] K. Etschberger et al. *CAN Controller–Area–Network, Grundlagen, Protokolle, Bausteine, Anwendungen.* Hanser Verlag, 1994.

[2] F. Fischer, T. Kolloch, A. Muth, and G. Färber. A configurable target architecture for rapid prototyping high performance control systems. In H. 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] G. Färber, F. Fischer, T. Kolloch, and A. 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, Oct. 27–29 1997.

[4] C. 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.

[5] Philips Semiconductors, Eindhoven, The Netherlands. *PCA82C200, Stand–alone CAN Controller, Product Specification*, 1992.

[6] R. Pinzinger. Speichersubsystem und flexible Prozeßanbindung für einen Rechnerknoten eines Rapid–Prototyping–Systems, 1997. Diplomarbeit (masters thesis) am Lehrstuhl für Prozessrechner, Technische Universität München.

[7] S. Rudolph. Modellierung und Realisierung eines CAN–Bus Controllers als Testszenario der Rapid Prototyping Umgebung *REAR*, 1997. Diplomarbeit am Lehrstuhl für Prozessrechner, Technische Universität München.