A Prototype for Interprocess Communication Support, in Hardware

Johan Furunäs, Joakim Adomat, Lennart Lindh, Johan Stärner, Peter Vörös
Mälardalen University
IDT/ Department of Real-Time Computer Systems, Sweden
email: lennart.lindh@mdh.se, joakim.adomat@mdh.se, johan.stärner@mdh.se, johan.furunas@mdh.se, peter.voros@mdh.se

Abstract

In message based systems, interprocess communication (IPC) is a central facility. If the IPC part is ineffective in such a system, it will decrease the performance and response time. By implementing the IPC facility in hardware, the administration (scheduling, message handling, time-out supervising etc.), is reduced on the CPU, which leads to more time left for the application and a more deterministic time behaviour. This paper describes an hardware implementation of asynchronous IPC in an RTU based architecture. RTU is a hardware implementation of a real-time kernel for uniprocessor and multiprocessor systems. In addition, our implementation of IPC supports message priority, priority inheritance on message arrival, and task time-out on message send/receive. An increased performance and message flow, in a message intense system, can be realized by implementing IPC functions in an RTU architecture.

1. Introduction

Different methods have been used to improve IPC e.g. using registers [Cheriton], implement IPC functions in microcoded firmware [Jeff], or by kernel design [Liedtke]. This paper describes a hardware implementation of asynchronous IPC in an RTU based architecture(cf. section 2). The motivation is to understand how performance and message flow in a message intense system can be increased by adding some functionality (for instance message priority) to the IPC functions and implement it in an RTU architecture.

In the event driven systems, where a message is produced out of every event or other message intense systems, it is important to get an efficient message flow through the system. Some messages are urgent, e.g. exceptions, which means that such messages should be handled before less important ones. Therefore message priority is included in our IPC model. To further decrease the response time (the time between a message is sent until it is read) priority inheritance on message arrival is included. By implementing the IPC facility in hardware, the administration (scheduling, message handling, time-out supervising etc.) is reduced on the CPU, which leads to more time left for the application. In multiprocessor systems, the IPC facility generally result in accesses to shared data and locks along critical sections. This is solved for instance in software [Gamsa] and in hardware with an RTU architecture.

![Figure 1: Our model of IPC](image)

In our model of asynchronous IPC (cf. fig. 1), slots are OS (operating system) resources that can be allocated by tasks. Each slot consists of a message queue, which holds the priority of messages and references to the messages which
are stored in a message buffer. By task, we mean the execution unit which is scheduled by the OS. Every message has a priority, which is set by the sender. The messages in the queues are sorted by their priority or in FIFO order. A task can inherit priority from messages. A sender task can use time-out constraints on full queues and a receiver task can do the same on empty queues, e.g. a receiver task can be set to wait a specified time for a message. Our solution of IPC works on both uniprocessor and multiprocessor systems.

In section 2 the RTU based architecture is described. The prototype board is described in section 3 and the design method in 4. Section 5 describes the IPC implementation and section 6 some problems concerning our IPC. Finally, in section 7 results and further work is discussed.

2. RTU Based Architecture - System Architecture

The goal of the architecture was to verify the implementation of the IPC functions and to produce a prototype in a short time. The system consists of standard commercial CPU boards, ram, I/O, and a specially designed prototype board. The prototype board was designed and produced, since we could not find any commercial board with enough FPGA (Field Programmable Gate Array) capacity.

The system contains of five main parts (cf. fig. 2):
1. Three application processors boards [FORCE]. These are connected via the VME bus. Each processor has a local memory for local tasks.
2. A separate prototype board. The prototype board contains two X4025 and some logic (cf. section 3).
3. A global memory board for global tasks. The global tasks are dynamically scheduled on all three processors, the local tasks are locked to one processor and the local memory.

4. The analyzer processor board is for graphical display of system status and operation to provide developers with better visibility into their applications. A PC is used for the graphical interface. In the prototype system the VME bus master arbiter is on the analyzer board, but it is possible to change it to some of the other boards.

5. VME Bus is an asynchronous parallel bus, see [VME].

![Figure 2: Overview of the system architecture](image)

The Real-Time Kernel and the IPC functions are implemented in the prototype board in a co-processor we call RTU (Real-Time Unit, more information see [RTU-data] [RTU-History]). The motivation to implement the IPC functions into the RTU is to re-use existing functions as e.g. Scheduling algorithms and VME bus interface.

The main feature of RTU is the ability to control multiple heterogeneous CPUs. The following real time functions have been implemented in RTU:
semaphores, event flags, watch dogs, interrupt handler, periodic start of tasks, relative delay of tasks, scheduler (priority algorithm), activate and terminate task, on/off task switch, IPC support and VME bus interface.

The scheduling concept is a priority-based, preemptive algorithm. The RTU consists of three schedulers, one for each CPU. There is one ready queue for each CPU (local queues), and one queue for the tasks which can be executed on any CPU (global queue). Each scheduler checks both the local and global queues. Since the RTU has information about the load on each CPU, it is possible to dynamically load balance the CPUs.

The interface between a hardware-based Real-Time Kernel such as RTU and application tasks is based on registers and interrupts. Registers are used for service calls e.g. allocate IPC slot. Interrupts are used to inform CPUs about task switching. For more information about registers and interrupts, see [RTU-step],[RTU-data].

3. Prototype Board

Our ASIC (Application Specific Integrated Circuit) development needs a flexible FPGA based development platform. Rapid prototyping with FPGA’s is convenient for validation. In a few minutes the system can be reconfigured with a new VHDL code version, ready for test in the system. It can also be used as a temporary substitute for the ASIC, in program development, while the chip is being processed.

Two Xilinx XC4025E [XILINX] together provides a maximum of 90k programmable logic gates including on-chip RAM. The board is connected to the system via a fully buffered VMEbus interface. It can be configured to act as a arbitrary mix of busmaster, bus arbiter and slave. Onboard logic can control all of the interrupts on the bus. The FPGA’s can be configured via a serial link from a host computer or alternatively the configuration can be programmed into a standard EPROM for sub-second board initialisation on powerup. An extensive set of jumpers are incorporated for maximum board flexibility, i.e. the ability to configure the board to act as a VME master and/or bus arbiter, changing the board clock frequency, etc.

The board is a multilayer PCB with three signal layers and a split powerplane made with the Mentor Graphics’ Board Station [Mentor]. The physical dimension is single Europe format (100mmx160mm).

Important features for the 50k prototype PCB:
- Two Xilinx XC4025E.
- 8 LEDs/IO per XILINX are incorporated for diagnostic/flexibility purposes.
- Four individually programmable clocks signals ranging from 2 ms to 16 MHz to support various degree of utilization of the FPGA’s.
- All signals fully buffered according to the VME bus specification.
- Programmable board address.
- Serial or parallel FPGA configuration (onboard EPROM).
- Access to IRQ and bus arbitration control signals

4. Design Method

Roughly these major steps were followed:
step 1: Specification included descriptions of the different IPC - Calls. The system architecture was reusing subsystems from older designs and one new IPC subsystem was defined. The IPC-RTU is an augmentation of the ordinary RTU instruction set.

As this was a industrial co-operation project, ABB Robotics was involved developing the initial IPC specification. Standard VME Motorola-based CPU-boards was to be used in conjunction to the FPGA prototype board.

step 2: The subsystems were decomposed into smaller components and designed in technology independent VHDL. Each block was simulated using Qsim [Mentor].

step 3: The partitioning was quite easy in this case (cf. conclusion), putting the stripped-down RTU base-system including VME control logic
in one Xilinx and the stripped-down IPC component (supporting 4 slots with 28 messages) in the other. All tools were Mentor Graphics except the FPGA place & route. The Qvhdl tool was used to compile the code. Synthesis and optimization were carried out in AutologicII. Xilinx XACT tools mapped the construction onto the FPGA’s giving us the configuration bitstream.

Step 4: System test start with design of a I/O program in assembler between the IPC-RTU prototype board and application processor and main test program in C.

5. IPC-Implementation

The idea with our model (cf. fig. 1) of IPC is that tasks must allocate a slot to be able to communicate with each other. This leads to checks, if tasks are connected to slots. It is possible to solve that in software or in hardware. Notice that the prototype support slot owner check, i.e. a slot can be set to check if a receiver task own the slot, when it reads a message. The prototype does not follow the model strict, i.e. tasks which are not connected to slots can send a message to any slot and a slot can be read by a task which does not own it.

To be able to work on a IPC bus, some primitives are needed:

Initialization of slots

*init*: Initializes all slots to work on priority inheritance on message arrival or not, and the sorting algorithm for messages, FIFO or priority.

*input*: priority inheritance on or off, the sorting algorithm for messages (FIFO or priority)

*output*: nothing

Allocation and deallocation of slots

*allocate*: Allocates a slot for a task.

*input*: the slot number, default priority of the slot, one owner check

*output*: error code or OK

*deallocate*: Deallocate a slot for a task.

*input*: the slot number

*output*: error code or OK

Sender primitives

*put*: Allocates a message buffer to the given slot and returns a reference to it. If the message queue is full, the sending task is set to wait state until time-out has occurred or until the queue is not full anymore.

*input*: the slot to send to, the message priority, a time-out value

*output*: error code or a reference to the message buffer were the message data can be placed.

*put_ready*: Is used when the CPU has copied the message to the message buffer, since the RTU must be informed when that is done. The message is then sorted by the RTU and it can be read.

*input*: the slot number, the reference which put returned

*output*: error code or a OK

Receiver primitives

*get*: Returns a reference, to the buffer were the first message in the queue is placed, for the given slot. When the message queue is empty, the receiving task is set to wait state until time-out has occurred or until the queue is not empty anymore.

*input*: the slot where to read from, a time-out value

*output*: error code or a reference to the message buffer were the message data is placed

*get_ready*: Is used when the CPU has copied the message from the message buffer, since the RTU must be informed when that is done. The message place is then deallocated by the RTU and it can be used for new messages.

*input*: the slot, the reference which get returned

*output*: error code or OK

Sending and Receiving Messages:
The sender and receiver task must allocate a slot before communication. Then the procedures addressed below must be followed. The sender procedure
1. The sender requests to send a message to a slot and gets (using the put primitive) a message buffer reference allocated for the message from the RTU.
2. The message must be copied to the message buffer by the sender. The RTU must be informed when the copying is done (using the put_ready primitive).
3. The RTU sorts the message queue and the receiver may read the message.

The receiver procedure:
1. The receiver requests to read from its slot and gets (using the get primitive) a message buffer reference to the first message in the queue from the RTU.
2. The message must be copied from the message buffer by the receiver, if the message has to be saved. The RTU must be informed when the copying is done (using the get_ready primitive).
3. The RTU deallocates the message buffer and it can be used for a new message.

The IPC facility is partitioned into a software and a hardware part as follows:

5.1 Software part of the IPC

The software can be divided into three parts
- The interface code towards the RTU. This part consists of 56 instructions.
- The C interface code. Which changes the arguments to RTU format. This part consists of 45 instructions in the receiver case and 55 in the sender case.
- The IPC bus code. Performs the sending (put, copy message, put_ready) and receiving procedure (get, copy message, get_ready). This part consists of 87 instructions in the receiver case and 114 in the sender case, when the instructions which performs the copying of a message are not included.

5.2 Hardware part of the IPC

The IPC administration, sorting message queues etc., is placed in hardware i.e. on the RTU.
In our implementation (cf. fig. 3) there are 32 slots. A task can allocate a slot and be set to one out of 32 different priority levels. A message can be set to one out of 4 different priority levels. The message queues are placed in hardware. Each message queue can hold 28 message buffer references. Furthermore, each message queue keep also track of each message’s priority. The message buffers are placed in RAM, one buffer per CPU, and the message data is stored there.

Figure 3: The message structure, in our IPC model.
Priority inheritance on message arrival:
If priority inheritance on message arrival is used, it is necessary to have primitives to change priority on receiver tasks. In this implementation, every task (connected to a slot) must be set to a default priority, which it never will be below, and bounded to a pinc (priority increment) table. A pinc table, is a transformation table from message priority to task priority e.g. a message with priority 0 makes the task run on priority 3 (if the default prio is not bigger than 3) see table 1. If the message priority is mapped direct to the task priority, the priority will be hard coded and not flexible. A pinc table increases the flexibility, since the priority mapping can be changed dynamically for the application. A disadvantage of using it, is that additional memory is needed.

<table>
<thead>
<tr>
<th>Message priority</th>
<th>Task priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Example of a pinc table.

6. IPC-Problems

In this section, methods for preventing full message queues and preventing loss of messages when deallocating slots are presented.

6.1 The full queue dilemma

The reason for using messages with priority is to be able to deliver important messages as quick as possible. But if the receiving message queue is full, then no more mail can be received and the idea with message priority will not work. One function which can partially solve that problem is to let get_ready return the number of remaining messages. With such information, the message read frequency can be increased when the queue is nearly full. This helps to prevent the queue to get full, but is not sufficient if a burst of messages arrives. Another method to prevent full queues, is to reserve the last place in the queue for important messages. That will ensure that at least one important message will reach the receiver in a quick way, as it was thought with message priority. In general it is impossible to prevent full queues but with a combination of the above addressed methods the problem is reduced to a fair level.

6.2 Deallocation of slots

Since sending and receiving are not atomic operations, deallocation can result in problems addressed below.

- A slot can be deallocated during a send operation, without the senders knowledge. This may result in lost messages or messages send to wrong slot.
- In the time span between last message read and deallocation, other messages may arrive and subsequently disappear.

One solution to the problems is to implement a close primitive, which must be performed before a deallocation. The close operation returns how many messages, including messages which are not put_ready, are left in the queue. By reading the remaining messages before deallocation, no messages will be lost.

7. Conclusions & Future Work

It was not possible to download the whole IPC (supporting 32 slots with 28 messages) design onto the prototype board, but it was enough to show that it is possible to implement IPC (supporting 4 slots with 28 messages) support in hardware.

It is not necessary to use priority inheritance as in our model. Another way is to use the priorities from the pinc table as increments instead of absolute priorities. It is also possible to have more pinc tables, which can be changed dynamically.

Since it is a FPGA prototype it is not possible to get the same performance as in an ASIC implementation.
Partitioning can be very tedious and time consuming when a function block, e.g. the IPC in the RTU, in a design is too big to fit into one FPGA. Our solution to that is to either get a FPGA with more gates or reduce the functionality of the block, e.g. IPC can handle less slot than planned in our case. Hardware emulators are still too expensive to be each man’s property, which makes our way of using a FPGA prototype board to verify hardware functions a realistic alternative. Our limit is the cost of FPGA’s. Further work is to test if it is possible to let the RTU manage the message copying e.g. through DMA (Direct Memory Access) channels and investigate how systems are affected by the IPC functionality we implement, e.g. how the cache is affected when the RTU manage the message copying etc.

Acknowledgements

This work was supported by KK-Stiftelsen, Mentor Graphics, and ABB Robotics.

References


[RTU-data] RTU Data Book, RF RealFast AB, Dragverksg 138, S-724 74 Västerås, Sweden, e-mail: realfast@quicknet.se, http://www.quicknet.se/realfast


[XILINX] XILINX, The programmable gate array Company, 2100 Logic Drive, San Jose, CA 95124 USA http://www.xilinx.com