Mälardalen University Licentiate Thesis

No.33

An Application Programming Interface for Hardware and Software Threads

Peter Nygren

2004-09

Department of Computer Science and Engineering

Mälardalen University

ISBN number: 91-88834-64-6

ISSN number: 1651-9256

Printed by Arkitektkopia, Västerås, Sweden

Distribution: Mälardalen University Press

ABSTRACT

Modern embedded computer systems contain an increasing number of software and hardware components. The most common way to communicate between these components is to interrupt the processor (CPU) and let the operating system manage the communication. In almost any operating system, the arrival of an interrupt event causes the execution of a service routine (which could be a device driver handling some external I/O). The advantage of this method is that it encapsulates all hardware details of the I/O device. In many cases these interrupt driven service routines interfere with the real-time behavior. In cases where the interrupt routine is not handled properly, priority inversion and unbounded delays of process execution can be introduced. The real time problem with software device drivers and the development of Field Programmable Gate Array (FPGA) technology motivate research on communication and synchronization between hardware and software components. This thesis presents an application interface called VCB (Virtual Communication Bus), which provides a standardized interface for communication and synchronization between hardware and software without the need to execute any driver software. The interface provides six different system calls; connect, disconnect, send, receive, send&wait, and broadcast. The VCB also has functions to avoid priority inversion problems. The interface is fully implemented in hardware, meaning that no software is used during communication and that several system-calls can be made simultaneously. This makes the system easier to analyze and design. The thesis presents the VCB concept, its implementation architecture and definition of hardware threads. Furthermore, the VCB is demonstrated and evaluated in a case study with device drivers that manage a Universal Asynchronous Receiver Transmit (UART). The two main contributions of this research are (1) that it shows that it is possible to design a uniform interface for communication between hardware and software threads, and (2) that this interface can be used to design device drivers in hardware that introduce almost zero overhead for the software system to manage the external device.

1 INTRODUCTION

In the modern society today, almost every mechanical system contains some kind of computer based system. The mechanical system could be for example a toaster, a toy for the kids to a more advanced system like a car, a train or an airplane. The computer based systems will be used to regulate and control external equipment connected to the system. The computer system has the task to react within precise time constraints to events in the environment. As a consequence, the correct behavior of these computer based systems depends not only on the calculated value but also on the time at which the results are produced. To improve the real time behavior for a system the use of more special designed hardware components is a desire. A standardize communication and synchronization mechanism is requirement to simplify the use of parallel logic in programmable devices, such as Field Programmable Gate Arrays FPGA together with the software system. The emphasis is on the thread-level abstraction. We will motivate that at this abstraction level the distinction between hardware and software threads is practically gone and all steps in the design space can then be shared for both software and hardware design [23]. To make the design process both shorter and more effective the designers use standard components in both software and hardware, for example Ethernet controllers, universal asynchronous receiver/transmitter, file systems, operating systems and databases. To facilitate the mapping of different components, either to hardware or software components, the communication and synchronization interface has to be more general. One of the major consequences is that the system cost can be significantly reduced and tighter timing/performance constraints can be met if more hardware components are used in exacting and time critical parts. The advantage is that it is more predictable, faster and easier to analyze. Every gate is a true executing unit, compared to software that often executes on a single unit, the CPU. The disadvantage can be additional cost of using more hardware gates instead of a software solution. On the other hand the number of gates in one chip has been doubled every 18 month following Moore’s law, and it still growing. This thesis is utilizing this development within the FPGA technique area, to show that a system can be built with a high abstract communication and synchronization mechanism. Furthermore it shows how to solve a real-time problem with the help of this mechanism, as well as demonstrating the hardware function in a case study.

1.1 Real Time Systems

There are many different definitions of a real-time system; generally they all state that a real-time system has to react on events in its environment within a specific amount of time. One good definition is the following: *A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated [17]*

There are different classifications of Real Time Systems. Depending on the consequences, the time constraints of real-time systems can be divided into two categories.

1. *Hard real-time system* A real time thread is hard if missing its deadline may cause catastrophic consequence on the environment under control.[17]
2. *Soft real-time system* A real time thread is said to be soft if meeting its deadline is desirable for performance reason, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.[17]

Some of the functionalities needed in a real time system are to,

- handle a thread set,
- close connection to process-I/O,
- predictable and fast manage to handle events,
- avoid priority inversion using special protocols,
- thread communication mechanisms,
- time management.

1.2 Why communication and synchronization between threads

The communication and synchronization between threads is an important task and the need is well documented [17][4][11][14]. The internet alone shows a tremendous need of communication. Also smaller system like a robot control computer system needs a lot of communication and synchronization. This is because of the partitioning of the systems controlling task into different software threads. Those system threads need allot of communication and synchronization to solve the controlling task [3]. These threads are connected together with a common communication mechanism, containing an API and protocol. The API is the syntax of the call and the protocol is the mechanism to handle the message. For example a letter has an API, the address to the recipient and the protocol is the post-office, managing the letter. In a computer system with different threads, use a shared memory area is use to exchange information. To avoid inconsistency in the communication system a mechanism is designed to handle that. The basic mechanisms are typically binary semaphores. If no access protocols are used to enter a critical sections like the shared memory, a number of undesirable phenomena can occur, such as priority inversion, chained blocking, and dead lock, which could introduce unbounded delays on real-time activities[15][17]. One problem with this type of communication and synchronization is that the application designer has to manage the communication and the protocol by hand in the application code instead of using a communication mechanism with a common API and protocol. Further, in this document a special designed communication and synchronization mechanism will be used to avoid the problems mention above. This mechanism has been developed and used for a while in a pure software thread solution.[11][3] But an extension to the mechanism with possibilities for hardware threads to use the mechanism to synchronize and communicate will be described in this thesis, for more detailed information of this communication mechanism see paper A and paper B.

1.3 Difference between hardware and software

Traditionally, software has been seen as “running programs” and hardware as “static”. Engineers and scientists have considered hardware and software as distinct entities with little in common in their design process [13]. In today’s research this concept is not always true. This is mostly due to different design styles that have been brought up in recent years that include very tight coupling between hardware and software and for that reason it is better to use other terms than only hardware and software to describe parts in a system.
If a designer, for example, takes a software program, translates it to a hardware description and then synthesizes it to target architecture like an FPGA or an ASIC, then a question arises: is the result software or hardware? The result can, for example, be seen as one large instruction doing all the computation or it can be seen as a custom dedicated hardware solution.

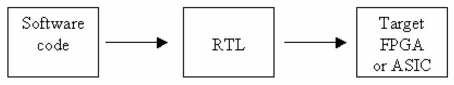


Figure 1 Translating a high level language to a hardware implementation

Yet another example of implementation style, that isn’t either hardware or software, is when instructions are placed in memories all around the chip, tightly coupled to pure hardware parts. This example shows that there are in some cases hard to tell when an implementation is a hardware or software approach.

A technique that is increasingly used in embedded system design is the use of programmable logic devices, like FPGAs. These devices bring a new terminology to system design due to that these devices can be configured as almost any computing element (microprocessor, DSP, custom hardware) and also are reconfigured in an easy way. This device is traditionally classified as a hardware component, but can today be shaped as a software solution with tightly coupled hardware parts.

A trend in today’s system development is to as much as possible only talk in terms of behaviors when handling different functions instead of software and hardware parts. This is the main idea in co-design: to select what goes into hardware respectively software as late as possible in the design process.

The progress in System-on-Chip implementations also tends to include a tighter and tighter mix of hardware and software parts, sometimes so tight that the hardware and software terms no longer are motivated. A problem when mixing hardware and software is the testing and validation process. It can be hard to apply the common test methodologies to a system where the coupling is tight. Co-simulation is the term used when hardware and software are tested in the same environment. Co-simulation tools exist today, but are a big challenge for researchers to improve while new technologies and architecture styles brings new concepts to computing.

1.4 **Programmable logic FPGA, CPLD**

The new technology introduced with the Field Programmable Gate Arrays (FPGA's) represents a leap forward in digital design as large as when the micro-controllers were made public in the 70's. The inherent enormous parallelism in these devices introduces new design challenges and trade-offs. One promising approach is to use standard and custom hardware components to build complete system architecture at on single chip. All hardware components are executed in truly massive parallel to each other and do not need to interfere with each other, as opposed to software implementations. A FPGA fundamental characteristic is that it consists of fine-grained programmable logic blocks interconnected via wires and programmable switches. Logic functionality for each block is specified via a small programmable memory, called lookup table (LUT). How these logic blocks are spread over the silicon differs in devices and families from different vendors. In some they are grouped with a matrix structure and in other they are grouped in slices over the chip. FPGA´s were earlier only used as interconnecting glue logic and to implement simpler Boolean functions. In recent years, new device families have quickly grown to capacities of tens of thousands of LUT´s containing millions of gates of logic [18] enabling them to be used as platforms for complex computing machines including processor structures and advanced calculations. Therefore, FPGA´s has become a technology to be considered at system level. Complex Programmable Logic Device (CPLD) is another device similar to an FPGA. CPLD´s are not that often used in research literature, therefore, in this publication the focus will be almost exclusively on FPGA´s, representing programmable logic devices.

1.5 Thesis Outline

This thesis consists of four main chapters describing the different parts in the scope of this thesis. It starts with a motivation. Further we present an overview of related work which gives insight into the problem area. Furthermore in section four, the result and contributions will be presented. Finally we sum up with the conclusion and experience from this area.

2 MOTIVATION

Real time systems are computing systems that must react within precise time constrains to events in the environment. The correct behavior of these systems depends not only on the logical result of the computation but also on the time at which the results are produced. A reaction that occurs to late or to early could be useless or even dangerous. To calculate and predict the response time to the environment for a given set of tasks could be hard if the time behavior of a system isn't predictable.

Today, the new FPGA technology will lead to digital computer systems that will fit into one chip [24], deterministic response time with predictable time behavior and faster parallel developments with standard hardware components [10]. The impact on the application will be shorter development time (component based), decreased production costs (one chip solution) and higher specially (utilization of hardware parallelism).

Special designed hardware components at the thread level of abstraction makes the design space much greater for the system designers to solve design or performance problems. This is the main motivation for this work, to explore a new way to implement hardware threads with a communication and synchronization concept we call VCB (Virtual Communication Bus).

Hardware threads or components that give the system-designer the following benefits:

- *Hardware threads always run in true parallel.*

  In the real word environment everything is parallel and different events are handled in true parallel. Moving software functionality into an equivalent implemented hardware function increases the simultaneousness in a system. A hardware function is always running and almost never uses any shared system resources and thereby the hardware will not be interrupted or stalled waiting for some shared resource. This makes it easier to calculate and analyze the execution time for the function.
- *Hardware gives better performance.*

  Hardware implemented functions give almost always better performance compared with equivalent implemented function in software [10], [11].

- *A hardware implemented function reduces the number of interrupts to the processor kernel.*

  A hardware design connected to the real word environment reduces in most cases the amount of interrupts to the software processor kernel. The hardware function can execute in parallel with the software processor and will not compete with other shared system recourses.

- *Hardware functions reduce the response time.*

  A hardware function reduces the response time especially if the locality of the function is placed in the same chip. A good example of that is the math processor in a modern processor CPU. It gives hundred or even thousand times better performance to implement math functions in hardware compared to implementing the same functionality in software.

All these points give a good motivation to believe that hardware functionality improves the time predictability for a real time system.

3 RELATED WORK OF COMMUNICATION INTERFACE FOR HW/SW THREADS

This chapter will give a brief overview of the different types of communication interfaces (see Figure 2). For more information in this area, the readers are referring to [16]. The communications between threads are divided in three parts:

A. (SW/SW) Software to Software thread communication, described in section 3.1.

B. (SW/HW) Software to Hardware and vice versa communication described in section 3.2

C. (HW/HW) Pure Hardware to Hardware communication described in section 3.3

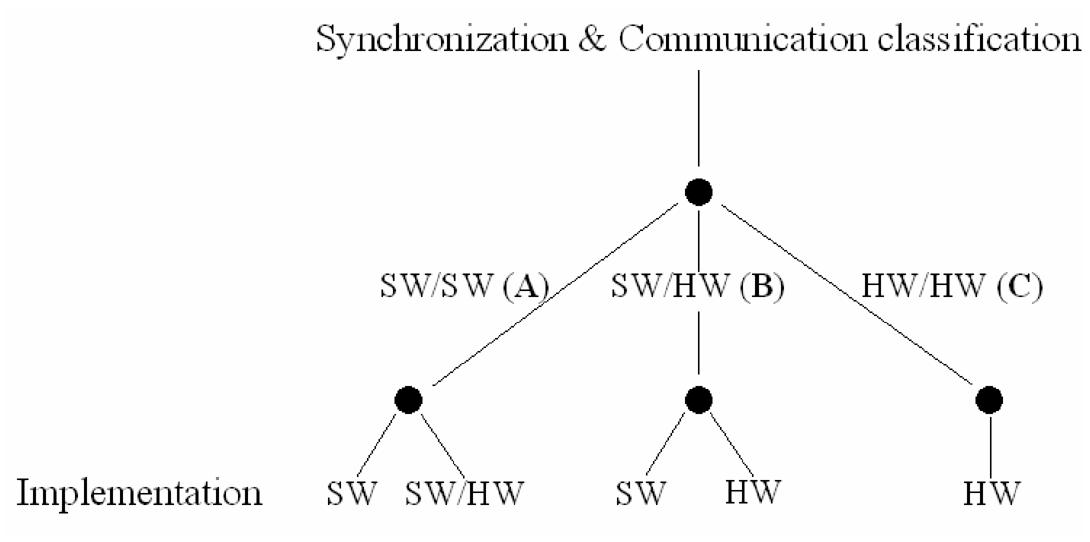


Figure 2 Classification of communication interfaces

The communication protocol and API are often implemented in software, but it can also be in hardware/software or only in hardware (VCB-API). This classification will focus at different communication mechanisms. The first section 3.1 will describe the communication between pure software threads. The second section 3.2 present two subclasses of the communication mechanism; the first one implemented in pure software, most common in commercial operating systems; the second subclasses are systems with special designed hardware. These type of system isn't common in the industry, but there are several academic works in this area [4][3][2][9][2]. The last section 3.3 will describe communication between pure hardware blocks. The focus on this brief description will be on communication between software and hardware threads.

3.1 Software thread communication SW/SW (A)

The thread communication is a wide and can be implemented in many different types of mechanisms. An example of different types of implementation of the thread communication could be:

- *Shared memory*

  The advantage of using shared memory is the performance; the user has the possibility to directly access the global shared memory. The disadvantage to use shared memory among many user threads is the inconsistence in the communication system. To make a mutual exclusion between those threads, the user has to take a semaphore in correct sequence and thereby making a mutual exclusion of shared resources [15].
- *Mailboxes*

  Mailboxes is a message passing mechanism in many real-time systems [29][28][30]. This mechanism is used to synchronize and communicate between different application threads. A mailbox is a message buffer located in the shared memory, protected by a protocol normally implemented in the operating system. Mailboxes buffer the messages in FIFO order and thereby permit the thread to read the message later on. The semantic of a mailbox may vary between different operating system but in general, blocking and none blocking read and write is the most common functionality.
- *Message Queue*

  Like the mailbox, the message queue mechanism is also used to synchronize and communicate between the application threads. The main difference between this and the previous on is the sorting algorithms. A message queue could sort the different messages either in FIFO, LIFO (Last In First Out) or priority order [3].

- *Pipes*

  Like the message queue the pipes can manage many messages but can only be used in one direction. When a thread wants to send and receive messages it has to open two pipes, one in either direction.

Interprocess communication is a critical issue in real time systems. The use of shared resources for implementing the message passing scheme may cause priority inversion and unbounded blocking on threads program execution. This would prevent any guarantee on the thread set and would lead to a highly unpredictable real time behavior. The most typical communication semantics used in different operating systems is the synchronous and asynchronous model.

Examples of commercial systems are

- The RTOS Real Time Operating System OSE is used in many telecommunication systems [28]. It uses a pure software implementation of the message passing mechanism.
- The largest RTOS in the world Vxwork's [29] uses message queues to implement the communication mechanism. It does not use any hardware support.
- Sierra16 [30] is a small RTOS used in small embedded systems. It uses semaphores or signals to implement the communication. Special designed hardware used to predict and speed up the communication.

The other class has support from special designed hardware to give the better performance and increase the predictability. This type of system isn't common in industry.

Examples of systems of hardware support are

- The Spring RTOS [4] is a distributed multiprocessor system. It uses a mix of software and hardware support to implement the network and the thread communication.
- SARA RTOS [3] is a multiprocessor system with thread communication with parts of the communication mechanism in hardware.

3.2 Software thread and device driver communication SW/HW (B)

This section will describe the synchronization and communication between the software application and a hardware device through a device driver. This section will be divided in two parts, one with the most common methods to solve the SW/HW communication (section 3.2.1). The other part will be our academic contribution (section 3.2.2).

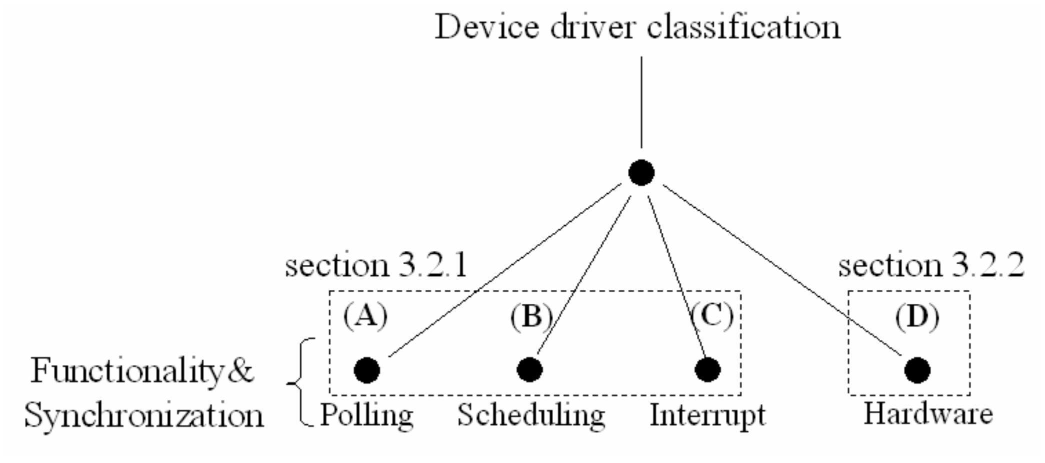


Figure 3 Functionality and synchronization classification of device drivers

A. The application has to poll the device. More detail in section 3.2.1.1
B. A specific driver encapsulates the hardware dependent code and the driver will be scheduled among other threads in the system. More details in section 3.2.1.2
C. The specific driver encapsulates all hardware dependent code and the synchronization will be through external interrupts. More detail in section 3.2.1.3
D. The driver for the functionality will be implemented in hardware and the synchronization will be through the VCB communication. More detail in section 3.2.2

When building applications that interact with different type of I/O devices and software which makes use of device drivers, one typically ends up with code that can only be used on the original platform. I. e., the code is hardware dependent since the device drivers used are much likely to be handcrafted for a specific hardware device [9]. Thus, the code cannot be used on other hardware platforms. To circumvent this problem, the dependency should be eliminated at a certain layer to allow portability. One way of accomplishing this is by using a consistent device driver interface that applications and system software should use to gain access to device drivers. By inferring a layer between the device drivers and the rest of the software, one ensures that the software can be applied on different hardware platforms, assuming that the same operating system is used on the new platform, and that corresponding device drivers are present. The link between a hardware device and the application is a device driver. Instead of forcing the applications to manually interface with the hardware device. The device driver is introduced to interacting with, and controlling the hardware directly. The driver for the device performs all the hardware specific processing. The application software operates on logical I/O objects provided by the driver. The logical I/O objects are presented to the applications as a set of functions calls for using the I/O device.

3.2.1 Common SW/HW communication

This section will give a small survey of the three most common solutions to interact with a hardware device.

3.2.1.1 Polling

Of those three *polling, scheduling, interrupt* is the *polling* the most radical solution. This is because of all interrupts from external devices are disabled and the peripheral devices must be handled by the application threads which have direct access to the registers of the interface to the device. Since no interrupt is generated, data transfer takes place through polling. The main disadvantage of this solution is low processor efficiency on I/O operations due to the busy wait of the thread while accessing the device registers.

3.2.1.2 Scheduling

As in the polling solution all external interrupts are disabled, but unlike polling the device is not handled by the application threads but is managed in turn by dedicated kernel routines, periodically activated by the operating system timer. This approach almost eliminate the unbounded delays due I/O access and confine all I/O operation to one or more kernel threads, whose computational load can be calculate and taken in account through a specification utilization factor. The advantages with this solution compared with the previous one is that all hardware details of the peripheral devices can be encapsulated and do not need to be known to the application. Ref [4][6]

3.2.1.3 Interrupt

The third approach enable all interrupt from the external devices, while reducing the drivers to a minimum size [17]. According to this method, the only purpose of each driver is to activate a proper thread that will take care of the device management. While activating the execution of a thread under direct control of the operating system, it's scheduled just like any other application thread. A priority can be assigned to the device thread completely independent from the application priority according to the application requirement. One major advantage of this approach is to eliminate the busy wait during the I/O operations. Unbounded delays introduced by the drivers during thread execution are also dramatically reduced. Ref [1]

Examples of academic and commercial and systems,

- Lynx [30] Used device driver is accessed through a set of *entry point* functions, which constitute the device driver's applications interface. A kernel thread and the basic device driver also composed of an interrupt service routine (ISR). Uses the Interrupt approach
- OSE [28]Uses device driver in access through a set of *entry point* functions and direct polled, or by a timer slots and interrupt driven. Uses the Interrupt approach
- Synthesis of DMA controllers from architecture independent descriptions of HW/SW communication protocols [9]. Use the Interrupt approach and special hardware to read and write to the hardware device.
- HARTIK a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution [1]. Uses the Interrupt approach

3.2.2 Our SW/HW communication approach

The last approach with respect to all others is to remove the external interrupts, routed directly into the processor kernel. Those interrupts will instead be managed by a hardware thread. This hardware thread will manage the functionality and act as a device driver. All data transfers take place through reads and writes through a (DMA) Direct Memory Access channel to the shared global memory. The hardware thread will synchronize with the application thread through send/receive messages through VCB-API. The main advantage of this solution is the true execution parallelism with the software processor kernel and the removing of the notification from the interrupts. No data read and write transfers take place through the processor kernel.

3.3 Communication between hardware blocks HW/HW (C)

Recent advances in hardware design now allow the integration of numerous functions onto the same single FPGA chip or silicon piece. Those functions could be from generic serial ports to complex memory controllers and processor cores. As a result, a hardware designer must now address issues as design for reuse and reuse of designs. To solve those problem, hardware designers has developed different types of virtual component interface [12]. This virtual interface is used together with different types of functionality and interconnections. The semantic of the interface is *request respond* and the protocol is adjusted to the interconnection. The interconnection is in most cases point to point in the HW/HW communication. But this technique could also be used to a shared system bus like IBM CoreConnect [32] ARM AMBA 2.0 High-performance Bus [25].

4 RECONFIGURABLE SYSTEM

This section will describe the chip device used during this project. The description will only talk about Xilinx devices [24] and architecture and will not give a general description of the different reconfigurable systems. The Altera solution with Cyclone, StratixII Nios or Arm9 will not be described [25]. The Virtex-II Pro is a platform FPGAs for designs that are based on IP cores and customized modules. The leading-edge at the device is 0.13 µm CMOS nine-layer copper process and have 18 Kb storage elements of True Dual-Port RAM. Embedded multiplier blocks are 18-bit x 18-bit dedicated multipliers. Digital Clock Manager (DCM) blocks provide self calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, and coarse- and fine-grained clock phase shifting. A new generation of programmable routing resources called Active Interconnect Technology interconnects all these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix.

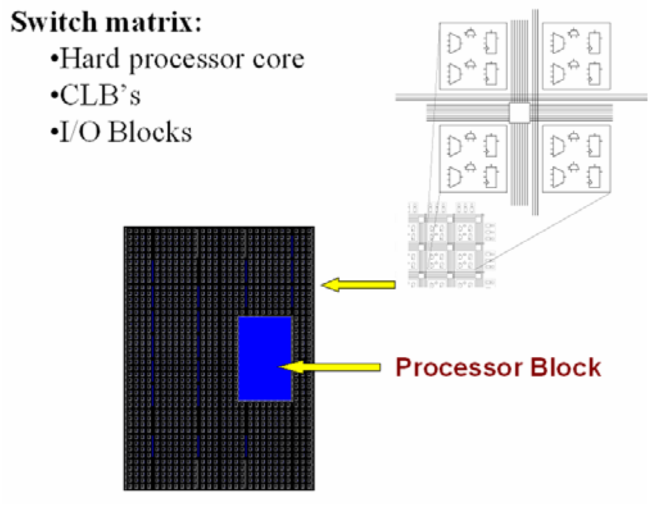


Figure 4 Xilinx virtexII_pro structure

Architecture:

The Virtex-II Pro devices are user-programmable gate arrays with various configurable elements and embedded blocks. The devices contain the following functionality.

- Embedded high-speed serial transceivers (RocketIO X).
- Embedded IBM PowerPC 405 RISC processor blocks

- SelectIO-Ultra blocks provide the interface between package pins and the internal configurable logic.
- Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.
- Block SelectRAM+ memory modules provide large 18 Kb storage elements of True Dual-Port RAM.
- Embedded multiplier blocks are 18-bit x 18-bit dedicated multipliers.
- Digital Clock Manager (DCM) blocks provide self calibrating, fully digital solutions for clock distribution

4.1 Processor architecture

The FPGA device today contains difference types of processor block. The biggest different of those two types is the implementation of the core. One type is a hard copy of the core the other type is a soft reconfigurable type, specially designed for a device family with regard to timing and area.

4.1.1 Hard processor core

The PPC405x3 is a 32-bit implementation of the *PowerPC™ embedded-environment architecture* that is derived from the PowerPC architecture. Specifically, the PPC405x3 is an embedded PowerPC 405D5 processor core (PPC405D5). The PowerPC architecture provides a software model that ensures compatibility between implementations of the PowerPC family of microprocessors. The PowerPC architecture defines parameters that guarantee compatible processor implementations at the application-program level, allowing broad flexibility in the development of derivative PowerPC implementations that meet specific market requirements [24].

4.1.2 Soft processor core

Optimized for Xilinx devices are the MicroBlaze soft processor core. This core is a 32-bit RISC processor and has 32 general purpose registers, separate instruction and data buses, and built-in interfaces to the on-chip memory and to IBM’s industry-standard On-chip Peripheral

Bus (OPB). In addition, implementations in Virtex-II and later devices support hardware multiply [26].

4.2 Shared system bus

The shared system bus could be any type of in house custom designed bus with special features. But in this architecture the bus is an IP component from IBM's core connect [32] The CoreConnect bus architecture is a standard SOC design, and serves as the foundation of IBM Blue Logic or other non-IBM devices. Elements of this architecture include the processor local bus (PLB), the on-chip peripheral bus (OPB), a bus bridge, and a device control register (DCR) bus. High-performance peripherals connect to the high-bandwidth, low-latency PLB. Slower peripheral cores connect to the OPB, which reduces traffic on the PLB, resulting in greater overall system performance.

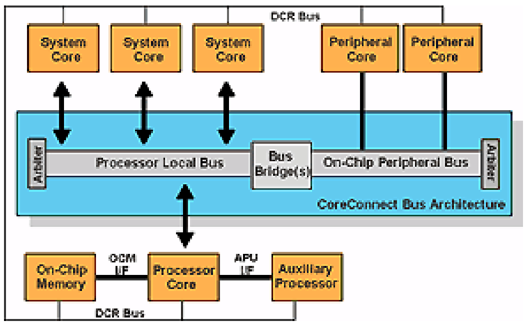


Figure 5 CoreConnect Block Diagram

Processor Local Bus

- Fully synchronous, supports up to 8 masters
- 32-, 64-, and 128-bit architecture versions; extendable to 256-bit
- Separate read/write data buses, enables overlapped transfers and higher data rates
- High Bandwidth Capabilities
    - Burst transfers, variable and fixed length supported
    - Pipelining
    - Split transactions
    - DMA transfers
    - No on-chip tri-states required
    - Cache Line transfers
    - Overlapped arbitration, programmable priority fairness

On-Chip Peripheral Bus (OPB)

- Fully synchronous
- 32-bit address bus, 32-bit data bus
- Supports single-cycle data transfers between master and slaves
- Supports multiple masters, determined by arbitration implementation
- Bridge function can be master on PLB or OPB
- No tri-state drivers required PLB Arbiter
- Arbitration for up to 8 PLB master devices on PLB bus
- Includes watchdog timer and separate address, read, and write data paths
- Supports address pipelining

PLB to OPB Bridges

- PLB slave and OPB master device
- Supports dynamic bus sizing for OPB connection
- Supports burst reads and writes
- Compliant with various bursts sizes
- Supports 4-, 8-, and 16-word line transfers
- Supports DMA transfers to/from OPB master peripherals

OPB Arbiter

- Arbitration for up to 4 OPB master peripherals on OPB bus

DCR Bus

- Provides fully synchronous movement of GPR data between CPU and slave logic

4.3 External I/O

The external I/O device in this test system includes an Asynchronous Receiver/Transmitter (UART) Intellectual Property (IP). This UART following the National Semiconductor PC16550D but have a few differences between the National Semiconductor implementation and the OPB UART. For more information [33] and data sheet for OPB UART Lite [24]

5 CONTRIBUTION AND RESULTS

This chapter will briefly present the contribution and results. More detailed descriptions in paper A, B, C.

The chapter is divided in three parts:

1. API for hardware threads; the contribution is defining the hardware thread and a case study is made to demonstrate how the API can simplify, achieve faster response time and make the design more predictable for device drivers.
2. Hardware thread architecture for device drivers; this is a case study on how the API can simplify and make the design more predictable for device drivers.
3. List of papers

5.1 API for hardware threads

This thesis presents an advanced communication and synchronization API at a higher level than usually used in software application design. The API is used in industry [3] as a software bus to communicate between software threads and is named Virtual Communication Bus (VCB). This thesis shows that the VCB can also be used to communicate between hardware threads or hardware and software threads.
The interface provides seven different types of system calls to synchronize and communicate between HW/HW HW/SW and SW/SW. The system calls is: *vcb_init, vcb_connect, svcb_disconnect, vcb_send, vcb_receive, vcb_broadcast and vcb_send_wait* (for more information see paper B).

5.1.1 Syntax notation for a hardware API

The hardware description language in this case is VHDL -93. The application use VHDL -93 and a package, defined as a VCB-API package. This package includes all seven different vcb calls for the hardware application designer.

The semantics is the same for both hard; and software calls, but the syntax is quite different. For a call e.g. *vcb_send* in the software case it is just an ordinary function call like "*ok=**vcb_send**(msg,size, address, priority);"* This call will send a message *msg* to an *address* at a specified *priority*. The same system call is used in the hardware case but the syntax description is different. Below the same system call *vcb_send* is described. The hardware thread sends a message to an address at a specified priority level.

```
case state is
  when SEND =>
    send_args<=send&
    msg_size&
    slot_address&
    msg_prio&
    thread_id;
  vcb_call <= '1';
  state <= WAIT_SEND;
…
```

The hardware application has to manage the state transactions sends or receives in the application code. For more details see paper B and appendix

5.1.2 API implementation

The API is divided in three different parts (see Figure 6) which today constitutes the VCB concept. Previous version of the VCB concept only allowed the software designers to use the VCB concept to synchronize and communicate between software threads *(SW_VCB_API)*.

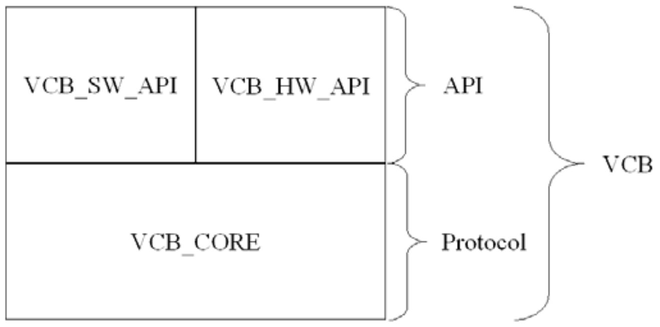


Figure 6 Structure of the VCB

This version of the VCB concept is augmented with an API for the hardware designers (HW_VCB_API) to synchronize and communicate between hardware and software threads. A modification of the protocol for the VCB_CORE concept, the extension of the protocol allows simultaneous access to the VCB_CORE. For more details of the implementation see paper B.

5.1.3 Simultaneous system calls to VCB_CORE

To allow simultaneous system calls from both software and hardware application the VCB-core had to be modified. The modification in the communication core compared to the original was to add a multiplexer for the data path form to the application and the controller block (see Figure 7). The rest of the design is the same as the original.

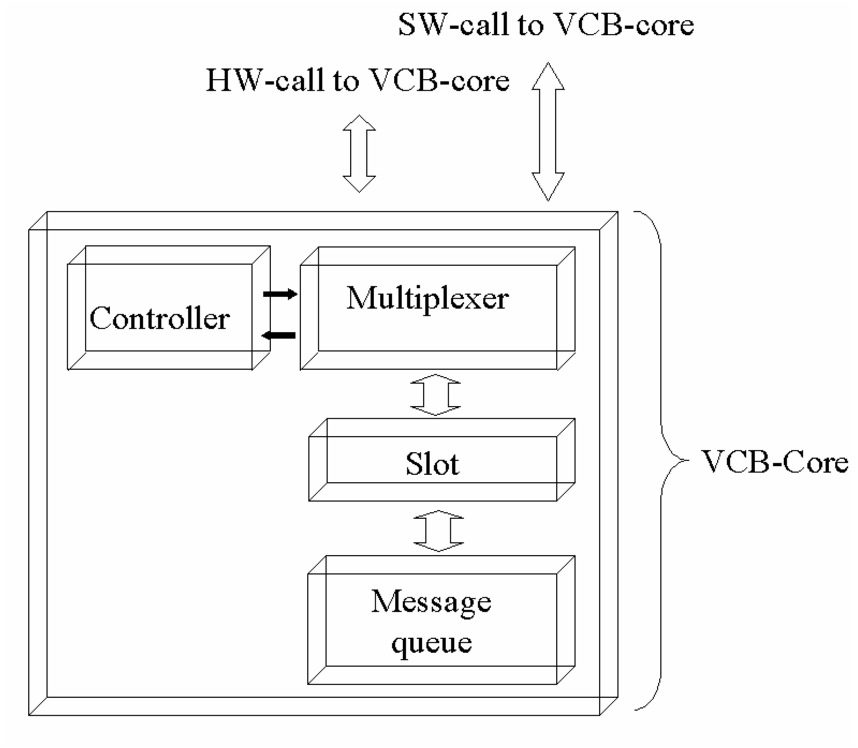
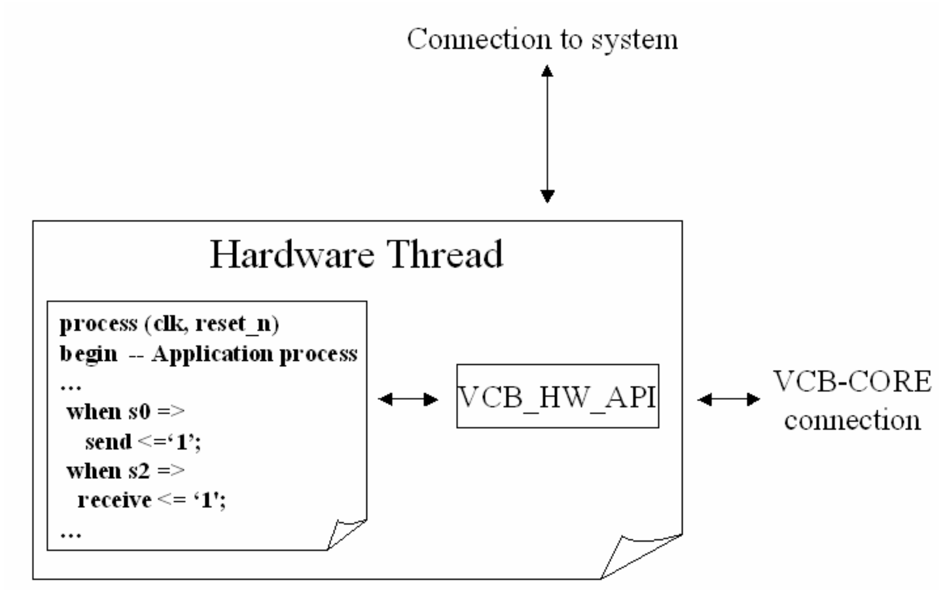


Figure 7 Internal architecture of the VCB-Core

5.2 Hardware thread architecture

This section will define and demonstrate the use of hardware threads in an implementation of a UART device driver. A hardware thread is defined in this thesis to consist of thread code (hardware description language) and a bus interface to shared memory and connection to VCB bus.



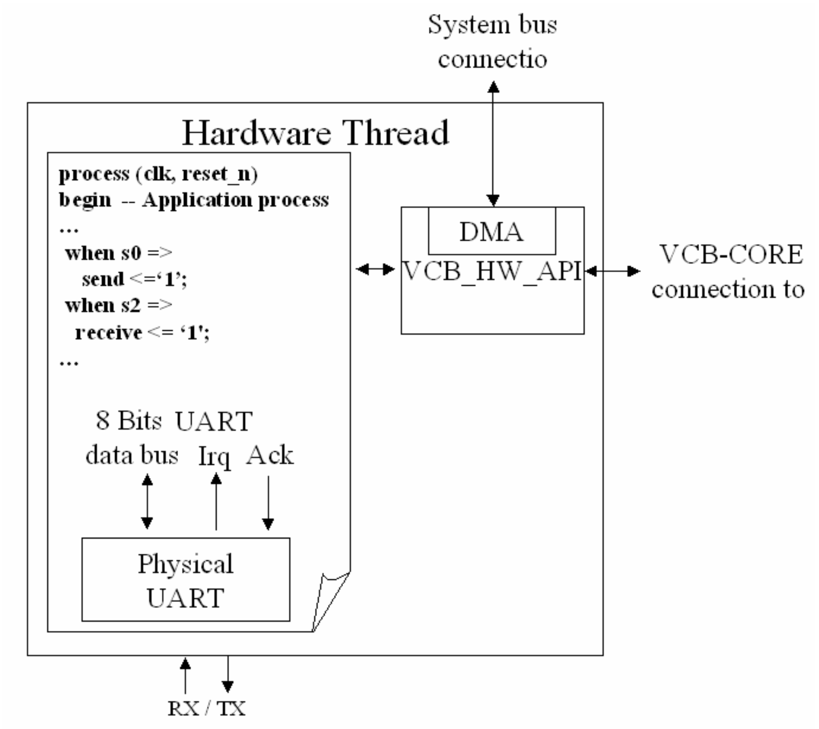
```
process (clk, reset_n)
begin -- Application process
...
 when s0 =>
   send <='1';
 when s2 =>
   receive <= '1';
...
```

Figure 8 Definition of a hardware thread

The thread code defines the behavior of the thread and is in our work given in VHDL.

1) The master bus Interface, *connection to system bus* used to read and write to the message buffers in shared memory.
2) *Connection to VCB-Core*, this is the connection to the hardware scheduler that provides the means to start, stop, and change priority of the software threads. This connection also contains the VCB_API used to communicate through VCB for both hardware and software threads.
3) The hardware thread also manages the functionality included in a driver for a device, *application process*. This is normally written in software but in this case the driver will be implemented in VHDL.

The device driver implementation for the UART in this case has a net list for the UART functionality. All the signals from the UART are routed directly to the hardware thread (see Figure 9). Normally the signals are routed to the system bus through a wrapper adjusted to fit the system bus.



```
process (clk, reset_n)
begin  -- Application process
...
 when s0 =>
   send <='1';
 when s2 =>
  receive <= '1';
...
```

Figure 9 Hardware thread architecture

Another component in the device driver is a DMA Direct Memory Access used to transfer data to/from the global shared memory. The application code for the device driver implements the functionality to send a message to the software every fourth received byte or when a carriage return is received to the UART. The application code also decides when the software application will use the UART. For more details see paper C

5.3 A case study; Implementation of a hardware device driver

Two test cases with different types of implementations, one traditional software solution and the other one is a hardware solution with the VCB-API.

The most common way to integrate an UART component in a system is to use software to transfer data to/from the UART and use interruption the processor kernel for attention when it’s necessary (see Figure 10).

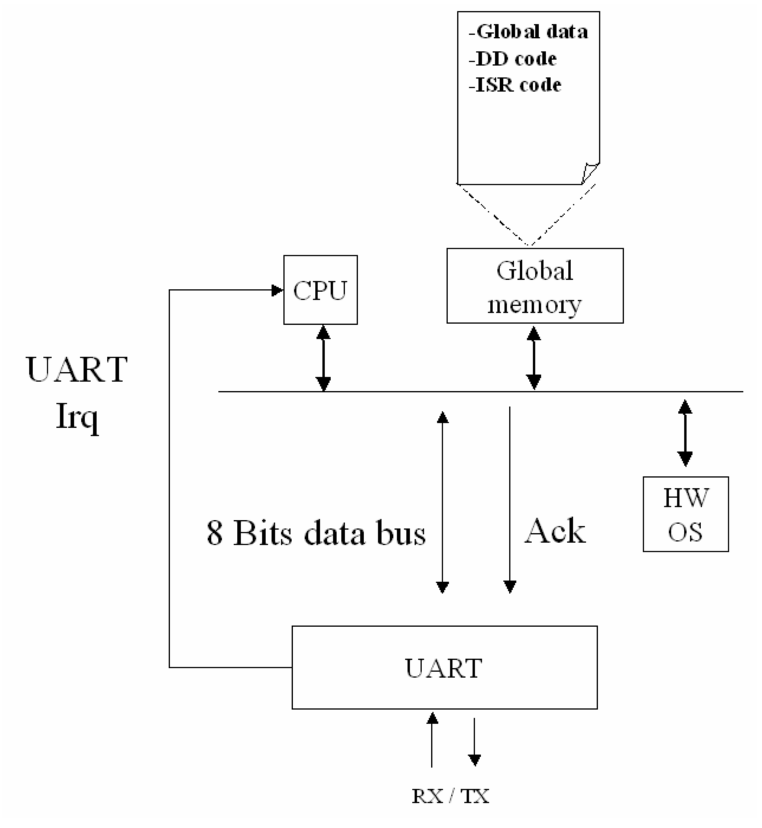


Figure 10 Normally software implementation of a UART device

An interrupt directly routed to the software processor kernel often have the highest priority in the system. This always extends the execution time for the software application and this could cause a missing deadline for a thread in a real time system and thereby make a useless reaction or in the worst case even a dangerous situation.

The VCB-API solution of this problem is to use a hardware thread to manage the UART. This thread is always in "running mode" and never utilizes any execution time from the software threads.

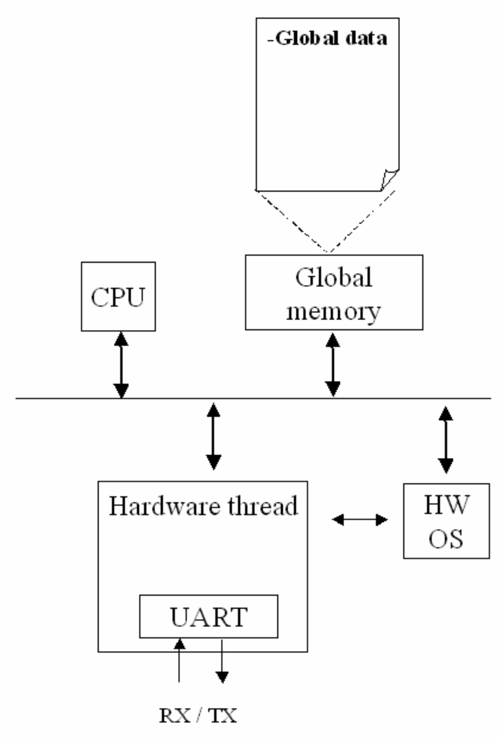


Figure 11 Architecture for the UART device driver

The processor kernel will never be interrupted to manage the UART device (see Figure 11). For more information see paper C.

List of papers

Following papers articles are included in this thesis:

Paper A) **Virtual Communication Bus with Hardware and Software Tasks in Real-Time System**

In *Proceedings for the work in progress and industrial experience sessions*, pages 3 12th Euromicro conferance on Real-time systems, June 2000.

Author(s): Peter Nygren, Lennart Lindh

My contribution: I wrote the paper under supervision of assistant professor Lennart Lindh

Paper B) **Uniform Interprocess Communication interface for Hardware and Software Threads**

In *International Workshop on Advanced Real-Time Operating System Services (ARTOSS)* Porto, Portugal , July 2003. IEEE

Author(s): Peter Nygren, Lennart Lindh

My contribution: I wrote the paper under supervision of assistant professor Lennart Lindh

Paper C) **Implementation of Uniform Communication Protocol and Interface for Hardware and Software Threads and a Device Driver Example**

Submitted for publication

Author(s): Peter Nygren, Lennart Lindh

My contribution: I and assistant Professor Lennart Lindh wrote the paper together.

Co-authored publications:

A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software

In *International Workshop on Advanced Real-Time Operating System Services (ARTOSS)* Porto, Portugal , July 2003.

Author(s): Tobias Samuelsson, Mikael Åkerholm, Peter Nygren, Johan Stärner, Lennart Lindh

My contribution: I was supervisor and review of the paper

6 CONCLUSION

The system architecture with the VCB mechanism has several advantages.

- Increases the real-time predictability.
- Can reduce the software complexity.
- Allows software functions to be moved into an equivalent hardware implementation without the need to rewrite other application software.
- Performance and real time problems can be solved.

It has been shown that a hardware communication and synchronization mechanism together with a hardware API solves several problems.

- External interrupts can be scheduled and managed simultaneously, without interference with the software application.
- Decreased CPU load, since applications can synchronize and communicate directly without CPU involvement.
- Reduced response time for devices, since drivers can be implemented in hardware.

Paper A

Virtual Communication Bus with Hardware and Software Tasks in Real-Time System

Presented at Euromicro conferance on Real-time systems

Stockholm, June 2000.

7 PAPER A

Virtual Communication Bus with Hardware and Software Tasks in Real-Time System

Abstract

The FPGA (Field Programmable Gate Array) of recent years has opened newer design possibilities of moving software into hardware. This paper is studied at two cases of transferring functionality from software into hardware. The paper describes the VCB (Virtual Communication Bus) concept and hardware tasks. The approach with VCB is focused today on existing systems with a main processor and slave processors or DSP (Digital Signal Processors). The first approach is to reduce the system load from the VCB bus and the second phase will be to eliminate the slave processors and move them to hardware tasks implemented in FPGA. Hardware tasks will reduce the response time and make the system more time-deterministic.

Keywords: ***FPGA, VCB, Task, hardware tasks, Real-Time System***

Introduction

Already today an FPGA have 10 million gates, it will not be long before FPGA's have hundreds of millions logic gates on-chip [24]. A FPGA can be programmed by a subset of C [34] or a hardware language such as VHDL [35]. A rule of thumb is that about 100 pages of C code fit into 30 000 gates. The cost of a 30K device is to day under 10 US $ [24]. Compilers today translate ordinary software from code into serial machine code. The hardware compilers translate the source code into concurrent gates, flip flops and memories by means of a synthesizer. The synthesizer during the last 10 years has developed from a simple state machine to behavioral translation. Today the designer can design in hardware design tools or ordinary software tools. With the help of new tools as CoWare N2C™ Design System [27] are dealing with this problem. The first phase will be to eliminate the operating system (OS) load generated from the VCB bus. The system load generated from the bus is an ordinary software task and this functionality could be moved from software into hardware. This will reduce the system load and make more execution time available for application tasks. The approach with VCB is focused today on an existing system (see [27]) with a main processor and slave processors or DSP. The goal of this approach is to eliminate the slave processors and move the software functionality in to hardware tasks implemented in FPGA. This means that a hardware task can consist thousand pages of C code.

Motivation and overview

The objective is to remove the functionality from the VCB bus, implement this functionality in software and move it into a hardware task. The purpose of this is to utilize the true parallelism in the hardware and their bye making available more execution time for the application tasks in the system. The second phase of the project will be to reduce the number of CPU's in a multiprocessor system with mixed architecture of generally micro-controllers and signal processors. The purpose of this is to decrease the response time from the external units and reduce the overhead for the ordinary system to handle communication with external devices. Industry today can reduce the cost and the design time for system architecture, if the system designers uses hardware task instead of CPU's. The cost of a system could thereby be reduced. An example of a commercial system is shown in Figure 12. In this type of system it is possible to eliminate the slave CPU's and move the functionality into hardware tasks.

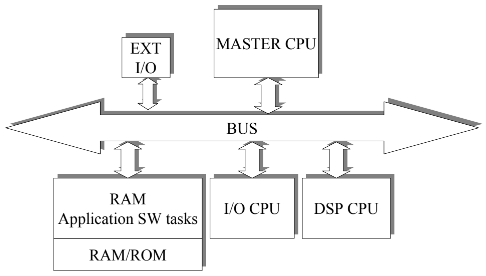


Figure 12 Logical architecture of a common system.

If we move the functionality from the CPU's and replace them with an FPGA and implement the functionality in hardware tasks could we reduce the necessity of mixed architectures in many types of systems *(see* Figure 13*)*. An advantage of using hardware instead of CPU's is that the external I/O can be connected directly into the FPGA. This reduces the communication on the shared bus. The communication possibilities on the VCB bus permit communication between hardware and software tasks in the system.

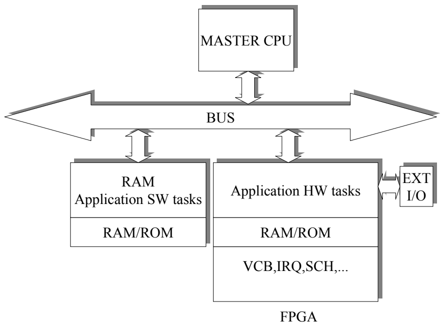


Figure 13 Logical architecture of the proposed system

To moving common software functionality into hardware could results in greater predictability and increase system speed. This new design method gives less complexity and reduces the cost of the whole system.

Phase one VCB implementation

The Virtual Communication Bus is used for inter process communication and for synchronization tasks in the system. Communication between different tasks usually consist some kind of message passing mechanism such as mailboxes, pipes, message queue. The VCB bus is such a message passing mechanism and the bus allowed task to task communication locally on one CPU and between several different CPUs in a system. The tasks could be either an ordinary software task or it a hardware task, the interface for the tasks will be the same.

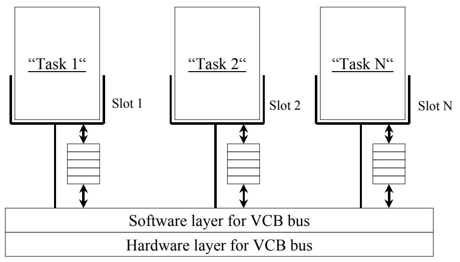


Figure 14 Logical architecture of a system with the VCB bus

The VCB bus is divided into two layers, the upper layer being the software implementation of the bus. In this layer the system provides support for different types of functionality from the bus. The other part of the bus consist the base primitives. Those primitives are implemented and integrated in the FPGA. When tasks want to communicate on the VCB the task hade to allocate one VCB-slot and their bye been the task will be connected to the virtual bus. The VCB connects the system and makes possible communication in two different ways, *synchronous* or *asynchronous*. When a task is connected to a slot it can communicate with all the other tasks in the system. The Send and receive communication primitives is the type of functionality mostly frequently used in the VCB bus. Other functions the bus could support are, for example *broadcast, send and wait, multicast, subscribe*. To reduce the system load from the OS (Operating System), the first step will be to reduce the execution time of the subscribe function. The subscribe function is a *Server<->Client* concept in which the *"Server"* is the functionality which handles all mail requests from other *"Clients"* in the system. The server runs each time *T* to decrement each individual timer *T*. When the time has expired for one or more *"Clients"* the *"Server"* sends a mail to every *"Client"* in the request list (see Figure 15*)*.

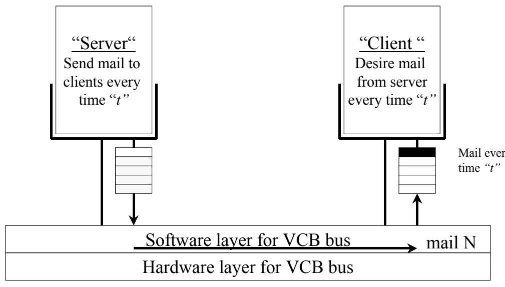

"Server"
Send mail to clients every time "t"
"Client"
Desire mail from server every time "t"
Mail ever time "t"
Software layer for VCB bus
mail N
Hardware layer for VCB bus

Figure 15 "Server" "Client" concept for subscribe function

The mail system uses some shared memory area either globally or locally. In some cases the local location is a better solution because it minimizes the accesses on the global common system bus and improves the performance of the mail system.

Phase two hardware task

The purpose of removing CPU's from a system is to increase the performance and predictability of some functionality. This type of implementation gives many advantages. One of the biggest differences is the performance as a hardware task runs in true parallel mode giving higher speeds as compared with a corresponding software implementation. Another advantage is the predictability of the hardware, the *max/min* execution time for the function being definable on the clock cycle level. The complexity and size of the code are reduced when hardware tasks are used. The design space increased if the designer used hardware tasks instead of ordinary software implementation. Hardware tasks can reduce the numbers of CPU's because of the possibility of using the VCB bus for communication and when functionality moved from the CPU into hardware tasks. The hardware task could be one instance or it could be divided into many small units in the same task. When using CPU's in the system *(see* Figure 12*)* it is difficult to handle the high frequency of the external I/O. The overhead for the interrupts reduces the CPU performance. The hardware task could handle the external I/O interrupts directly from the external I/O units without any overhead from the OS. The hardware task could also handle concurrent processing of the information in true parallelism. Results from the hardware tasks are sent directly to the software task at the master CPU via VCB bus. Device drivers for handling external I/O are not necessary

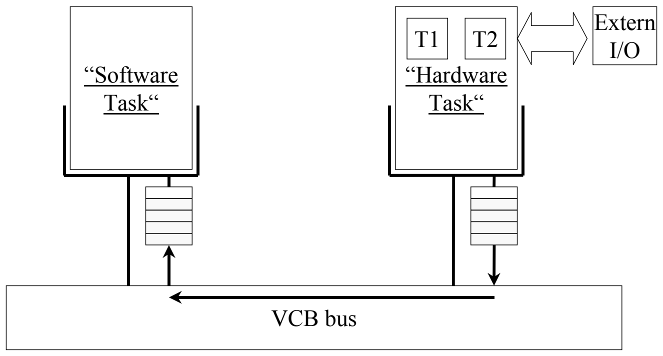


Figure 16 Logical picture of a hardware task

Larger FPGA's gives the possibility to use more of the memory on chip instead of common system RAM. This gives an improved performance of the task in the FPGA.

Advantages with Hardware Tasks

The implementation of the same function code in hardware is considerable different from its implementation in software It is easier to predict real-time behavior in hardware. Min and max times can be verified with tools. In software it is difficult or almost impossible. The background is that software uses the shared resources such as CPU, ALU but in the case of hardware the tasks in most case don't use any shared resources. The hardware can use the same resource for different "tasks", but this can be scheduled offline. In hardware it is also easier to deal with asynchronous events (such as interrupts). In a software solution an interrupt interrupts the entire system, in a hardware task it interrupts only the interrupt function in the system. Performance of a hardware task is much higher than of a software task, if the function can be held inside the same chip. The parallelism in hardware is very high, in a 30000 gates FPGA it is 30000 concurrent elements. Hardware tasks need no overhead, such as operating system, device drivers for interrupt. The response time from a hardware task is much shorter than from a common software task. FPGA hardware gives flexible hardware architecture. For example the number of ALU units, interrupt pins, I/O is configurable. In a processor there are nearly always overhead and restriction resources. For example if you need an floating point unit, it will be provided, or if you need more interrupt lines there must be added. When you must access a unit outside the CPU the response time will increase.

Discussion

- New hardware design methods [27] and higher capacity FPGA's create new possibilities of removing CPU's from ordinary system design and replacing the CPU's.
- The integration of heavy regulates algorithms into hardware tasks and the possibility of transparent communication between soft and hardware tasks gives new dimensions in this new type of architecture.
- New FPGA types with more internal memory need less external memory and will increase the speed of the function. In many cases the time behavior of hardware tasks will be much more predictable.
- As the price of FPGA's reduced by 50% each 18 month, the cost of architecture the same functionality in a system is almost the same.

Paper B

Presented at International Workshop on Advanced
Real-Time Operating System Services (ARTOSS)

Portugal, July 2003

8 PAPER B

Uniform Interprocess Communication interface for Hardware and Software Threads

Abstract:

A standardized communication and synchronization interface at the thread level abstraction is required to speed up and simplify the system design. This paper describes a novel implementation of a uniform communication interface for hardware and software threads. We call the communication interface for Virtual Communication Bus (VCB). This communication interface contains an Application Programming Interface (API) with seven different primitives {*init, connect, disconnect, send, receive, broadcast, send_wait*} those primitives are used in both the hardware and software cases. The communication interface is currently used in a system implemented in a single *Xilinx* device xc2v1000 [24]. This system contains one processor kernel from Xilinx [24] and a real time kernel [2]. These components and the VCB are connected through a shared system bus *core connect* [32].

Keywords: VCB, hardware and software threads.

Introduction

A system design could be described and implemented in a structural and hierarchic manner with threads communicating concurrently**.** Software systems have for a long time been developed in this way. It is now possible to design an entire system containing one or several processor kernels, custom-designed hardware, intellectual property components and an amount of programmable logic on a single chip. A system design could be a pure software solution, which might not be acceptable because its performance is inadequate or a pure hardware solution which is too expensive. In this case system designers need some form of communication interface at the thread level to make changes between software and hardware implemented threads and to avoid the need to rewrite new HW/SW interface each time the new partitioning is performed. In the software case the threading of an application is no problem because each modern real-time operating system (RTOS) supports a communication interface. The communication interface often performs some kind of message-passing by means of mail boxes, slots, pipes, etc. Unfortunately this feature is not supported in a natural manner in the most popular hardware description languages (HDL's) such as VHDL and Verilog. The HW/SW interface which transfers data between hardware and software components basically reads and writes hardware registers and memory locations. This interface could either be implemented in a normal program flow or in a small software layer connected to an interrupt routine (called a device-driver) [7]. Another solution could be the use of different types of hardware support [9][8]. Our objective is to develop a communication interface at the thread level and to introduce the possibility of transforming the system tasks to either a hardware or software thread containing the interface. The interface is to provide a uniform independent message-passing mechanism using a specified application programmable interface (API). This paper describes the API functionality and the particular implementation of this interface, the virtual communication bus VCB.

Virtual Communication Bus (VCB)

The VCB interface contains two main components a *slot* and a *message queue* (see Figure 17). The *slot* is used by the application threads to connect and communicate with other threads connected to the bus. The *slot* is a system resource allocated by a particular thread and cannot be used by other application threads. The slot contains information *owner id(entity),* if the *slot* is *open* or *closed*, and the default priority of the application thread. In this particular implementation there are four slots but this number is arbitrary.

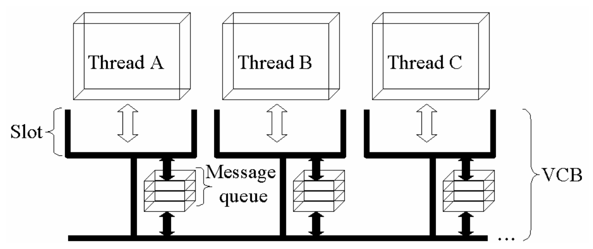


Figure 17 VCB structure

The other main component is the *message queue* which contains the particular message index and the message index number used to protect and point out the particular message buffer in the shared global memory. The message index is sorted by the sorting algorithm specified, these being based at PRIORITY or FIFO orders. The sorting algorithm must be specified when an application thread allocates a *slot.* The number of message indexes in this implementation is sixteen but the number could be increased or decreased.

Functional description of VCB

The VCB interface can be considered as blocks (see Figure 18), the first block (VCB-API) including the Application Programmable Interface. This block consists of three different layers of abstraction {*API, API_Slot, API_Basic*} these implementing well-defined functionality at each level of abstraction.

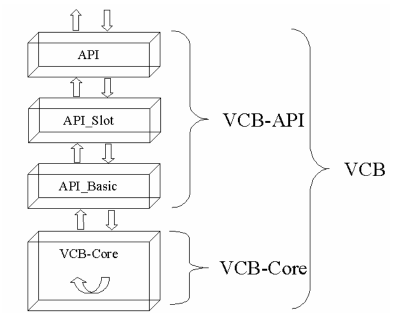


Figure 18 VCB structure

The VCB-API is mapped into two different implementations languages, either C-code used by the software threads, or VHDL used by the hardware threads.
Block number two, VCB-Core manages the lowest level of the interface functionality, this functionality including the authentication of the owner of the particular *slot,* whether it is opened or closed and the sorting of the index numbers of the messages. This functionality is mapped to a pure hardware implementation in VHDL.

API

The functionality of the VCB-API will be described below, beginning with the API layer. The *API* layer is the point of entry to the VCB interface and the most abstract. This layer implements the system calls {*init, connect, disconnect, send, receive, broadcast, send_wait*}.

- vcb_init:

The vcb_init system call resets the entire VCB-interface and removes all owner information and messages in the interface. This system call should only be used at system start-up.

- vcb_connect:

This system call connects a thread to VCB, enabling the owner of the slot to read or write messages to the queue. The caller must specify the name, slot number, message sorting algorithm and default priority of the thread which the caller intends to use.

- vcb_disconnect:

This breaks the connection to the VCB. The call must be managed by the owner of the slot. All messages will be removed and the slot will be released, thereby becoming available for use by other threads.

- vcb_send:

This system call is used to send a message to another thread. The user must specify a pointer to the message, the quantity of data to be transferred, the receiver of the message, the priority of the message and the sender identity. The sender function checks if the parameters are correct and the return value is either VCB_OK or VCB_ERROR.

- vcb_receive:

This system call is used to request permission to read a message from the VCB, The caller must specify the parameters associated with the message, a pointer associated with the mail received, a pointer to the bytes transferred, the sender of the message, and an indication if the caller intends to wait for mail if the message queue is empty. The last parameter specifies a call-back routine. The function returns either VCB_OK or VCB_ERROR.

- vcb_broadcast:

This system call sends message to all threads connected to the VCB. It uses multiple calls to system call *vcb_send.*

- vcb_sendwait:

The vcb_sendwait system call sends a message to a specified thread and waits until a response is received from the thread called or when a preset time expires. The return value is either VCB_OK or VCB_ERROR.

API_Slot

The *API_Slot* layer manages different parameters and also implements mixed calls of send and receives to build other higher level functions such as *send_receive, broadcast.* This layer incorporates a function for locating names associated with a slot number or slot numbers associated with a name. It also manages the copying of the message associated with the index number generated from the VCB-core layer.

API_Basic

The *API_Basic* level accessed from the *API_Slot* layer, is the lowest layer of the VCB-API interface. This layer implementing the lowest primitives and accesses the vcb-core. The different functions in this layer assign the input parameters to a specified bit structure associated with a particular system call. The different system calls manage information concerning the slots and the message queues. The caller can create, close, open or obtain information about a particular slot and can also post a message to, or remove a message from the message queue connected to the slot concerned.

- Init:

This call initializes the entire VCB system to a well-known state and prepares it for use. It should only be used at start-up.

- Allocate:

This call allocates a slot for the thread called and specifies that the messages in the message queue should be sorted by a FIFO or priority order. It saves the identity of the owner of the slot if the caller is specified to inherit the priority from the arrival message and saves the information that the caller is software or a hardware thread. All this information is specified by the caller.

- Open:

This call opens a message queue. It enables other threads to post new messages to the queue. The thread performing this request must be the owner of the slot.

- Close:

This system call closes a message queue. It prevents the posting of new messages to the queue. It should be used before the deletion of a queue as this call returns all the unread messages in the queue. The thread must be the owner of the slot.

- Delete:

This call erases all the information (in the on-chip memory) about the slot and the message queue. It allows other threads to allocate this particular slot and message queue. The caller thread must be the owner of the slot.

- Put:

This call allocates a message buffer in the message queue to slot specified and returns an index number to the caller. If the message queue is full, an error message will be returned to the caller.

- Put_ready:

This call is used when the CPU or DMA has copied the message to the message buffer. The VCB-core must be informed when this is done. The message is sorted in the sorting algorithm specified and the owner of the slot can then read the message.

- Get:

This call returns an index number to the buffer in which the first message in the queue is placed for the given slot. If the owner hade specified "wait for mail when the message queue is empty", the thread is set to the wait state until a message arrives at the queue. Otherwise, a message NO-MAIL will be returned.

- Get_ready:

This call informs the VCB-core that the index number from the *"get"* call has been read. If the owner of the slot has specified "priorities inherit", this call changes the priority back to the default priority.

- Info:

This system call gives information concerning the slot and queue. The caller must specify the type of information and the return value contains this. This information is used at higher levels in VCB when different program execution decisions are to be made.

Differences between Hardware/Software threads in VCB-API

There are certain minor restrictions in the hardware implementation of VCB-API. The priority inherit functionality is not available for use by hardware threads and it is not possible change the priority dynamically or block/restart a hardware thread. The VCB-API interface implemented in VHDL contains, however, the same set of system calls as for software threads to permit communication with the VCB.

VCB-core

The VCB-core consists of four different parts as shown in Figure 19. This block is used to implement the VCB-core functions and to manager the globally shared system memory. This contains all relevant messages and these can only be accessed through the VCB.

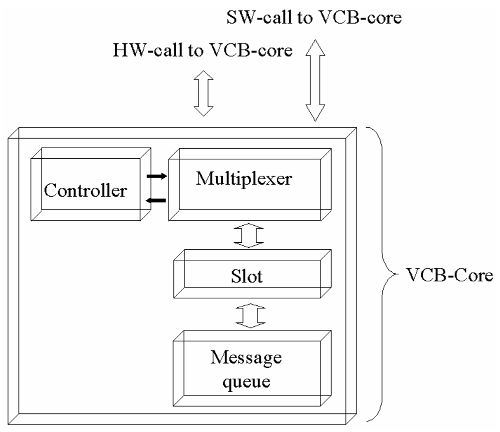


Figure 19 VCB-core architecture

The *multiplexer* block is used to manage different types of accesses, either HW (Hardware) or SW (Software) calls. The different system calls are routed through the multiplexer to the *slot* block. The *controller* block determines which of simultaneous accesses to the core is to be given precedence and the other two blocks manage the lowest level of system calls in the VCB-API (see Figure 18).Multiplexer

The multiplexer incorporates an important part of the VCB-core functionality. To avoid restricting the use of VCB-API to software threads only, the VCB-core multiplexer is augmented with a request-driven component which controls the communication with the VCB-core. The multiplexer enables hardware threads to communicate with software threads via the VCB-core. The multiplexer acts as a sort arbitrator granting access to one of the units if more than one are simultaneously requesting access to the core. The objective is to obtain a common interface for communication, irrespective of whether the caller is implemented as software threads or as hardware threads.

The state transitions take all relevant signals into account in order to preserve the protocol of the VCB communication. The owner (the unit granted access to the VCB-core) must remain owner of the entire transaction. All state transitions and their conditions are illustrated in (see Figure 20), and are explained below.

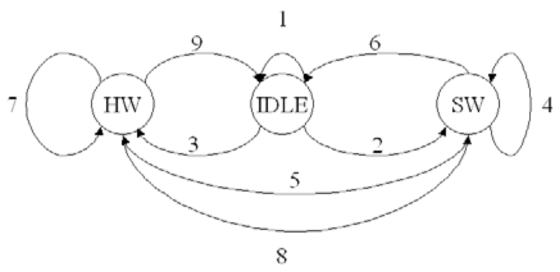


Figure 20. State transitions s for the different transactions

[IDLE -> IDLE:]

No unit requests communication.

[IDLE -> SW:]

A SW-thread request for VCB-core communication.

[IDLE -> HW:]

A HW-thread wishes to utilize VCB-core functionality.

[SW -> SW:]

SW-thread transaction in progress

[SW -> HW:]

The previous SW-thread transaction has been finished and a HW-thread is waiting to gain access.

[SW -> IDLE:]

The previous SW transaction has been finished and there is no pending HW or new SW request.

[HW -> HW:]

HW-thread transaction in progress.

[HW -> SW:]
The pending SW-thread request is accepted for processing when the current HW-thread transaction is completed.
[HW -> IDLE:]
There are no requests from the units and no current HW-thread transaction.

Slot
The VCB-core functionality is divided into separate blocks (see Figure 19), *slot* and *message queue* to reduce the complexity of the VCB-core functionality. All system calls enter the *slot* block, and the calls {*allocate, open, close, delete*}, determine the slot information (see Figure 21) they do not use the functionality in the *message queue.*

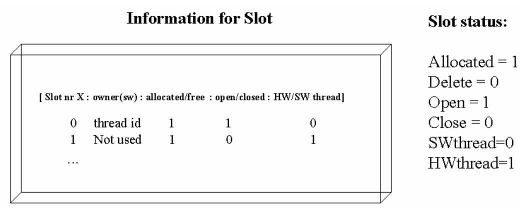
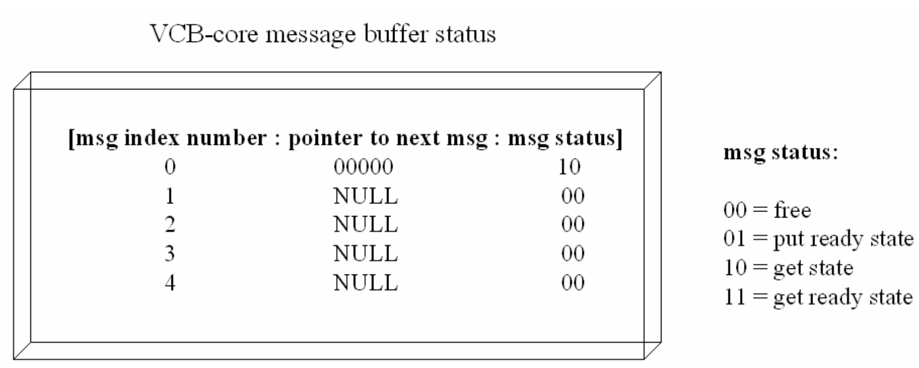


Figure 21 Slot information in on-chip memory

The other system calls use the functionality defined in the *message queue* block. Data is first entered into the *slot* block which determines if the *slot* is open or closed. If the slot is open, the *slot* block routes the system call information into the message block and then sends a call, (*put, put_ready, get, get_ready,* or *info*) to the *message queue*. The return value from this call is passed by the *slot* block to the VCB-API together with a supplement to the system call concerned.

Message queue

The different index numbers of the messages located in the global shared memory are managed by this block. The system calls {*put, get*} return an index number from the message queue associated with the relevant *slot.* This index number identifies the particular messages in the buffer. The different system calls {*put, put_ready, get, get_ready*} transform the particular message buffer into different states (see Figure 22) and the messages posted are sorted in a selected algorithm, FIFO or PRIORITY, defined by the system call *allocate*.


VCB-core message buffer status

msg index number	pointer to next msg	msg status
0	00000	10
1	NULL	00
2	NULL	00
3	NULL	00
4	NULL	00

msg status:

00 = free
01 = put ready state
10 = get state
11 = get ready state

Figure 22 Message queue information and status in on-chip memory

The system calls must be received in the correct sequence, e.g. to send a message the sequence must be *put* follow by *put_ready.* These two system calls transform the particular message to the *get* state. When the message is in the *get state,* it is ready for delivery to the owner of the queue. To read a message, the owner must make a *receive* call which generates a sequence of calls, the first of which is the *get* call. The return message from this call will be given the first index number in the queue with the *get state* status. This call also transforms the index number to the *get-ready* state. The next system call will be *get_ready* to complete the read operation.

System architecture

The system architecture of the complete implementation includes a software processor kernel *CPU* from Xilinx [26]. This kernel executes the software threads in the system. The *real time unit* is an accelerator which manages the software thread scheduling, interrupt and semaphore handling in addition to time management control in an external hardware component. The OS accelerator supports two different scheduling algorithms, fixed priority scheduling and Round Robin.

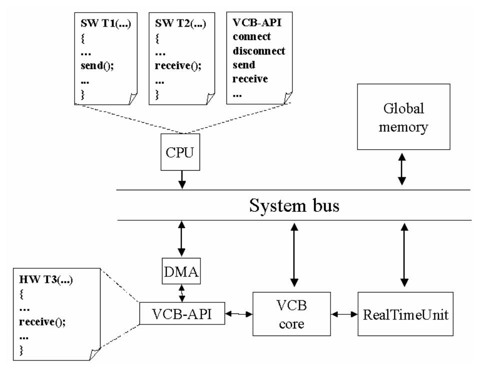


Figure 23 Hardware and Software architecture

The memory of this system is in two parts, an on- chip block RAM reserved for program code and local data structures and an off-chip memory [36] which contains all message information in the VCB interface.

All devices in the system are connected through the IBM CoreConnect [32] bus. The on-chip peripheral bus (OPB) is designed for easy connection of on-chip peripheral components. The OPB is a fully synchronous bus which functions independently at a separate level of the bus hierarchy. The processor core can access the slave peripherals on this bus through the processor local bus (PLB) to the OPB bridge unit which is a separate core.

Application code example

The VCB concept is implemented in both software and hardware. Some parts are implemented in hardware to achieve higher performance and/or to attain a higher level of abstraction.

Sending and receiving messages:

The sender and receiver threads (HW/SW) must allocate a slot before communication. The procedures described below for making a send/ receive call must be followed.

The sending procedure

1. The sender requests permission to send a message to a slot and receives a message index number allocated to the message by the VCB core
2. The sender copies the message to the message buffer. The VCB core must be informed when the copying is completed.
3. The VCB core sorts the message queue and the receiver may then read the message.

The receiving procedure

1. The receiver requests permission to read a message from its slot and receives a message buffer reference to the first message in the queue from the VCB-core.
2. The receiver must copy the message from the message buffer, if the message is to be saved. The VCB core must be informed when the copying is completed.

The VCB core transforms the state of the message queue to "available" and the message buffer is then available for new messages.

Receive service call in Software thread (C-code)

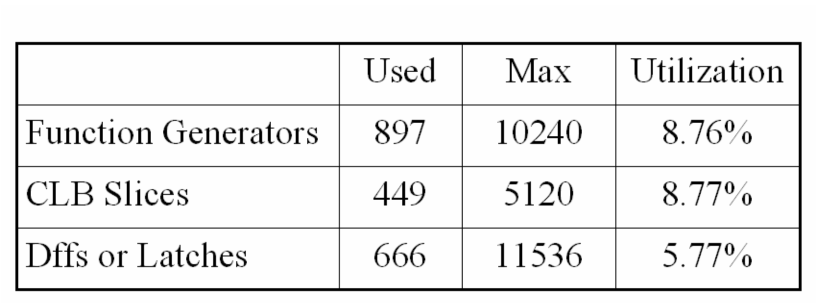
```
-- T1 Connect to the slot ------------------------------
retval=vcb_connect("t1",SLOT_1, INHERIT,FIFO_SORT);
-- Other code
...

-- Thread T1 Send  msg -----------------------------------------
if( vcb_send("Hello word",strlen("Hello word" ),SLOT_2,T1)!=VCB_OK)
   error handler
}

-- T2 Connect to the slot ------------------------------
retval=vcb_connect("t2",SLOT_2, INHERIT,
FIFO_SORT);
-- Other code
...
-- Thread T2 Receive msg -----------------------------------------
if(vcb_receive(msg,&size,SLOT_2,&sender,w4m,print_string)!=VCB_OK)
   error handler
}
```

Receive service call in Hardware thread (VHDL-code)

```
-- T1 connect to the slot 1-------------------------------
-- do a CONNECT
when CONNECT =>
  vcb_call <= '1';
  vcb_args(nroffunc+defaultarg-1 downto defaultarg) <= VCB_FCONNECT;
  vcb_args(nrofslot downto 0) <= myslot & msg_sort_prio;
  state <= WAIT_CONNECT;

-- Other code
-- Send msg to thread
when SEND =>
   vcb_call <= '1';
   vcb_args(nroffunc+defaultarg-1 downto defaultarg) <= VCB_FSEND;
   vcb_args(nrofslot+1 downto 0) <= destslot & "10";  -- slotid, msgprio
   vcb_sendmsg <= X"ABBA_ACDC";
  state <= WAIT_SEND;

-- Other code

-- T2 connect to the slot 2--------------------------------
 when CONNECT =>
    vcb_call <= '1';
   vcb_args(nroffunc+defaultarg-1 downto defaultarg) <= VCB_FCONNECT;
    vcb_args(nrofslot downto 0) <= myslot & msg_sort_prio;
    state <= WAIT_CONNECT;

-- Other code

-- T2 Receive msg ----------------------------------------
 when REC =>
 -- signals to RECEIVE
    vcb_call <= '1';
    vcb_args(nroffunc+defaultarg-1 downto defaultarg) <= VCB_FRECEIVE;
   vcb_args(0) <= VCB_WAITFOREVER;
   state <= WAIT_REC;
```

Result

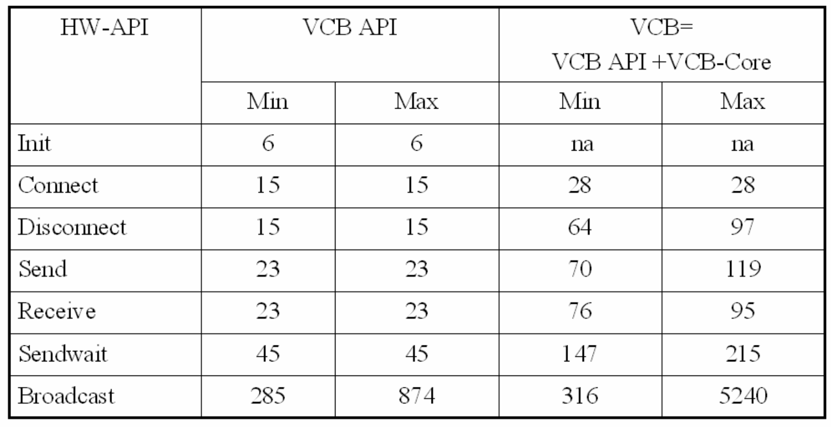
The hardware implementation of the VCB-API is not fully optimal and could be further optimized in certain respects.



	Used	Max	Utilization
Function Generators	897	10240	8.76%
CLB Slices	449	5120	8.77%
Dffs or Latches	666	11536	5.77%

Figure 24 VCB-API physical size in slices

If the design of the VCB-API and the VCB-core is made more parallel, its size will be increased. The design, including only the VCB-API, already has the footprint (see Figure 24) of a 1M-gates device [24].



HW-API	VCB API		VCB= VCB API +VCB-Core	
	Min	Max	Min	Max
Init	6	6	na	na
Connect	15	15	28	28
Disconnect	15	15	64	97
Send	23	23	70	119
Receive	23	23	76	95
Sendwait	45	45	147	215
Broadcast	285	874	316	5240

Figure 25 Timing diagram for VCB-API and VCB in clock cycles.

The timing would be more deterministic if the design were more parallel. A redesign of the VCB-core could reduce the variation in the number of clock cycles to give a more deterministic behavior (see Figure 25). This, however, would also result in an increased size. If a particular application should demand a more deterministic behavior, a redesign is possible.

Conclusion

- In this paper we present a novel uniform interface between hardware and software threads. It shows how to construct a combination of software and hardware threads at the same level of abstraction.
- The interface conceals the lower level of hardware/software communication and a system designer could reuse the different application threads without rewriting the hardware/software communication interface for each new partitioning.
- The interface also permits the designer to reuse the complete software or hardware thread or the complete design. The interface currently offers the same semantic behavior in both cases but with different notations of the API. This is because of limitations in the VHDL-93 languages which do not allow signals on one level of abstraction to affect signals on another level of abstraction.

Paper C

A hardware device driver implementation using an Application Programming Interface for Hardware and Software Threads

Submitted at DATE ,

Munich 2005

9 PAPER C

A hardware device driver implementation using an Application Programming Interface for Hardware and Software Threads

Abstract

The main motivation for this work is to utilize the enormously powerful characteristic of hardware parallelism to move functions from software to hardware in embedded systems. This study conceptualizes a hardware-software communication and synchronization mechanism at the thread level abstraction. We call the communication and synchronization mechanism Virtual Communication Bus (VCB). The hardware based VCB controller has been used by software threads for some years, but with this extension also allows hardware threads to use VCB. This mechanism together with hardware functionality is used to enabling hardware independent access to external I/O devices. Results during this work are an implementation of one Application Programmable Interface (API) for hardware threads. We also achieve zero overhead and 36 times faster response time through the additional mechanisms for hardware thread API. The articles validate the concept at a new developed printed circuit board (PCB) prototype with a design in one chip (XILINX FPGA). This chip contains one single processor PowerPC405 processor and about 800 000 programmable gates.

Introduction and background

Software has for many years used standardized communication protocols between software threads.[1][4][22] The last year's new FPGA (Field programmable Gate arrays) devices from XILINX [24]and Altera [25]have opened up for easy implementations of a mixture between hardware and software components. Components in software are often called threads, tasks or process. Hardware components can be called block, task, process or thread, in this article we call software and hardware components threads. The hardware thread could be a combination of parallel, sequential and conditional execution of operations and the software threads are by definitions sequential. A standardized API on HW/SW thread level and the programmable FPGA technology open up for easier technology choice between hardware and software threads. Standardization gives less misunderstanding, easier reuse of components and also facilitates design for reuse of complex hardware threads. Attempts to solve some of the real time problems related to the communication between hardware and software thread are given in [1][4][17], where an interface between HW/SW is designed and implemented. This article deal with the real time problems, related to the communication between the software application and an ordinary external I/O device universal asynchronous receiver / transmitter (UART).

System overview

A common architecture for a computer system contains a processor unit CPU, an operating system, peripheral devices and an application. In Figure 26 A, the common system and device drivers are in software. Another approach for system architecture is shown in Figure 26 B, were the architecture includes the same amount of functionality, but the shared processor only runs the application code. Operating system functionality and the device driver management is removed from the software architecture and implemented in concurrent hardware units. This is for utilizing the parallelism to CPU and the massive concurrency in hardware.

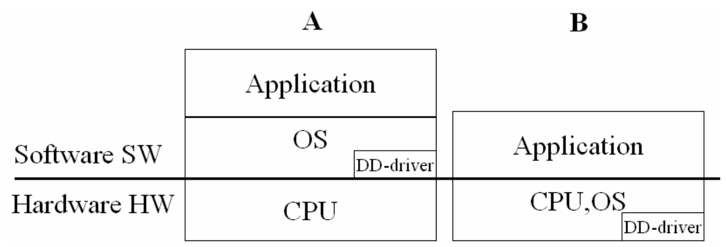


Figure 26 A and B; Common approach and the papers approach for system architecture

To understand the architecture we first describe different hardware components in this work.

VCB concept

The VCB is an inter process communication (IPC) system implemented in hardware [11]. Like any other IPC system the VCB is used by threads to communicate information. This is done by writing to and reading from slots that are “connected” to the “virtual bus”. The VCB can provide communication between threads on a single processor system as well as on a multi processor system. In this work we use single processor systems. The VCB (see Figure 27) contains a logical structure with three main components; an *application programmable interface* (we call VCB-API), a *slot* and a *message queue*. To each slot a hardware or software thread can be connected and communicate with other slots without knowing if they are in hardware or software. In Figure 27 there are three slots, two software threads and one hardware thread. A message transfer contains a specified amount of data and has a fixed maximum length specified in the configuration of the interface. All messages are stored in the global system memory and could only be accessed by using the VCB.

VCB-API is an interface for the service calls to the VCB bus and is standardized for both soft/hard API, se next section.

The ***slot*** provides the abstraction of a communication channel as a data structure and must be allocated and initialized. The slot is a system resource allocated by one thread and cannot be used by other application threads. The data structure for a slot contains information about the *owner id(entity),* if the *slot* is *open* or *closed*, and the default priority of the application thread.

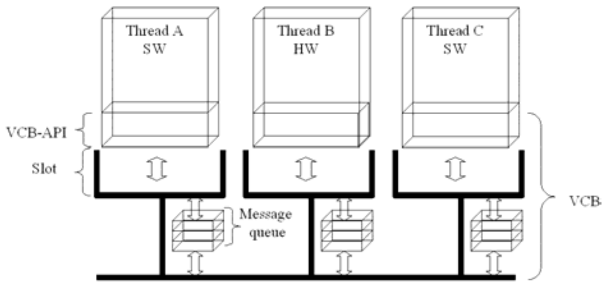


Figure 27 VCB thread communication

The message queue contains the message to the thread. A specified algorithm sorts the messages in either PRIORITY or FIFO order, the sorting algorithm must be specified when a thread allocates a *slot.* A thread can inherit highest priority from the messages in the queue to avoid priority inversion. 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 and a sender can wait a specified time if the buffer is full. All this is handled by hardware.

Real time Scheduler in Hardware

In the system a thread scheduler is also implemented in hardware and we call it in this paper for a (RT-SCH). The VCB use the scheduler to change priority and block or allow the software threads. Also the RT-SCH works in true parallel to the CPU and the other hardware units. It is called RTU (Real-Time Unit) or Sierra in our previous research, for more information see [2][11].

VCB architecture

The logical view of VCB (figure 3) is divided in two different layers. The first layer is divided in two technological dependent interfaces; one for hardware VCB_HW_API and one for software VCB_SW_API, They represent the interface to the software/hardware communication mechanism. The second one is VCB_core used both by hard and software threads and contains the message passing protocol. The VCB_API is technology dependent, for software threads it is designed in C and for hardware threads in VHDL.

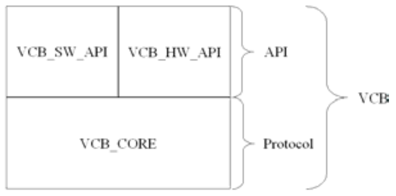


Figure 28 VCB structure

The VCB_core have three connections; HW treads API (point to point), system bus and the real-time scheduler (RT_SCH). It contains three components; multiplexer, slot controller and message queue controller.

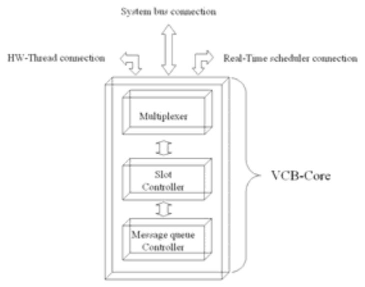


Figure 29 VCB_core architecture

Connections to the VCB_core are the

- **System bus connection:**
  Only for software threads communication. The VCB is a slave on the system bus and the CPU reads and writes to the memory mapped registers file.
- **Hardware threads connections:**
  Only for the hardware threads. Point to point to the hardware threads without using the system bus.
- **Real-time scheduler connection:**
  Used to start, stop and change priority of the software threads. The communication is done point to point without using the system bus. Hardware threads do not need any scheduler or priority change, because they are always running.

The VCB_core manages all information about the VCB status. The core is designed with three different components.

VCB_core components the

- **Multiplexer**
  Acts as a form of arbitrator to prevent more then one VCB_API call to be served by the slot controller. The calling threads can not be interrupted during the entire VCB_core transaction. To the Multiplexer is all HW_VCB_API and system bus connected.
- **Slot controller**
  All system calls enter the *slot controller from Multiplexer*, and the calls {*allocate, open, close, delete*} can be handled by the slot. Other system calls must be manages by the message queue controller [5]. The slot controller also manages to inform the hardware based real-time kernel to start/stop and change priority of the software threads.
- **Message queue controller**
  Handle the message queues located in the system memory and sorted in a selected algorithm, FIFO or PRIORITY. This is done with index number identifying the particular messages. The index is used by VCB_API to inform the thread of the address to the message [5].

Application Programming Interface for software and hardware threads

This chapter gives a brief description of the application programming interface. The application programmer use the interface containing those different system calls, *vcb_init, vcb_connect, vcb_disconnect, vcb_send, vcb_receive, vcb_broadcast and vcb_send_wait* . The VCB_API has technology mapped API in C for software threads and VHDL for hardware threads. Table 1 gives a summary of all the system calls. Further in this chapter a small example is presented of one soft- and hardware thread communicating through VCB. This small example will show how to use *vcb_connect* and *vcb_send,vcb_receive* in an application.

Table 1, service calls for VCB

SVC	**Description**
Vcb_init	Resets the entire VCB-interface and removes all owner information and messages in the interface.
Vcb_connect	Connects a thread to VCB, enabling the owner of the slot to read or write messages to the queue. The caller must specify the name, slot number, message sorting algorithm and default priority of the thread, which the caller intends to use.
Vcb_disconnect	Disconnect the thread from the vcb_slot. All messages will be removed and the slot will be released.

Vcb_send	A thread sends a message to another slot. The user must specify a pointer to the message, the quantity of data to be transferred, the receiver of the message, the priority of the message and the sender identity.
Vcb_receive	A thread requests the permission to read a message from the message queue.
Vcb_broadcast	This system call sends message to all threads connected to a slot. Multiple use of vcb_send will be used to implement this call.
Vcb_sendwait	The vcb_sendwait system call sends a message to a specified thread and waits until a response is received from the thread called or when a preset time expires.

The sender and receiver thread must allocate a slot before communication (vcb_connect). Then the procedures below must be followed.

Syntax description of two service calls, vcb_connect and vcb_receive in C code.

Description of vcb_connect:

extern int vcb_connect(const char *name, int conn_slot_id, int prio_inher, int prio_sort);

Arguments:

```
name                    /* ASCII name at the owner of the slot*/
conn_slot_id /* Identity of the connected slot*/
prio_inher    /*Allow priority inherit or not from the arrival mail*/
prio_sort     /*Type of sorting mechanism either FIFO or priority*/
```

Description of vcb_receive:

extern int vcb_receive(void *msg, int *size, int owner, int *sender, int w4m, void (*handler)(void *msg, int size,int sender));

Arguments:

```
void msg	/* Location of transfer data */
int size			/* Size of transferred data */
int owner	/* The slot owner identity*/
int sender	/* Identify the sender of the mail*/
int w4m			/*Defines if the caller should be wait for mail or not*/
void (*handler)(void msg, int size,int sender)		/*Handel the received message*/
```

Returns: VCB_OK or an error on illegal input

Synopsis

```
/* Connect thread T1 to slot number one */
retval=vcb_connect("t1",SLOT_1, INHERIT,FIFO_SORT);

if(vcb_receive( msg_data, msg_size,sende_id, w4m,
handler(msg_data,msg_size,sender_id))!=VCB_OK){
  error handelr
else
  other code …
```

The return value from this system call will be *VCB_OK* if the call was successfully.

Syntax description of vcb connect and vcb_receive in VHDL code.

Figure 30 describes the logical structure of a hardware threads we later give an example of connect and receive from the VCB. A hardware thread is described in VHDL code. Application VHDL code describes the semantic functionality of a hardware thread; the description could be either parallel to use the hardware parallelism and/or sequential execution. One VCB_HW_API is instantiated in each hardware thread and it has connection to both the VCB_core and the system bus sees Figure 30. The system bus is used to move messages to main memory.

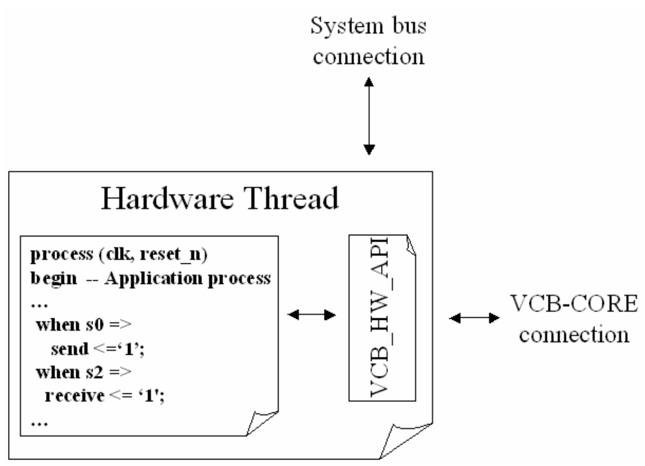


Figure 30 General hardware thread

The hardware thread could implement any type of functionality without interfere with the software application. The only system communications a hardware thread will require are the VCB together with the system bus, used to communicate with other software threads in the system. For more information see [5]. Below a description of the system call *vcb_connect* and *vcb_send* included in the VCB-API package will be described in the VHDL case.

Description of vcb_connect:

Except for the name parameter the system call will be the same as for the software case. The name parameter has been excluded in the hardware case.

Arguments:

conn_slot_id /* Identity of the connected slot*/
prio_inher /*Allow priority inherit or not from the arrival mail*/
prio_sort /*Type of sorting mechanism either FIFO or priority*/

Description of vcb_send:

The vcb_send call makes by specify call types; in this case the number is 6. Then the other parameters are the same as the software case. Except from the pointer in the software case, the solution in the hardware case will be to use a separate bit-vector to transfer the data.

31 – 28	27–16	15-12	11-8	7-4	3-0
6	Not used	Size	Address	Priority	Slot ID

Arguments:

Size	--Amount of data to be transferred
Address	--Identity of the sender
Priority	--Mail priority
Slot ID	--Address to the recipient of the message (Slot user)

Note:

send_data -- Send data is transferred by a separate bit-vector

Return codes: svc_ret[0] return code 0=OK, 1=not free

Synopsis

```
-- T2 connect to the slot 2
case c_state is
 when CONNECT =>
   connect_args<=connect&X"2"&msg_inherit&sort_alg –Slot id number 2, no inherit, FIFO
order
   c_call_api <= '1';
   c_state <= WAIT_CONNECT;
 when WAIT_CONNECT =>
     if rdv_api = '1' then
        connect_ok<='1';
        c_vcb_call <= '0';
        c_state <= IDLE_CONNECT;
     else
        c_state <= c_state;
     end if;
```

```
--When hardware thread T2 sends a message to T1 at slot number one,
case s_state is
  when SEND =>
     send_data<=X"ABCD";
     send_args<=send&size&address&priority&slot_id;
     s_call_api <= '1';
     s_state <= WAIT_SEND;
  when WAIT_SEND =>
     if rdv_api = '1' then
        send_ok<='1';
        s_call_api <= '0';
        s_send <= IDLE_SEND;
     else
        send_s <= send_s;
     end if;
```

Software and Hardware device driver cases

A device driver acts as a translator between a hardware device and software threads. One benefit to use device drivers is to break the dependency between the application and the lowest level hardware dependent software code. Every physical device, whether it is a printer, disk drive, or keyboard, must have a driver program. A device driver example for an UART is used to demonstrate and proof the concept and monitor the overhead The UART device is the connection between a serial port (RS232) and the system bus. A UART converts asynchronous serial data bits communication into a parallel byte stream and vice versa. Output and input to the UART device is two serial busses (one receive and one transmit) connection. The UART has a memory mapped register interface, where the register is used to control, read/write information and read status. We call this part the physical UART device.

A software device driver can be designed, described and handled in some ways and a standard software device driver always has a specified API. The standard driver API often provides five different types of calls; *init, opened, close, read and write.* In the test cases A and B the device driver will be handled as an ordinary thread, an UART thread. This thread will manage the hardware dependent code. A thread context switch (CTX) is the computing process of storing and restoring the state of a CPU (the context) at a memory location.

Case A Software device driver (software UART threads)

The first system architecture shown in Figure 31 contains one processor kernel, RT-SCH, UART, and VCB without support for hardware threads. The VCB core has only connection with the system bus and HW-SCH. The System busses are from IBM (Core connect) [32]the bus architecture include the processor local bus (PLB), an on-chip peripheral bus (OPB), a bus bridge and a device control register (DCR) bus. The real-time thread scheduling is managed by a hardware implemented scheduler. The real-time scheduler has support for 16 software threads at 8 priority levels and has support for time and semaphore handling [2]. The memory hierarchy in this system is divided in three levels. The first level is the on-chip cache in the processor kernel, not used during our test cases described below and in section 0. The second level is the on-chip block RAM and the third level is the off-chip memory, those two different memory levels contain system data and program code. The I/O block is a UART (Universal Asynchronous Receiver/Transmitter). This UART have a simple register interface with no buffering support. The interface is memory mapped and reaches from a base address X and five consecutive addresses ahead.

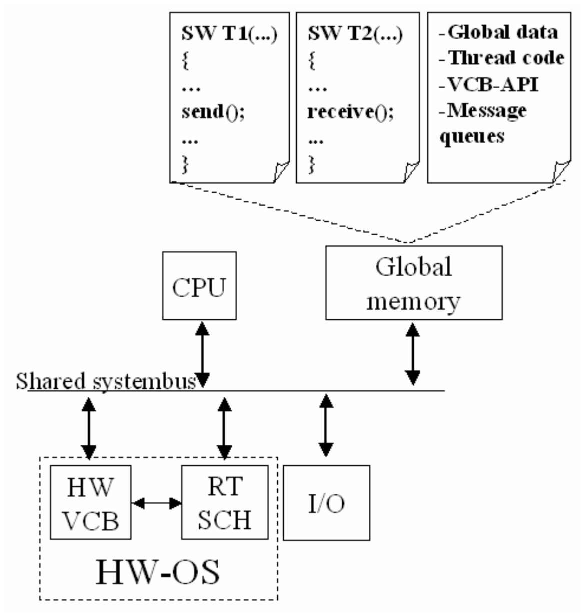


Figure 31 System architecture without hardware threads.

The UART-thread is scheduled as an ordinary application thread. The software interrupt service routine ISR is executed when the physical UART device fires an interrupt. The ISR manages the communication with the HW-VCB to generate a system event. A system event in this case is a rescheduling of the software threads.

The UART-thread use *init*, *opened* and *read*. And also use the *vcb_connect*, *vcb_send* and *vcb_receive* to communicate with the application. The terminal thread could read/write data from/to the physical UART device through the UART-thread.

The Figure 32 shows the sequence of processes which handle the physical UART device in the "software case".

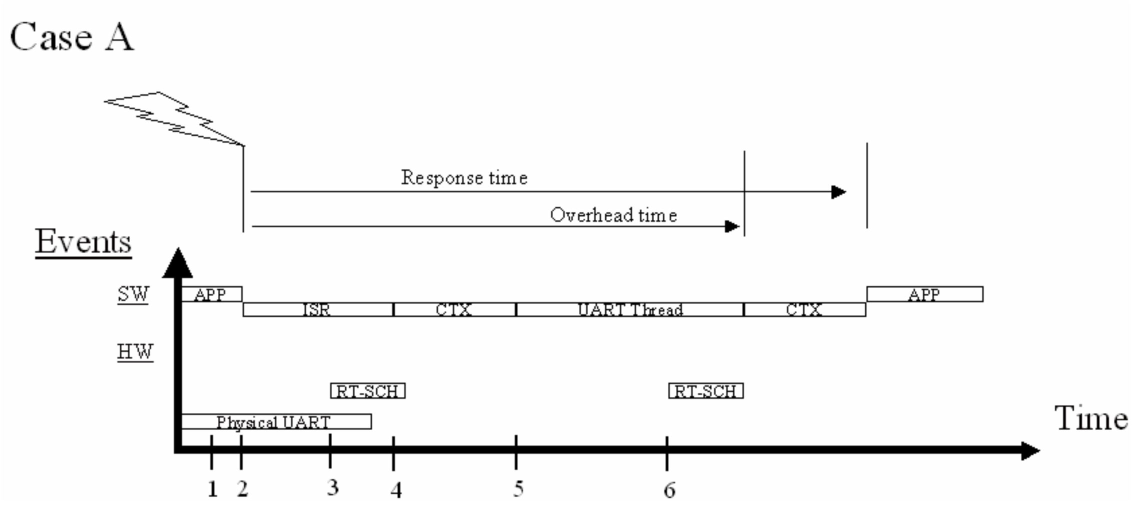


Figure 32 Response time in case A

1. One 8 bits ASCI data is arriving to the physical UART device and convert to one parallel byte.
2. Interrupt to the CPU
3. ISR starts and acknowledge the interrupt and send an event to start UART-Thread (device driver) through the HW-VCB unit
4. Tread Context switch to UART-Thread (CTX)
5. UART-Thread read the ASCI from physical UART device and if it is "carriage return" or the right amount of bytes has been read it starts terminal-thread else it collect the bytes in a memory array and block it's self.
6. Thread CTX, and next thread starts from ready queue.

Case B; Hardware device driver (hardware UART threads)

The second system architecture has the same configuration as in case A, but it also has an extension with support for HW threads. The I/O device in this case is removed from the shared system bus and is included in the hardware thread. The hardware thread manages the functionality connected to the UART device.

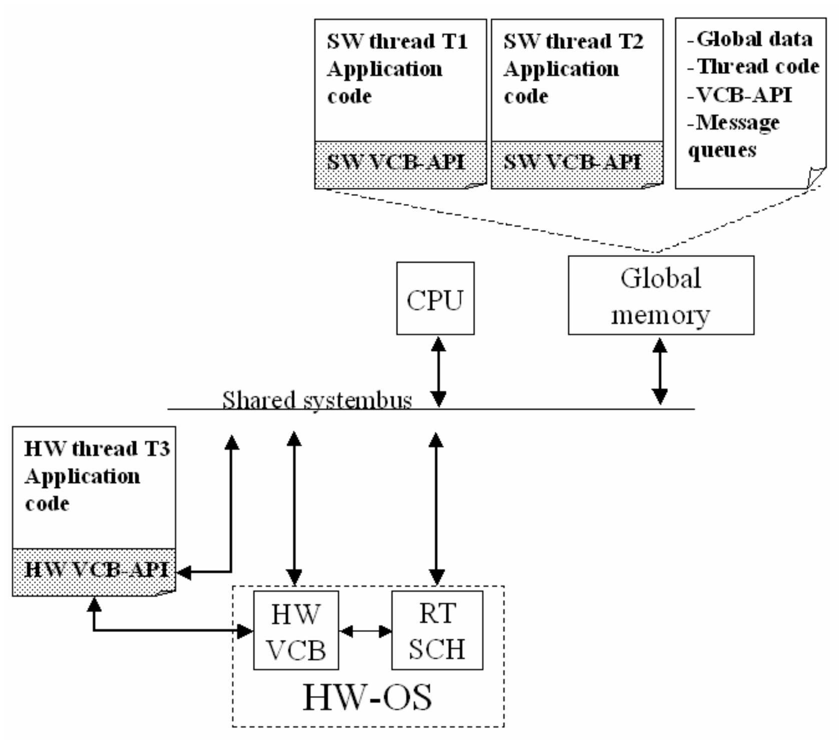


Figure 33 System Architecture overview with support for hardware threads

A hardware UART thread has the same semantic function as the software UART thread. Both threads will use the same interface to communicate with other software application threads. The difference between the software and hardware UART thread is that the hardware thread is always running in the system and will not utilize the CPU to manage the physical UART device. A hardware UART device driver thread interfaces the physical UART device directly through a point-to-point connection. The point-to-point connection includes 8 bits wide data bus, interrupt and acknowledge signals from/to the UART device. The HW_VCB_API connects the hardware thread to the software application through the VCB. The

HW_VCB_API has a master connection to the system bus used to transfer message data to the main memory with Direct Memory Access (DMA), see Figure 34.

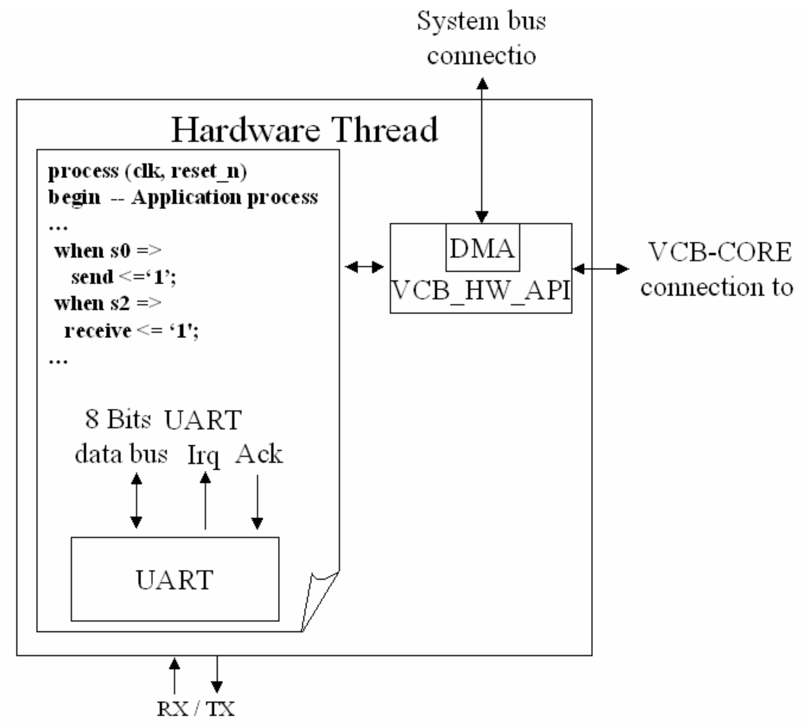


Figure 34 Physical UART and Hardware thread architecture

The physical UART component has a direct connection to the external environment through the two serial busses RX and TX. A hardware thread handle the arrived interrupt instead of directly interrupt the processor kernel, as in case A. The incoming interrupt request signal trigs a sequence of different statements, that we call ISR (to get equivalent semantic as in case A). The UART HW thread reads out characters form the data buffer. The sequence to read out one character from the physical UART takes three system clock cycles. The read characters are stored in a buffer. When the buffer is full or on carrier return arrival the hardware thread will send a VCB message to the subscriber of data from the UART device. The sequence of activity in case B will be described below Figure 35.

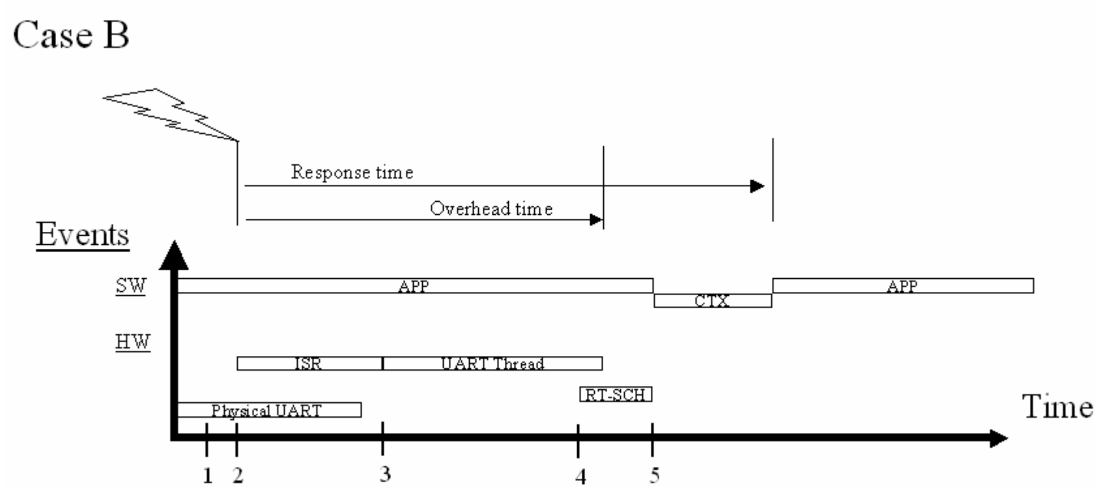


Figure 35 Response time in case B

The sequence of events in case B,

1. One 8 bits ASCI data is arriving to the physical UART device and convert to one parallel byte,
2. Interrupt to the ISR (hardware),
3. ISR starts and acknowledge the interrupt and start UART thread,
4. UART-Thread read the ASCI from physical UART device and if it is “carriage return” it start terminal-thread else it dose nothing more.
5. When the hardware-thread sends a message to the Terminal thread a CTX occur in the system.

Test platform description

A new PCB was developed with one FPGA, peripheral components, external RAM etc, see Figure 36. The FPGA architecture includes a hard CPU from (IBM PowerPC 405), and about 800 000 programmable gates and 18k RAM in a single FPGA chip from XILINX.

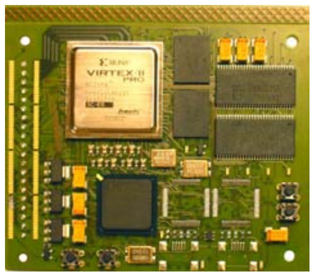


Figure 36 Test board

The programmable gates are programmed with hardware architecture except the CPU. The CPU is an ASIC within the FPGA chip. The system platform development tools used to generate the system platform are the Embedded Development Kit EDK and ISE from XILINX [24]. Those tolls generate a downloadable hex file for selected device; in this case a Virtex II PRO (XC2VP4). The hex file contains a complete system and the on chip memory configuration. The tools used to developed and generate the software are GNU tools [24] adjusted to fit into the XILINX design flow.

Results

Two main results are shown in this article, which are the HW/SW VCB-API and two different device driver cases A/B. Every hardware component in the hardware architecture is running at 50 MHz.

VCB-API results

The first result was the response time for different service calls for hardware/software VCB-API. The measurement was done with a timer on the System bus (in the FPGA), counting with system clock frequency and the probe effect was subtracted from the result.

Table 2 HW/SW VCB Service Calls response times (System clock 50MHz).

API/Calls	SW	HW	Quota
Connect	96.72us	1,65us	58,6
Disconnect	220.02us	2,5us	88,0
Send	135.12us	1,51us	89,5
Receive	490.66us	2,48us	197,8
Send&Wait	640.36us	3,99us	160,5
Broadcast	630.82us	6,20us	101,7

The result shows a decrease in response time between 60 and 200 times, where the most complex service calls gives the largest differences.

UART Driver case A and B

The second result shows the implementation of hardware device drivers and the response time, measured when the interrupt occur until the interrupt routine ISR and the UART-Thread manage the event. Table 4 shows the time when the interrupt occur to the character from UART is send to the consumer thread. In the software case A, 100us is needed to service one interrupt and the communication with the physical UART. Those 100us seconds is an overhead time for the software application. The same sequence in case B took 2,76us. In this case everything is done in hardware and running simultaneous with the software application. Thereby the overhead for the application will be zero. The latency from the UART thread in those two cases will be (OHT+VCB_send+CTX+VCB_receive) see Table 3.

Table 3 Response time from the physical UART see Figure 32 and Figure 35

CASE	A	B
Response time	749,98us	519,33us

Table 4 Overhead time for CASE A and B see Figure 32 and Figure 35

CASE	A	B	Relation
ISR	18,54us	1.23us	15,1 times
CTX	24,20us	***	Infinity
UART-Thread	57,30us	1.53us	34,5
Overhead time (OHT)	100us	2,76us	36,0

The design size for the extra logic for VCB-API is described in next table.

Table 5 Footprint for a HW_VCB_API in a hardware thread

	Used	Max	Utilization
Function Generators	897	10240	8.76%
CLB Slices	449	5120	8.77%
Dffs or Latches	666	11536	5.77%

Conclusions and discussion

- We have proved that a standardized VCB interface for SW/HW threads is a feasible way to increase the effectively of design device drivers with less overhead time, no interrupt from the physical devices and zero overhead for the CPU. This study was made by adapting communication to the hardware scheduler, with interface to hardware threads.
- If the serial port is set to 19200 baud. That means there are 19200/8 interrupts per second. At 100us each, this represent about 20% CPU overhead for a single channel using the standard CPU in our study. For 5 serial channels with 19200 baud, the result will be in worst case 100% load only for the interrupts. Today's new design technique gives the possibility to avoid the use of general architectural solutions.
- The FPGA technology has open a possibility to solve architecture in more an application demanding way. The design space has increased, because of the programmable gate arrays; they open up for an incredible lot of new hardware architectural freedom. Future work in this area could be to optimize the interface to get a cost-effectiveness of the VCB-API. But also look at how could a system with many hardware threads be implemented in an effective way.

10 APPENDIX

VCB is divided in two layers the highest layer is the application interface VCB_API, used by the application designers to communicate and synchronies the application. This layer could be implemented either for software designers or the hardware designers. The lower part VCB_CORE includes the protocol functionality of the VCB and this part is always implemented in hardware. Used by both the software and hardware VCB_API.

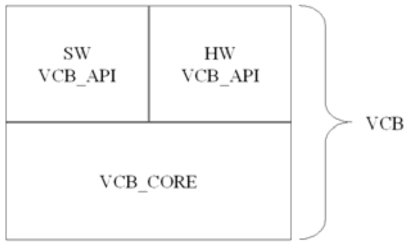


Figure 37 VCB structure

The VCB supports communication through message queues. A queue can be owned by a thread, which means that only one thread can read or write messages from a "Slot". Read or write function call supports *"synchronous"* or *"asynchronous"* calls. A "Slot" can be configured to support priority inherit from the messages priority. Message priority will be inherited to the software thread. Messages in a queue can be sorted by FIFO or PRIORITY order.

10.1 Software Application Programmable Interface VCB-API

Used by the software designers to communicate and synchronize the software application.

Disconnect

void **disconnect**(int slot_id);

Description:

Disconnects a software thread if it has been connected with the method connect earlier. Returns: VCB_OK or VCB_ERROR if the method failed.

Argument:

int **slot_id:** Numerical number of the slot

Connect

int **connect**(const char *name, int slot_id,bool prio_inher, bool prio_sort,bool task_prio_sort);

Description:

This system call connects a thread to the bus. The name may not exceed VCB_MAXNAMELEN. The slot_id should be positive and not exceed VCB_MAXSLOTNUMBER. Name: A name instead of a number Slot_id: Returns: VCB_OK if connected else an VCB error.

Argument:

const char ***name**: Name of the slot

int **slot_id** : Numerical number of the slot

bool **prio_inher**: Used to inherit priority or not from the mail

bool **prio_sort**: Type of sorting algorithm FIFO or PRIORITY order

Receive

int **receive**(void *msg,int *size,int *sender,int timeout,void (*handler)(void *msg, int size, int sender));

Description:

Receive a message Wait for a message to arrive and then copy it to <msg> of size <size> and call the <handler>. The <timeout> defines how long should be waited until a message arrives.

Argument:

void ***msg** : Message location

int ***size**: Amount of transferred data

int ***sender** : Sender identity

int **timeout**: Amount of time, waiting at a mail

void **(*handler)**(void *msg, int size, int sender): Call back routine, used to read a mail at the inherit priority.

Send

int **send**(void *msg,int size,int address,int priority);

Description:

Send the message <msg> of size <size> to <address> with priority
<priority>. The <size> may not exceed VCB_MAXMSGSIZE. This method is blocking until the message successfully is sent. Returns: VCB_OK or an error on illegal input.

Argument:

void ***msg**: Location of data

int **size**: Amount of data

int **address**: Address to the reader of the message

int **priority**: Mail priority

Send & Wait

int **sendwait**(void *msg, int size, int address, int priority, void *rcvmsg, int *rcvsize, int timeout);

Description:

Send a message and wait for answer

Argument:

void ***msg**: Location of data

int **size**: Amount of data to be transfered

int **address**: Address to the reader of the message

int **priority**: Mail priority

void ***rcvmsg**: Received mail location

int ***rcvsize**: Amount of transferred data

int **timeout** Amount of time, waiting at a mail

Broadcast

int **broadcast**(void *msg, int size, int priority);

Description:

Broadcast the message <msg> of size <size> with priority <priority>.

Argument:

void ***msg**: Location of data

int **size** Amount of data to be transferred

int **priority**: Mail priority

Init

int **init**(void);

Description:

This method is only needed to be called once in the entire system.

It should be called before any other VCB method is called. Returns: VCB_ERROR if something went wrong else VCB_OK.

Argument:

None

10.2 Hardware Application Programmable Interface VCB-API

Used by the hardware application designers to communicate and synchronize the hardware application threads.

Disconnect

call_args <= disconnect&slot_id;

Description:

Disconnect a thread *N* from the VCB-bus at slot number *X*. If the calling thread is the owner of the slot number X the return value will be OK otherwise the value is VCB_ERROR.

Argument:

31 – 28	27–16	3-0
1	Not used	Slot ID

Slot ID:

Specifies the slot ID (0-3).

Return codes

svc_ret[0] return code 0=OK, 1=not owner of the slot

Connect

call_args <= connect&msg_sort_alg&prio_inherit&slot_id;

Description:

Connect a thread *N* to the VCB-bus at slot number *X*. If the slot number is available the return value from the function call will be OK otherwise the return value will be VCB_ERROR

Argument:

31 – 28	27–16	5	4	3-0
2	Not used	Msg sort algorithm	Priority inherit	Slot ID

Slot ID:

Specifies the slot ID (0-3).

Priority inherit

The thread inherit or not from the arrival mail, not available for hardware threads.

Message sort algorithm:

Specified the type of sorting algorithm 0=FIFO 1=Priority.

Return codes

svc_ret[0] return code 0=OK, 1=not free

Receive

call_args <= receive&w4m& slot_id;

Description:

Receive a message from slot *number X* to read a message the user of the slot hade to bee the owner of the slot and the user hade to specify the blocked or non-blocked function call. The blocked functionality will only bee available for the software threads. If the destination slot is available and is opened the return value from this function call will be the particular message and the OK value, otherwise the return value will be VCB_ERROR.

Argument:

31 – 28	27–8	7-4	3-0
4	Not used	Wait for mail or not	Slot ID

Slot ID:

Specifies the slot ID (0-3).

Priority inherit

Define if the thread will be blocked or not if the reader hasn't any mail.

Address

Return codes

svc_ret[0] return code 0=OK, 1=no mail

Send

call_args <= send&msg_size&msg_adr&prio_inherit&slot_id;

Description:

Send a message to a thread through the specified slot_id. If the destination slot is available and if the slot is opened the return value from this function call will be OK otherwise the return value will be VCB_ERROR

Argument:

31 – 28	27–16	15-12	11-8	7-4	3-0
6	Not used	Size	Address	Priority inherit	Slot ID

Slot ID:

Specifies the slot ID (0-3).

Priority inherit

The thread inherit or not from the arrival mail, not available for hardware threads.

Address

Specified the destination slot.

Size

Amount of data to be transferred

Return codes

svc_ret[0] return code 0=OK, 1=not free

Send & Wait

call_args <= send_wait&msg_size&msg_adr&prio_inherit&slot_id;

Description:

Send a message to a thread through the specified receiver specified at a slot_id. If the destination slot is available and if the slot is opened the return value from this function call will be OK otherwise the return value will be VCB_ERROR. Wait for replay from the receiver.

Argument:

31 – 28	27–16	15-12	11-8	7-4	3-0
7	Not used	Size	Address	Priority inherit	Slot ID

Slot ID:

Specifies the slot ID (0-3).

Priority inherit

The thread inherit or not from the arrival mail, not available for hardware threads.

Address

Specified the destination slot.

Size

Amount of data to be transferred

Return codes

svc_ret[0] return code 0=OK, 1=not free

Broadcast

call_args <= broadcast&msg_size&msg_prio;

Description:

Send a message to every connected thread at the VCB. If the destination slot is available and if the slot is opened the return value from this function call will be OK otherwise the return value will be VCB_ERROR

int **broadcast**(void *msg, int size, int priority);

Argument:

31 – 28	27–8	7-4	3-0
8	Not used	Size	priority

Slot ID:

Specifies the slot ID (0-3).

Priority inherit

The thread inherit or not from the arrival mail, not available for hardware threads.

Address

Specified the destination slot.

Size

Amount of data to be transferred

Return codes

svc_ret[0] return code 0=OK, 1=not free

10.3 Supported service calls in VCB_BASIC

Every system call contains information to the VCB module about the type of system call. Every call could be described in a package with a fixed bit width.

Register specification

31 – 28	27-0
Module	Block specific description

This message block includes three types of register, each register is memory mapped and could accessed from the base address and two consecutive addresses ahead.

Version register

31- 0
Inform the caller about number of slot and message buffers

Service call register

31 – 28	27-0
Module	Block specific description

Service call response register

12	11-8	7-4	3-0
Status bit	Thread id	Buffer nr	Error code

Status bit:

Indicate the status at the block, either busy or ready

Thread id:

Next running thread ID

Buffer nr:

Buffer place to copy message to

Error code:

Indicate OK or some error (depends of which type of service call)

Create

call_args <= create&hw_sw&sort_alg&prio_inherit&thread_prio&thread_id&slot_id;

Description:

Create a VCB queue slot. Initialize the queue slot to allowed priority inherit a message priority arrived at the message slot. The messages sort algorithm could bee initiated in FIFO or by priority order. The priority order bit in the configuration register below controls the priority sort order of messages. The configuration of the queue slot must include information about software or hardware thread.

Argument:

31 – 28	27–16	14	13	12	11-8	7-4	3-0
0	Not used	HW/SW thread	sort algorithm	Priority inherit	Thread priority	Thread ID	Slot ID

Slot ID:

Specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Thread priority:

The default priority of the thread, used only when priority inheritance is allowed and the queue has one owner. This priority assigns to the thread after the get_ready call.

Priority inherit

The software thread inherit the priority from the mail

Message sort algorithm:

0=FIFO 1=Priority.

Hardware or software thread:

Used to indicate if the owner of the slot is hardware or software thread. (Not implemented in the latest version of VCB)

Return codes

svc_ret[0] return code 0=OK, 1=not free

Close slot

call_args <= close_slot&thread_id&slot_id;

Description:

Close a message queue. Prevents new messages to be sent to the queue. Should be used before deletion of a queue. The thread must bee the owner of the slot.

Argument:

31 – 28	27 – 8	7-4	3-0
1	Not used	Thread ID	Slot ID

Slot ID:

Specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Return codes

svc_ret[3:0] return code 00=OK, 01=Not owner of slot, 10=queue not created or queue is deleted.

svc_ret[7:4] number of messages in queue (0-15).

<u>Open slot</u>

call_args <= open_slot&thread_id&slot_id;

Description:

Open a message queue. Enable new messages to be sent to the queue. The thread performing this request must be the owner of the slot.

Argument:

31 – 28	27 – 8	7-4	3-0
2	Not used	Thread ID	Slot ID

Slot ID:

specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Return codes

svc_ret[3:0] return code 00=OK, 01=Not owner of slot, 10=queue not created or queue is deleted.

svc_ret[7:4] number of messages in queue (0-15).

Delete slot

call_args <= delet_slot& thread_id&slot_id;

Description:

Delete a slot. All information of the slot disappears. The thread must bee the owner of the slot.

Argument:

31 – 28	27 – 8	7-4	3-0
3	Not used	Thread ID	Slot ID

Slot ID:

Specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Return codes:

svc_ret[3:0] return code 00=OK, 01=Thread not owner of the slot, 10= slot is deleted or not created.

Get

call_args <= get&condition_flag&thread_id&slot_id;

Description:

Get a message from a queue the return value dependence on the argument to the *get* service call. Conditions are error code or suspended to message arrival.

Argument:

31 – 28	27 – 12	8	7-4	3-0
4	Not used	Condition flag	Thread ID	Slot ID

Slot ID:

Specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Condition:

Bit value 1= the thread will not be waiting if no mail is available.

Bit value 0= the thread will be waiting until a mail is available.

Return codes

svc_ret[3:0] return code 00=OK, 1=no mail ,2=Slot not opened, 3= slot is deleted or not created, 4=Thread not owner of the slot.

svc_ret[7:4] buff place pointer to message buffer (0-15) was messages has bean placed.

Get_ready

call_args <= get_ready&buffer_nr&thread_id&slot_id;

Description:

Inform the VCB that a message is read from the buffer placed *"get"* returned. This call must be performed after the *"get"* service call. The thread performing this request gets the priority changed back to its normal priority (when priority inheritance is used). The priority inherit functionality don't used for the hardware threads.

Argument:

31 – 28	27 – 12	11-8	7-4	3-0
5	Not used	Buffer place	Thread ID	Slot ID

Slot ID:

Specifies the ID of the queue (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Buffer place:

Specifies the buffer were the message is read from (0-15). This value must be the index number returned from the *"get"* service call

Return codes

svc_ret[3:0] return code 00=OK, 1=Thread not owner of the slot, 1=message not in get_ready state, 11= slot is deleted or not created.

<u>Put</u>

call_args <= put&message_prio&thread_id&slot_id;

Description:

Post a message at end of a queue, when FIFO else placed after messages at same priority. The first thread waiting for a message, inherit priority from the message when priority inheritance is used and ready (see below) is set to 0.

Argument:

31 – 28	27 – 12	11-8	7-4	3-0
6	Not used	Message priority	Thread ID	Slot ID

Slot ID:

Specifies the ID of the slot (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Message priority:

Specifies the message priority (0-3), 0=priority 0, 1= priority 2, 2= priority 4, 3= priority 6.

Return codes

svc_ret[3:0] return code 00=OK, 1=queue full,2=slot is closed,3=queue deleted or not created.

svc_ret[7:4] Index number for the location of the message in the buffer(0-15).

Put_ready

call_args <= put_ready&buffer_nr&thread_id&slot_id;

Description:

Inform the VCB that a message is placed on the buffer placed *"put"* returned. This call must be performed after a *"put"*. When priority inherit is used the thread priority inherit the message priority (Only available for the software threads).

Argument:

31 – 28	27–12	11-8	7-4	3-0
7	Not used	Buffer place	Thread ID	Slot ID

Slot ID:

Specifies the ID of the queue (0-3).

Thread ID:

Specifies the ID of the thread (0-15), which owns the queue

Buffer place:

Specifies the buffer were the message is placed (0-15). This value must be the buffer place that was returned by *"put"*.

Return codes

svc_ret[3:0] return code 0=OK,1=Not opened,2= queue not created or queue is deleted, 3=VCB_CORE error, 4=Not owner of the slot

Flush

call_args <= flush&slot_id;

Description:

Flush all messages on a slot. The thread performing this request must be owner of the queue.

Argument:

31 – 28	27 – 2	3-0
8	Not used	Slot ID

Slot ID:

Specifies the ID of the queue (0-3).

Return codes

svc_ret[2:1] return_code00=OK, 01=Not owner of slot,10=queue not created or queue is deleted.

Slot_info

call_args <= slot_info&info&slot_id;

Description:

Returns queue information.

Argument:

31 – 28	27 – 8	7-4	3-0
9	Not used	Info1/2/3	Slot ID

Slot ID:

Specifies the ID of the queue (0-3).

Info0/1/2:

Specifies if infoN=0 (see below for return codes).

Return codes

Info 0:

svc_ret[0] return code 0 = OK, 1 = queue is deleted.

svc_ret[1] 0=open, 1=closed

svc_ret[2] message sorting algorithm 0=FIFO,1=Priority.

svc_ret[7:4] number of messages in queue (0-15).

Info 1:

svc_ret[0] return code 0 = OK, 1 = queue is deleted.

svc_ret[1] 0=no owner, 1=one owner

svc_ret[7:4] thread ID (0-15).

Info 2:

svc_ret[0] return code 0 = OK, 1 = queue is deleted.

svc_ret[1] thread sorting algorithm 0=FIFO, 1=Priority

svc_ret[2] priority inheritance on message arrival 0=off ,1=on.

svc_ret[7:4] Thread default priority. (0-7).

Init

call_args <= init;

Description:

Initiate whole VCB to zero this system call should bee called once at the system init sequence.

Argument:

31 – 28	27-0
10	Not used

None

Return code:

None

Internal memory description

This description include an over view of the internal memory in the VCBIF and VCB main core blocks.

Slot memory (19 down to 0)

Bit [0] = Inherit message priority

Bit [1] = Type of sort algorithm FIFO or PRIO

Bit [2] = Owner is HW or SW thread 0=SW thread

Bit [3] = Thread blocked and waiting for mail

Bit [4] = Software thread has change priority

Bit [5] = Slot *n* is created

Bit [6] = Slot number *n* is opened

Thread default priority {generic ***taskprio***}

Thread identity only used to identified software threads {generic ***thread_id***}

Pointer memory (19 down to 0)

This memory contains four pointers, one for each message priority level. If the slot n is opened for FIFO sort algorithm the pointer 0 is used to hold the ready message buffer. { ***pointer 3, pointer 2, pointer 1, pointer 0*** } The length of the pointer memory depends on the generic variable ***pointersize***

Message memory

Bit [1 - 0] = Used to point out correct CPU number

Bit [3 - 2] = Message state {*free, put, put_ready* }

Pointer to next message in the message queue {generic ***pointersize*** }

11 GLOSSARY

- FPGA

  A Field Programmable Gate Array (FPGA) fundamental characteristic is that it consists of fine-grained programmable logic blocks interconnected via wires and programmable switches. Logic functionality for each block is specified via a small programmable memory, called lookup table (LUT).
- API

  Application Programmers Interface, The sum of all function calls available to an application programmer
- ISR

  Interrupt service routine, the routine that's called when an interrupt occur
- Task/Thread

  A task/thread is a sequential program performing certain functions, real time application is usually made up of one or more sets of communicating tasks/threads.
- Real-time system (RTS)

  A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated.
- RTOS

  Real time operating system, an operating system designed to be used in real time systems
- RT-SCH

  Real Time Scheduler, a real time resource scheduler implemented in pure hardware
- VCB

  Virtual Communication Bus, communication and synchronization mechanism used by both hardware and software threads.

12 REFERENCES

[1] HARTIK: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution Buttazzo, G.C.; Di Natale, M.
Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on , 2-6 May 1993

[2] Real-time scheduling co-processor in hardware for single and multiprocessor systems Starner, J.; Adomat, J.; Furunas, J.; Lindh, L.; EUROMICRO 96. 'Beyond 2000: Hardware and Software Design Strategies'., Proceedings of the 22nd EUROMICRO Conference , 2-5 Sep 1996

[3] L. Lindh, T. Klevin, J. Furunäs, "Scalable Architecture for Real-Time Applications - SARA", Swedish National Real-Time Conference SNART'99, August 1999, Linköping, Sweden

[4] Stankovic, J.A.; Ramamritham, K.: "The Spring kernel: a new paradigm for real-time systems", Software, IEEE , Volume: 8 , Issue: 3 , May 1991

[5] Uniform Interprocess Communication interface for Hardware and Software Threads
In International Workshop on Advanced Real-Time Operating System Services (ARTOSS) Porto, Portugal , July 2003. IEEE
Author(s): Peter Nygren, Lennart Lindh

[6] Distributed fault-tolerant real-time systems: the Mars approach
Kopetz, H.; Damm, A.; Koza, C.; Mulazzani, M.; Schwabl, W.; Senft, C.; Zainlinger, R.;
Micro, IEEE , Volume: 9 , Issue: 1 , Feb. 1989

[7] A Predictable Device Driver Model for a Variable-Rate Software-Controlled Switch Matrix
David B. Stewart and Melissa Moy Dept. of Electrical Engineering and Institute for Advanced Computer Studies University of Maryland

[8] Hardware/Software Co-Design of I/O Interfacing Hardware and Real-Time Device Drivers for Embedded Systems David B. Stewart and Bruce L. Jacob Dept. of Electrical and Computer Engineering, and Institute for Advanced Computer Studies University of Maryland

[9] Synthesis of DMA controllers from architecture independent descriptions of HW/SW communication protocols O'Nils, M.; Jantsch, A. VLSI Design, 1999. Proceedings.

[10] Successful Prototyping of a Real-Time Hardware Based Terrain Navigation Correlator Algorithm
Euromicro symposium on Digital System Design, Belek, Turkey. 2004

[11] Interprocess Communication Utilizing Special Purposed hardware
Technology Licentiate thesis by Johan Furunäs Åkesson ISBN 91-88834-24-7

[12] VSI Alliance Virtual Component Interface Standard

[13] N. Wirth, “Hardware Compilation: Translating Programs into Circuits”, Published by IEEE Computer Society in Computer (Vol. 31, No. 6), June 1998

[14] Distributed fault-tolerant real-time systems: the Mars approach
Kopetz, H.; Damm, A.; Koza, C.; Mulazzani, M.; Schwabl, W.; Senft, C.; Micro, IEEE , Volume: 9 , Issue: 1 , Feb. 1989

[15] Priority inheritance protocols: an approach to real-time synchronization Sha, L.; Rajkumar, R.; Lehoczky, J.P.; Computers, IEEE Transactions on , Volume: 39 , Issue: 9, Sept. 1990 Pages:1175 - 1185

[16] Survey of device driver management in Real-Time Operating Systems Sebastian Penner, Peter Nygren

[17] Hard Real-Time Computing systems, Predictable Scheduling Algorithms and Applications by Giorgio C. Buttazzo. KLUWER ACADEMIC PUBLISHERS

[18] R. Tessier, W. Burleson, ”Reconfigurable Computing for Digital Signal Processing: A Survey”, Journal of VLSI Signal Processing 28, Kluwer Academic Publishers, 2001

[19] Hardware/Software Co-Design of I/O Interfacing Hardware and Real-Time Device Drivers for Embedded Systems David B. Stewart and Bruce L. Jacob Dept. of Electrical and Computer Engineering, and Institute for Advanced Computer Studies University of Maryland

[20] Synthesis of DMA controllers from architecture independent descriptions of HW/SW communication protocols O'Nils, M.; Jantsch, A. VLSI Design, 1999. Proceedings.

[21] A Predictable Device Driver Model for a Variable-Rate Software-Controlled Switch Matrix David B. Stewart and Melissa Moy Dept. of Electrical Engineering and Institute for Advanced Computer Studies University of Maryland

[22] Intertask communications in an integrated multirobot system
Kang Shin; Epstein, M.;
Robotics and Automation, IEEE Journal of [legacy, pre - 1988] , Volume: 3 , Issue: 2 , Apr 1987

[23] Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems
Authors: Filip Thoen and Francky Catthoor, Publiched by Kluwer Academic Publichers

12.1 Unpublished references and www material

[24] http://www.xilinx.com

[25] http://www.altera.com

[26] http://www.xilinx.com/products/design_resources/proc_central/

[27] http://www.CoWare.com

[28] http://www.ose.com/products/

[29] http://www.windriver.com/

[30] http://www.lynuxworks.com/

[31] http://www.realfast.se/

[32] http://www-3.ibm.com/chips

[33] http://www.national.com/pf/PC/PC16550D.html

[34] CoWare N2C Design System www.CoWare.com

[35] VHSIC Hardware Description Language VHSIC stands for Very High Speed Integrated Circuit

[36] Samsung 128MB K4S280832D-TC75 16MX8

13 INDEX