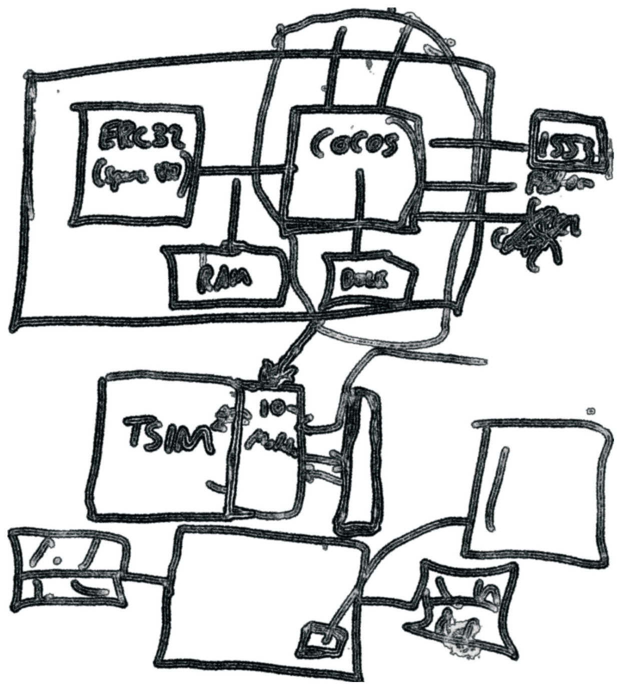
2003:113 CIV

LULEÅ UNIVERSITY OF TECHNOLOGY

MASTER'S THESIS

Remote Control of Embedded System Software

XML-RPC Over a Compact PCI Ethernet Device and TCP/IP in an ERC32 and RTEMS Real-Time OS Environment



Niclas Johansson
Olof Larsson

MASTER OF SCIENCE PROGRAMME

Department of Computer Science and Electrical Engineering
Division of Computer Science and Networking

2003:113 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 03/113 - - SE
SE D-G-REP-00030-SE

Abstract

This report describes the implementation of means to control tests of software in embedded systems.

When developing software for embedded systems it is crucial to be able to do test runs to ensure proper operation. This is often done on the target system, i.e. the real hardware with some peripheral test equipment attached. Being able to do this commonly requires physical presence at the test site. To facilitate this procedure, support for remote testing via a network would be desirable.

To achieve this, the target real-time operating system has been complemented with a network device driver, a PCI system bus API, an application level protocol and a test application server and client. The implementation of these complementations is described in this report.

Preface

First and foremost we would like to thank our thesis mentor, John Alexandersson, without his initial sketches[1] and ideas this thesis would never had seen the light of day, and also for his solid support throughout the project.

We also would like to thank our university thesis examiner, Anders Lindgren, for scrutinizing our results.

A number of other people at Saab Ericsson Space were also crucial to this thesis project - we would like to thank the head of the software section, Anna-Lena Johansson, our PCI guru, Håkan Jidmar, and our general hardware guru, Stefan Asserhäll, for their support.

Olof Larsson & Niclas Johansson

Luleå, Mars 12th, 2003

[1] See front-page block diagram for John's very first enlightening sketch effort, explaining the fundamentals of the project hardware environment.

Table of Contents

1. Introduction

Developing software of any kind often results in repeated test runs to check that new additions and changes are working properly. In some cases the test runs can be performed locally in the development environment, in other cases the developer has to move to a remote lab to test his work. Certain tests need to be supervised and monitored which means that developers have to remain at the test location or move between office and test location.

This also applies when developing software for embedded systems. Tests are often performed in target systems consisting of a hardware platform together with peripheral testing equipment, located in labs separate from developers' office.

If tests could be controlled and supervised over a network, instead of developers having to physically be present at the test site, this would facilitate the development and testing process.

1.1 Purpose

RTEMS [10] (Real-Time Executive for Multiprocessor Systems) is currently tried out at Saab Ericsson Space AB as a candidate real time operating system for future projects. Executing on a SPARC v7 compatible processor, ERC32, it co-operates with an ASIC called COCOS that supports the CPU and provides a number of I/O interfaces.

The purpose of this thesis is to enable communication between an RTEMS application running in the target environment and a remote application, using the local area network. The target system is equipped with a commercial PCI Ethernet card.

1.2 Method

The first initial step was to compile and install RTEMS and some essential tools, e.g. a cross compiler, for ERC32. An ERC32 simulator, TSIM[5], was used to execute our first code examples and to investigate the RTEMS operating system.

This provided extensive tracing facilities into the OS. For example tracing of the networking stack utilisation, down to the initial peripheral hardware setup procedures in the device driver layer. Further studies required the presence of PCI bus and network interface hardware, hence remaining work was performed on lab equipment.

The lab equipment was, for the client/server control software, an installation on an ordinary i386 based PC. This made it possible to start developing this application layer software without being dependant on first getting the BSD stack lower layers working on the ERC32 target platform. It also provided a reference platform for studying existing working drivers for another hardware architecture during the development of the device driver and lower stack layers.

The network interface device driver lab equipment was the target ERC32 and COCOS board, connected to the outside world via an RS232 serial console.

2. Requirements

Develop the software needed to communicate over a network with the ERC32 based target system. This via the RTEMS BSD stack, including the link, network, transport and application layers. Also implement an interface for controlling this hardware using some application layer protocol.

2.1 Requirement Specification

- Develop drivers for the CompactPCI interface in an ASIC developed by Saab Ericsson Space AB.
- Develop drivers for some commercial Ethernet card connected with a CompactPCI bus.
- Implement control software, controlled over the network, used as an interface to perform tests on software in development.
- Create a user interface for communication with control software.
- Investigate and implement a suitable communication protocol between target environment and remote host.

2.2 Additional Requirements

A general requirement has been to provide a test case scenario, for all the components involved, of a more complex and real world type than specific test software commonly used during the development of a new platform. Specifically:

- The RTEMS real-time operating system
- The BSD TCP/IP stack included in RTEMS
- The COCOS ASIC used as a PCI bridge

Another requirement was to provide a sample test application server for controlling test software. This server had the following feature requirements:

- Generic control
- Logging
- Data input stimuli

3. Implementation

3.1 Control Software

In order to control and supervise an application during tests, there is a need for communication between the application and supervising software. RTEMS contain an HTTP server, which constitutes the main connection point in our solution. A client connects to the server and sends commands using the XML-RPC protocol [20]. The client contains functionality to invoke procedures in the server and send or receive data from the application. An overview of the system is shown in Figure 1. Commands are parsed, marshalled, and executed by a handler in the server.

There are two versions of the client, an applet that can be downloaded from the server, and an application that can be installed on the host. The applet does not contain log functionality due to applets' restrictions in file access at client's host. Apart from logging capabilities, functionality is identical.

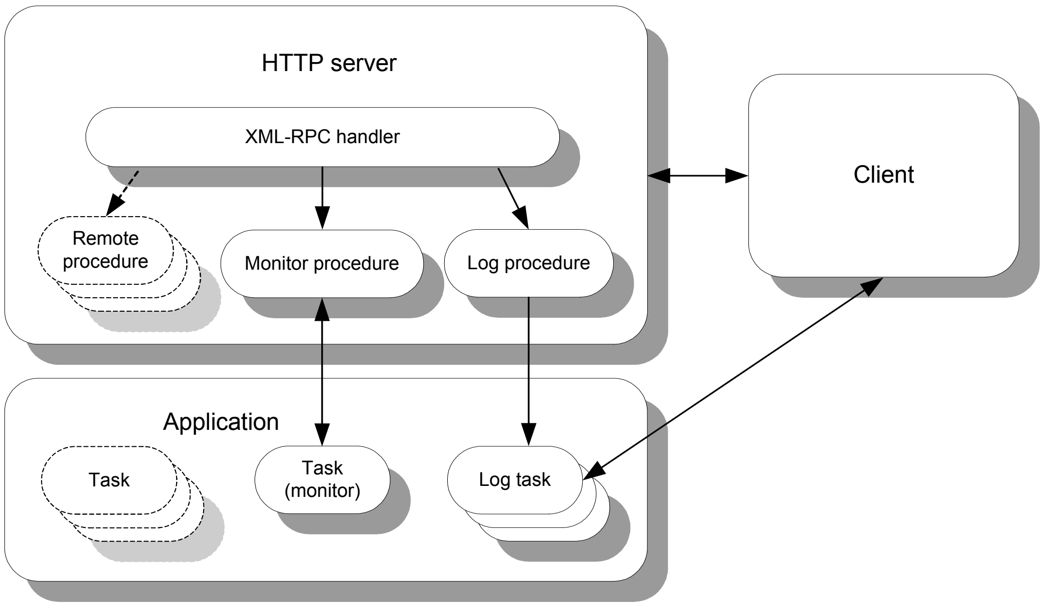


Figure 1: System overview

3.1.1 Server

When an HTTP POST request is received, the server checks the header for information. A handler is invoked to deal with the incoming message. Which handler to use depends on information in the HTTP header. In our solution we use XML-RPC to exchange information with the server so a handler, `websXmlRpcHandler`, is implemented. The `websXmlRpcHandler` procedure can parse and marshal a command that is encoded in XML. The handler creates a table where functions to be called by these requests are added.

On receipt of an XML-RPC request a method name and possible arguments are parsed. If the method name is defined in the table, the procedure belonging to this entry is invoked.

3.1.1.1 Remote Procedures

It is possible to define procedures in the server that can be invoked by a request sent from the client. These procedures should be written in file `webmain.c` in RTEMS directory `rtems_webserver` as ordinary functions. For `xmlRpcHandler` to know that a procedure exists, a pointer to the procedure has to be added to the symbol table. This is done by a call `websXmlRpcDefine`.

An example of a remote procedure is shown in Figure 2. The procedure name is `xmlRpcTest` and it takes two arguments. The first argument is a struct that contains information about the connection between the server and the client. The following arguments are optional and used as preferred. If any result should be sent back to the client, it must be embedded in XML code in order for the client to parse the response properly.

```
static void xmlRpcTest(webs_t wp, arg 1, …, arg 9,arg 10)
{
 /* Perform task.                                        */

 /* Send result back to client. A single <params> contain
    a single <param> that contain a single <value> of type
    int, string, boolean or double                       */


 websWrite(wp, T(" <params>\n"));
 websWrite(wp, T("  <param>\n"));
 websWrite(wp, T("   <value><string>%s
                          </string></value>\n"),”Done”);
 websWrite(wp, T("  </param>\n"));
 websWrite(wp, T(" </params>\n"));
}

websXmlRpcDefine(T("xmlRpcTest"), xmlRpcTest);
```

Figure 2: Code example of remote procedure definition.

When a procedure is correctly defined it can be invoked by sending a request containing method name and possible parameters, see Figure 3. The message is created according to the XML-RPC protocol and is built as a method call. The `xmlRpcHandler` procedure can, in current version, handle up to 10 parameters to C functions that define remote procedures, but the number of parameters can be easily increased.

After invoking a procedure, the client waits for a response from the server. The response is built as a method response and can contain result parameters originating from execution of procedures or a fault message. The client parses the response and the result is presented.

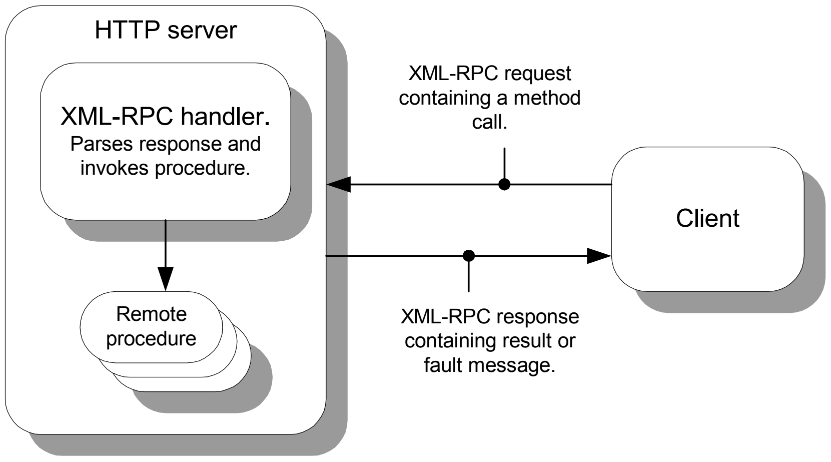

HTTP server
XML-RPC handler.
Parses response and invokes procedure.
Remote procedure
XML-RPC request containing a method call.
Client
XML-RPC response containing result or fault message.

Figure 3: A remote procedure call

3.1.1.2 Monitor

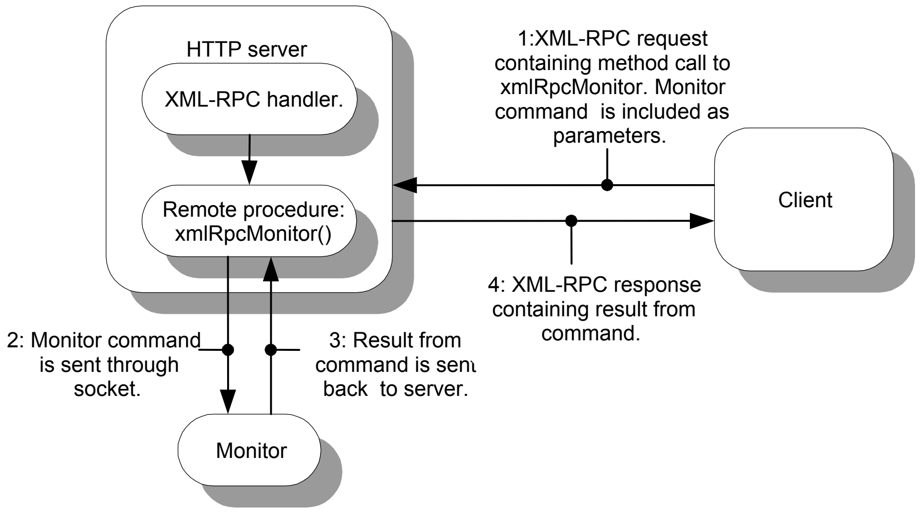

HTTP server
XML-RPC handler.
Remote procedure: xmlRpcMonitor()
1:XML-RPC request containing method call to xmlRpcMonitor. Monitor command is included as parameters.
Client
4: XML-RPC response containing result from command.
2: Monitor command is sent through socket.
3: Result from command is sent back to server.
Monitor

Figure 4: Monitor call overview, numbers represent course of events

Monitor is an optional task in that it is able to retrieve information from RTEMS data structures and present them in a user readable way. There are a number of different commands that can be given to monitor. Some of these commands can be passed optional parameters to further specify what kind of information that should be presented. For our purpose we modified the monitor so that, instead of printing the information to the screen, the information is sent to the client for presentation. An overview of a monitor call is shown in Figure 4.

In order to forward the information to the client we have modified monitor so that it waits for a connection from the server. *Standard out* is redirected to a socket waiting for a connection. Sending a monitor command with optional parameters from the client invokes the monitor procedure `xmlRpcMonitor` in the server. When this procedure is invoked it connects to the monitor and forwards the command.

When a connection is established and a command has been received, monitor executes the command and the information asked for is sent back to the server. This information is embedded in a response and sent to the client. The client parses the response and the result from the command is presented to the user.

3.1.1.3 Task Log

In order to supervise and control tests of software running in this environment there is a need for a communication link between the client and the tasks that constitute the application to be tested. A task in RTEMS is comparable to a process in other operating systems.

RTEMS contain a signal manager that allows a task to optionally define an asynchronous signal routine (ASR). ASRs are used in asynchronous communication between tasks. When a signal is sent to a task the task's execution path is interrupted and its ASR is invoked. When the ASR is finished the execution path of the interrupted task is resumed.

By using signals it is possible to control the execution of procedures in the application. Provided is a file that contains two predefined ASR's, one for sending data or log information to the client and one for receiving data sent from the client. This file can be included in the application and by calling a function resident in this file it is possible to enable a task to send or receive data.

3.1.1.4 Sending Data or Log Information

For ability to send data to a task from the client, the task has to make a call to `enable_log(int direction)` in file `logenable.c` which should be included in the task file. This is done at task start-up. When direction is set to SEND_DATA the ASR `process_asr_send` is defined as signal handler. The course of events is started in the client, see Figure 5. The client sends a message to the server containing a *start send* command and the name of the task to be started. The server then sends a *start send* signal to the task. When the task receives the signal it makes itself available for a connection from the client. When server has sent the signal successfully it sends a response back to the client, containing the port number that should be used to open a connection to the task. The port number is created from the tasks' class and index. When the client receives the response it connects to the task and starts receiving and presenting data.

When the task receives the signal it redirects its *standard out* to the socket. This way a user can send already existing printouts in the application code without further manipulation of the program. When started, a task can be put on hold and wait for a start signal to start the execution. This is done by a call to `waitForSignal`. To stop the transmission of data the client sends a stop message containing task name to the server, which in turn sends a stop signal to the task. The task then closes the socket, restores *standard out* and resumes execution at the point where it was interrupted by the signal.

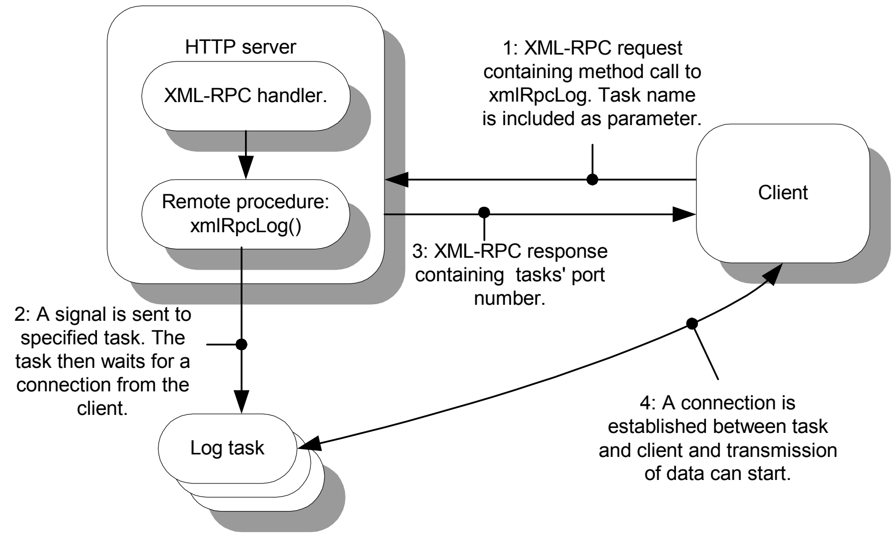

HTTP server
XML-RPC handler.
Remote procedure: xmlRpcLog()
1: XML-RPC request containing method call to xmlRpcLog. Task name is included as parameter.
Client
3: XML-RPC response containing tasks' port number.
2: A signal is sent to specified task. The task then waits for a connection from the client.
4: A connection is established between task and client and transmission of data can start.
Log task

Figure 5: Log call, numbers representing order of events

3.1.1.5 Receiving Data Sent From Client

Setting up a task for receiving data is done in the same way as setting up a task that sends data (See Figure 5). The difference is that the parameter to `log_enable` is RECV_DATA, which makes `Process_asr_recv` the ASR to be used. When a *start receive* signal is caught the task makes itself available for a connection but instead of redirecting *standard out* it starts reading from the connecting socket. Alternating reads and acknowledging writes to the socket at the task end achieve the transfer flow control.

3.1.2 Client

We provide a client that can communicate with the server. The client contains functionality to call arbitrary remote procedures, to use monitor task and to transfer log information to or from tasks. An applet version of the client can be downloaded from the server. This version lacks log functionality and can only call remote procedures and use monitor task. If log capabilities are required, the application version of the client has to be installed at the host. Both versions are developed using Java™ 2 Platform, Standard Edition (J2SE™). The client host should provide a runtime environment supporting this version so that the client works properly. For information about installation and downloads, see [19].

The applet version is built by making the main class, in `Head.java`, into an applet with calls to create log functionality removed. Together with classes in `Gui.java` and `TestControl.java` this class constitute the applet residing at the HTTP server. Since the applet version, more or less, is a subset of the application version, the following sections describe the classes needed for the application version of the client.

The client is built using the following files:

- Head.java
- Gui.java
- TestControl.java
- Log.java
- LogWindow.java
- LogFile.java

The following additional files might also optionally be used:

- rtemsfunctions.ini
- rtemstasks.ini

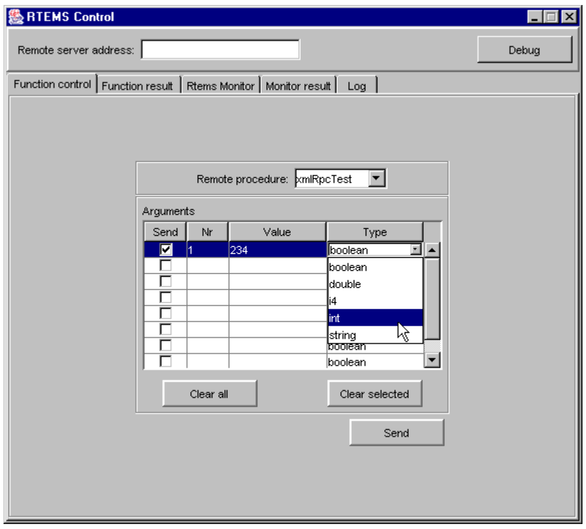
Files `rtemsfunctions.ini` and `rtemstasks.ini` can be placed in the installation directory. Remote procedure names separated by whitespaces can be specified in file `rtemsfunctions.ini`. If this file is found when the application is started, procedure names are available as choices and the user does not have to enter the names manually. File `rtemstasks.ini` is similar only it should contain names of RTEMS tasks of interest.

3.1.2.1 Head.java

This is the main class that initiates the application. It creates a frame that contains the graphical components created in class `Gui`. Instances of `Gui`, `TestControl` and `Log` are created.

3.1.2.2 Gui.java

The class `Gui` creates the graphical components. A tabbed pane is used to get a logical structure of the functionality of the client. Each functionality resides in its own tab, starting with function control, function result, RTEMS monitor, monitor result, log and finally a debug tab.



Remote procedure: xmlRpcTest

Send	Nr	Value	Type
☑	1	234	boolean
☐			boolean
☐			double
☐			i4
☐			int
☐			string
☐			boolean
☐			boolean

Figure 6: RTEMS control screenshot

In order to control presentation of data, all accesses to graphical components goes through this class. Results from commands are printed to text areas using public functions and input from different components is retrieved using public functions.

3.1.2.3 TestControl.java

This class controls the invocation of remote procedures and monitor commands. It listens to actions triggered by buttons in function control and monitor.

When a remote procedure is invoked a simple check is performed to see if arguments have the specified type. Arguments and types are then put in two separate vectors. Together with procedure name the vectors are passed to function `buildRequest` that assembles a request message. The message is passed as a string to `sendRequest` that creates a socket for connection to the server. Two data streams are created for reading and sending data.

When a message is sent the response is received it is passed to `parsePrintResponse`. The response is parsed using a simple XML parser resident in function `getXmlRpcTagValue` and then presented in appropriate result tab.

3.1.2.4 Log.java

Class `Log` controls logging of tasks. Data can be sent from a task to the client and be viewed in a window. As an alternative, data sent from the task can be saved to a file.

A third option is to send data, stored in a file at client host, to a task.

In order to keep track of logged tasks and corresponding files, task names and files are kept in a vector called `tasks`. Each started session puts these entries as an element into the vector and when session is finished or halted the element is removed. When a new log session is started a check is performed to see if the task and possible file is occupied by another session. If that is the case the log cannot be started until the session that occupies the file or task is halted.

3.1.2.4.1 Reading Data to Window

Using function `buildRequest` in class `TestControl`, function `readData` creates a log request message containing two strings as arguments. First string contains information about what kind of log session that should take place, if it should read or send data. The second string contains the name of the task that constitutes source or target. If all goes well when the message is sent the client can, from the response from the server, retrieve the port number that corresponding task is listening to. A socket is created and a connection is established between client and task.

When a connection is established, class `LogWindow` is instantiated and passed the socket. `LogWindow` extends thread to allow concurrent execution of multiple log sessions. When started it creates a window in which the received data is presented. The data is then read from the socket and presented to the user until socket is closed.

3.1.2.4.2 Reading Data To and From File

In function `readFile` a log request message is assembled in the same way as when reading data to a window. Before the message is created and sent, a file dialog is used for selecting a target file. If the file is not found in vector `tasks`, the message is sent and a socket is created and connected to the task. Class `LogFile` is the `LogWindow` counterpart and executes as a separate thread handling transfers to and from file.

When data should be written to a file two data streams are created. An input stream is used for reading data from the socket connected to the task and an output stream is used for writing to the selected file.

In the case of sending data from a file to a task, there is a need for transfer synchronisation to prevent buffer overflow at task end of connection. Client and task alternates writes and reads to stay synchronised.

3.1.2.4.3 Stop Logs

If the task name is present in vector `tasks`, function `stopLog` builds a stop log request and sends it to the server. The element holding the task and possible file in vector `tasks` is removed.

The corresponding thread that logs this task gets its connection closed by remote part and closes the streams and socket at his end.

3.1.3 RPC Protocol Analysis

The client needs a way to invoke procedures on the server side of the system. Any result should be sent back to the client. For both sides to understand each other there is a need of a common protocol. There are a number of RPC protocols, each with advantages/disadvantages depending on requirements and type of environment they are supposed to be used in.

- XML-RPC, Extensible Markup Language-Remote Procedure Call.

  XML-RPC can use HTTP as transport protocol and XML as encoding of RPC calls. XML is similar to HTML in that tags define input. HTML defines a set of tags that specify presentation of data. XML allows specification of arbitrary tags and can be used to define what kind of data is contained in the tags. XML is a subset of Standard Generalized Markup Language, SGML.

  A call is embedded in an HTTP-POST request and the payload of the request is the command encoded in XML. The server must provide capabilities to handle this kind of requests. In the header of an HTTP-POST request it is possible to define the type of request that is sent. If the server is supposed to handle different types of requests, not only XML-RPC requests, it can check the header to invoke the appropriate request handler.

  XML-RPC provides an encoding scheme to translate application semantics to XML code. The request payload or XML encoded command, contain the information about the procedure to be invoked. It is built as a single `<methodCall>` structure that must contain a `<methodName>` sub-item and optional parameters to the procedure. An XML-RPC response has its payload built as a single `<methodResponse>` structure containing either a `<params>` structure or a `<fault>` structure.

  XML-RPC defines a finite number of XML tags equivalent to data types, procedure names and other information needed to marshal a complete command. A set of rules is needed to be complied to when structuring these tags into a request. Tags and rules are described further in the XML-RPC specification [20].

There are many parsers for XML and XML-RPC API's available for many programming languages. Since this protocol is well-defined and kept small and simple, it is also possible to create a parser from scratch. This way the solution can be customised according to the requirements at hand.

XML-RPC became our choice of protocol due to its simplicity and adaptability. Our solution does not support structures and arrays. According to the requirements these types are not necessary and were left out. Should they be needed, only minor additions will be required for parsing these specific data types.

- CORBA, Common Object Request Broker Architecture.

  CORBA [13] is a protocol for writing distributed object-oriented applications. It is widely used in large enterprises and works well with several of the most common programming languages. Procedures to be invoked are defined as objects and for each type of object an interface has to be defined. This definition is made according to OMG Interface Definition Language, and specifies operations that are allowed to be performed on the object. A client uses the interface to send a request to the target object. When server side receives the request, it uses the same interface definition to retrieve arguments and invoke the procedure. For our requirement that only needs a simple invocation mechanism, this protocol seems to complex. The amount of time needed to apprehend and properly apply this protocol is too large for this project.

- SOAP, Simple Object Access Protocol.

  SOAP [21] can use HTTP to transport remote procedure calls just as XML, but can also use other network protocols. Like XML-RPC it provides an encoding scheme to translate application semantics to XML. The XML message is created using a predefined structure and namespace. The server uses the same encoding scheme to parse and interpret the message to build a command and execute it. SOAP is currently during development by a W3C working group. During this development a lot of features has been added. SOAP is similar to XML-RPC but contain an extensive amount of features not needed in this project, which is a reason why XML-RPC was preferred in our solution.

3.2 Network Interface Driver

3.2.1 Network Processes and Data Flow

RTEMS has a networking stack imported from FreeBSD. The following picture (Figure 7) from the RTEMS networking documentation [11] illustrates how data is passed back and forth between different processes.

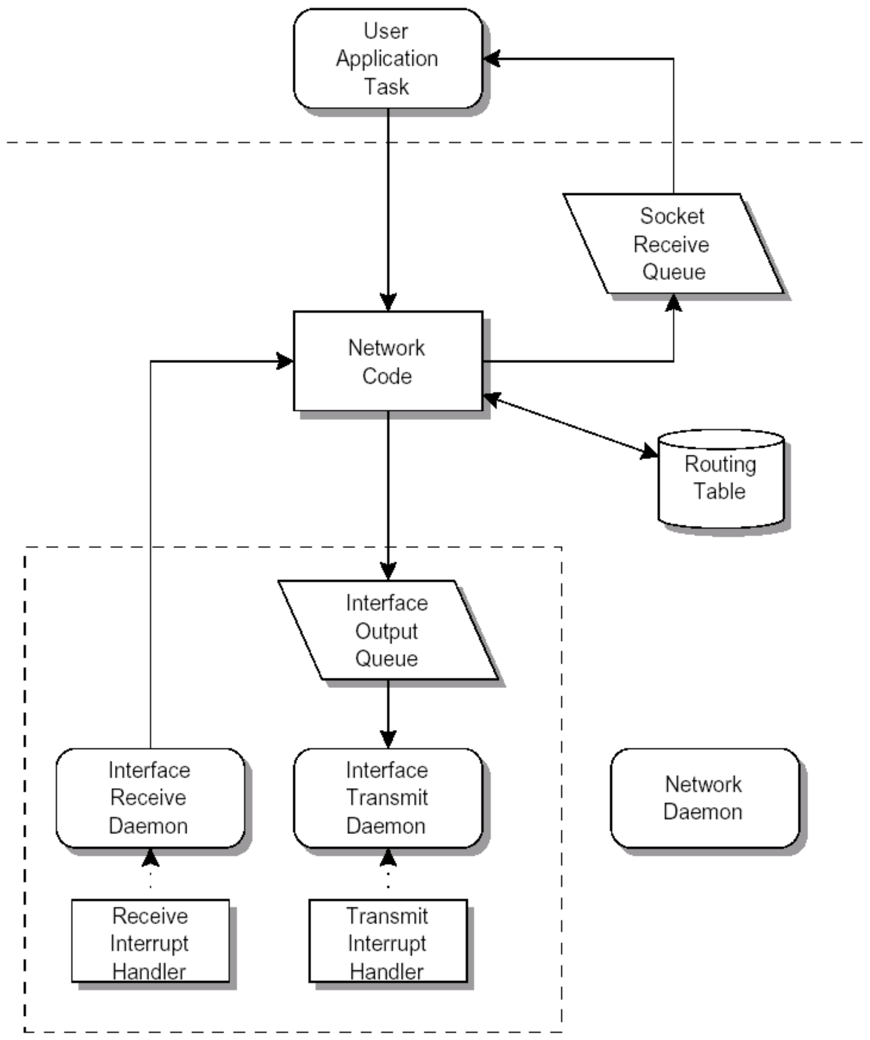


Figure 7: Network processes and data flow

The upmost level is the user process level. This is any application that includes networking facilities. The lower left-hand side box is the device driver code, while the rest is the general BSD stack implementation.

The device driver hosts two processes, both are usually blocked waiting for a signal indicating that there are data to be sent or received, respectively.

The transmission daemon is signalled via calls from the upper layers when there is data available for transmission, either application data or network control traffic, e.g. ARP broadcasts.

The receive task is signalled when the network device issues a hardware interrupt to indicate that there are incoming packets ready for processing. An interrupt handler does this.

The routing table is being maintained by the network process, which also takes care of assembling and processing incoming Ethernet frames passed on by the receive process.

3.2.2 RTEMS Network Setup Call Structure

The networked user process has to carry out two things to be able to reach the network. First it has to set up a struct called `rtems_bsdnet_config` which carries information about the network, such as IP, DNS, hostname and attached network interfaces. For each network interface available is also provided a pointer to the corresponding device driver's setup routine. Second, the network initialisation procedure is called. This is a coarse overview of what happens during this setup phase:

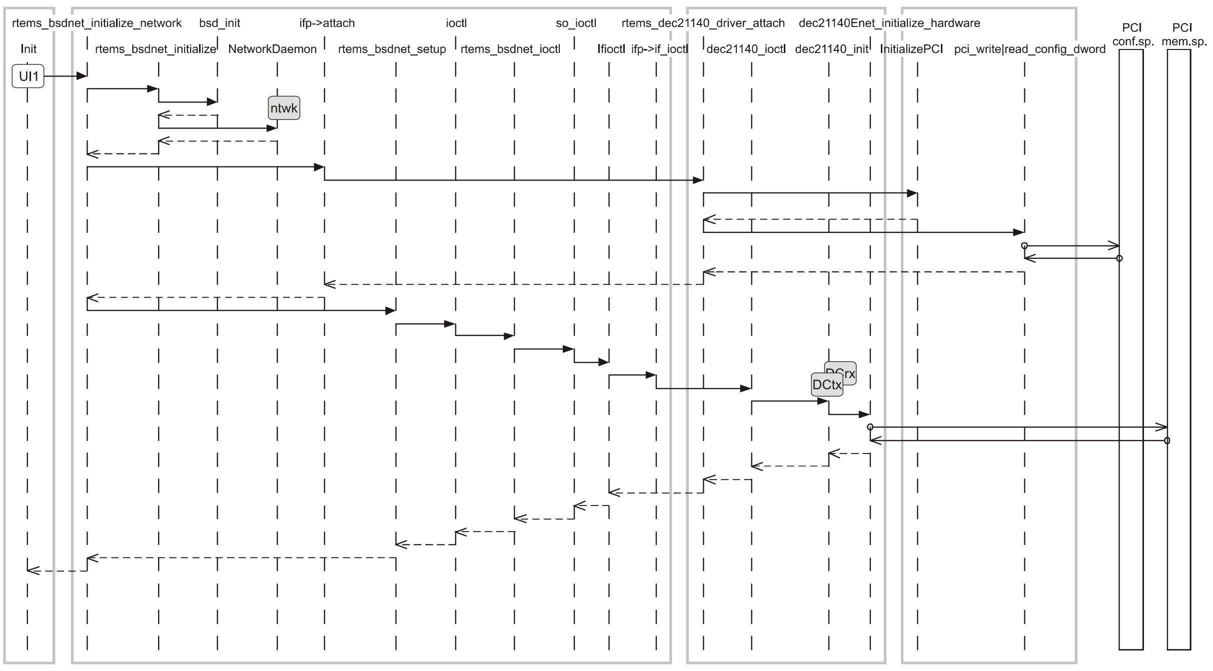


Figure 8: Networking setup calls

Comparing the above Figure 8 to Figure 7, Network processes and data flow, it is possible to spot all four process, or task, blocks. The *User Application Task* has the task identity *UI1* (Default value, may be set using a C language define statement), and is the first initial task that we are following in this UML like sequence diagram. The three other tasks marked in grey are being forked off as we go. First, the *Network Daemon* is started, it has RTEMS task id *ntwk*. The last two tasks are the *Interface Receive Daemon* and the *Interface Transmit Daemon*. They have different labels depending on what network interface is used, but commonly have the form *XXrx* and *XXtx*. The DEC 21140A network driver tasks have the identities *DCrx* and *DCtx*.

The division of the calls into four framed blocks shows what part of the code these calls belong to. The first call, *Init*, is the user application code. Next follows quite a few calls to the general BSD networking code. The BSD networking code makes calls to device driver procedures, which in turn uses the PCI code to access the PCI address spaces.

The three major calling sequences clearly identifiable above does the following:

- The first one starts, as previously mentioned, the *Network Daemon*. This daemon takes care of the BSD TCP/IP stack housekeeping tasks.
- The second sets up the PCI aspect of the network interface, by accessing PCI configuration address space, making it available to the system in the third phase:
- The last calling sequence configures the network interface via the PCI memory address space, setting network interface specific operation parameters, such as transmission rates, packet buffer pointers etc. It also starts the network interface receiving and transmitting tasks.

The network initialisation procedure called by the user application *UI1* program is:

`rtems_bsdnet_initialize_network`
This procedure makes three main subcalls to set up the network.

`rtems_bsdnet_initialize`
Which in turn calls:

`bsd_init`
This call sets up the message buffers for storing network data, e.g. IP packets and ARP packets.

`NetworkDaemon`
The network daemon is spawned and gets blocked waiting for a signal from underlying layers that a packet is ready for processing. Once it gets signalled it checks whether it is an IP or an ARP packet, and deals with them accordingly.

`ifp->attach`
This calls the first setup procedure for network interfaces. The *ifp* is a pointer into the user program network setup structure that defines this *attach* procedure.

`rtems_bsdnet_setup`
Sets up hostname, DNS, NTP etc. from the setup struct. Initialises network interfaces via *ioctl* calls.

`ioctl`
Starts interface with a call to the UNIX device control system call.

`rtems_bsdnet_ioctl`
The network interface is a device belonging to the network.

`so_ioctl`
We want to manipulate an interface belonging to a socket.

`Ifioctl`
It is an operation on a network interface.

`ifp->if_ioctl`
Finally we call the second part of the network interface setup. While the name unfortunately is the same as in previous call above, *ifp* is a pointer into a structure set up by the driver during previous setup phase containing pointers to methods for starting, stopping, setting the *mtu* (maximum transfer unit) and otherwise access and control the device.

Most noticeable from this report's point of view is the call to the device driver setup procedures.

3.2.3 Device Driver Setup Procedure

3.2.3.1 Defining Device and Attach Procedure

This is the logical entry point for the work on porting the DEC 21140A driver to the BSP (Board Support Package) consisting of the ERC32 CPU and the attached COCOS ASIC. Within the specific subdirectory of each BSP in RTEMS resides the file `bsp.h`. In this file are listed definitions of the device names and setup procedures of the network interfaces that are known to be supported on this specific RTEMS BSP. Once a suitable entry for your network device has been found you make a C definition that is later used when setting up the network configuration structure in user processes.

The very first basic step is thus to define that we have support for the DEC 21140A

```
#define RTEMS_BSP_NETWORK_DRIVER_NAME   "dc1"
#define RTEMS_BSP_NETWORK_DRIVER_ATTACH rtems_dec21140_driver_attach
```

Device drivers' setup procedures typically have names of the form *rtems_xxx_driver_attach*. These definitions provide the RTEMS BSD stack with a reference to an entry point into the driver.

3.2.3.2 The Device Driver Attach Procedure

The `rtems_dec21140_driver_attach` sets up the DEC 21140A for operation. Although the network interface chip is certainly the same on original and the target BSP, the way it is set up and what settings are done differs.

As this driver software is used for all BSPs supporting this particular piece of hardware the more architecturally dependent parts are littered with conditional defines, compiler directives defining different setup operations on different BSPs. The target BSP for our ERC32 based board is aptly referred to as *erc32*. This attach procedure does the following:

3.2.3.2.1 Initializing the PCI Bus

This first operation in the `rtems_dec21140_driver_attach` procedure is a call to setup the PCI bus, `InitializePCI`. This procedure resides in a file, `pci.c`, for which there exists one for each BSP supporting some kind of PCI bus. This setup procedure makes the subsequent accesses to the DEC 21140A device, via the PCI memory and configuration address spaces, possible. An account of how the PCI bus is set up for use is given in a separate paragraph (See 3.3.5.3.1, API Call for PCI Initialisation). The procedures for dealing with configuration address space accesses are also hosted in this file (See 3.3.5.3.2, API Calls for Configurations Space Access).

3.2.3.2.2 Scanning for a Network Device

Before we can access our network device we need to find on what PCI slot the device is physically located. This is done by reading the identity information of each slot on the PCI bus, in the configuration address space, until something that reports to be the sought after device is found, in this case the DEC 21140A. One PCI card can also be the host of several devices, so each slot is also scanned for additional devices on the same physical card, these are commonly referred to as multifunction PCI devices. The DEC 21140A is however an ordinary single function device, which contains one function only, function zero, the Ethernet adapter functionality.

Once the device has been located the driver registers the presence of this device in the internal per-driver data structure array. This structure stores configurational information used by the driver, such as address space references. The array of these structures are in place for device drivers supporting more than one physical Ethernet device, but as the DEC 21140A driver has no such support, this array only contains this single currently used structure.

3.2.3.2.3 Configuration Space Setup Operations

After having found where our device is located we can start setting it up. We write to registers that lie on offsets from the address that belongs to the slot on which our device is located (See 3.3.5, Using the PCI Bus, for more information on how this works). The first write is to the *Command and Status Register*, we set up some parity error responses, tell the device to respond to memory access and enable the capability to act as PCI bus master.

Next write is to the *Base Memory Address Register*, this is to tell the DEC where in the PCI memory address space we want it to map its memory space registers. This value is then read back and stored in the internal per-device structure for later use, i.e. when memory space registers are accessed. All PCI bus accesses from now on are performed on PCI memory address space.

3.2.3.2.4 Finding Device MAC Address

If the MAC address has been defined in the network configuration struct, provided by the user program, that entry is copied. Otherwise the device's EEPROM is accessed to retrieve the address. The location of the EEPROM can be read from the *Boot ROM, serial ROM and MII management* memory address space register.

The rest of the attach function sets a default value for the *mtu* (maximum transfer unit), and fills in the external network interface structure with pointers to driver procedures used by the upper layer to access and control the driver. Among those is the `dec21140_ioctl` procedure that becomes the agent for the next step in the setup phase - now that we have become able to read and write to the DEC 21140A memory space registers - the communication setup.

3.2.3.3 The Device Driver Initialisation Procedure

Once the device is attached it is open for Unix POSIX type *ioctl* configurational system calls. It is via such a call to the driver's *ioctl* control interface that the second, final, setup procedure is initiated. The command given to this *ioctl* call is *SIOCSIFFLAGS* which makes drivers consider the interface flags, provided via the external network interface structure. If these flags tell the driver it should be up (*IFF_UP* flag) the driver ensures that the interface is running (i.e. the *IFF_RUNNING* flag is set), which is done via a call to the initialisation procedure, `dec21140_init`. This procedure calls the DEC hardware initialisation procedure, starts the receiver and transmitter processes, unless of course they are detected already running, and sets the *IFF_RUNNING* flag. The hardware initialisation procedure is named `dec21140Enet_initialize_hardware`.

3.2.3.4 The Device Driver Hardware Initialisation Procedure

Sets up the device for operation. All DEC 21140A register writes are done using CPU address space previously mapped onto the PCI memory address space.

3.2.3.4.1 Setting the Operation Mode

The *Operation Mode Register* (CSR6) [3] sets the transmit and receive operation modes. The register is set up to a MII transmission rate of 10 Mbit/s and to use store and forward, that is, to await the availability of a full packet in the transmission FIFO before starting to transmit.

3.2.3.4.2 Software Reset

When setting the reset bit in the *Bus Mode Register* (CSR0) the DEC 21140A resets all internal hardware except for settings in the configurational area, or the previously set port selection.

3.2.3.4.3 Setting the Bus Mode

The *Bus Mode Register* (CSR0) is set up to use a round-robin bus arbitration scheme, resulting in fair bus sharing between the transmission and the receive process in the device driver. We also set the alignment requirement to be on eight bit boundaries. Further the DEC 21140A is set to interpret the data buffers as being *big endian* (See section 4.1.1).

3.2.3.4.4 Setting Up Transmit and Receive Buffers

First we allocate space for the MD (Message Descriptor) structure records in the transmit and the receive linked lists plus the amount of memory we want for storing a given number of maximum size frames for transmission.

Then, *Receive List Base Address Register* (CSR3) is pointed to the first position in the allocated receive Message Descriptor list. This receive structure is then traversed and initiated with proper values and each record is given pointers to a message buffer to store incoming data.

Next, the *Transmit List Base Address Register* (CSR4) is set to the position of the transmit descriptor list. This structure is then initiated with pointers to memory within the buffer space that follows right after the descriptor lists and that was allocated for at the same time.

3.2.3.4.5 Setting up Interrupts

This part of the setup procedure sets up the routing of interrupts from the external PCI device, the DEC 21140A, to the CPU via the COCOS Interrupt Controller module. How this is done is described in section 3.2.4.

3.2.3.4.6 Building the Setup Frame

Together with configuration and memory address space writes a third way is used to provide the DEC device with setup information. The DEC 21140A reads this setup frame [3] and use the data it provides to filter packages. Therefore Ethernet addresses on packets that the interface is expected to forward should be entered here, most commonly the address of the interface itself and the broadcast address. The filter buffer fits 16 addresses, of which all 16 have to be provided with the setup frame. If not 16 unique addresses are needed, the unused positions should be filled with duplicates of a previous entry.

Next we set the frame length field of the structure to be of specified fixed length, 192 bytes, and the *setup frame indication* flag is also set to tell the DEC 21140A that this frame is meant as a setup frame and not for transmission.

3.2.3.4.7 Processing the Setup Frame

First we write to the leftmost bit in the status field of the message descriptor structure that belongs to the buffer containing the setup frame. This indicates that the message buffer is ready for processing and the CPU hands the ownership over to the DEC 21140A. Then a write to the *Operation Mode Register* (CSR6) starts the DEC transmission process. The DEC device starts looking through the descriptor list, using DMA, for a packet with its owner bit set. Once it finds one, e.g. a descriptor structure harbouring a setup frame - the device copies it, the setup frame bit setting is checked and the frame is processed accordingly.

When the DEC 21140A device is done reading a buffer it resets the message descriptor owner bit of that message buffer. This is to return ownership and to inform the driver that setup frame processing is complete. The driver initialisation process does a busy wait for this to happen before it can proceed and complete the setup.

3.2.3.4.8 Enabling Network Device Interrupts

Finally the DEC 21140 is instructed on what events should trigger interrupts. Most notable should be interrupts upon reception of packets. Setting the corresponding bits in the *Interrupt Enable Register* (CSR7) does this. Any pending interrupts for those events are cleared before enabling them by writing the same bits to the *Status Register* (CSR5). Finally the *Operation Mode Register* (CSR6) is written to, enabling the DEC reception process.

3.2.4 Interrupt setup

This section of the setup procedure is highly dependent on the surrounding hardware, and thus rather platform specific. First of all, however, we turn off DEC 21140A interrupt generation. Things are set up in order from the OS and the CPU outwards, towards the external COCOS ASIC modules and the DEC 21140A chip. Figure 9 below provides an overview of the interrupt system.

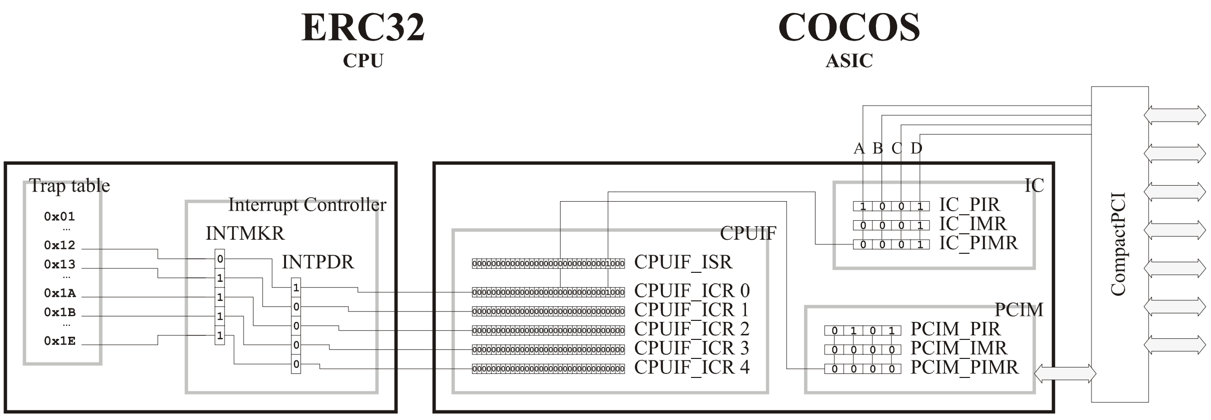


Figure 9: Interrupt overview

Except for the CPU Interface (CPUIF) registers only a few of the bits in the registers has been included – all registers are really 32-bit. The missing entry positions in the CPU *trap table* are used for internal interrupts and exceptions, only those entries belonging to the five external interrupts are shown. All the unused bits in the CPU Interface registers (all except bits 3 and 13) belong to other COCOS modules that may need to use interrupts.

Also worth noting is the potentially confusing circumstance that the COCOS mask registers let interrupts through when the bit is set, while the CPU mask register works the opposite way, letting interrupts through only when mask bit is cleared.

3.2.4.1 RTEMS

First called is `rtems_interrupt_catch` to make RTEMS install the interrupt handler for our chosen interrupt, external interrupt 0, *extint0*. Looking at the documentation for our ERC32 CPU (*TSC695F*) we can see that the entry for *extint0* is located at position 0x12 in the trap table. We provide an address reference to the driver's interrupt handler, `dec21140Enet_interrupt_handler`, to set what RTEMS does when this type of interrupt is triggered.

3.2.4.2 CPU

Next the *CPU Interrupt Shape Register* (CPU_INTSHR) is used to set the ERC32 to level triggered, rather than edge triggered, external interrupts (See section 4.1.7). Interrupts are cleared (CPU_INTCLR), external interrupt is unmasked in the *CPU Interrupt Mask Register* (CPU_INTMKR) and pending interrupts are cleared by a register read (CPU_INTPDR).

3.2.4.3 CPU Interface

The CPU Interface is the interface between the COCOS ASIC and the CPU. We set it to only route interrupts from the Interrupt Controller on *extint0* to the CPU, and all others are disabled.

3.2.4.4 Parallel Internal Bus Master

The PI Bus Master control is the interface used to enable and disable COCOS modules. It also controls what COCOS module gets DMA access to the CPU system memory and the ASIC registers and administers the memory map of the ASIC. As we are about to use the Interrupt Controller module we want to make sure it is available.

It takes two register writes to enable the Interrupt Controller. Those are the *PIM Enable Arm Register* (PIM_EAR) and the *PIM Enable Register* (PIM_ER).

We also want to perform a reset on the Interrupt module. This is likewise done by two separate register writes. These two are the *PIM Reset Register* (PIM_RR) and the *PIM Reset Arm Register* (PIM_RAR).

3.2.4.5 Interrupt Controller

The interrupt controller module in the COCOS can be used to increase the number of external interrupts available to the processor.

The Interrupt Controller is set to trigger on the level of external interrupts, and to consider a low signal as active. Also, it should be set to receive interrupts from, rather than generate interrupts to, external devices. These things are done via the *IC Input Mode Configuration Register* (IC_IMCR), the *IC Polarity Configuration Register* (IC_PCR) and the *IC IO Configuration Register* (IC_IOCR).

Finally we clear possible pending interrupts and select which external interrupt should be listened to. This is done via the *IC Pending Interrupt Register* (IC_PIR) and the *IC Interrupt Mask Register* (IC_IMR).

The steps in the above paragraph is dependant on what PCI Interrupt (See section 3.3.5.2) is used, which is determined by what slot the PCI device is inserted into. This is handled in a less than ideal way; see the Future Work section (sect. 5.2). Further, not all of the PCI interrupts were physically connected from the CompactPCI bus to the Interrupt Controller on the hardware used in this project.

3.3 The PCI Bus

The PCI, Peripheral Component Interconnect, bus interface used in this project has been developed by Saab Ericsson Space AB. It uses a modified version of the PCI standard referred to as CompactPCI.

3.3.1 About the PCI Bus

Intel originally developed the specification for the PCI bus system [16]. Their work on the PCI standard was published with revision number 1.0 and transferred to a separate organisation, the PCI Special Interest Group (PCI SIG). The SIG was responsible for producing the PCI Local Bus Specification revision 2.0, in 1993, based upon engineering requests from its members. The latest PCI specification at the time of this report was revision 2.3.

Due to the development within the i386 based PC industry PCI bus technology was established as a well-known standard by 1994.

3.3.2 About the Compact PCI Bus

CompactPCI, CPCI, [14] is an industrial bus based on the standard PCI electrical specification.

It is intended for application in areas where demands are higher with regard to physical robustness and reliability. Target areas for this technology specifically mentioned by the CompactPCI specification committee are, among others, telecommunications, industrial automation, real-time data acquisition, instrumentation and military systems.

Thus the differences are essentially physical, compared to the standard PCI used with desktops. The pin-and-socket connector of CompactPCI is more shock and vibration resistant than the card edge connector of standard PCI, and it has card guides on both sides in the card cage. The CompactPCI standard only supports eight slots on each bus. Future versions of CompactPCI are said [15] to include an additional hot swap feature.

The standard defines two options regarding physical shape of CompactPCI devices, referred to as 3U and 6U.

- 3U: 100 x 160mm
- 6U: 233.35mm x 160mm

The format factor of the DEC 21140A device is the smaller 3U.

3.3.3 Physical and Electrical Considerations

The PCI bus specification supports cards operating at both 3.3 and 5.0 volts. The network chip used on this card uses 5 volt. A shift towards using the increasingly popular 3.3 volt devices is currently underway at Saab Ericsson Space AB.

Since revision 2.1 the PCI bus standard supports not only a maximum of 33 MHz operation frequency, but also 66 MHz, doubling the theoretical throughput. The PCI bus used in this project is however clocked at the same frequency as the CPU. This frequency can relatively easily be changed by selecting a different crystal oscillator. The equipment used with the DEC 21140A network interface ran at the relatively moderate frequency of 14 MHz.

Special PCI devices referred to as *PCI-to-PCI bridges* are used to connect two PCI busses together. The PCI standard theoretically supports up to 255 busses (with each bus having up to 32 devices and each device having up to 8 functions).

The PCI standard of a maximum of 32 devices (In practice, the maximum number is usually significantly lower, though [7].) on each bus is not supported however, as we use the CompactPCI standard, only 8 devices are supported.

3.3.4 PCI and RTEMS

RTEMS for ERC32 does not have very elaborate PCI support. Many OSs that commonly run on PCI equipped hardware handle the generic PCI device setup procedure during some initial OS setup phase. This commonly involves items such as setting up the PCI host bridge for operation, scanning the PCI bus for devices, assigning PCI memory address space ranges, and building a list containing information about each device found.

For example, the Linux 2.2 kernel builds a list of PCI devices at boot time. Each entry is a `struct pci_dev`, which contains PCI configuration information about the device. This would typically fit into an extended `InitializePCI` procedure, which would be called upon during the initialisation of RTEMS, before user applications tasks are called upon.

However, in RTEMS, each driver currently does a lot of this work separately – which makes sense in highly embedded systems. The driver, e.g., does its own scanning of the PCI bus (either directly in the driver or by calling a procedure in the PCI API that does the same) to find its corresponding hardware. Drivers also do a lot of device setup work within configuration space. This is also quite commonly done by the OS during some earlier setup phase, and the driver usually only have to touch device registers within the memory space. These configurations space settings includes for example things such as claiming memory address space area, instead of having them assigned from the OS.

3.3.5 Using the PCI Bus

3.3.5.1 Address Spaces

The PCI bus can be accessed in three different address spaces, memory, I/O, and configuration space. All PCI devices must support some mandatory configuration space registers defined in the PCI standard. Most devices also present I/O mapped and/or memory mapped resources.

Generally, the PCI configuration space concerns settings that are done to a PCI device because of it being just that, a PCI device. Memory or I/O space settings are done to set up the device according to what function it has, e.g. Ethernet interface, audio card, et cetera.

Before devices on the bus can be used, they must be configured. Essentially, configuration will assign I/O and/or memory address ranges to each device and then enable that device. The DEC 21140A device uses memory space, and I/O space is therefore not described in this report.

That is, first of all we need to access the device in configuration space. These accesses are done using a PCI configuration API provided by the OS. On Unix type systems these methods typically looks something like:

```
pci_write_config_dword(char bus, char slot, char function,
                        char where, int val)
```

All PCI devices have a unique address in configuration space. This address is determined from a slot number, a device number, and a function number. It is thus fixed in this address space for each device - depending on where the device is physically located on the bus.

However, device specific registers are available in memory space, and what memory space address range is used is set up in configuration space. This PCI feature of setting what address range a device should use is a major advantage to previous bus technologies, e.g. Industry Standard Architecture - ISA, in that it does not allow devices to be built using fixed addresses.

3.3.5.2 Interrupts

The PCI bus uses its own internal interrupt system for handling requests from the cards on the bus. These interrupts are commonly referred to as *#A*, *#B*, *#C* and *#D* to avoid confusion with the normal, numbered system IRQs. These interrupts, if they are needed by cards in the slots, are then mapped to regular system interrupts.

Thus, the interrupts are not touched in the PCI setup procedure, as the need for them depends on what hardware is present and how it is used. For this reason the interrupt initialisation is done in the DEC 21140A driver.

3.3.5.3 The PCI API Procedures

As stated in the PCI specification [16] the PCI operations are provided via an API outside the device drivers. The PCI API provides seven functions, one for initialising the PCI bus, and six others for reading and writing PCI configuration space data of byte, word and double word sizes.

```
pci_write_config_dword          pci_read_config_dword

pci_write_config_word           pci_read_config_word

pci_write_config_byte           pci_read_config_byte

InitializePCI
```

In these procedure names a word is regarded as being 16 bits long, a double word is 32 bits, and the byte is 8 bits.

3.3.5.3.1 API Call for PCI Initialisation

The aptly called `InitializePCI` procedure initialises the PCI bus. What this procedure does is basically setting up some parameters on how the PCI module should operate and making it accessible by setting up some mappings between PCI address space and CPU local address space.

Four types of registers are accessed.

- PCI Module standard registers

  These are the PCI standard registers located on the PCI Module host bridge in the COCOS ASIC. The PCI bridge is a PCI device like others in that it occupies a PCI slot and carries the same standard PCI configuration registers.

- PCI Module device specific registers

  As most other devices the PCI Module also requires some device specific registers. These registers corresponds to those that would reside in memory space, had this not been the PCI bridge device.

- PI Bus Master registers

  The PI Bus Master is responsible for setting up the mapping of the address spaces available in the COCOS modules onto the COCOS address space.

- CPU Interface register

  The CPU Interface is called to enable COCOS' response to accesses in the CPU address space area directly mapped onto the COCOS.

PCI Module Device Specific Registers

The first register written to is the *PCIM Configuration Register*. This write initiates the COCOS PCI Module to be able to act as master and as target. It also sets that it should respond to memory space accesses, and what size this memory address space has. The rest of the registers contain settings on what worst-case latencies to expect when carrying out DMA operations.

Next we decide, using the *PCIM Retry Limit Register* how many retries should be done accessing a target device before the access is failed. We choose this setting to be zero, which means indefinite retries. This was a value used by some Saab Ericsson Space test software and no reason to change this was found.

There are two interrupt registers to clear, both are cleared by reading them, The first is the *PCIM Interrupt SW Mask Register* which hold pending PCI bus interrupts from external devices, e.g. our DEC 21140A Ethernet device. The second register is the *PCIM Interrupt Mask Register* that contains information about PCI Module reported events, typically errors, e.g. parity errors, timeouts. Clearing this interrupt is vital only when properly taking care of interrupt events reported by this register, which is something that is currently not done.

The *PCIM Direct Access Type Register* is set up to specify what address space should be targeted when doing direct access operations, i.e. configuration space or memory space. The default is set to be memory space accesses, as all configuration space accesses are supposed to be done using the PCI API procedures, which ensures that this register is set accordingly when they are called.

PCI Module Standard Registers

The first write to these PCI Module (PCIM) registers is paired to the memory address space size setting in the *PCIM Configuration register*. This setting sets the space size, and writing to the *PCI Base Address Register* sets from what offset this memory space should be available. As we map the whole memory space, we set this offset to zero.

Next, we set the latency timer, *PCI Latency Timer Register*, to establish how much time (in PCI bus clock cycles) master operations carried out by the PCI Module are guaranteed before they may be timed out (this setting has been done by example, and the practical implications of this specific value have not been investigated).

The PCI Module is activated by a write to the *PCI Command and Status Register.* The upper half-word are status bits only, of which some are fixed value, and others are cleared by this write. The lower half-word command bits are set to enable the PCI Module acting as master and responding to memory accesses.

Finally, the last PCI Module register setting, *PCIM Target DMA Base Address Register*, we set the lowest address which the PCI Module should be able to access with DMA reads and writes. This should in the DEC 21140A case at least cover the area where the receive and transmit buffers might be allocated. Simply setting this value to zero, i.e. the beginning of the CPU address space, ensures this.

PI Bus Master Registers

The PI Bus Master handles the COCOS memory map, using seven memory map units, *MUnits*. One for the COCOS registers, four are used for mapping bulk memory, one for mapping to CPU local memory, and the one of interest to us, the memory map unit for the PCI address space.

First the *PIM MUnit Parameter Register PCIM* of the memory map unit is set to what DMA page we want the PCI address spaces to be mapped on. The second DMA page, referred to as DMA page 1, is most appropriate for this. It is there with PCI in mind and is free from default mappings. All the default COCOS memory mappings are put on DMA page 0.

Next a definition of the start and end address of this mapping on DMA page 1 is needed. This is done using the *PIM MUnit Address Low Register MPCI* and the *PIM MUnit Address High Register MPCI*.

We map the entire PCI address space. The only time we potentially might need to access the whole 32-bit address space, in this DEC21140A case, is when probing for PCI devices in the configuration address space.

CPU Interface Register

In memory space we only use some address area above 0x40000000. This specific area is because of the CPU address space being directly mapped onto the same addresses of COCOS DMA page 1. This facilitates memory space accesses in that we do not need to use any of the *Extended I/O area map pages* like done when accessing configuration space. Enabling the COCOS response to the use of this area beyond 0x40000000 is done by setting the *CPU IF enable extended I/O* register.

Telling the DEC to map its memory space registers to 0x40000000 is part of the Ethernet device setup procedure, and thus not covered here. In other OS environments is this address commonly assigned by the OS PCI setup procedures, which are called upon prior to device driver setup is performed.

3.3.5.3.2 API Calls for Configurations Space Access

The API contains three procedures for reading and three for doing writes of byte, word and double word sizes. The read or write operation itself is trivial once the surrounding mappings and address resolutions have been performed. These are similar in both read and write cases, and given that the byte and word procedures call their double word sibling to carry out their task, only one of these procedures needs to be accounted for here.

This is what happens:

- PCI interface is set to access configuration space.
- Determine target address
- Address mapping of target address area is performed
- Access the mapped area with appropriate read or write operation

Setting PCI interface to use configuration space

This is simply done by setting the *PCIM Direct Access Type Register*. We set this to direct PCI address space accesses to configurations space. Before the procedure call is completed we restore this register, setting it to the memory address space access type initially set in the `InitializePCI` procedure.

Finding the PCI target configuration address

The PCI devices are located at fixed addresses in the 32-bit configuration space. A device inserted in the first slot is located at 0x8000000, i.e. where the most significant bit is set, and all other bits are zero. Shifting this bit one step to the right will for each shift provide the location of the following PCI slots. Given that this is a CompactPCI type bus there is, as previously accounted for, a maximum of eight devices, including the PCI bus master device. This means that the last device will reside at 0x02000000.

Configuration access on the i386 PC platform

What happens next is perhaps best illustrated by first looking at the i386 PC platform approach. This following Figure 10 is from the PCI specification [16]:

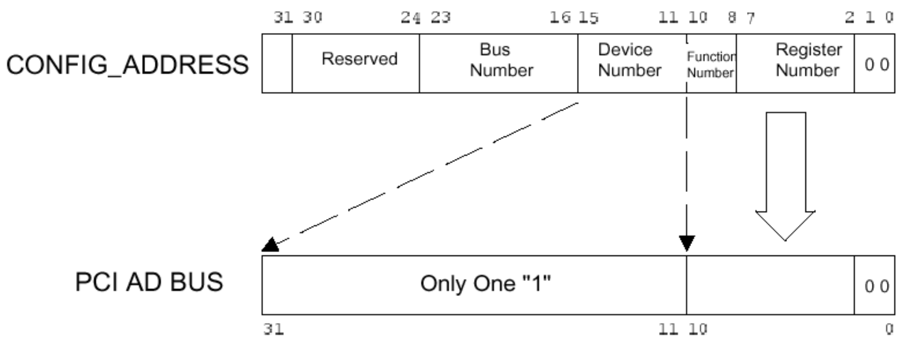


Figure 10: Host bridge translation for Type 0 configuration space addresses

Here, PCI configuration space operations are carried out by writing to two registers. On the i386 PC implementation of PCI, the above register CONFIG_ADDRESS is filled with the following information. These fields should be fairly trivial:

- Bus number indicates what PCI-to-PCI bus the target device is located on. Typically this field is all zeros, indicating that the device is available on the current PCI bus system. The maximum number of such bridges is 256.
- Device number could also be referred to as slot number. As this field is 5 bits wide, the maximum number of devices is 32 according to specification, as this is not CompactPCI.
- Function number, three bits wide, which limits the number of functions a multifunction PCI device can host to 8.
- Register number, finally, may also be referred to as the offset. Indicates what double word in the configuration space header for the target device do we want to read or write.

The data read or written during operations on this address is stored in a second register, the CONFIG_DATA register.

Next is the crucial part. The register information in CONFIG_ADDRESS is translated to a PCI address space access, as is depicted in Figure 10. There are two types of configuration space operations that can be done, referred to as *Type 0* and *Type 1* operations. The most common operation type is the *Type 0*. The *Type 1* is used for accessing PCI-to-PCI bridges, i.e. to access PCI devices that are attached to slots on PCI bus systems that are attached as devices to the PCI system on which the operations performed originates.

The *Type 0* regards the 5-bit device number field, and uses this value to determine what bit should be set in the translated request. That is, the most significant bit is set, and then shifted down a number of steps indicated by the device number field. E.g. the first device, device number 0, will become 0x80000000, the following device number 1 at 0x40000000, etc.

This works fine for the CompactPCI standard, supporting 8 devices. However, given that this available bit field is 21 bits wide, while the maximum device number can identify at most 32 devices, which is also the maximum mentioned in the PCI standard specification, it may seem like there are some potential problems with the translation here. That is indeed the case, and the PCI standard does not really provide any solution to this situation ([16] section 3.2.2.3.5), using this way of addressing devices. However, in practical, the device limit is usually set even lower than that due to electrical constraints and other reasons [7].

Configuration access using the COCOS

In the COCOS ASIC, these two registers, CONFIG_ADDRESS and CONFIG_DATA (see previous section, Configuration access on the i386 PC platform) do not exist. Therefore the PCI request address translation has to be done by the PCI API procedures, and accesses are done via a CPU address space mapping of the PCI configuration space.

An RTEMS PCI API procedure declaration:

```
pci_read_config_dword(char bus, char slot, char function,
                      char offset, int *val);
```

Bus. This parameter is not used. It may be used for generating Type 1 operations (see previous section) for PCI-to-PCI bridges, but since no such devices are expected to be used, setting this value to anything but 0, the current bus, will return an error.

Slot. Shift a leading one the slot value (again, see previous section) number of times to the right, ranging from 0 to 7 in this Compact PCI case.

Note that this depends somewhat on your numbering scheme. In this case things work out like described above, as the DEC 21140A device driver starts probing from device 0. Should the numbering scheme be 1-8 instead, or even 2-8 with the intention to avoid probing the master PCI device itself, this certainly has to be accounted for.

Function. The driver only supports single function devices. Other values than 0 for the function number will return an error. Adding this capability to the driver should require little effort, if a multifunction device should become available.

Offset. Just as depicted in the i386 PC translation illustration, the offset can be copied right in on the bits 2 to 7.

Note that that this offset is in 32-bit words. In this case, however, we have a file, `pci.h`, defining the compulsory standard PCI configuration space registers that give this value as byte offsets, which mean that the two least significant zero bits are included. Thus, we can enter this 8-bit value into bits 0 to 7.

This results in this rather simple expression:

```
pciaddr = (0x80000000 >> slot)+(offset & 0xfc);
```

Which, with 32-bit word offset values and a number scheme having the first available device as device 2, would look something like:

```
pciaddr = 0x80000000 >> (slot - 2)+(offset & 0x3f) << 2;
```

Accessing PCI address space through map pages

The whole PCI address space has been mapped onto the COCOS address space DMA page 1 (See 3.3.5.4, Address Spaces and Mapping Overview). However, mapping this entire 32-bit PCI in COCOS onto the CPU address space is, given that the CPU space also is 32-bit wide, obviously not possible. Thus we need to map a smaller subset of this space, the portion we need to access. This is done using a *CPU IF Map register*. There are eight such mapping registers, numbered 0 to 7, and we use *CPU IF Map register 7*.

Each mapping register maps a certain area of the CPU address space onto COCOS address space. In this case the mapping area, *Extended I/O area Map page 7*, resides between 0x3C000000 and 0x40000000 (Again, see 3.3.5.4, Address Spaces and Mapping Overview) as we have selected *CPU IF Map register 7*. All accesses within that region are then redirected to wherever the register is set up to map them.

These registers are set up accordingly:

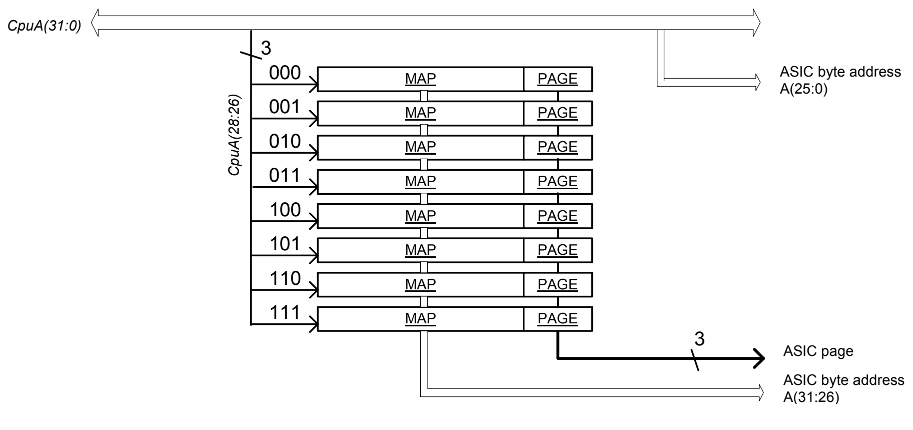


Figure 11: Memory mapping

The mapping is triggered by writes in the mapped area. This is addresses from 0x20000000 to 0x40000000, i.e. addresses starting with 001_b on bit positions 31 to 28. What happens next, is that the three bits 26 to 28 are used to determine what mapping area is accessed. In the case of *Extended I/O area Map page 7* those bits are set to 111_b, and thus the *CPU IF Map register 7* is checked for (See Figure 11 above) mapping information.

The mapping registers contains the significant bits of the target address space location, while the last three bits of these registers are used to indicate what COCOS DMA page should be accessed. In the current PCI case this value is always 001_b, for DMA page 1. The lower bits 0 to 25 are copied; the offset into the mapping area is simply kept and becomes the offset in the target address space.

Setting this up is then simply done using the following operation:

```
mapvalue = ((pciaddr >> 23) & 0x1f8) | 0x1;
```

The six most significant bits of the desired PCI address are entered in the register. They are shifted 23 bits to the right, instead of 26, to leave room for the three bit page field. Thus we end up with a 32 bit register of 23 zeroes, followed by the 6 most significant bits of the mapping, and a 3 bit field controlling the DMA page selection (For information regarding the *pciaddr* component, see previous section).

This is then written to *CPU IF Map register 7*:

```
WRITE (CPUIF_MR7, mapvalue);
```

Next we need to figure where to do our access, so that it will be mapped to the target area of interest:

```
mapaddr  = 0x3c000000 | (pciaddr & 0x03ffffff);
```

As previously mentioned the map area belonging to this register, *Extended I/O area Map page 7*, begins at 0x3c000000. Use the 6 most significant bits of this area, remove the old significant bits, and keep the 26 least significant bits' offset.

Doing PCI configuration space operations

Now that we have an address to a mapping of our target PCI address, we simply do the operation, e.g. a write of variable `val`, to this address.

```
WRITE (mapaddr, val);
```

In the `pci_read_config_dword` case, this is of course a read operation instead. READ and WRITE are just macros to perform simple memory read and write operations along the lines of:

```
*(volatile unsigned int *)(addr) = (data);
```

Restoring PCI interface to memory space access

This is done by another write to the *PCIM Direct Access Type Register*. The direct PCI address space accesses are directed back to memory space, as was the case before this PCI configuration space API was called.

3.3.5.4 Address Spaces and Mapping Overview

This is an overview of the address spaces of interest, and some illustrations on what mappings are done.

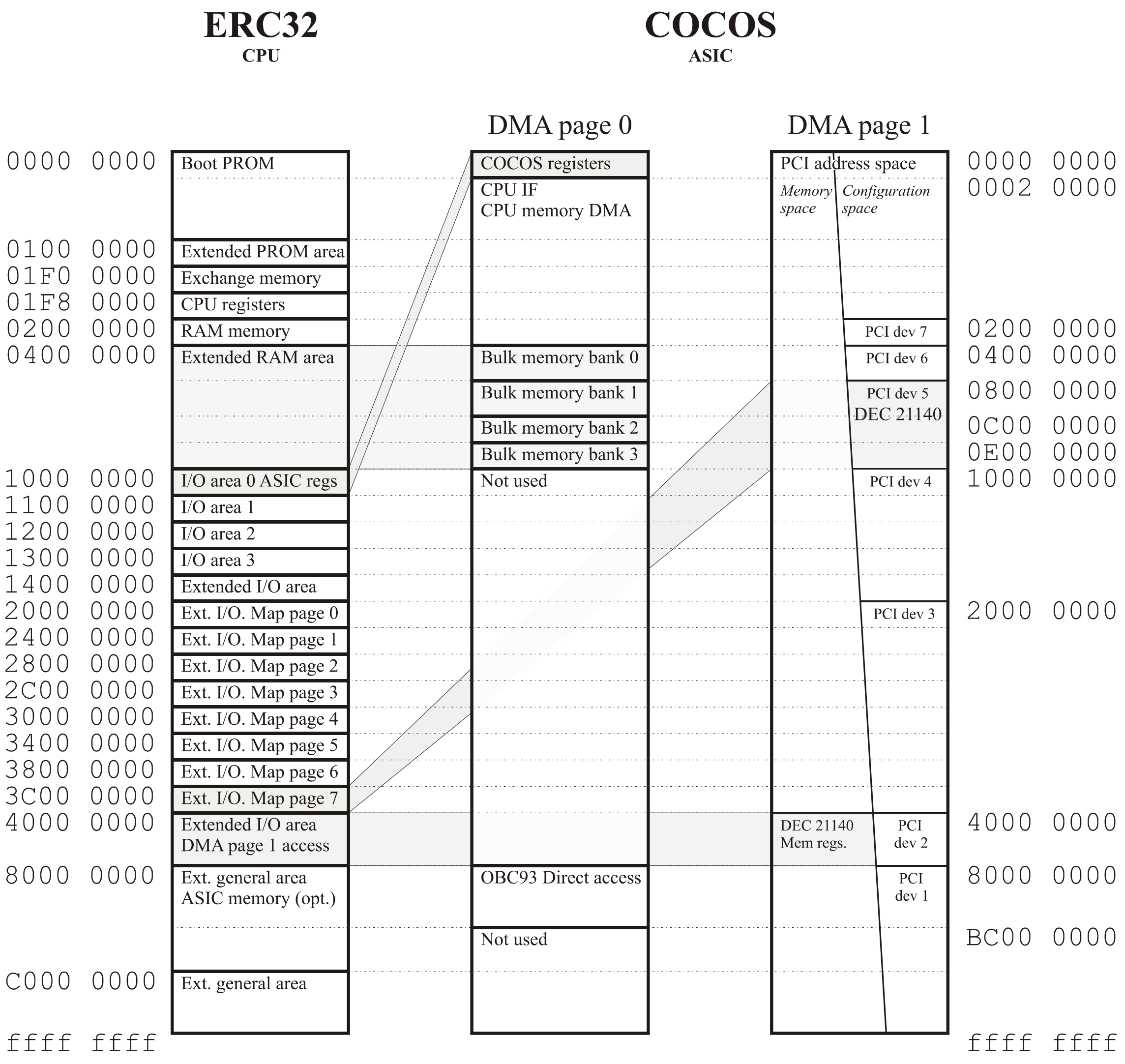


Figure 12: CPU and COCOS memory maps

Notes regarding the mappings (In CPU address space order):

- First mapping is the COCOS ASIC bulk memory in an area referred to as the *Extended RAM area*. This is not used in this project.
- There is a default mapping of the COCOS registers in *I/O area 0 ASIC registers*. This is key to being able to use the COCOS.

- Next we have all the *Extended I/O area Map page* areas. The mapping illustrated in Figure 12 is the *Extended I/O area Map page 7* being mapped to the PCI configuration space onto the PCI slot where the DEC 21140A happened to be inserted. This map page is however repeatedly remapped over almost the whole PCI configuration space during the initial PCI bus device scan (In Figure 12 the scan is started from below, at the higher configuration space slot addresses, and stopped where the DEC 21140A is found).
- The last mapping in this picture is the straight mapping of CPU space addresses between 0x40000000 and 0x80000000 to the same addresses on DMA page 1. As it can only be used for accessing the configuration space of one PCI device, we do not use this mapping in configuration space. We do however, tell the DEC 21140A to map all its memory space registers in this area, as this makes us able to access them directly, without the use of any *CPU IF Map register*.

Please also note how the PCI address space mapping in DMA page 1 is split to illustrate the its two address space modes, configuration space and memory space.

4. Experiences

One major experience has been the value of a stable, feature-complete and dependable development environment, as problems that occur are not necessarily best addressed by changes in the code that is currently being worked on and that caused them to trigger.

4.1 Hardware Related Problems Encountered

This is a description of some of the problems that occurred during our work. There were of course the common issues during development work - programming mistakes, misconfigurations of registers, problems finding documentation – included here are only some of those problems that would have occurred regardless of approach or that carries some other general relevance.

4.1.1 Endian Conflict

Probably the most obvious issue in porting a driver is endianness.

Words that were read and written to memory from the DEC 21140A were interpreted with the wrong endianness. This became obvious from analysing the initial ARP broadcast message in Ethereal, looking at the contents of the PCI bus and comparing the Ethereal interpretation of the interface's MAC address to what it was known to be and how those bytes where grouped into words on the PCI bus.

The ERC32, being a SPARC type CPU, runs natively in big-endian mode, whereas the PCI bus is little endian. That is, when a word is stored in memory, the ERC32 stores the most significant byte at the lowest address, while words are transferred across the PCI bus with the least significant byte at the lowest address. Thus the processor and the PCI bus agree on which byte is at the lowest address, but disagree on whether it is the most or the least significant.

4.1.2 Processor Status Register

Upon writing to the PSR register, the *Processor Status Register*, the system would occasionally hang. This was dealt with by switching off the power to the equipment and leaving it off for a few hours.

Later investigations of this issue carried out by Saab Ericsson Space personnel indicated that this might be due to problems with the floating point, FPU, registers. The version of the PROM based boot-up software used on this model of the test board did not initialise the FPU registers. The parity problems that may occur form this could be the source of the randomly experienced parity problems. Further, upon enabling the FPU registers in RTEMS, there were not enough idle operations (*nop*) between the register write enabling these registers, and the first access to them. That is, the FPU enabling operation had not had time to complete, and they were not ready for use. Both of these events could trigger unwanted interrupts that were not taken care of.

4.1.3 Failure on Large System File Sizes

Upon uploading larger files to the target board it was discovered that when the data segment grew and the total file size exceeded a certain value, RTEMS would not boot any more. The failing address was located in the area occupied by RDBmon, the remote debug monitor to which GDB was connected. The RDBmon software has been widely used before without problems. When the same thing was tried on reference hardware things worked fine, though. This made thorough testing of the extended HTTP server difficult. Only small data files could be attached to the server binary, effectively making it hard to test the network with the HTTP server under anything but light loads.

4.1.4 IP Header Checksums

When trying to initiate TCP/IP connections, a calling client would not accept the TCP acknowledgement sent from the server back to the client to establish the connection.

Table 1: Etherreal traffic log

No	Time	Source	Destination	Port	Info
6	13.9416	Ziatech_01:5c:b6	Compaq_e3:67:b7	ARP	10.131.109.12 is at 00:80:50:01:5c:b6
7	13.9416	10.121.109.4	10.131.109.12	TCP	2522 > 24742 [SYN] Seq=1259371 Ack=0 Win=8192 Len=0
8	13.9454	10.131.109.12	10.121.109.4	TCP	24742 > 2522 [SYN, ACK] Seq=0 Ack=1259372 Win=17520 Len=0
9	16.8640	10.121.109.4	10.131.109.12	TCP	2522 > 24742 [SYN] Seq=1259371 Ack=0 Win=8192 Len=0
10	16.8651	10.131.109.12	10.121.109.4	TCP	24742 > 2522 [ACK] Seq=1 Ack=1259372 Win=17520 Len=0
11	22.8727	10.121.109.4	10.131.109.12	TCP	2522 > 24742 [SYN] Seq=1259371 Ack=0 Win=8192 Len=0
12	22.8737	10.131.109.12	10.121.109.4	TCP	24742 > 2522 [ACK] Seq=1 Ack=1259372 Win=17520 Len=0

As can be seen from the above *Ethereal* network log in Table 1 the remote client tries to set up a TCP connection, and sends a SYN to our listening server. For some reason the acknowledgement, ACK, (and the accompanying server side SYN) to the clients initial SYN is not accepted by the client, which timeouts and retransmits its SYN.

By looking at the packet contents in Ethereal this was established to be due to RTEMS producing a bogus IP header checksum. The reason for this problem was never clearly determined. However, when switching from the optimised SPARC v7 assembler checksum routine to a generic C routine the problem went away. Using the optimised version should however have worked, at least theoretically, as the ERC32 is using the SPARC v7 instruction set. This problem was not further investigated.

Another problem that can be seen from the above list is that there are no retransmissions of the server side SYN on packet no. 8. The reason to that is the absence of a working clock interrupt feature, thus the timeout for that SYN never triggers, and no retransmit occurs, see section 4.1.5.

Looking at the times of the client side SYN retransmissions the TCP back-off behaviour is easy to identify. First timeout and retransmission occurs after three seconds [2] between packet 7 and 9, and then the client waits for double that time for a response, until packet 11.

4.1.5 Clock Interrupt

The clock interrupt problems were perhaps the most significant problem encountered, and could well have jeopardised the whole project. Whenever the clock interrupt was enabled in RTEMS on our test board, RTEMS would hang upon initialisation of the clock interrupt. Thus, we had to try to do without that rather basic feature.

This problems was one of the very first encountered. During the RTEMS network initialisation phase, in a procedure called `rtems_bsdnet_initialize` (See section 3.2.2) the following call was made:

```
rtems_task_wake_after(rtems_bsdnet_ticks_per_second);
```

Execution would not proceed after this line. At that time we did not know what caused this problem, we could not find any proper way to get it to work. However, simply removing that call would allow things to proceed passed this point, down to the device layer, and this enabled us to continue with our work despite the clock problem for the time being.

It is remarkable that it was possible to get anything at all working without this very basic feature of timed interrupts. Most notably in this context, the TCP stack which is heavily dependant on timeouts to occur.

Time interrupts are however also used in other critical places, in the RTEMS kernel.

For example one thing that was also affected by this was time slicing between processes. It does however seem most simple test processes end up in some blocked state sooner or later and therefore CPU control was yielded still, when test software were run during development.

It was not until some of the later stages this caused problems.

When trying to run the control client and server software with several applications on the target ERC32 hardware, these applications would not end up blocked often enough for things to work. Once an application had received control of the CPU it would not surrender control and yield to another application process. Some solution with explicit application initiated yielding or artificial blocks could have been tried to address this problem.

From the network point of view it is hard to tell exactly what impact it had. It was possible to maintain TCP connections. However, the BSD network stack would be unable to handle IP packet loss. Since development took place on a LAN network, the Ethernet adapters took care of retransmissions in case of packet loss due to collisions on the link layer. Therefore IP layer packet loss was a rare thing. An interesting display of the potential problems did however still show up in connection to the IP checksum problems mentioned in section 4.1.4.

Looking at the table (Table 1: Etherreal traffic log, page 42) in 4.1.4 we can see that even though the active client never replies to the ERC32 server's SYN, there is no recorded retransmission. There are several retransmissions from the client due to the missing ACKs (or broken and discarded ACKs rather, due to the IP checksum problem), but none from the server.

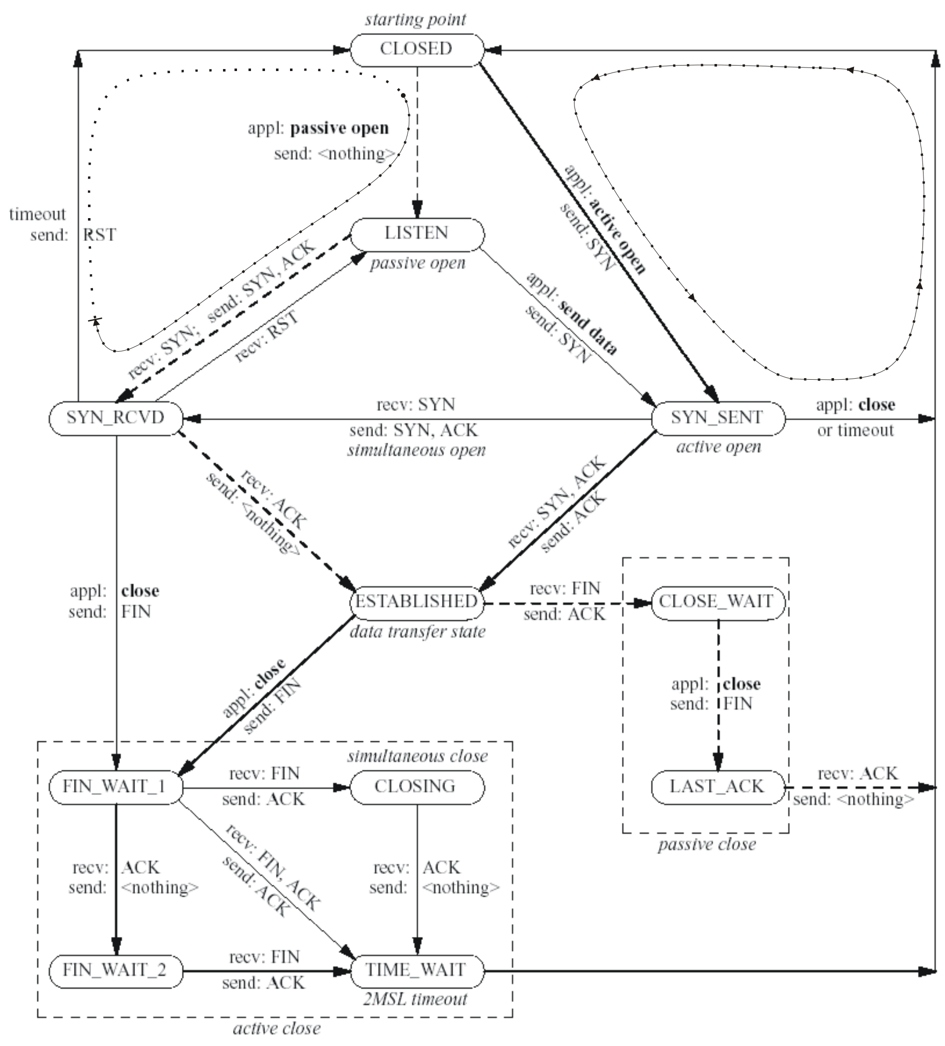


Figure 13: TCP state transition diagram

The two loops in Figure 13 (Illustration is from [18]) shows what is going on. To the right, the *active open* part, the Compaq client, repeatedly gets timeouts in the *SYN_sent* state as it does not understand the bogus IP (due to the broken checksum) SYN and ACK packets the server sends as response and that it needs to proceed to state *established*. It will continue in this circle until it decides to give up and fails the connection attempt.

On the left side, the *passive open* part, the RTEMS server, receives the SYN packets from the client and responds to them with ACK packets, moving to the *SYN_rcvd* state. It also sends its own SYN packet - however, as the Compaq does not respond with the ACK it needs to receive to get to *established*, it too should timeout and end up in a similar circle of retries. The server, however, never gets timed out of the *SYN_rcvd* state.

These clock interrupt problems appeared to be associated with this specific test board model, as another board of the same model experienced the same problems, whereas the newer version, Tiger-ALF, a 3.3 volt board model did not.

4.1.6 Problems Using 100 Mbit/s

Upon connecting to the local network things would work reasonably well (Sometimes connecting to an RTEMS application using telnet did require a retry), while it would not when connecting directly to another single machine running Ethereal. This appeared unlikely, as the local network environment should cause a lot more stress on the interface, considering the amount of traffic of many protocol types.

The only apparent difference that could cause problems was the use of 100 Mbit/s when connecting directly to the other single host. Studying the return status bits indicated that the interface ended up in a transmit underflow condition. This indicating that the transmit buffer were not able to fetch data from the system memory fast enough for keeping up with transmission. This, while being seemingly reasonable, should however not occur, as the interface was set to operate in the *Store and Forward* mode. This mode makes the device wait until the whole frame is in place in buffer memory before starting the transmission. Looking at the PCI bus log it was also established that the frame transfer really took place. Buffer underruns should thus not be possible.

After some research on the Internet, the DE520 network interface errata [4] was found, regarding transmit underflow problems. The DE520 is an interface that, alike the ZT 6650 in this project, uses the DEC 21140A chip.

Studying this errata it was found that there were problems running 100 Mbit/s if the PCI bus frequency was 25 MHz or below. As can be seen with the PCI bus analyser, *BusView*, the clock signal for the Compact PCI bus on the COCOS on our hardware ran at 14 MHz. Thus, avoiding 100 Mbit/s for this combination of low bus frequency and this particular network device chip would be a good idea.

4.1.7 Lost Interrupts

Receiving packets would work reasonably well with small amounts of packets arriving reasonably evenly temporally distributed. However, on a busy network the system would seize to respond on TCP/IP connection requests. This turned out to be due to use of edge-triggered interrupts. The busy hardware would not always trigger when the interrupt signal edge came and thus the interrupt would be lost. Changing to interrupt triggering on signal levels took care of this, as the interrupts now are available for detection until cleared.

4.1.8 IP Address Alignment

When calling methods in the upper network stack layers to construct packets for transmission the system would hang when accessing the message structure. It turned out the IP address was not properly aligned. A method, `ipalign`, which was used by a National Semiconductor Sonic device driver in the development version of RTEMS, was imported to get proper alignment.

4.1.9 ASIC Synthesis Versions

At some occasions there were problems finding an appropriate ASIC synthesis. We learned that the FPGA did not fit all of the functionality of the COCOS ASIC when we tried to get PCI working using an ASIC synthesis that did not include the PCI module. Further, there had been a redesign of the Interrupt Controller, that caused some problems with regard to what documentation was to be used and what synthesises included what version of the Interrupt Controller module.

4.2 RTEMS Related Problems Encountered

4.2.1 IPC

RTEMS has gaps in its POSIX IPC support. It does not support POSIX pipes or AF_UNIX protocol type calls to sockets. As some of those gaps were not too explicitly stated some unfortunate tries to use unimplemented calls were made.

4.2.1.1 Pipes

The `pipe` system call does not exist. This was discovered once a pipe implementation was done and those calls returned error *Function not implemented*.

4.2.1.2 Sockets

The IPC implementation of sockets in the BSD stack is not supported in RTEMS. Those protocol types (e.g. AF_UNIX) are however defined in `socket.h` and the BSD source files, `svc_unix.c` and `clnt_unix.c`, for this feature are present in the source tree. This as the stack is a direct import of the FreeBSD stack.

The presence of those definitions and of the corresponding source code initially made us incorrectly assume that AF_UNIX type socket IPC is possible.

4.2.1.3 Local IP Communication

When the absence of the two above IPC methods had been established, the software was changed to do regular socket calls via the IP layer. The initial approach failed for unknown reasons, as RTEMS would not respond when trying to setup the TCP connection. This despite having no problems connecting to RTEMS sockets several times from outside hosts. It seems RTEMS had issues accepting connections originating from its own IP address. Once a loopback device was defined we could however communicate using the localhost IP.

4.2.2 Descriptor Handling

Means for redirecting and duplicating descriptors is commonly provided by C library functions `dup` and `dup2`. In this case the *standard out* file descriptor to a socket type descriptor. When working reference implementations in Linux had been produced to assure that the function interfaces were properly used to no avail, the *RTEMS users mailing list* were turned to for advice. This bug was unknown to head developers of OAR Corporation, however the networking stack developer promptly confirmed that `dup` and `dup2` indeed were broken.

He kindly presented a workaround solution, involving the use of `fdopen`, to make a file descriptor copy.

4.2.3 Misplaced Assertion

Another problem that occurred was an *assert* in a procedure, `pcib_init`, that checked for PCI BIOS presence on the i386 architecture. However, before the procedure was able to report the absence of such a device an assert checking the very same condition put the system to a halt. This assert was probably left there by lapsus, and was likely triggered by the probably uncommon circumstance that an RTEMS binary was compiled on a PCI capable system, but then executed on non-capable one.

5. Conclusion

5.1 Results

The most important result from this project has been to test all components involved and to show that they can successfully operate together, and to identify problems that need to be addressed. This includes testing the core RTEMS functionality, the BSD TCP/IP stack provided with RTEMS and some of the COCOS ASIC features, in particular the PCI module working towards a real world application PCI bus device. The project has also presented an operational approach to remote application control.

5.2 Future Work

There is room for many refinements and enhancements of this work.

- Enabling the timer interrupt feature.

  This is either done by finding out what is causing it to fail on the ERC32 board, or by simply moving over to the next generation ERC32 board, the Tiger-ALF board. See the item about moving to 3.3 volts technology.

- Properly taking care of all PCI related interrupts from the PCI Module.

  There are a number of error situations, which does trigger interrupts from the PCI Module. E.g. interrupts triggered by the detection of parity errors on the PCI bus, by master or target aborts or by PCI operations getting timeouts. No policy for handling these interrupts is in place today, and they are currently disabled via the interrupt mask.

- Properly initiate all interrupts.

  Ideally, all external COCOS interrupts should be turned off to start with, outside and before the driver, and then enabled as needed by each device.

- Port a device driver supporting 3.3 volts technology.

  Moving away from 5 volts technology has made the continued use of the Ziatech ZT6650 or any other DEC21140A based Ethernet chip impossible. This as this chip is not available in any 3.3-volt versions as is used in the newer Tiger-ALF board.

- Slot dependent PCI interrupt selection.

  The driver is dependent upon the interface being inserted into the proper slot. Even though the driver is dynamic in the sense that it is capable of doing all the address space setup independently, the PCI interrupt setup still ties the Ethernet device to certain slots only. This as different slots uses different PCI interrupts, and the matching that ideally should make the driver set up the correct interrupt for the detected slot is not implemented. Also, some of the physical interrupt lines were not laid out on the PCB, further inhibiting proper interrupt selection.

6. References

1. Atmel (2001). *TSC695F - Rad-Hard 32-bit SPARC Embedded Processor - User's Manual, Revision F*. Nantes, France. Atmel Nantes S.A.
2. Braden, R. (1989). *Requirements for Internet Hosts – Communication Layers*. RFC 1122.
3. Digital Equipment Corp. (1996). *Digital Semiconductor 21140A PCI Fast Ethernet LAN Controller Reference Manual*. Maynard, USA. Digital Equipment Corp. EC-QN7NB-TE.
4. Digital Equipment Corp. (1997). *DE520 Driver Development Information*. Maynard, USA. DEC OEM Business Group.
5. Gaisler, J. (2002) *TSIM Simulator User's Manual, version 1.1*. Gaisler Research.
6. Hjalmarson F. (2002). *COCOS ASIC User's Manual*. Gothenburg, Sweden. Saab Ericsson Space AB. P-ASIC-NOT-00047-SE.
7. Ingraham, A. *PCI-SIG Mailing list - Number of devices*. Posted Nov 2002. Available at <http://www.pcisig.com/reflector/msg05272.html>.
8. Jidmar, H. (2002). *PCI Module User's Manual*. Gothenburg, Sweden. Saab Ericsson Space AB. P-ASIC-NOT-00060-SE.
9. Leffler, S. (1993) *Networking Implementation Notes - 4.4 BSD Ed*. Berkeley, USA. Dept. of Electrical Engineering and Computer Science. SMM: 18.
10. OAR Corp. (2000). *RTEMS C User's Guide*. Huntsville, USA. OAR Corp. 1999-09-25.10
11. OAR Corp. (2000). *RTEMS Network Supplement*. Huntsville, USA. OAR Corp. 1999-09-25.10
12. OAR Corp. *OAR Corp. Home Page*. Available at <http://www.OARcorp.com>.
13. Object Management Group, Inc. *CORBA BASICS*. Updated May 2002. Available at <http://www.omg.org/gettingstarted/corbafaq.htm>.
14. PCI Industrial Computers Manufacturers group. (1997). *CompactPCI Specification Short Form*. Wakefield, USA. Roger Communications. PICMG 2.0 R2.1.
15. PCI Industrial Computers Manufacturers group. *About CompactPCI*. Updated Jan 2003. Available at <http://www.picmg.org/compactpci.stm>.
16. PCI Special Interest Group. (1998). *PCI Local Bus Specification, Revision 2.2*. Hillsboro, USA. PCI Special Interest Group.

17. Peterson, L. (2000). *Computer Networks – A Systems Approach*. San Francisco, USA. Morgan Kaufmann Pub. ISBN 1-55860-514-2.

18. Stevens, R. (1995). *TCP/IP Illustrated, Volume 2 - The Implementation*. Addison-Wesley Publishing Company, Inc.

19. Sun Microsystems, Inc. *Java™ 2 Platform SE Introduction.* Updated January 2003. Available at <http://java.sun.com/j2se/>.

20. Userland Software, Inc. *XML-RPC Specification.* Updated April 27, 2002. Available at <http://www.xmlrpc.com/spec>.

21. W3C. *Simple Object Access Protocol (SOAP) 1.1.* Available at <http://www.w3.org/TR/SOAP/>.

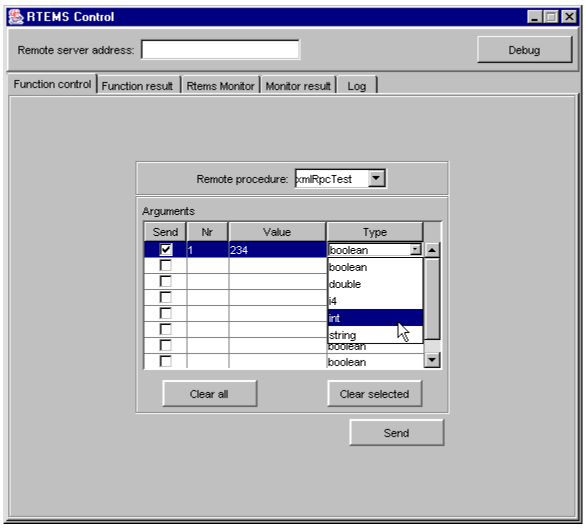
22. Ziatech Corp., (1998). *Hardware manual For ZT 6650 Revision A*. San Luis Obispo, USA. Ziatech Corp. 10256402

7. Appendix

7.1 User Manual

7.1.1 Function Control

The function control tab is used when remote procedure invocation is wanted. In Figure 14 we can see the input field for procedure name and the table where arguments can be defined. In the applet version the procedure name has to be entered manually. The application version tries to find and read a file called `rtemsfunctions.ini` where procedure names can be specified. The names are then available as choices although it is still possible to manually input procedure names.



Remote procedure: xmlRpcTest

Arguments

Send	Nr	Value	Type
☑	1	234	boolean
☐			boolean
☐			boolean
☐			boolean

Figure 14: Function control

Arguments are specified in the table. In each row there is a checkbox that should be checked if the argument should be enabled and included in the procedure call. If it is not checked the argument is ignored. Arguments that are enabled get a number, shown in the second column in the table. This number shows in which order arguments are marshalled and presented to a function. The numbering is a simple enumeration starting at the first argument that is enabled.

When using several possible remote procedures it is convenient to specify arguments of all procedures and for each call enable the ones that belong to that procedure. A call is sent using the send button and possible result of the procedure is presented in function result tab.

7.1.2 Monitor Control

Monitor commands can be invoked using function control tab. Using `xmlRpcMonitor` as procedure name, the command can be entered as argument. Since there is a predefined set of commands that monitor provide, this course of action can be simplified. *Rtems monitor* tab provides a button for each monitor command and possible input can be entered to each command that can take arguments, see Figure 15. A command is sent when corresponding button is pressed and the result is presented in monitor result tab.

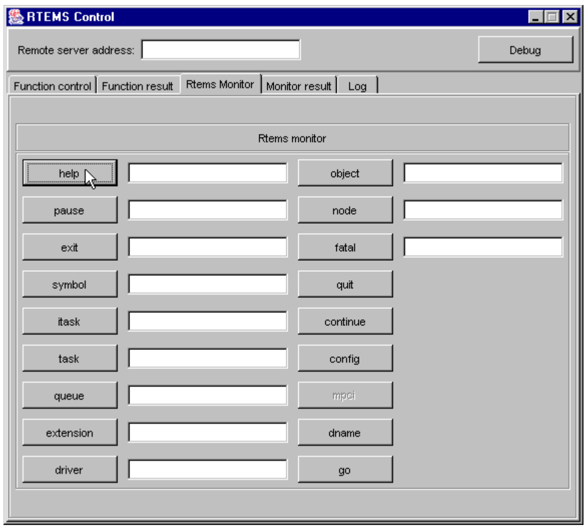


Figure 15: RTEMS Monitor

The commands that can be given to monitor are:

Help	Provides information about commands. Default is to show a basic command summary.
Pause	Monitor goes to sleep for specified number of ticks (default is 1). Monitor will resume at end of period or if explicitly awakened.
Exit	Invoke `rtems_fatal_error_occurred`.
Symbol	Display value associated with the specified symbol. Default to displaying all known symbols.
Itask	List init tasks for the system.
Task	Display information about the specified task. Default is to display information about all tasks on this node.
Queue	Display information about the specified message queues. Default is to display information about all queues on this node.

Extension	Display information about specified extensions. Default is to display information about all extensions on this node.
Driver	Display the RTEMS device driver table.
Object	Display information about specified RTEMS objects.
Node	Specify default node number for commands that take ids.
Fatal	Exit with fatal error, default error is RTEMS_TASK_EXITTED.
Quit	Alias for exit.
Continue	Put the monitor to sleep waiting for an explicit wakeup from running task.
Config	Show the system configuration.
Dname	Display information about named drivers.
Go	Alias for continue.

7.1.3 Log

The log feature is not available in the applet version of the control software. This is due to the security restrictions imposed upon applets.

When a task is enabled to log we can control the task from Log tab, see Figure 16. Data can be presented in a separate window, saved to a file or sent to a task from a file. To start or stop logging a task, its name has to be entered. Tasks must have unique names; RTEMS have no way to guarantee that the proper task is identified if two tasks are given the same name. Task name can be entered manually or can be read from a file called rtemstasks.ini in the client directory. In this file, task names can be specified and therefore available as choices.

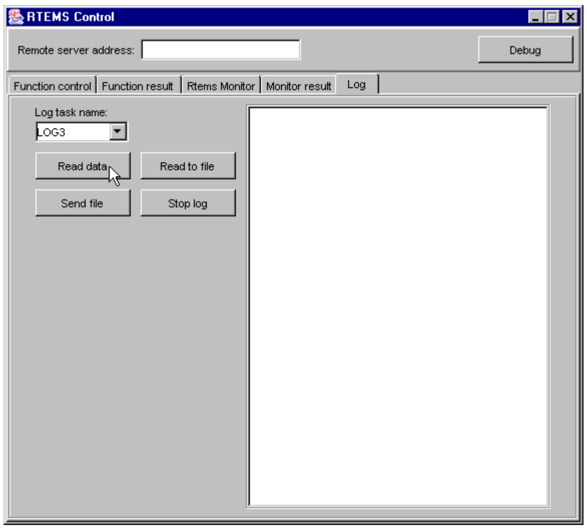


Figure 16: Log

To start reading data from a task, a task is selected. If data should be presented immediately and not saved to a file, button *Read data* is used. Received data appears in a separate window. If the information should be saved to a file button *Save to file* is used and a filename is entered or selected where data is to be put.

Sending data to a task involves the button *Send data*. The source of data is the selected file and data is sent to the task.

To stop logging a task, select task name and press *Stop log* button.

7.1.4 Debug

Inspection of messages sent between client and server is possible. By enabling debug, all XML-RPC messages sent or received are presented in tab *Debug*.

7.2 PCI Bus Operation Log with Comments

This is a commented PCI bus operations' log. It has been of great value when developing the device driver. Though it may seem a bit deterrent it probably provides the best view possible for technical understanding. This, as it shows how the network interface is set up by the device driver, it shows how the BSD network stack deploys the device driver, all from a perspective that also shows how the PCI bus work. Getting this PCI bus view on things is unfortunately rather difficult, as it does require special PCI bus logging hardware.

This is what things can look like on the ERC32 CompactPCI bus when connected to a small two computer's network (This just to reduce the amount of irrelevant traffic):

```
Sample  TimeRel Wait Size Burst Command Address   Data     Status  Err INTx# Ext74
TRIG:     0.0ns    5 AD32 .     ConfRd  80000000  ........ MAbort  --  ----  1111
   1:   2.944us    3 AD32 .     ConfRd  40000000  000116D1 Tdwd    --  ----  1111
   2:   902.7ns    3 AD32 .     ConfRd  40000000  000116D1 Tdwd    --  ----  1111
   3:   902.7ns    3 AD32 .     ConfRd  40000000  000116D1 Tdwd    --  ----  1111
   4:   902.7ns    3 AD32 .     ConfRd  40000000  000116D1 Tdwd    --  ----  1111
   5:   2.071us    5 AD32 .     ConfRd  20000000  ........ MAbort  --  ----  1111
   6:   885.0ns    5 AD32 .     ConfRd  20000000  ........ MAbort  --  ----  1111
   7:   885.0ns    5 AD32 .     ConfRd  20000000  ........ MAbort  --  ----  1111
   8:   885.0ns    5 AD32 .     ConfRd  20000000  ........ MAbort  --  ----  1111
   9:   2.036us    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  10:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  11:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  12:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  13:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  14:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  15:    29.5ns    3 AD32 .     ConfRd  10000000  ........ TdwodTr --  ----  1111
  16:    29.5ns    2 AD32 .     ConfRd  10000000  09608086 Tdwd    --  ----  1111
```

The above sample sequence is continuous, without gaps. However, samples 9 to 15 have status *TdwodTr*. This means *Target Disconnect Without Data* and *Target Retry*. This means the operation failed and will, according to our settings, be retried until successful completion. What causes these to occur has not been investigated, nor what can be done about them. It is however likely that some different timeout setting might affect this behaviour. To reduce the amount of information operations resulting in *TdwodTr* has been filtered away throughout the rest of this PCI log. This is the reason to the gaps in the PCI operation log number sequence.

```
  40:    29.5ns    2 AD32 .     ConfRd  10000000  09608086 Tdwd    --  ----  1111
  46:    29.5ns    2 AD32 .     ConfRd  10000000  09608086 Tdwd    --  ----  1111
  53:    29.5ns    2 AD32 .     ConfRd  10000000  09608086 Tdwd    --  ----  1111
```

What has happened so far is the initial scanning of the PCI bus by the device driver. We can clearly see how the device driver is reading PCI configuration address space, *ConfRd*, scanning for a device driver with a PCI identity that corresponds to the DEC 21140A chip. It starts its scanning the configuration space address with the most significant bit set ($1000\ldots0000_{32}$), that is address 0x80000000, moving on shifting the bit (This bit is called IDSEL in the PCI Specification [16]) one step to the right ($0100\ldots0000_{32}$) to find the next slot's address, i.e. 0x40000000, 0x20000000 and so on until we have found the DEC 21140A.

On our way we encounter slots that have either no devices in them, which results in a *Master Abort* (Mabort), or in the finding of a device other than the one we are looking for, we can see identities such as 000116D1 and 09608086.

```
54:   2.059us    2 AD32 .     ConfRd  08000000  00091011 OK       --  ----  1111
```

We have found what we were looking for, 00091011 is the identity of the DEC 21140A device. The first half of this 32-bit word is the device number, and the last 16 bits identifies the manufacturer. Many of these identities are available on the Internet. Using this resource (http://www.yourvote.com/pci/) to look these numbers up yields:

Vendor ID: 0x1011	**Device ID:** 0x0009
Short Name: DEC	**Chip Number:** DC21140
	Description: Fast Ethernet Ctrlr

Which is just what we would have expected.

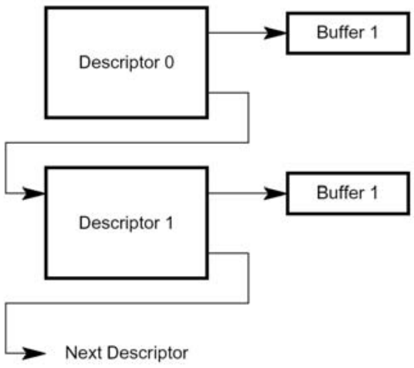
```
55:   979.4ns    2 AD32 .     ConfWr  08000004  00000146 OK       --  ----  1111
56:   802.4ns    2 AD32 .     ConfWr  08000014  40000000 OK       --  ----  1111
57:   796.5ns    2 AD32 .     ConfRd  08000014  40000000 OK       --  ----  1111
58:  777.86us    2 AD32 .     ConfWr  0800001C  00000012 OK       --  ----  1111
59:   15.96us    2 AD32 .     ConfRd  08000008  02000022 OK       --  ----  1111
```

For a more thorough explanation of what goes on here, please check the relevant part of this report, the DEC 21140A documentation or the source code. This is where we do our writes to the PCI configuration space registers, *ConfWr*. Most important to note here is the write to a register at offset 14. This is how we tell the DEC 21140A where we want it to map its memory space registers which, as can be seen by inspecting the data field, is at 0x40000000. This value is then immediately read back and stored by the driver.

```
60:  8.9884ms    3 AD32 .     MemWri  40000030  026CC000 OK       --  ----  1111
61:   64.95us    3 AD32 .     MemWri  40000000  00000001 OK       --  ----  1111
62:   64.95us    3 AD32 .     MemWri  40000000  00404182 OK       --  ----  1111
63:   2.484us    3 AD32 .     MemWri  40000018  02794A58 OK       --  ----  1111
64:   30.21us    3 AD32 .     MemWri  40000020  02794D58 OK       --  ----  1111
65:   2.496us    3 AD32 .     MemWri  40000038  00000000 OK       --  ----  1111
66:   12.14us    3 AD32 .     MemWri  40000030  026CE000 OK       --  ----  1111
```

Most obvious thing to note here is that we have abandoned PCI configuration address space. All operations from now on are done on PCI memory address space. That is, *MemRd* and *MemWri* instead of *ConfRd* and *ConfWr* in this list. Another important thing to note are registers write to offsets 18 and 20. These registers harbour the CPU space addresses of the receive and transmit buffer linked lists, located wherever *malloc* has placed them, in this case, 02794A58 and 02794D58.

Let us look at two pictures, Figure 17 and Figure 18, from the DEC 21140A documentation before moving onto to operations 69 to 78:



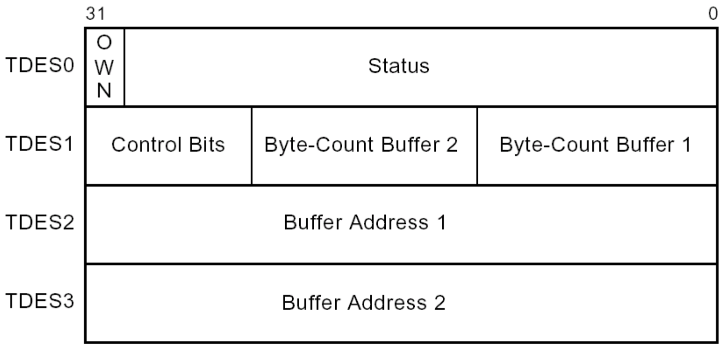


Figure 17: Descriptor chain structure Figure 18: Transmit descriptor format

This left side figure is what the buffer structure looks like, as we are using a *chain structure*, as opposed to a *ring structure*, which also is an option. Which is used is for author of the device driver to decide.

The right side figure shows the contents of one of the two rather similar types of descriptor structure we use, a transmit descriptor.

```
69:     53.1ns    2 AD32 .     MemRd   02794D58  80000000 OK      --  ----  1111
```

We can see how the DEC interface reads the first address (TDES0) in the RAM transmit buffer previously reported to the DEC (See operation 64). Also visible in 69 is that the OWN bit is set, and all others bits are zero – this means that the driver informs the DEC that it owns the descriptor and should process it. By owning we refer to who has the right to write to the memory area in question. It would be a bad thing if the CPU and the DEC would manipulate the same RAM area at the same time.

```
72:     53.1ns    2 AD32 .     MemRd   02794D5C  090000C0 OK      --  ----  1111
```

Next position is the TDES1. The control bit value 9 tells the DEC that the second *Buffer Address* (TDES3) in the descriptor really points to the next descriptor – which is the case for chained descriptor lists. It also tells the DEC interface that this is a *setup frame*, and not an ordinary frame meant for transmission.

```
75:     53.1ns    2 AD32 .     MemRd   02794D60  02794ED8 OK      --  ----  1111
```

This is the pointer (TDES2) to the buffer containing the setup frame. It is located in CPU address space position 0x02794ED8.

```
 78:    53.1ns    2 AD32 .     MemRd   02794D64  02794D70 OK       --  ----  1111
```

This is the pointer to the next descriptor.

```
 81:    53.1ns    2 AD32 .     MemRd   02794ED8  00800080 OK       --  ----  1111
```

The DEC has jumped to the beginning of the buffer containing the setup frame (See 75).

```
 84:    53.1ns    2 AD32 .     MemRd   02794EDC  50015001 OK       --  ----  1111
 87:    53.1ns    2 AD32 .     MemRd   02794EE0  5CB65CB6 OK       --  ----  1111
 90:    53.1ns    2 AD32 .     MemRd   02794EE4  00800080 OK       --  ----  1111
 93:    53.1ns    2 AD32 .     MemRd   02794EE8  50015001 OK       --  ----  1111
 96:    53.1ns    2 AD32 .     MemRd   02794EEC  5CB65CB6 OK       --  ----  1111
 99:    53.1ns    2 AD32 .     MemRd   02794EF0  00800080 OK       --  ----  1111
102:    53.1ns    2 AD32 .     MemRd   02794EF4  50015001 OK       --  ----  1111
```

We skip a big chunk of the log between 102 and 198 here, as it contains identical data. Looking at the contents of the operations 81 to 213, it is easy to spot the recurring pattern in the data read which we have reduced.

The reason to this is that the setup frame contains the Ethernet MAC addresses that the DEC interface should respond to and forward. Also, the setup frame has to be exactly 192 bytes long. This means that we have to enter the same address several times, unless we have enough unique addresses to fill all 192 bytes. Since the this PCI log represents the common case, we only have interest in receiving Ethernet frames destined for our own address and the broadcast address.

As our Ethernet address is 00:80:50:01:5c:b6, it is stored in a somewhat cumbersome way. Only two bytes of each MAC address entry are stored in each word-sized read, which means parts of two entries are read with every 32-bit word. I am confident the reader will find the pattern.

```
198:    53.1ns    2 AD32 .     MemRd   02794F74  00800080 OK       --  ----  1111
201:    53.1ns    2 AD32 .     MemRd   02794F78  50015001 OK       --  ----  1111
204:    53.1ns    2 AD32 .     MemRd   02794F7C  5CB65CB6 OK       --  ----  1111
207:    53.1ns    2 AD32 .     MemRd   02794F80  00800080 OK       --  ----  1111
210:    53.1ns    2 AD32 .     MemRd   02794F84  50015001 OK       --  ----  1111
213:    53.1ns    2 AD32 .     MemRd   02794F88  5CB65CB6 OK       --  ----  1111
216:    53.1ns    2 AD32 .     MemRd   02794F8C  FFFFFFFF OK       --  ----  1111
219:    53.1ns    2 AD32 .     MemRd   02794F90  FFFFFFFF OK       --  ----  1111
222:    53.1ns    2 AD32 .     MemRd   02794F94  FFFFFFFF OK       --  ----  1111
```

The last two positions in the DEC 21140A MAC address table, that is the last three word read operations, contains a new Ethernet address - the *broadcast address*, ff:ff:ff:ff:ff:ff.

Had we not erased some lines in the middle we would have had 192 bytes of setup frame transferred to the DEC by now. That would correspond to 48 lines in this log, as each operation reads a 4-byte word.

```
223:    70.8ns    1 AD32 .     MemWri  02794D58  7FFFFFFF OK      --  ----  1111
```

The DEC returns to the descriptor (TDES0) again and restores the OWN bit, telling the device driver that it is done with the descriptor's contents and that might again be written to by the CPU. All other bits are set to 1, which complies with the DEC specification. The device driver has been waiting in a busy wait for this to happen.

```
227:    53.1ns    2 AD32 .     MemRd   02794D70  00000000 OK      --  ----  1111
```

The DEC checks out the next descriptor by following the pointer previously provided by the second *Buffer Address* field (TDES2), this was done at operation 78. The OWN bit for this descriptor is not set, so the DEC knows that it should not read this descriptor's buffer.

```
230:    53.1ns    2 AD32 .     MemRd   02794D74  01000000 OK      --  ----  1111
```

Comparing to operation 72 - this tells us it is not a setup frame, but it is chained.

```
233:    53.1ns    2 AD32 .     MemRd   02794D78  027954D8 OK      --  ----  1111
236:    53.1ns    2 AD32 .     MemRd   02794D7C  02794D88 OK      --  ----  1111
```

The address of the descriptor's buffer (233), and the address to the next descriptor (236) are read.

```
237:  194.09us    3 AD32 .     MemWri  40000028  000100C0 OK      --  ----  1111
238:   147.5ns    3 AD32 .     MemWri  40000038  000100C0 OK      --  ----  1111
239:   147.5ns    3 AD32 .     MemWri  40000030  026CE002 OK      --  ----  1111
```

When the setup frame has been processed the driver writes to a number of registers (CSR5, CSR7, and CSR6), enabling reception and transmission on the interface and also enabling the generation of interrupts.

```
242:    53.1ns    2 AD32 .     MemRd   02794A58  80000000 OK      --  ----  1111
```

Here it comes, the first access to the receive descriptor chain. Please note that all previous descriptor structure accesses have been to the transmit descriptor chain structure. As with the transmit descriptor case we can see that the OWN bit is set to the DEC for this first descriptor.

```
245:    53.1ns    2 AD32 .     MemRd   02794A5C  FDC00600 OK      --  ----  1111
248:    53.1ns    2 AD32 .     MemRd   02794A60  027CB800 OK      --  ----  1111
251:    53.1ns    2 AD32 .     MemRd   02794A64  02794A70 OK      --  ----  1111
```

The DEC checks out the rest of the descriptor. The status information, the pointer to the first receive buffer, and the pointer to the next frame. Nothing else is done at the moment, as the DEC will have to wait for the arrival of some Ethernet frame before writing into this acquired buffer located at 0x027CB800.

```
252:  9.0850ms    3 AD32 .     MemWri  40000008  00000001 OK      --  ----  1111
```

Here we clearly have a write to some special register address at offset 8. This write, with arbitrary data, by the device driver is to the *Transmit Poll Demand Register* (CSR1), which tells the DEC to go look for something to send.

```
255:    53.1ns    2 AD32 .     MemRd   02794D70  80000000 OK      --  ----  1111
258:    53.1ns    2 AD32 .     MemRd   02794D74  E100003C OK      --  ----  1111
261:    53.1ns    2 AD32 .     MemRd   02794D78  027954D8 OK      --  ----  1111
264:    53.1ns    2 AD32 .     MemRd   02794D7C  02794D88 OK      --  ----  1111
```

Comparing to operation 227, we can see that this time the OWN bit is set. Thus the DEC is informed that this transmit descriptor contains something to send. Looking at the status information in 258, we see that it is still chained (of course) but not a *setup frame*. This will then become our very first transmission! The rest of the status information (the initial 0xE) informs the DEC that this is the first and the last buffer harbouring this frame – thus we will not need to read any additional descriptors before this frame can be transmitted – it all fits in the current buffer.

```
267:    53.1ns    2 AD32 .     MemRd   027954D8  FFFFFFFF OK      --  ----  1111
270:    53.1ns    2 AD32 .     MemRd   027954DC  FFFF0080 OK      --  ----  1111
273:    53.1ns    2 AD32 .     MemRd   027954E0  50015CB6 OK      --  ----  1111
276:    53.1ns    2 AD32 .     MemRd   027954E4  08060001 OK      --  ----  1111
279:    53.1ns    2 AD32 .     MemRd   027954E8  08000604 OK      --  ----  1111
282:    53.1ns    2 AD32 .     MemRd   027954EC  00010080 OK      --  ----  1111
285:    53.1ns    2 AD32 .     MemRd   027954F0  50015CB6 OK      --  ----  1111
288:    53.1ns    2 AD32 .     MemRd   027954F4  0A836D0C OK      --  ----  1111
291:    53.1ns    2 AD32 .     MemRd   027954F8  00000000 OK      --  ----  1111
294:    53.1ns    2 AD32 .     MemRd   027954FC  00000A83 OK      --  ----  1111
297:    53.1ns    2 AD32 .     MemRd   02795500  6D0C0000 OK      --  ----  1111
300:    53.1ns    2 AD32 .     MemRd   02795504  00000000 OK      --  ----  1111
303:    53.1ns    2 AD32 .     MemRd   02795508  00000000 OK      --  ----  1111
306:    53.1ns    2 AD32 .     MemRd   0279550C  00000000 OK      --  ----  1111
309:    53.1ns    2 AD32 .     MemRd   02795510  00000000 OK      --  ----  1111
```

The DEC reads the contents of the frame we want to send, operations 267 to 309. This is 15 word-sized operations, 60 bytes, which also happens to be the minimum Ethernet frame length. Now, what really is this frame we want to send? The obvious solution is reaching for some Computer Networking handbook (In our case [17]) on the bookshelf, and browsing for the specification of the Ethernet format. This first part is however trivial, it is a frame to the broadcast address, ff:ff:ff:ff:ff:ff to our own address, 00:80:50:01:5c:b6. To help us with the rest in this presentation, the Ethereal network traffic analyser has been running on a receiving host.

The Ethereal traffic summary says:

```
No Time        Source            Destination  Protocol  Info
 1 0.000000    Ziatech_01:5c:b6  Broadcast        ARP   Who has 10.131.109.12?
                                                        Tell 10.131.109.12
```

So we have an ARP broadcast, from the ERC32 to the rest of the network. It is asking for its own assigned IP address, checking that no one else has any claims on owning it. As long as no replies are received everything should be fine.

Studying this packet in more detail is also possible in Ethereal:

```
Frame 1 (60 bytes on wire, 60 bytes captured)
    Arrival Time: Jan 24, 2003 17:10:29.288806000
    Time delta from previous packet: 0.000000000 seconds
    Time relative to first packet: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 60 bytes
    Capture Length: 60 bytes
Ethernet II, Src: 00:80:50:01:5c:b6, Dst: ff:ff:ff:ff:ff:ff
    Destination: ff:ff:ff:ff:ff:ff (Broadcast)
    Source: 00:80:50:01:5c:b6 (Ziatech_01:5c:b6)
    Type: ARP (0x0806)
    Trailer: 00000000000000000000000000000000...
Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 00:80:50:01:5c:b6 (Ziatech_01:5c:b6)
    Sender IP address: 10.131.109.12 (10.131.109.12)
    Target MAC address: 00:00:00:00:00:00 (00:00:00_00:00:00)
    Target IP address: 10.131.109.12 (10.131.109.12)

0000  ff ff ff ff  ff ff 00 80  50 01 5c b6  08 06 00 01   ........P.\.....
0010  08 00 06 04  00 01 00 80  50 01 5c b6  0a 83 6d 0c   ........P.\...m.
0020  00 00 00 00  00 00 0a 83  6d 0c 00 00  00 00 00 00   ........m.......
0030  00 00 00 00  00 00 00 00  00 00 00 00                ............
```

From this last raw, uninterpreted part, it is easy to compare and see that Ethereal picks up the same data as is being read from memory by the DEC.

```
 312:    53.1ns    2 AD32 .     MemRd   02794D88  00000000 OK      --  ----  1111
 315:    53.1ns    2 AD32 .     MemRd   02794D8C  01000000 OK      --  ----  1111
 318:    53.1ns    2 AD32 .     MemRd   02794D90  02795AD8 OK      --  ----  1111
 321:    53.1ns    2 AD32 .     MemRd   02794D94  02794DA0 OK      --  ----  1111
```

Next transmit descriptor is at 0x02794D88, as we knew from read operation 264. No owner bit (OWN) is set in this descriptor, so nothing is done except just reading the control value (0x01000000, i.e. chained), the pointers to the buffer (0x02795AD8) and the next descriptor (0x02794DA0).

```
 322:   4.584us    1 AD32 .     MemWri  02794D70  7FFF3000 OK      --  ----  1111
```

Comparing 322 to operation 255, we recall that this write is to the status word of the descriptor belonging to the buffer containing the recently transmitted ARP frame. This is the DEC writing to return the descriptor control to the CPU, and to inform the device driver about the outcome of the buffer read. The upper 0xFFF3 bits are however do-not-care, and the lower status bits 0x000 are all zero, indicating that things went just fine.

```
 323:  188.43us    3 AD32 .     MemRd   40000028  FC660005 OK      --  ----  1111
```

The device driver reads the DEC 21140A *Status Register* (CSR5) for the outcome of the frame transmission.

```
 324:  707.59ms    1 AD32 .     MemWri  027CB800  FFFFFFFF OK      --  ----  1111
 326:    53.1ns    2 AD32 .     MemWri  027CB804  FFFF0002 OK      --  ----  1111
```

```
328:    53.1ns    2 AD32 .     MemWri  027CB808  A526768E OK      --  ----  1111
330:    53.1ns    2 AD32 .     MemWri  027CB80C  08060001 OK      --  ----  1111
332:    53.1ns    2 AD32 .     MemWri  027CB810  08000604 OK      --  ----  1111
334:    53.1ns    2 AD32 .     MemWri  027CB814  00010002 OK      --  ----  1111
336:    53.1ns    2 AD32 .     MemWri  027CB818  A526768E OK      --  ----  1111
338:    53.1ns    2 AD32 .     MemWri  027CB81C  0A797200 OK      --  ----  1111
340:    53.1ns    2 AD32 .     MemWri  027CB820  00000000 OK      --  ----  1111
342:    53.1ns    2 AD32 .     MemWri  027CB824  00000A72 OK      --  ----  1111
344:    53.1ns    2 AD32 .     MemWri  027CB828  C84A0000 OK      --  ----  1111
346:    53.1ns    2 AD32 .     MemWri  027CB82C  00000000 OK      --  ----  1111
348:    53.1ns    2 AD32 .     MemWri  027CB830  00000000 OK      --  ----  1111
350:    53.1ns    2 AD32 .     MemWri  027CB834  00000000 OK      --  ----  1111
352:    53.1ns    2 AD32 .     MemWri  027CB838  00000000 OK      --  ----  1111
```

Another familiar 60 byte ARP packet shows up 0.7 seconds later, but this time it is being received and written to the CPU memory at 0x027CB800 as established by read operation 248. To the Ethernet broadcast address from some host called 00:02:a5:26:76:8e. This happens to be the other host on the network, on which our Ethereal session is running. Looking at the current state of Ethereal traffic summary tells us what we have encountered:

```
No Time         Source            Destination  Protocol  Info
 1 0.000000     Ziatech_01:5c:b6  Broadcast        ARP   Who has 10.131.109.12?
                                                         Tell 10.131.109.12
 2 8.602848     Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
```

The other host, which is a Compaq, with a Compaq Ethernet interface, is looking for some other, unkown, remote host. This is due to some spoilt service being run on the Compaq that will not be able to reach its target host, as it has been connected from the big LAN network, and is only connected to our ERC32 host.

```
354:    53.1ns    2 AD32 .     MemWri  027CB83C  5A81279F OK      --  ----  1111
```

What is this extra junk at the end of the ARP message? For some reason it seems the interface is including the Ethernet CRC checksum into the buffer. The minimum Ethernet frame size is 60 bytes, which has been achieved by padding with zeros, as can be seen in write operations 346 to 352. Added to this are 4 bytes of Ethernet checksum, which are automatically calculated and added by Ethernet interfaces before transmission. Thus the minimum Ethernet frame is 60 bytes, which becomes 64 bytes when transmitted on the medium.

```
356:    53.1ns    2 AD32 .     MemWri  02794A58  00400720 OK      --  ----  1111
```

This is the descriptor picked for storing received messages by operation 242. It is being handed over, via the OWN bit, to the CPU, which reads the frame and does the rest of the processing within the BSD networking stack structure.

```
360:    53.1ns    2 AD32 .     MemRd   02794A70  80000000 OK      --  D---  1111
363:    53.1ns    2 AD32 .     MemRd   02794A74  FDC00600 OK      --  D---  1111
366:    53.1ns    2 AD32 .     MemRd   02794A78  027CB000 OK      --  D---  1111
369:    53.1ns    2 AD32 .     MemRd   02794A7C  02794A88 OK      --  D---  1111
```

The next descriptor is being read and we can see that the PCI interrupt #D goes high to inform the CPU that the previously received ARP packet is ready for processing.

```
370:   513.3ns    3 AD32 .     MemRd   40000028  FC670045 OK      --  D---  1111
371:   147.5ns    3 AD32 .     MemWri  40000028  FC670045 OK      --  D---  1111
```

This is the interrupt handler reading the *Status Register* (CSR5) of the DEC 21140A, and then writing to the same to clear the interrupt.

```
372:  123.34ms    1 AD32 .     MemWri  027CB000  FFFFFFFF OK      --  ----  1111
374:    53.1ns    2 AD32 .     MemWri  027CB004  FFFF0002 OK      --  ----  1111
376:    53.1ns    2 AD32 .     MemWri  027CB008  A526768E OK      --  ----  1111
378:    53.1ns    2 AD32 .     MemWri  027CB00C  08060001 OK      --  ----  1111
380:    53.1ns    2 AD32 .     MemWri  027CB010  08000604 OK      --  ----  1111
```

These operations 372 to 380 are the first few operations of receiving and transmitting two irrelevant ARP frames, not terribly interesting to study in great detail as we have already seen what this looks like. Let us settle for the Ethereal traffic summary:

```
No Time         Source             Destination  Protocol  Info
 1 0.000000     Ziatech_01:5c:b6   Broadcast        ARP   Who has 10.131.109.12?
                                                          Tell 10.131.109.12
 2 8.602848     Compaq_26:76:8e    Broadcast        ARP   Who has 10.114.200.74?
                                                          Tell 10.121.114.0
 3 10.098734    Compaq_26:76:8e    Broadcast        ARP   Who has 10.114.200.74?
                                                          Tell 10.121.114.0
 4 11.539243    Compaq_26:76:8e    Broadcast        ARP   Who has 10.131.109.12?
                                                          Tell 10.121.114.0
```

Having skipped ARP frames 3 to 4 in the Ethereal traffic summary, we join in again before transmitting frame 5.

```
468:   79.30us    3 AD32 .     MemWri  40000008  00000001 OK      --  ----  1111
```

The *Transmit Poll Demand* register is written to. We want to send something.

```
471:    53.1ns    2 AD32 .     MemRd   02794D88  80000000 OK      --  ----  1111
474:    53.1ns    2 AD32 .     MemRd   02794D8C  E1000040 OK      --  ----  1111
477:    53.1ns    2 AD32 .     MemRd   02794D90  02795AD8 OK      --  ----  1111
480:    53.1ns    2 AD32 .     MemRd   02794D94  02794DA0 OK      --  ----  1111
```

Checking the current transmit descriptor.

```
483:    53.1ns    2 AD32 .     MemRd   02795AD8  0002A526 OK      --  ----  1111
486:    53.1ns    2 AD32 .     MemRd   02795ADC  768E0080 OK      --  ----  1111
489:    53.1ns    2 AD32 .     MemRd   02795AE0  50015CB6 OK      --  ----  1111
493:    53.1ns    2 AD32 .     MemRd   02795AE4  08060001 OK      --  ----  1111
496:    53.1ns    2 AD32 .     MemRd   02795AE8  08000604 OK      --  ----  1111
499:    53.1ns    2 AD32 .     MemRd   02795AEC  00020080 OK      --  ----  1111
502:    53.1ns    2 AD32 .     MemRd   02795AF0  50015CB6 OK      --  ----  1111
505:    53.1ns    2 AD32 .     MemRd   02795AF4  0A836D0C OK      --  ----  1111
508:    53.1ns    2 AD32 .     MemRd   02795AF8  0002A526 OK      --  ----  1111
511:    53.1ns    2 AD32 .     MemRd   02795AFC  768E0A79 OK      --  ----  1111
514:    53.1ns    2 AD32 .     MemRd   02795B00  72000000 OK      --  ----  1111
517:    53.1ns    2 AD32 .     MemRd   02795B04  00000000 OK      --  ----  1111
520:    53.1ns    2 AD32 .     MemRd   02795B08  00000000 OK      --  ----  1111
523:    53.1ns    2 AD32 .     MemRd   02795B0C  00000000 OK      --  ----  1111
526:    53.1ns    2 AD32 .     MemRd   02795B10  00000000 OK      --  ----  1111
529:    53.1ns    2 AD32 .     MemRd   02795B14  A1C2D090 OK      --  ----  1111
```

The ARP packet is 60 bytes, with the addition 4 byte checksum this makes 64 bytes or 16 word-sized operations.

```
532:    53.1ns    2 AD32 .     MemRd   02794DA0  00000000 OK      --  ----  1111
535:    53.1ns    2 AD32 .     MemRd   02794DA4  01000000 OK      --  ----  1111
538:    53.1ns    2 AD32 .     MemRd   02794DA8  027960D8 OK      --  ----  1111
541:    53.1ns    2 AD32 .     MemRd   02794DAC  02794DB8 OK      --  ----  1111
542:   4.844us    1 AD32 .     MemWri  02794D88  7FFF3000 OK      --  ----  1111
```

Reading the descriptor of the next transmit buffer, and returning the old one. The Ethereal summary:

```
No Time         Source            Destination  Protocol  Info
 1 0.000000     Ziatech_01:5c:b6  Broadcast         ARP   Who has 10.131.109.12?
                                                          Tell 10.131.109.12
 2 8.602848     Compaq_26:76:8e   Broadcast         ARP   Who has 10.114.200.74?
                                                          Tell 10.121.114.0
 3 10.098734    Compaq_26:76:8e   Broadcast         ARP   Who has 10.114.200.74?
                                                          Tell 10.121.114.0
 4 11.539243    Compaq_26:76:8e   Broadcast         ARP   Who has 10.131.109.12?
                                                          Tell 10.121.114.0
 5 11.540431    Ziatech_01:5c:b6  Compaq_26:76:8e  ARP   10.131.109.12 is at
                                                          00:80:50:01:5c:b6
```

Now, as previously stated, the ARP frames 2 and 3 are rather irrelevant, as they involve a host not present on the network. In frame 4, however, the Compaq remote host want to know the MAC address of some IP 10.131.109.12. This IP is the ERC32 host, and ARP frame 5 from the ERC32 host answers this request.

```
543:   17.52us    1 AD32 .     MemWri  027CA000  00805001 OK      --  ----  1111
545:    53.1ns    2 AD32 .     MemWri  027CA004  5CB60002 OK      --  ----  1111
547:    53.1ns    2 AD32 .     MemWri  027CA008  A526768E OK      --  ----  1111
549:    53.1ns    2 AD32 .     MemWri  027CA00C  08004500 OK      --  ----  1111
551:    53.1ns    2 AD32 .     MemWri  027CA010  002CC1CD OK      --  ----  1111
553:    53.1ns    2 AD32 .     MemWri  027CA014  40008006 OK      --  ----  1111
555:    53.1ns    2 AD32 .     MemWri  027CA018  44F60A79 OK      --  ----  1111
557:    53.1ns    2 AD32 .     MemWri  027CA01C  72000A83 OK      --  ----  1111
559:    53.1ns    2 AD32 .     MemWri  027CA020  6D0C0E97 OK      --  ----  1111
561:    53.1ns    2 AD32 .     MemWri  027CA024  60A6002A OK      --  ----  1111
563:    53.1ns    2 AD32 .     MemWri  027CA028  24010000 OK      --  ----  1111
565:    53.1ns    2 AD32 .     MemWri  027CA02C  00006002 OK      --  ----  1111
567:    53.1ns    2 AD32 .     MemWri  027CA030  2000F0B5 OK      --  ----  1111
569:    53.1ns    2 AD32 .     MemWri  027CA034  00000204 OK      --  ----  1111
571:    53.1ns    2 AD32 .     MemWri  027CA038  05B40000 OK      --  ----  1111
573:    53.1ns    2 AD32 .     MemWri  027CA03C  29624CC8 OK      --  ----  1111
```

The end of the received frame.

```
575:    53.1ns    2 AD32 .     MemWri  02794AA0  00400320 OK      --  ----  1111
```

This is the descriptor of this received message. It is being handed over, via the OWN bit, to the CPU.

```
579:    53.1ns    2 AD32 .     MemRd   02794AB8  80000000 OK      --  D---  1111
582:    53.1ns    2 AD32 .     MemRd   02794ABC  FDC00600 OK      --  D---  1111
585:    53.1ns    2 AD32 .     MemRd   02794AC0  027C9800 OK      --  D---  1111
588:    53.1ns    2 AD32 .     MemRd   02794AC4  02794AD0 OK      --  D---  1111
```

The DEC reads the next descriptor and PCI interrupt #D informs the CPU that there is something new available for processing in the receive buffer.

```
 589:   501.5ns    3 AD32 .     MemRd   40000028  FC670045 OK      --  D---  1111
 590:   147.5ns    3 AD32 .     MemWri  40000028  FC670045 OK      --  D---  1111
```

Interrupts are cleared by the interrupt handler, the `dec21140Enet_interrupt_handler`.

Let us check the Ethereal traffic log:

```
No Time         Source           Destination  Protocol  Info
 1 0.000000     Ziatech_01:5c:b6  Broadcast        ARP   Who has 10.131.109.12?
                                                         Tell 10.131.109.12
 2 8.602848     Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
 3 10.098734    Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
 4 11.539243    Compaq_26:76:8e  Broadcast        ARP   Who has 10.131.109.12?
                                                         Tell 10.121.114.0
 5 11.540431    Ziatech_01:5c:b6  Compaq_26:76:8e  ARP   10.131.109.12 is at

 6 11.540514    10.121.114.0     10.131.109.12    TCP   3735 > 24742 [SYN]
                                                         Seq=2761729  Ack=0
                                                         Win=8192     Len=0
```

We have a TCP connection request, from the Compaq on port 3735 to the ERC32 on port 24742. Skipping a few more operations, we can see how the TCP connection is set up with two additional Ethernet frames, 7 and 8, with TCP packet payloads.

```
 6 11.540514    10.121.114.0     10.131.109.12    TCP   3735 > 24742 [SYN]
                                                         Seq=2761729  Ack=0
                                                         Win=8192     Len=0
 7 11.544149    10.131.109.12    10.121.114.0     TCP   24742 > 3735 [SYN, ACK]
                                                         Seq=0        Ack=2761730
                                                         Win=17520    Len=0
 8 11.544273    10.121.114.0     10.131.109.12    TCP   3735 > 24742 [ACK]
                                                         Seq=2761730  Ack=1
                                                         Win=8760     Len=0

 699:    53.1ns    2 AD32 .     MemRd   02794AD0  80000000 OK      --  D---  1111
 702:    53.1ns    2 AD32 .     MemRd   02794AD4  FDC00600 OK      --  D---  1111
 705:    53.1ns    2 AD32 .     MemRd   02794AD8  027C9000 OK      --  D---  1111
 708:    53.1ns    2 AD32 .     MemRd   02794ADC  02794AE8 OK      --  D---  1111
 709:   501.5ns    3 AD32 .     MemRd   40000028  FC670045 OK      --  D---  1111
 710:   147.5ns    3 AD32 .     MemWri  40000028  FC670045 OK      --  D---  1111
```

These are the last few operations following frame, or IP packet, number 8. Interrupt #D is being cleared to acknowledge the handover of this TCP ACK from the DEC to the Compaq.

Next is another burst of 7 irrelevant packets, which we skip.

```
 9 11.598681    Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
10 13.098680    Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.75?
                                                         Tell 10.121.114.0
11 14.598646    Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.75?
                                                         Tell 10.121.114.0
12 16.098593    Compaq_26:76:8e  Broadcast        ARP   Who has 10.114.200.75?
                                                         Tell 10.121.114.0
13 17.598540    10.121.114.0     10.255.255.255   NBNS  Name query NB NTSRV9<20>
14 18.348531    10.121.114.0     10.255.255.255   NBNS  Name query NB NTSRV9<20>
15 19.098532    10.121.114.0     10.255.255.255   NBNS  Name query NB NTSRV9<20>
```

Moving right along in the PCI log; next come two TCP/IP packets, in two frames:

```
1090:  153.89ms    1 AD32 .     MemWri  027C5800  00805001 OK      --  ----  1111
1092:    53.1ns    2 AD32 .     MemWri  027C5804  5CB60002 OK      --  ----  1111
1094:    53.1ns    2 AD32 .     MemWri  027C5808  A526768E OK      --  ----  1111
1096:    53.1ns    2 AD32 .     MemWri  027C580C  08004500 OK      --  ----  1111
1098:    53.1ns    2 AD32 .     MemWri  027C5810  002CCACD OK      --  ----  1111
1100:    53.1ns    2 AD32 .     MemWri  027C5814  40008006 OK      --  ----  1111
1102:    53.1ns    2 AD32 .     MemWri  027C5818  3BF60A79 OK      --  ----  1111
1104:    53.1ns    2 AD32 .     MemWri  027C581C  72000A83 OK      --  ----  1111
1106:    53.1ns    2 AD32 .     MemWri  027C5820  6D0C0E97 OK      --  ----  1111
1108:    53.1ns    2 AD32 .     MemWri  027C5824  60A6002A OK      --  ----  1111
1110:    53.1ns    2 AD32 .     MemWri  027C5828  24020000 OK      --  ----  1111
1112:    53.1ns    2 AD32 .     MemWri  027C582C  00015018 OK      --  ----  1111
1114:    53.1ns    2 AD32 .     MemWri  027C5830  2238274B OK      --  ----  1111
1116:    53.1ns    2 AD32 .     MemWri  027C5834  00006F6C OK      --  ----  1111
1118:    53.1ns    2 AD32 .     MemWri  027C5838  6F660000 OK      --  ----  1111
1120:    53.1ns    2 AD32 .     MemWri  027C583C  3BDD5F7B OK      --  ----  1111
1122:    53.1ns    2 AD32 .     MemWri  02794B78  00400320 OK      --  ----  1111
1126:    53.1ns    2 AD32 .     MemRd   02794B90  80000000 OK      --  D---  1111
1129:    53.1ns    2 AD32 .     MemRd   02794B94  FDC00600 OK      --  D---  1111
1132:    53.1ns    2 AD32 .     MemRd   02794B98  027C5000 OK      --  D---  1111
1135:    53.1ns    2 AD32 .     MemRd   02794B9C  02794BA8 OK      --  D---  1111
1136:   513.3ns    3 AD32 .     MemRd   40000028  FC670040 OK      --  D---  1111
1137:   147.5ns    3 AD32 .     MemWri  40000028  FC670040 OK      --  D---  1111
```

The first one above has a TCP packet payload being read during operations 1116 and 1118, containing the data 0x00006F6C and 0x6F660000, which corresponds to ASCII characters “olof”.

```
1138:   3.664us    1 AD32 .     MemWri  027C5000  00805001 OK      --  ----  1111
1140:    53.1ns    2 AD32 .     MemWri  027C5004  5CB60002 OK      --  ----  1111
1142:    53.1ns    2 AD32 .     MemWri  027C5008  A526768E OK      --  ----  1111
1145:    53.1ns    2 AD32 .     MemWri  027C500C  08004500 OK      --  ----  1111
1147:    53.1ns    2 AD32 .     MemWri  027C5010  002ACBCD OK      --  ----  1111
1149:    53.1ns    2 AD32 .     MemWri  027C5014  40008006 OK      --  ----  1111
1151:    53.1ns    2 AD32 .     MemWri  027C5018  3AF80A79 OK      --  ----  1111
1153:    53.1ns    2 AD32 .     MemWri  027C501C  72000A83 OK      --  ----  1111
1155:    53.1ns    2 AD32 .     MemWri  027C5020  6D0C0E97 OK      --  ----  1111
1157:    53.1ns    2 AD32 .     MemWri  027C5024  60A6002A OK      --  ----  1111
1159:    53.1ns    2 AD32 .     MemWri  027C5028  24060000 OK      --  ----  1111
1161:    53.1ns    2 AD32 .     MemWri  027C502C  00015018 OK      --  ----  1111
1163:    53.1ns    2 AD32 .     MemWri  027C5030  2238F911 OK      --  ----  1111
1165:    53.1ns    2 AD32 .     MemWri  027C5034  00000D0A OK      --  ----  1111
1167:    53.1ns    2 AD32 .     MemWri  027C5038  00000000 OK      --  ----  1111
1169:    53.1ns    2 AD32 .     MemWri  027C503C  E6630033 OK      --  ----  1111
1171:    53.1ns    2 AD32 .     MemWri  02794B90  00400320 OK      --  ----  1111
1175:    53.1ns    2 AD32 .     MemRd   02794BA8  80000000 OK      --  D---  1111
1178:    53.1ns    2 AD32 .     MemRd   02794BAC  FDC00600 OK      --  D---  1111
1181:    53.1ns    2 AD32 .     MemRd   02794BB0  027C4800 OK      --  D---  1111
1184:    53.1ns    2 AD32 .     MemRd   02794BB4  02794BC0 OK      --  D---  1111
1185:   501.5ns    3 AD32 .     MemRd   40000028  FC670040 OK      --  D---  1111
1186:   147.5ns    3 AD32 .     MemWri  40000028  FC670040 OK      --  D---  1111
```

The second one has a TCP packet payload being read during operations 1165 and 1167, containing the data 0x00000D0A and 0x00000000, which corresponds to ASCII characters *line feed* and *carriage return*.

```
16 20.969533   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
                                                        Seq=2761730  Ack=1
                                                        Win=8760     Len=4
17 20.969592   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
                                                        Seq=2761734   Ack=1
                                                        Win=8760      Len=2
```

What happens next?

```
1187:  631.35us    3 AD32 .     MemWri  40000008  00000001 OK      --  ----  1111
```

Transmit Poll Demand, this time we send something.

```
1190:    53.1ns    2 AD32 .     MemRd   02794DB8  80000000 OK      --  ----  1111
1193:    53.1ns    2 AD32 .     MemRd   02794DBC  E100003C OK      --  ----  1111
1196:    53.1ns    2 AD32 .     MemRd   02794DC0  027966D8 OK      --  ----  1111
1199:    53.1ns    2 AD32 .     MemRd   02794DC4  02794DD0 OK      --  ----  1111
```

Operations 1190 to 1199, reading the transmit descriptor.

```
1202:    53.1ns    2 AD32 .     MemRd   027966D8  0002A526 OK      --  ----  1111
1205:    53.1ns    2 AD32 .     MemRd   027966DC  768E0080 OK      --  ----  1111
1208:    53.1ns    2 AD32 .     MemRd   027966E0  50015CB6 OK      --  ----  1111
1211:    53.1ns    2 AD32 .     MemRd   027966E4  08004500 OK      --  ----  1111
1214:    53.1ns    2 AD32 .     MemRd   027966E8  002E0001 OK      --  ----  1111
1217:    53.1ns    2 AD32 .     MemRd   027966EC  40004006 OK      --  ----  1111
1220:    53.1ns    2 AD32 .     MemRd   027966F0  46C10A83 OK      --  ----  1111
1223:    53.1ns    2 AD32 .     MemRd   027966F4  6D0C0A79 OK      --  ----  1111
1226:    53.1ns    2 AD32 .     MemRd   027966F8  720060A6 OK      --  ----  1111
1229:    53.1ns    2 AD32 .     MemRd   027966FC  0E970000 OK      --  ----  1111
1232:    53.1ns    2 AD32 .     MemRd   02796700  0001002A OK      --  ----  1111
1235:    53.1ns    2 AD32 .     MemRd   02796704  24085018 OK      --  ----  1111
1238:    53.1ns    2 AD32 .     MemRd   02796708  4470F800 OK      --  ----  1111
1241:    53.1ns    2 AD32 .     MemRd   0279670C  00006F6C OK      --  ----  1111
1244:    53.1ns    2 AD32 .     MemRd   02796710  6F660D0A OK      --  ----  1111
1247:    53.1ns    2 AD32 .     MemRd   02794DD0  00000000 OK      --  ----  1111
1250:    53.1ns    2 AD32 .     MemRd   02794DD4  01000000 OK      --  ----  1111
1253:    53.1ns    2 AD32 .     MemRd   02794DD8  02796CD8 OK      --  ----  1111
1256:    53.1ns    2 AD32 .     MemRd   02794DDC  02794DE8 OK      --  ----  1111
1257:   4.584us    1 AD32 .     MemWri  02794DB8  7FFF3000 OK      --  ----  1111
```

These previous two TCP packets are responded to with the above packet, operations 1202 to 1256. Let us look at the full interpreted details of this packet:

```
Frame 18 (60 bytes on wire, 60 bytes captured)
    Arrival Time: Jan 24, 2003 17:10:50.266278000
    Time delta from previous packet: 0.007880000 seconds
    Time relative to first packet: 20.977472000 seconds
    Frame Number: 18
    Packet Length: 60 bytes
    Capture Length: 60 bytes
Ethernet II, Src: 00:80:50:01:5c:b6, Dst: 00:02:a5:26:76:8e
    Destination: 00:02:a5:26:76:8e (Compaq_26:76:8e)
    Source: 00:80:50:01:5c:b6 (Ziatech_01:5c:b6)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 10.131.109.12 (10.131.109.12), Dst Addr:
10.121.114.0 (10.121.114.0)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 46
    Identification: 0x0001
    Flags: 0x04
```

```
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0x46c1 (correct)
    Source: 10.131.109.12 (10.131.109.12)
    Destination: 10.121.114.0 (10.121.114.0)
Transmission Control Protocol, Src Port: 24742 (24742), Dst Port: 3735 (3735),
Seq: 1, Ack: 2761736, Len: 6
    Source port: 24742 (24742)
    Destination port: 3735 (3735)
    Sequence number: 1
    Next sequence number: 7
    Acknowledgement number: 2761736
    Header length: 20 bytes
    Flags: 0x0018 (PSH, ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 1... = Push: Set
        .... .0.. = Reset: Not set
        .... ..0. = Syn: Not set
        .... ...0 = Fin: Not set
    Window size: 17520
    Checksum: 0xf800 (correct)
Data (6 bytes)

0000  00 02 a5 26  76 8e 00 80  50 01 5c b6  08 00 45 00   ...&v...P.\...E.
0010  00 2e 00 01  00 00 40 06  46 c1 0a 83  6d 0c 0a 79   ....@.@.F...m..y
0020  72 00 60 a6  0e 97 00 00  00 01 00 2a  24 08 50 18   r.`........*$.P.
0030  44 70 f8 00  00 00 6f 6c  6f 66 0d 0a               Dp....olof..
```

The only thing I am going to point out in this context is the echoed reply of the two previous packets, containing the sequence 6f 6c 6f 66 0d 0a, this is the “olof” string followed by the *line feed* and the *carriage return*.

What we have done is talking to an echo server on the ERC32 host. Telneting to this server running on port 24742 from the Compaq computer, writing olof, followed by hitting the *carriage return* key, and getting it straight back as an echo from the server on the ERC32 host.

This was frame 18. No more operations will be accounted for. A few more packets are sent, and then the TCP shutdown sequence is initiated. The whole sequence look like:

```
No Time         Source            Destination  Protocol  Info
 1 0.000000     Ziatech_01:5c:b6  Broadcast        ARP   Who has 10.131.109.12?
                                                         Tell 10.131.109.12
 2 8.602848     Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
 3 10.098734    Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.74?
                                                         Tell 10.121.114.0
 4 11.539243    Compaq_26:76:8e   Broadcast        ARP   Who has 10.131.109.12?
                                                         Tell 10.121.114.0
 5 11.540431    Ziatech_01:5c:b6  Compaq_26:76:8e  ARP   10.131.109.12 is at
                                                         00:80:50:01:5c:b6
 6 11.540514    10.121.114.0      10.131.109.12    TCP   3735 > 24742 [SYN]
 7 11.544149    10.131.109.12     10.121.114.0     TCP   24742 > 3735 [SYN, ACK]
 8 11.544273   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [ACK]
 9 11.598681   Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.74?
                                                        Tell 10.121.114.0
10 13.098680   Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.75?
                                                        Tell 10.121.114.0
11 14.598646   Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.75?
                                                        Tell 10.121.114.0
12 16.098593   Compaq_26:76:8e   Broadcast        ARP   Who has 10.114.200.75?
                                                        Tell 10.121.114.0
13 17.598540   10.121.114.0      10.255.255.255   NBNS  Name query NB NTSRV9<20>
14 18.348531   10.121.114.0      10.255.255.255   NBNS  Name query NB NTSRV9<20>
15 19.098532   10.121.114.0      10.255.255.255   NBNS  Name query NB NTSRV9<20>
16 20.969533   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
17 20.969592   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
18 20.977472   10.131.109.12     10.121.114.0     TCP   24742 > 3735 [PSH, ACK]

19 21.129743   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [ACK]
20 29.017364   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
21 29.017456   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [PSH, ACK]
22 29.025247   10.131.109.12     10.121.114.0     TCP   24742 > 3735 [PSH, ACK]
23 29.223486   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [ACK]
24 37.923518   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [FIN, ACK]
25 37.924646   10.131.109.12     10.121.114.0     TCP   24742 > 3735 [ACK]
26 37.929675   10.131.109.12     10.121.114.0     TCP   24742 > 3735 [FIN, ACK]
27 37.929823   10.121.114.0      10.131.109.12    TCP   3735 > 24742 [ACK]
```

7.3 Development Environment

The development environment can be divided in the following items:

- Hardware
- The RTEMS operating system
- Software development tools

7.3.1 Hardware

ERC32 is a radiation-tolerant embedded SPARC V7 instruction set compatible processor developed for space applications. It is an open 32-bit CPU, developed with support from the European Space Agency (ESA), where the VHDL models have been made freely available under the GNU LGPL license.

The ERC32 implementation used in this project is called TSC695F and is manufactured by Atmel Wireless & Microcontrollers.

The ERC32 is connected to a general CPU and IO controller ASIC called COCOS. The CPU support functions include a programmable watchdog and an alarm signal generator. IO interfaces available via this ASIC includes UART, 1553, Spacewire and the PCI interface deployed in this project.

7.3.2 RTEMS

RTEMS stands for Real-Time Executive for Multiprocessor and was developed by On-Line Applications Research Corporation (OAR) for the U.S. Army. The original goal of RTEMS was to provide a portable, standards-based real-time executive with available source code. RTEMS is licensed under a modified version of the GNU General Public License (GPL).

In order to support different hardware environments, RTEMS contains a number of BSPs (Board Support Package). A BSP is a collection of device drivers, startup code, linker scripts, and compiler support files that tailor RTEMS for a particular target hardware environment.

RTEMS is available for the following processor families:

Motorola MC68xxx	Motorola MC683xx	AMD A29K
Motorola ColdFire	Hitachi SH	Intel i386
Intel i960	MIPS	PowerPC
SPARC	Hewlett-Packard PA-RISC	

A set of managers in the executive core provides functionality that applications can use, see Figure 19. Unused managers, except interrupt manager, can be optionally excluded from the runtime environment in order to save memory space.

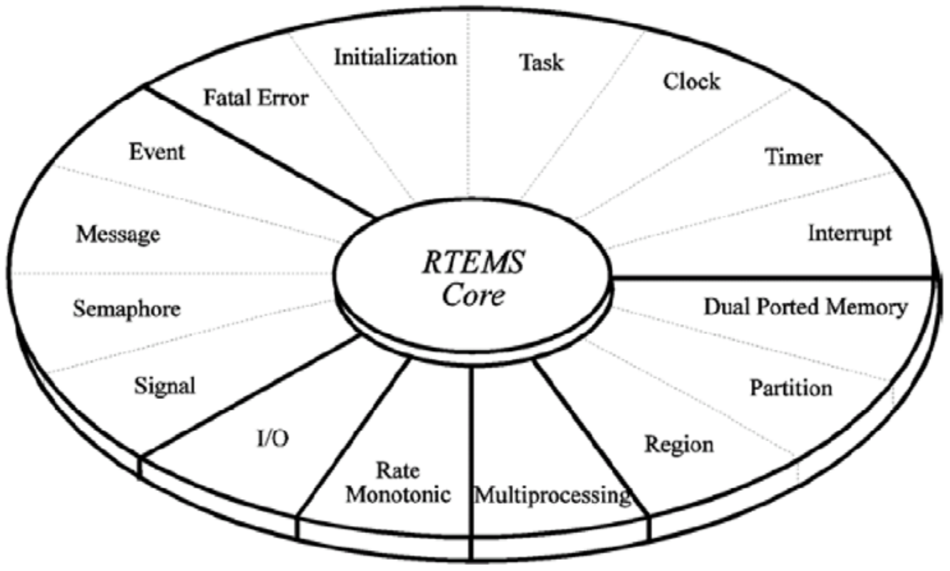


Figure 19: Overview of RTEMS managers

RTEMS is of object-oriented nature, and objects are assigned an ID and a name. Objects such as tasks, queues, events, signals, semaphores, and memory blocks can be designated as global objects and accessed by any task regardless of which processor the object and the accessing task reside.

A task is RTEMS's version of a thread and for each task configured during system initialisation, RTEMS reserves a control block where context of the task is stored. The tasks compete for CPU time and the scheduling of each task is based on its priority and current state.

Applications in RTEMS are started using a special type of task, the *Initialization task*. This task is automatically created and started by RTEMS in the initialisation sequence. Its purpose is to reserve memory, create application tasks and transfer initial control to the user's application. It can also be used to create other objects used by application tasks.

RTEMS contain a TCP/IP network stack, which is a port of the FreeBSD network stack. Networking in RTEMS is described in section 3.2.1, Network Processes and Data Flow.

RTEMS is distributed by OAR Corporation. Source code and documentation are available at their site [12].

7.3.3 Software Used

BusView 3.01 by VMetro

The software used for accessing the PCI log device. Presents a log of operations, interrupts and other useful information when doing development on involving PCI technology.

GNU tools

Standard software development tools like Emacs and GDB were used. We also used the terminal software Tip to connect to the ERC32 RS-232 console interface.

HyperTerm 690170 by Hilgraeve Inc

Terminal software provided with Microsoft Windows NT, version 4.

JTAG Programmer 3.3WP8.x by Xilinx Inc.

Software used for programming the FPGA using the JTAG interface.

TSIM for ERC32 SPARC 1.1.3 by Gaisler Research

Hardware simulator initially used for exploring the RTEMS OS environment.

RDBMon 1.3.3 by Gaisler Research

Remote GDB debugger run on the ERC32 platform.

ERC32SC SPARCMon 2.0 by Saab Ericsson Space AB

PROM software used for booting the ERC32 hardware.

Ethereal 0.9.7 by Gerald Combs

Used for analysing individual Ethernet packets sent by the interface as well as monitoring TCP/IP communication.