The Implementation of a Core Architecture for Geophysical Data Acquisition

THESIS

Submitted in fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

of

Rhodes University

by
Ray Edward Heasman

January 2000

Abstract

This thesis describes the design, development and implementation of the core hardware and software of a modular data acquisition system for geophysical data collection. The primary application for this system is the acquisition and realtime processing of seismic data captured in mines. This system will be used by a commercial supplier of seismic instrumentation, ISS International, as a base architecture for the development of future products.

The hardware and software has been designed to be extendable and support distributed processing. The IEEE-1394 High Performance Serial Bus is used to communicate with other CPU modules or peripherals.

The software includes a pre-emptive multitasking microkernel, an asynchronous mailbox-based message passing communications system, and a functional IEEE-1394 protocol stack.

The reasons for the end design and implementation decisions are given, and the problems encountered in the development of this system are described. A critical assessment of the match between the requirements for the project and the functionality of the implementation is made.

Acknowledgements

I would like to express my gratitude to ISS International and the Rhodes University Physics & Electronics department for making this MSc possible. Special thanks must go to my supervisors, Richard Grant, Justin Jonas, and Peter Mountfort for making this project a pleasant and worthwhile learning experience. Peter Mountfort deserves special thanks for his thoughtful mentoring while I worked for ISSI in Welkom. I would also like to thank those at ISSI that made my life so pleasant by being friendly and helpful.

Finally, this thesis would not be complete without mention of the love and support from my family that has made all that I have done possible.

Preface

This thesis describes the design and implementation of the core of an architecture that will replace and extend systems currently in use by ISS International (ISSI)[1] for gathering geophysical data. Although this is a research project, submitted in fulfillment of a MSc degree, it has its own context that is flavoured by business considerations. To fully understand the reasoning behind the requirements, one has to have a feel for the context in which the architecture was designed, and the situation that made its development necessary.

The status quo

The current systems provided by ISSI for data acquisition are optimised for a particular environment, namely deep gold mines. Although the systems are very successful and well regarded by customers and competitors alike, their current implementation makes them inflexible and difficult to tailor to new applications.

They are designed for an environment that is relatively harsh, where communications are provided through (usually) sub-standard cables, but where electrical power is plentiful. Also, they are designed to trigger on pertinent events, and to store the resultant data for subsequent forwarding to a central site. They have limited sample-rate and dynamic range, mediocre distortion requirements, and are simple gatherers of data.

Problems with the status quo

Although the current range of "intelligent seismometers" (as called by ISSI) are a good match to their requirements, it was felt that ISSI should try to gain market share in other areas of geophysical data acquisition. Better flexibility means lower risk for the company, and hopefully more profits and happier customers. Customers frequently ask for some level

[1]ISSI can be contacted at:
P.O. Box 12063
Die Boord
7613
South Africa
Tel : +27 (0)21 809-2060
Fax : +27 (0)21 809-2061
Email : iss@cape.issi.co.za

of customisation in their implementations, and this has often resulted in ISSI making small production runs of a customised printed circuit board and the associated firmware.

Also, the current seismometers are based on the INMOS Transputer range of CPUs, and ISSI has been compromised by the cessation of all support for, and production of, this CPU. It is at this point that ISSI decided to start a research project on potential solutions to their problems, and I became involved.

Contents

1 Introduction **1**

1.1 The current system 1

1.1.1 The current system hardware 1

1.1.2 The current system firmware 2

1.2 My objectives 3

2 The design requirements **4**

2.1 What was asked for 4

2.2 Discussion 6

2.3 Deciding on a solution 7

2.3.1 Serendipity 7

2.3.2 Other contenders for connecting modules 8

3 My proposed solution **9**

3.1 The IEEE-1394 High Performance Serial Bus 9

3.2 Finding a suitable CPU 10

3.2.1 The ARM7TDMI CPU core 12

3.3 Partitioning the problem 14

3.3.1 Constituents of a module 14

3.4 My objectives revisited 15

4 The hardware implementation **16**

4.1 The overall board design 16

4.2 Problems encountered 18

4.2.1 Building and testing the prototype 20

4.2.2 Other problems with the prototypes 22

5 The software implementation **25**

5.1 The language and development environment 26

5.2 The implementation of a kernel 27

5.2.1 The communication subsystem 28

5.2.2 A suggested programming model for this kernel 31

5.2.3 The primitives provided 32

5.2.4 The kernel structure 33

5.3 Problems encountered 36

5.4 A simple test 39

5.5 The IEEE-1394 protocol stack 42

5.5.1 Overview of a generic IEEE-1394 stack 43

5.5.2 The significance of the CSR Architecture primitives 44

5.5.3 My 1394 protocol stack implementation 45

5.6 Testing the 1394 stack 49

5.6.1 Stages of implementation and testing 49

5.6.2 Summary 50

6 Consolidation **51**

6.1 The hardware 51

6.2 The software 53

6.2.1 The kernel 53

6.2.2 The Inter-Process Communication system 54
6.2.3 The IEEE-1394 stack 55
6.3 Parts of the implementation that turned out well 55
6.3.1 The kernel and message passing system 55
6.3.2 The hardware 56
6.4 Conclusion 56

References 58

A Schematics 60

B PCB Artwork 66

C Photograph 75

D Source Code 76

List of Figures

1.1 The legacy ISSI hardware. 2

3.1 An example of how an IEEE-1394 address is formed. 11

4.1 An example of the envisaged system. 17

4.2 An overview of the prototype board. 19

4.3 A screenshot of the program used to design the PCB artwork, and the completed 6-layer prototype board. The PCB artwork can be found in appendix B. 21

5.1 A simple example of how processes would communicate via mailboxes. Each process only has to know the name of a mailbox to communicate effectively. 30

5.2 The functional dependencies of the functions provided by the kernel and the message passing system. 34

5.3 The interrupt handling in the kernel and how it fits together with the context switcher and message passing (MP). 35

5.4 The structure of an IEEE-1394 stack. More detailed versions of this diagram can be found in [IEEE1394, pg 22] and [And98, pg 43]. 43

5.5 An overview of my IEEE-1394 stack implementation. 46

5.6 The queue structure used to implement the transaction layer functions. . . 48

List of Tables

3.1 A summary of the processors considered. 13

5.1 The different possible types of packet available in IEEE-1394, from [IEEE1394, 6.2.4.5]. Unspecified values are reserved. 47

Chapter 1

Introduction

This project was intended to provide a potential software and hardware solution that would meet most, if not all, requirements of the company for the near and, hopefully, also the unforeseen future. A solution that matched all the requirements would be impossible, but ISSI wanted some idea of how such a system could be built. If the concept and implementation were to show promise, they would develop it further for their own use. It would be my task to try and provide an implementation that would be a good compromise between the given requirements and other factors such as cost and extensibility.

1.1 The current system

The current “Multi Seismometer” (the most recent seismic data-collection device available from ISSI) is a device for the collection of seismic data. At the user level, it is a device that samples and buffers data from several geophones or accelerometers, triggers on "meaningful" events, and passes them to a central site. These devices are scattered around a volume within a mine, and their results are processed by the central site to provide the location and intensity of seismic events occurring.

1.1.1 The current system hardware

The current system comprises an Analogue to Digital Converter (ADC) card, an Inmos Transputer-based CPU card, a boot ROM card and a modem card, all communicating

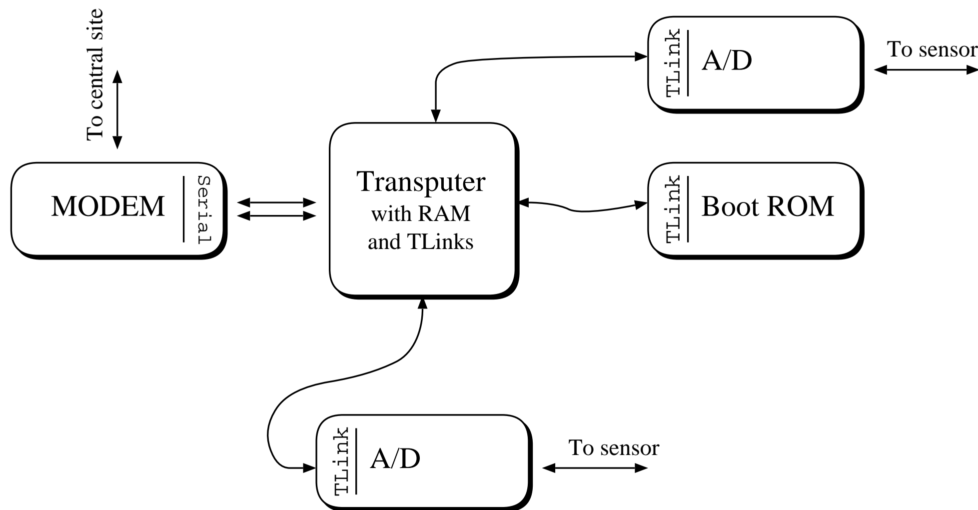


Figure 1.1: The legacy ISSI hardware.

with each other. The ADC card and Transputer communicate with each other using the transputer's serial links.

There are limitations to what the current system can do. It is limited to one processor per box - and that processor is not powerful enough for many potential applications. Although the transputer can physically connect to three ADC cards, each of which has three channels, it cannot process the data from those cards fast enough. This is because in applications where it becomes useful to place seismic sensors close to each other, they need to be sampled at higher rates. At the moment, the on-board CPU handles all control logic (such as data buffering and packetising for the modem), as well as deciding whether the incoming sampled data is "meaningful" (See section 1.1.2). For some potential applications, the CPU simply doesn't have sufficient processing capabilities. For others, the speed of the Transputer communication links becomes a limitation.

1.1.2 The current system firmware

The current system has one transputer per device, and this CPU handles all the processing for this device. It collects data from the input ADC cards via the transputer serial links, runs the data stream through a triggering algorithm to decide whether some event worth recording has occurred, and adds the stream to an event queue if it qualifies. A central site is kept aware of the status of the event queue and requests events in its own time. The CPU responds with the requested data.

The trigger algorithm usually used in the current system is based on a comparison between long and short term averages of the magnitude of the incoming data. If the short term average suddenly increases to some multiple of the long term average, then something interesting is deemed to have occurred. Future systems will use more complex triggering algorithms, no triggering algorithms at all, or other algorithms that characterise the activity rather than recording it.

1.2 My objectives

My objectives for this project were to produce a hardware unit, with attendant firmware, that would test the feasibility of my hardware and software design. A complete implementation was deemed to be beyond the scope of an MSc project, but my prototype should show that the overall design philosophy would be a good match to the problem.

Chapter 2

The design requirements

The design requirements for this project resulted from interviewing members of several different departments in ISSI. Each department had its own priorities which often conflicted with those of other departments, therefore the final requirements and their relative priorities were mine to choose.

2.1 What was asked for

The Research, Development, and Production departments were the three that I had to poll. Below is an unprioritised list of the requirements provided by these three departments. They are copied directly out of my notebook, with some contextual comments added:

- In some situations, the system might be run as a standalone device - so it would have to store data locally.
- It might have to support a telephone modem, if used as a standalone device.
- A digital system should have short (if any) analogue cable-runs, to minimise the distortion of the signal to be measured.
- The new system should have better data processing capabilities than the current system.
- The new system should be able to support a Digital Signal Processor (DSP) if necessary.

- It should be able to accept many different sensor inputs, such as strain gauges or radiation detectors. For example, one system could monitor both seismic and non-seismic information in a mine.
- It should be able to synchronise sampling across several boards; this is useful for seismic reflectometry.
- It must be able to communicate via a long distance cable, modem, or with another like system; thus improving the flexibility of the system in the field.
- Certain applications could require at least 4 MB of on-board memory per ADC, as a result of using higher input sampling rates.
- It should be able to drive outputs for on-site warnings, indicators or the like; since electric signals travel faster than seismic waves, a system that activates a remote safety measure as soon as it detects a large local event could result in the difference between mild injury and death.
- It would be beneficial if multiple systems, or system CPUs, could co-operate, allowing seamless expansion of a system to higher volume data inputs.
- It should provide a consistent, flexible, software interface that would allow migration of processing algorithms down the system hierarchy from the central servers towards the sensors.
- It should consume as little power as possible, as required by standalone systems that are expected to be powered by batteries or solar panels.
- The base system should be as cheap as possible, preferably cheaper than the current Transputer-based system sold by ISSI.
- It should be modular.
- It should be flexible, and allow for a simple base configuration.
- It must allow for daisy chaining of systems.
- It must be simple for the end user to configure and use.
- It should not be tied to the transputer platform.

- It should use a CPU with a clean memory model and instruction set.
- It should not incur license fees of any type on ISSI.

These requirements are presented in no particular order and are often contradictory. It is practically impossible to make a device that would match all of the specifications, particularly that of price. It is difficult to make a single board that would work well and allow for unspecified expansion in the future.

2.2 Discussion

As I spoke to more people, it became clear that flexibility and modularity would be two of the most important needs for the future. The Research department had hopes of implementing new processing algorithms at the sensor that would be CPU intensive. However, only some systems would need these algorithms. The Development department was concerned about costs associated with customising designs for individual customers, and the power requirements and cost of a new system. The Production department wanted a system that could allow the minimum number of standard Printed Circuit Boards (PCBs) to be made.

Other requirements come from envisioned future applications of a system. For instance, seismic reflectometry would require sensors to be sampled synchronously. However, one potential application cannot dictate a feature for a design at the expense of other goals. I needed to come to a compromise that would provide a simple architecture, with a low base cost, and that would still serve many of the potential applications.

I decided that the truly important and feasible requirements were the following:

- It must be modular.
- It must be flexible, and allow for a simple base configuration.
- It must use as little power as possible.
- Multiple systems, or system CPUs, must co-operate, so as to allow seamless expansion (within practical limits) of a system to higher volume data inputs.

- It must provide a consistent, flexible, software interface that allows migration of processing algorithms down the system hierarchy from the central site towards the sensors.
- It must be simple for the end user to and configure and use.
- It must be able to communicate via a long distance cable, modem, or with another like system.
- The base system should be as cheap as possible, preferably cheaper than the current system sold by ISSI.
- It must allow for timing information to be propagated from, or to, all ADC boards.

By satisfying these requirements I hoped to meet the others in a cost effective manner - perhaps not all at once, but at least those required for a particular application.

2.3 Deciding on a solution

A modular solution to a problem implies that the individual components in a system are grouped into modules which are reasonably independent of each other, except for the exchange of control and data signals with the rest of the world. The more autonomous each module, the easier it becomes to re-arrange and mix-and-match modules. However, this is balanced by a lack of flexibility in the more complex module, and a higher component cost per module. Thus, modules can only be made more flexible for a given complexity by making them more expensive.

2.3.1 Serendipity

It was while I was pondering this annoying conflict of requirements that I was made aware of the IEEE-1394 High Performance Serial Bus (henceforth called IEEE-1394) by a fellow MSc student, Robert Laubscher, who was using it in his project [Lau99]. It is also known by two different trade names, "FireWire", trademarked by Apple, and "iLink", trademarked by Sony. On further investigation, it became clear that IEEE-1394 would solve many of my problems, making it simpler and cheaper to separate modules and yet allowing for a powerful and flexible method of interfacing them. A survey of other buses did not provide a better solution.

2.3.2 Other contenders for connecting modules

There are many ways to join electronic modules. I considered using a shared bus, TTL serial links, SHARC DSP serial links, line drivers and others. A shared bus was not attractive because it involved far too many signals, and problems such as signal skewing seemed likely. This left many variations on serial links. A pure TTL synchronous serial connection is reasonably fast but has limited distance at higher speeds. It also requires hop-to-hop wiring and software routing, which would load the communicating MCUs. Additionally, it would not provide the sort of data rates that I thought I might need. If I used an Analog Devices SHARC DSP as the MCU, then I would get high speed serial communications for free, but the SHARCs were too expensive for my needs.

Line drivers would increase the distance between modules, which I didn't really need. The Universal Serial Bus (USB) looked inviting, yet didn't meet my requirements for bandwidth. The IEEE-1394 bus had many of the advantages of USB, but was also much faster (400 Mbps available today as against 12Mbps).

Chapter 3

My proposed solution

If I assumed that I could get a cheap processor that was still good enough for most applications, then I could afford to put a CPU and an IEEE-1394 chipset in each module. This drove me to survey the processors available. Ideally, I wanted a CPU that was very cheap, and provided DSP-like facilities without using use a DSP-like instruction set (this was one of the requirements that the Development department stipulated - they wanted high-level compilers to generate efficient code for the target architecture). After some searching, I managed to find such a processor. (See section 3.2)

3.1 The IEEE-1394 High Performance Serial Bus

The IEEE-1394 bus is a group of point-to-point links that emulate a bus at the chipset level. Each node signals only to those nodes connected directly to it. However, those nodes repeat the transmission to their neighbours. This is done at the bit level [IEEE1394, pg 98], where each bit is received, retimed to a local bit clock, and resent.

IEEE-1394 allows 63 nodes on a bus and a total of 1023 buses. Nodes are connected in a tree-like structure, with the limitations that no more than 16 nodes can be connected in a row, and that there are no closed loops in the tree. (An example of how this could be configured is shown in Figure 4.1). Each node can communicate at 100Mbps, 200Mbps, or 400Mbps, occurring over two twisted pairs. There are two variants of the IEEE-1394 cable. The standard cable has the two twisted pairs, a shield, and power and ground lines. The cable intended for consumer applications, where size is critical, does not include the power and ground lines.

An IEEE-1394 chipset provides all the services up to the Link Layer, and in the incarnation that I used (a chipset from Texas Instruments), comes in two chips - the LLC, or Link Layer Controller, and the PHY, or the Physical media interface.

The IEEE-1394 bus can be viewed as a write-only bus. It provides a read, a write, and a lock transaction, where each type of transaction can be interpreted as a write that may request a write in return. Thus, a read is a write requesting a write in return from the other node.

IEEE-1394 also provides an implementation of the IEEE-1212 Control and Status Register Architecture [IEEE1394, pg 10]. This is a useful abstraction wherein each logical unit on the IEEE-1394 bus provides memory-mapped registers that are used to communicate with the unit. In effect, each device is memory-mapped into a 64-bit address space. 10 bits are used to specify the bus, 6 bits to specify the node. A node address of 63 (i.e. all ones in binary) is the broadcast address, and a bus address of 1023 (again, all ones in binary) corresponds to the local bus. The remaining 48 bits are memory-mapped within a node. The memory is not directly mapped to the node's internal RAM; the mapping is emulated by the node software. One section of this mapped area is the configuration ROM of the node. This ROM can have one of several different layouts, depending upon the level of support the node provides for the CSR Architecture. Some layouts allow for a node to define its own memory areas for one or more logical units, where a unit is a logical service or device on that node. An example clarifying the layout of the IEEE-1394 address space is given in Figure 3.1.

The CSR Architecture allows for a node to export certain services using a well understood interface. External hardware or software need never know how those facilities are provided, and the implementation can be hidden from external devices.

3.2 Finding a suitable CPU

Once again, I was forced to decide what my priorities were in choosing a component. In this case, I needed a CPU that was both cheap and powerful. It had to be sparing in power consumption and integrated enough to be simple to use (usually, these integrated CPUs are called MCUs, short for "Micro-Controller Units"). Also, the CPU would probably be required to do arithmetic on 24-bit data. This ruled out many potential candidates.

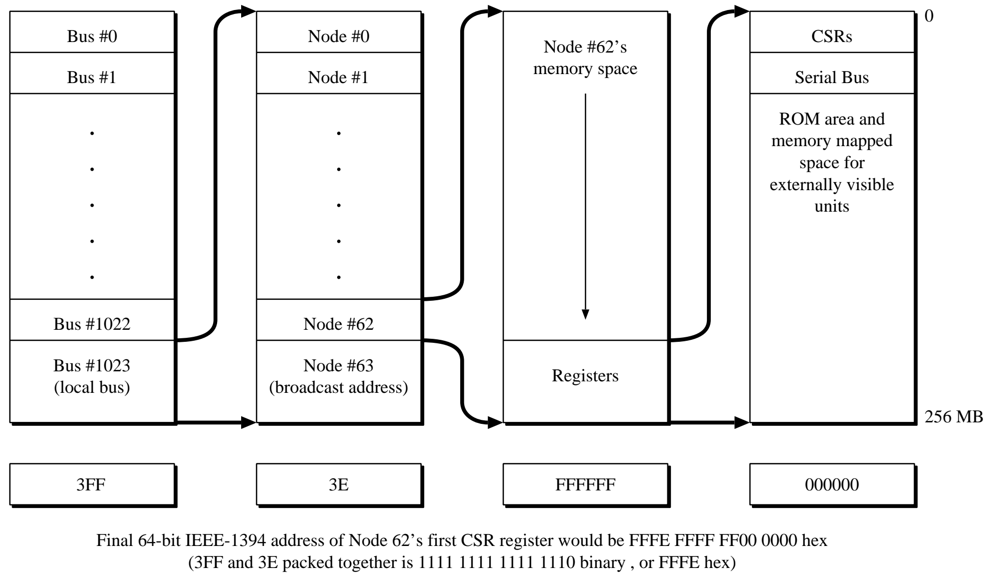


Figure 3.1: An example of how an IEEE-1394 address is formed.

I considered a wide range of CPUs and culled the undesirables using more and more strict requirements. Initially I wanted a CPU that would be "fast enough" in arithmetic and had the ability to access at least two megabytes of memory without paging. This ruled out any 8-bit processors such as the 8051, Z80, and the Atmel AVR RISCs. The larger CPUs tended either to have complicated and expensive memory interfaces, used too much power, or were too expensive. MCUs were a better choice, as their higher level of integration saved on board space and number of components. This left me with the Motorola ColdFire MCUs, the embedded 68000 derivatives (MC683xx), the SGS Thomson 16x microcontrollers, and several ARM-core based MCUs such as the Sharp LH77790, the GEC Plessey Butterfly, the Samsung KS32C6000, and the Atmel AT91 series.

It became clear that a chip with a 16-bit bus was preferable to a chip with a 32-bit bus. Fewer lines to drive meant better power consumption. It also meant fewer pins on the chip, less board space, and fewer RAM chips would be required.

The MC68331 used too much power compared to some of the other chips available, wasn't as fast, and was slightly too expensive. Yet it did have the advantage of being readily available.

The ColdFire CPUs were simply too expensive, given their capabilities.

The KS32C6000, designed for use in laser printers, had functionality that I didn't need; nevertheless, it was attractive in that it could use DRAM without any extra circuitry,

DRAM being a lot cheaper per megabyte than SRAM. However, I eventually decided that it would be more and more difficult to obtain the DRAM that we required, as the consumer market is moving away from SIMMS. Also I was hoping to make the entire board work at 3.3V, to save power, and the KS32C6000 was a 5V component.

The Butterfly was very attractive, as it ran at 3.3V (albeit at 15MHz), with good integration. But despite using the ARM7 core, it was outclassed by other ARM7-based MCUs that had the T and M extensions. (See section 3.2.1.)

The SGS Thomson ST10R163 was tempting, with the ability to address 16 MB of memory, multiply two 16-bit values at 25MHz in 10 clock cycles (i.e. 400 ns), and had DMA controllers for handling its on-board peripherals. It also had a clever system of register banks, making context switches extremely fast. The ST10R163/165/166/167 MCUs had much the same attributes, with variations in peripherals integrated and the amount of internal RAM.

After a fairly exhaustive search, I settled on a short list of three MCUs, namely, the SGS Thomson ST10R163, the Atmel AT91M40400, and the Motorola MC68331. The ST10R163 was only 16-bit but was cheap and fast for its class. The MC68331 was easily available, with a 32-bit instruction set, yet slightly expensive.

All of these were reasonably integrated, fast, and low cost. However, the Atmel chip, because it used an ARM7TDMI core and could run at 33MHz at 3.3V, matched my requirements far better than the others. Also, since the AT91 series is based on a core licensed to many manufacturers, it is unlikely that ISSI would again find itself in the same predicament as it does now with the discontinued Transputer. For instance, future designs could be based on a better MCU from another manufacturer using the same CPU core.

3.2.1 The ARM7TDMI CPU core

The ARM7TDMI core is produced by Advanced RISC Machines and licensed to parties who need a CPU but do not wish to develop their own. The ARM7 is a full 32 bit RISC processor, providing impressive performance and power consumption using very little silicon real estate. The "TDMI" suffix signifies certain additional features:

The "T" indicates that the core supports the "Thumb" instruction set, a subset of the ARM instructions which are compressed down to 16 bits per instruction. These are decompressed "on-the-fly" by the core into ARM instructions and allow the ARM to run much faster

Processor	For	Against
ColdFire	Clean architecture	Price too high
GEC Butterfly	Well integrated, available	Low clock freq.
KS32C6000	Supports DRAM	5V part
ST10R163 and variants	Clever architecture	16-bit mult.
MC68331	Easily available	Slow
Z80, 8051, AVR	Cheap, low power	8-bit
LH77790	Highly integrated	Low clock freq.
AT91M40400	$32 \times 32 + 64$ MAC	No cache
CL-PS7110	Highly integrated	Too many extras
V850/V831	Fast MIPS based	Not available

Table 3.1: A summary of the processors considered.

[ARM96, Sec 1.2.1] on a 16-bit bus. It has been shown that the ARM7TDMI core running in Thumb mode out of 16-bit RAM runs at about 160% of the speed of the same core using ARM mode. Code size is also reduced to about 65% of the size of an equivalent program expressed in ARM opcodes. The "DI" suffixes together mean that the core integrates a debugger called an "EmbeddedICE". This allows the user to stop, start and breakpoint the CPU while it is running. This is done using a synchronous serial connection to the "JTAG" pins on the CPU. These pins are used on other chips for testing the chip during production, but in this case they are used at run-time for debugging. Other facilities for breakpointing the CPU when a particular bit pattern is fetched or written are also included. The "M" shows that the core has an extended multiplier to accelerate multiplication operations.

The ARM7TDMI is capable of multiplying two 32-bit operands to return a 64-bit result, and adding the result to a 64-bit accumulator register pair, doing so in only seven clock cycles, worst case [ARM96, Sec 4.8.3]. It uses an early exit multiplier, so the best case execution time for the instruction is four clock cycles.

The AT91M40400 is an MCU with 4 KB of on-board RAM, an extended interrupt controller that allows vectored interrupts, two UARTs with DMA facilities, three 16-bit timers, and some parallel IO ports. It can run at 3.3V at 33MHz and draws an absolute maximum of 60mA while doing so [Atm98, pg 4]. Experience has shown that the power consumption of this MCU is typically much lower than this and is usually dwarfed by the other devices on the board.

3.3 Partitioning the problem

It is all fine and well to say that each module will come complete with a CPU and an IEEE-1394 chipset, but this begs the question, “what constitutes a module?”

3.3.1 Constituents of a module

As has been stated in section 1.1, in situations where many sensors need to be connected to the data acquisition box, the sensors need to be sampled faster. This means that the processing requirements increase disproportionately as the number of sensors is increased. One of the problems with the current systems is that one CPU can't handle this load, and that the memory requirements for the system increase with the number of sensors. This is one reason not to allocate one CPU for all the ADCs. Also, if a system does not require multiple ADCs in one box, having more than one ADC per module implies having more than one type of ADC module - something of which the production department would not approve.

It follows fairly naturally from this way of thinking that provided the extra components are cheap enough, a single ADC should be a complete module. Once this decision has been made, it becomes natural to stipulate that any other components, such as a modem or line driver, should each be a single module.

It is gratifying to realise at this point that, at an abstract level, the IEEE-1394 bus and its associated software can be seen as a replacement for the Transputer serial links. Thus, at the very least, we have a design that is a more general incarnation of the legacy system. However, the IEEE-1394 chipset and the associated CPU provide far more processing power and bandwidth than the current system, at a lower parts cost.

It is interesting to note that this system would provide bandwidth in excess of custom-designed multiprocessors produced only a few years ago [Tan92, pg 371]. Also, the amount of processing power and memory scales with the number of sensors, as each data acquisition card comes with a built-in CPU.

3.4 My objectives revisited

Now that I had an idea of how I would solve the problem, I had to decide which implementation would be sufficient to show that my idea has merit. I would have to implement enough of the hardware that it would become obvious that any other required hardware would be easy to integrate, enough of the software to show that software development on the hardware would be convenient, and also show that the software I produced was appropriate and functional.

I decided on the following objectives:

- I would design and build a single printed circuit board that had enough IO connections for ISSI to interface IO modules of their choice, such as serial ports or ADCs, to it.
- I would write a kernel that would support a runtime environment and communications system that would have at least the same functionality as the old Transputer-based system.
- I would write software to demonstrate the utility of the kernel features such as multitasking or communication.
- I would write at least one device driver, to show how kernel drivers would support hardware.
- I would write an IEEE-1394 driver and enough of the protocol stack to show how it would be done and that it would work. This would provide further proof of the stability and functionality of the kernel as well.

Chapter 4

The hardware implementation

Once I was sure what a sample system would look like (see Figure 4.1), I could begin my part of its implementation - the part that would make all the modularity and scalability possible. I decided to build a board that would provide an IEEE-1394 chipset, memory, and a MCU capable of interfacing to whatever ISSI felt they would like to connect to a prototype, along with enough software to show that this approach was feasible.

I had already chosen the AT91M40400 MCU but I was stuck, as the chip was not available by the time I got to the stage where I needed it. I had to wait several months but eventually managed to get samples of a beta version of the chip from Atmel. Once I had made the major decision, namely the choice of the MCU, I could design the rest of the board.

4.1 The overall board design

The prototype board had to show that my concept for an architecture could work, and should also make it simple to interface other circuitry (such as an ADC) to it. It had to show that using the IEEE-1394 interface was feasible, and that the ARM core could be included in such a way as to be fast enough for our requirements.

I decided that the firmware for the board would be stored in Flash ROM, as it would be convenient to be able to change the firmware often while debugging. Seeing as the AT91M40400 came with a built in EmbeddedICE module, programming the Flash ROM should be easily done using the MCU, through its JTAG port. I decided to include RAM

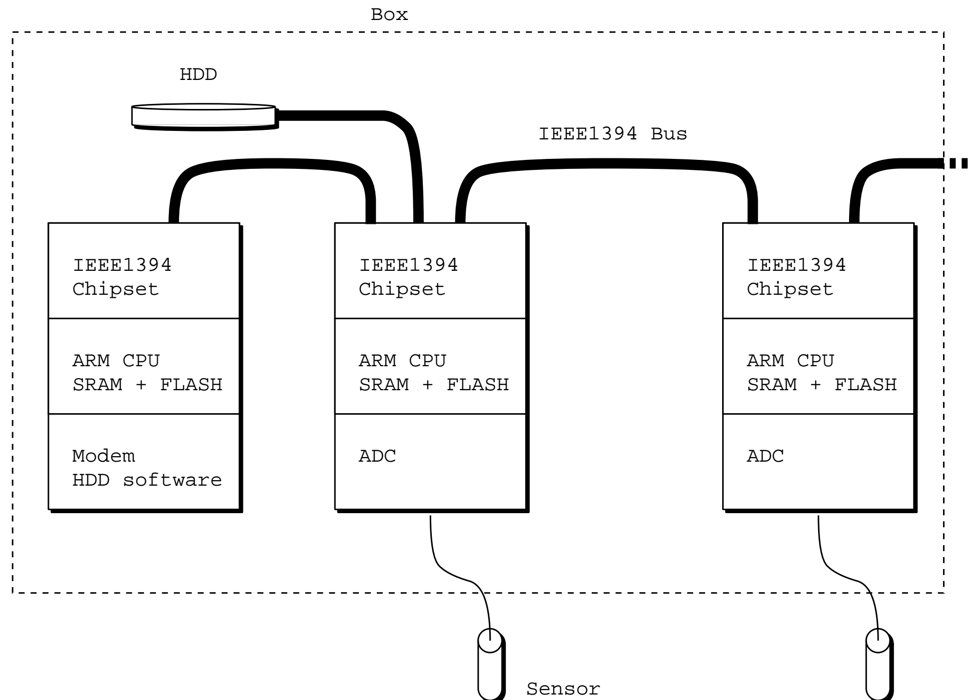


Figure 4.1: An example of the envisaged system.

that would allow the core to run at full speed, and also RAM that was slower (and used less power), but allowed more capacity for its price.

This left me the choice of a FireWire chipset. At the time of design, only one chipset was easily available - that made by Texas Instruments and comprising the TSB12LV31 LLC and the TSB21LV03 PHY. This made my choice very simple; in the future, I expect ISSI will probably select another chipset. The TSB12LV31's main problem is that it does not have large enough FIFOs for communications. The TSB12LV32 fixes this, but did not seem to be available at the time of writing. There are also other chipsets available now that fit both the LLC and PHY into one package, saving space and making the board layout simpler.

Motorola provided a fast (10, 12, or 15ns) 64K by 16-bit SRAM [Mot98] that matched my requirements exactly. Samples were readily available, and I decided to use this chip, the MCM6323A, as my fast SRAM.

The choice of what to use for less expensive, lower power RAM, was not so simple. DRAM is difficult to interface to: it requires multiplexing buses and meeting strict timing requirements. There were chips available that would do the interfacing for you - but at twice the

price of the MCU. Also, at the time, the only DRAMs available required 5V. Furthermore, DRAM uses too much power for some of the envision applications. I eventually found some Pseudo-Static RAM, the TC51V8512AF, made by Toshiba. This is DRAM modified to behave more like SRAM. However, I found that this had its own problems, and required me to use a timer on the MCU to generate refresh pulses. This would cause data loss if I breakpointed the CPU and stopped the timer during debugging. All in all, they were almost as bad a cure as the original disease.

It is at this point that I was made aware of some Toshiba SRAMs that had excellent power consumption (55mW peak, $4\mu A$ standby, at 25°C, [Tos97]), and were cheaper than Pseudo-Static RAMs. They also provided performance comparable to DRAMs (85ns quoted read cycle time at 5V, but they seem to behave similarly at 3.3V [Tos97]).

How to interface the IEEE-1394 chipset to the MCU was not immediately apparent to me. The TSB12LV31 provided two methods for interfacing to CPUs, and could also be used on either an 8- or 16-bit bus. It had an asynchronous handshake or a pulse mode handshake interface, selected by tying specified pins high or low [Tex98, pp 2-1 - 2-7]. I eventually decided to use the asynchronous method and hold the wait line asserted on the MCU until the TSB12LV31 signalled that the current transaction was complete.

The final design block diagram of the prototype module can be seen in Figure 4.2. A circuit diagram can be found in appendix A.

4.2 Problems encountered

The schematic capture software that I used (Seetrax Ranger 3) was adequate for the purposes of producing a schematic. However, the facilities it provided for producing the six-layer PCB artwork were terrible. The auto-router failed to connect more than about 15% of the connections detailed in the schematic, even failing at trivial tasks such as wiring a bank of memory. The manual routing facilities were obtuse, idiosyncratic, and buggy. Furthermore, there was no undo facility. It did provide the facility to plug in a third party router (which, I suspect, was the reason that the built-in router atrophied). Unfortunately, the dongle that came with the autorouter I had was faulty. The replacement came only after four months, and in the meantime, I had routed the board manually in Ranger, which took almost a month.

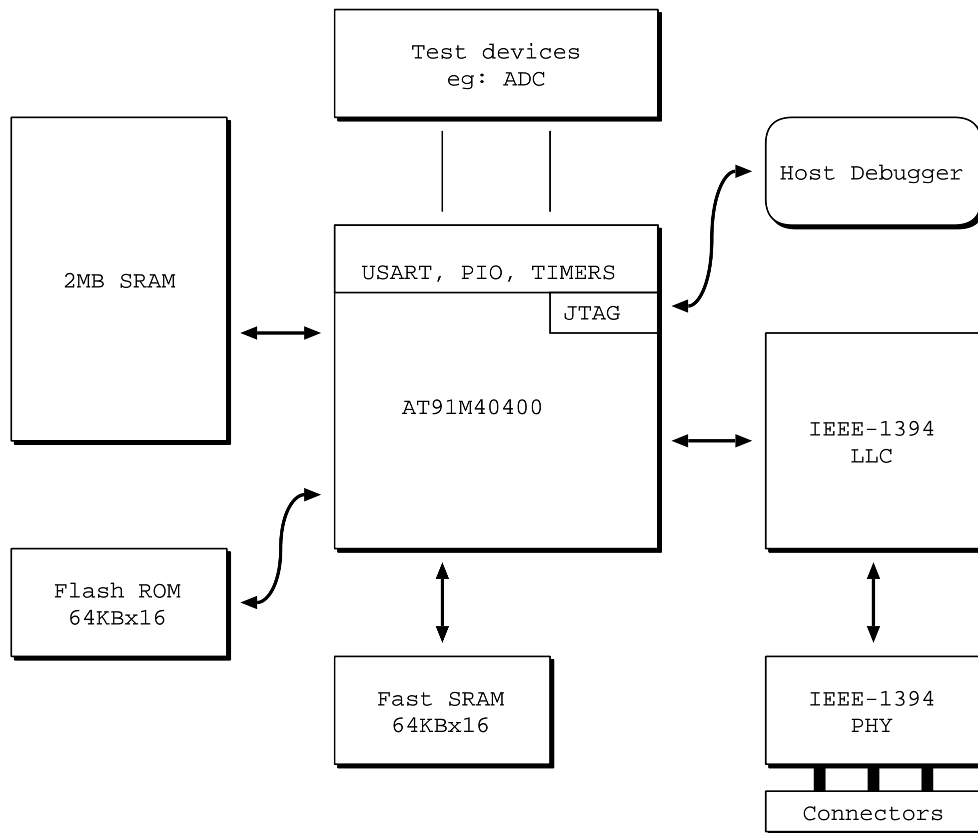


Figure 4.2: An overview of the prototype board.

I took care to keep the higher frequency lines as short as possible, and tried to isolate lines that I thought could cause interference. I split my power and ground planes into an analogue and digital section, with the IEEE-1394 section being mostly analogue.

The final layout was a six layer board, but two of the layers were mostly used to take debug signals to external connectors. I also took those signals which I thought could be used to test peripherals (such as ADCs) to external connectors. A screen capture can be seen in Figure 4.3.

There were additional delays as the MCU was not yet available from Atmel. After waiting several months, and signing a Non Disclosure Agreement, I received a beta version of the chip.

4.2.1 Building and testing the prototype

Once I had the PCB artwork, I sent it to a production facility to have the board produced. I received the blank board and immediately found that the analogue ground and power planes were shorted together. I connected a 20 amp power supply between them, sprayed the board with a coolant spray and turned on the power supply. The short circuit generated heat and thawed the iced condensation faster than elsewhere on the board. The fault turned out to be in an area that was not correctly drilled in the footprint of one of the IEEE-1394 connectors. This took a few minutes with a high speed drill to fix.

After that, I populated the board with the absolute minimum number of components required to get the MCU to boot. In this case, that meant the crystal oscillator, some glue logic and reset supervisor, as well as the MCU. I consciously designed the board so that it could be populated and tested in stages. I reasoned that the extra time spent testing and populating would probably save far more time trying to find errors in the schematic.

After soldering on the MCU and glue logic, and some LEDs used for debugging, I applied power to the board. In order to monitor the CPU, I had the JTAG cable to the CPU core debugger connected. The oscillator started up smoothly, and the power supervisor generated a valid reset pulse for the MCU, but the MCU was completely dead. After much fiddling, I noticed that the LED connected to the MCKO output (used to drive a crystal, but reconfigurable as an output), was not being pulsed at 25MHz, as it should. The MCKO output is usually the inverse of the MCKI, unless programmed otherwise.

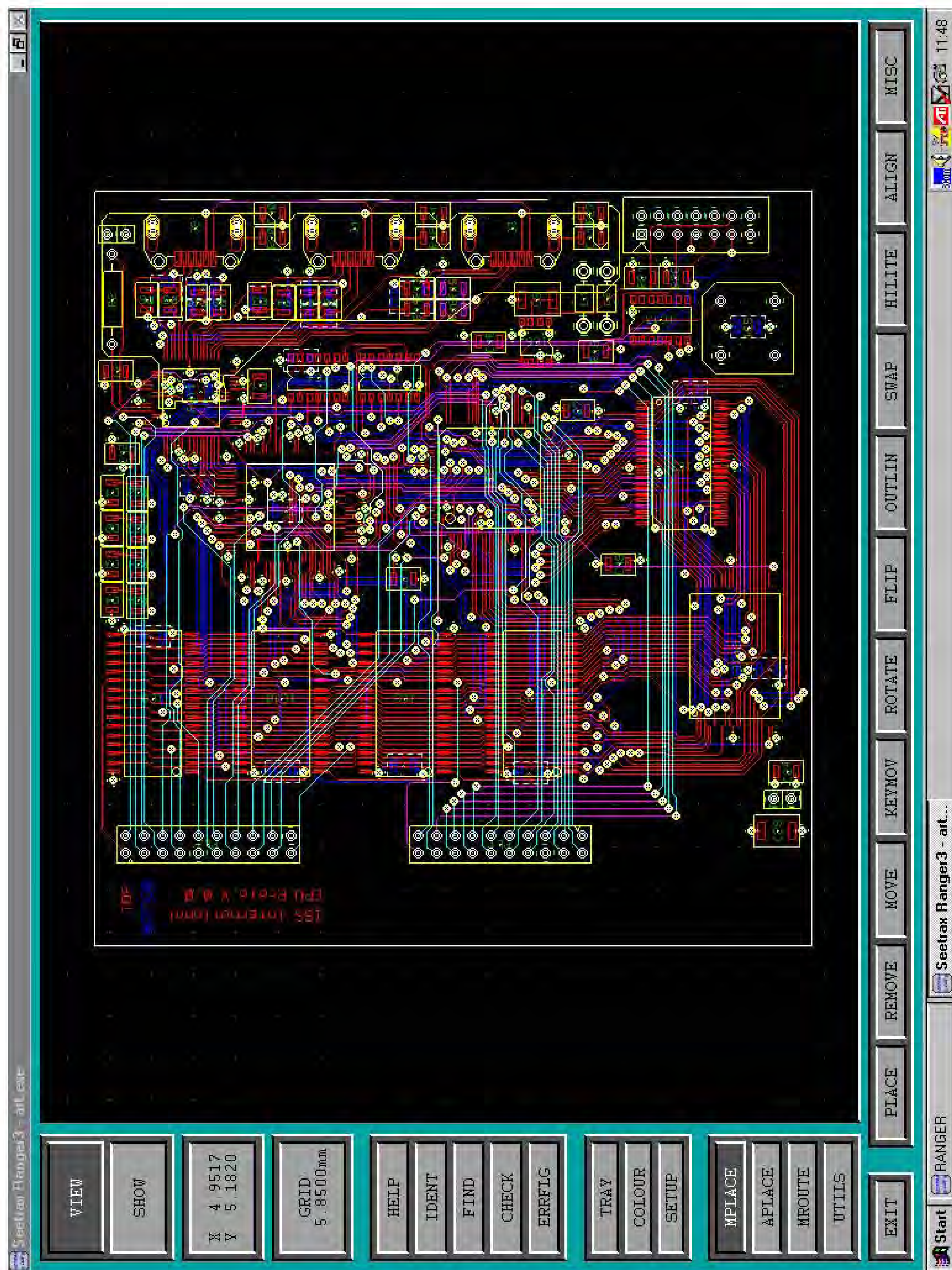


Figure 4.3: A screenshot of the program used to design the PCB artwork, and the completed 6-layer prototype board. The PCB artwork can be found in appendix B.

Logically, it should default to being the the inverse of MCKI, as otherwise it would be impossible to use an ordinary crystal (I used a crystal oscillator, which does not need any driving inputs). This meant that the MCU was either damaged or in some sort of tri-state mode. It turned out that the MCU shared the JTAG lines with other inputs, one of which is checked at bootup to see whether the MCU should be switched into tri-state mode. The ARM Embedded ICE debugger module held this line low by default, forcing the MCU into tri-state mode. Unplugging the JTAG connector before switching on the board solved this problem.

I attached the FlashROM next and tested reads and writes to it; they seemed satisfactory.

I then attached the Motorola x16 SRAM, and found that reads and writes were mangled. After some testing, I realised I had made a trivial mistake in the schematic and had to reroute one line. Things were better at this stage, but looking at the output of the MCU on a data analyser showed that the A0 line was behaving in an almost random manner when accessing 16-bit wide RAM. I reported this fault to Atmel and rewired the interface so it would not use that line.

The final step was to solder on the slow SRAM banks, which I did, and on testing them, they worked well.

4.2.2 Other problems with the prototypes

As time went on, hardware that initially seemed well-behaved turned out to have problems. While the Flash ROM on the prototype seemed to program well, and retained its contents when power was removed, it did not occur to me to check to see that the built-in write protection on the chip worked. When trying to debug some code I had written to the ROM, I found that entire sectors (256 bytes) of the ROM were being scrambled. It became clear that the debugger was trying to write to the ROM to create a software breakpoint. The ROM accepted any write as being valid (although there is a specific sequence you are supposed to go through, according to the data sheet), and when a single address in a sector was written, all other addresses in that sector were also programmed. However, if an address had not had data written to it before the program cycle was entered, the address was written with a sequence of all ones. This meant that I would lose large sections of ROM every time something went wrong and a process wrote to random memory locations. Atmel did not have errata available on their web site; an email query to them returned a reply

that “there are no known errata on that chip”. I fixed this by adding some extra logic to implement a write protect/enable signal. This I connected to a general purpose pin on the MCU, which disallowed writes until the pin had been software-configured to a particular state. The general purpose pin is tri-stated by default, and held in the “protected” state by a resistor.

For a while, I had a great deal of trouble trying to make the MCU communicate with the IEEE-1394 chipset. Writes to the chipset were simply unreliable. The Link Layer Controller (LLC) has a built-in byte stacker [Tex98, pg 2-8], to collect multiple 8 or 16-bit reads and writes and execute an internal 32-bit read or write when the transaction is complete. My problem was that the stacker would accept individual 16-bit reads or writes, but would not stack them in any useful manner. Initially, I thought this might be because I was using the LLC incorrectly. Eventually, I decided it had to be a hardware problem.

I used a data analyser to watch transactions between the MCU and the LLC. At first, it was quite confusing, as it seemed that the LLC could guess which transaction the CPU would request, and then changed its behaviour accordingly. I never did find out what caused this apparently non-deterministic behaviour. After watching several transactions, and verifying that the correct signals changed in the the right order, I decided that I may have been violating the setup time of signals into the LLC. The observed signals were within the published constraints [Tex98, pg 6-3], but to within less than a nanosecond, and the data sheet states that the published values are not production tested. The LLC times everything relative to the rising edge of the microprocessor clock. The MCU, however, references transactions to the falling edge of the clock[Atm98, pg 7]. With a 24.576MHz clock, this gave me approximately 20ns to setup all the required signals. On a loaded bus, with the extra logic required to interface to the LLC semi-synchronous interface, meeting the required 9.6ns setup time was difficult. I decided to invert the clock signal being provided to the LLC, meaning that it now referenced its signals to the same edge that the MCU used. This effectively gave me another 20ns to setup my signals. Once I made this change, the MCU-LLC interface became reliable.

This sort of problem would have been easy to fix, if I could have built more than one prototype and compared them. Unfortunately, of the three PCBs I had made, two showed faults before I populated them and the third randomly connected and disconnected various signals as I gently twisted it. The first board had a short circuit between power and ground that was easy to fix. The second had the same fault but it was not clear where the short circuit was. The third seemed alright, but was unusable due to intermittent connections

on signal lines. From this, I think it was reasonable to distrust my one (partially) working prototype made from that run of boards. Any problem that I turned up could be as a result of the debugger (which is not perfect, and has its foibles), my design, my software, or a faulty connection due to a faulty PCB. In the earlier stages, the MCU was also under suspicion, as it was a beta version. This has made debugging especially challenging.

Chapter 5

The software implementation

This chapter describes my efforts to write the software components of my project. I cover the structure of my kernel, the communications subsystem, and the IEEE-1394 stack that I implemented. I try to explain why I implemented various functions, and why I chose the solutions I did.

The concept of a "real-time system" needs to be defined. Many authors provide their own definition. However, as an elegant summary, the authors of the IEEE-1394 standard [IEEE1394, pg 2] make the point that the correctness of a real-time system is dependent not only on the logical result of a computation, but also on the time at which the results are produced. They go on to define the difference between a "hard" and "soft" realtime system, stating that a "hard" real-time system is one in which it is imperative that responses occur within a specified deadline. Their definition of a "soft" real-time system is one in which the deadlines are important, but not absolute: they may be missed from time to time. Other authors, such as Magalhaes [Mag96], treat deadlines in a far more complex and general manner which I think is unnecessary for the purposes of this thesis.

The software response-time requirements for this architecture are awkward. I could not find another system like it in the literature. In short, the software in this system is expected to be overloaded. This makes it difficult to characterise the new software as a pure real-time system, as the point of real-time system design is to guarantee the meeting of deadlines.

The software will be expected to work well most of the time, but it is known that situations will occur where it will not be able to handle the sheer volume of data presented to it. One example is the surge of seismic activity just after blasting in a gold mine. With the dramatic increase in the number of seismic events, more data is captured than could ever

be relayed back to the base station. The CPU is unable to process the data before new data arrives, nor does it have sufficient storage space to buffer either the processed or the raw data.

The software could be considered to be a soft realtime system that is expected to meet its processing requirements most of the time. Due to the wide range of possible loads, designing the system for any eventuality would result in expensive hardware that spends most of its time completely underutilised. A more realistic approach would be to aim for a system that degrades gracefully under excessive load.

I first attempted to find a commercial realtime OS (RTOS) for this project. However, ISSI were not willing to pay license fees to another company for part of their system. Most RTOS implementations[1] either cost an exorbitant amount in a once-off fee, or charged "per-CPU" fees that effectively doubled the cost of the hardware. Also, the development environments for these systems usually cost more than the entire cost of my MSc. ISSI was simply not interested in such arrangements. This meant that I had to implement my own operating system, or at least a subset sufficient for their needs.

As the author of the kernel under which any processing will run, I have to ensure that I provide the primitives to allow a quick response when possible and a graceful degradation when necessary.

5.1 The language and development environment

The choice of CPU greatly affects the software and languages available for development. The ARM Software Development Toolkit, produced by ARM Ltd, provides support for assembler and C. It was also the only development environment available for the target processor at the time. It provides an Integrated Development Environment, a mature C compiler and assembler, ARM CPU core simulator, debugger and profiler. This meant that C and assembler were already looking attractive as the languages that I would use for development. Another factor that I had to consider is that the legacy software that

[1] I am aware of eCos, a free open source RTOS produced by Cygnus Solutions (See http://www.cygnus.com), which was not available while I was implementing my kernel. At the time of writing, eCos has only just started supporting the ARM7 core. I have advised ISSI of its availability, and I think that as it becomes more mature, it is likely to become a better solution for ISSI than my implementation.

would have to be ported to this system was written in C. Although other languages have advantages compared to C [Coo96], simple practical considerations determined my choice.

5.2 The implementation of a kernel

In some ways the new design is for a realtime system and has various requirements:

- It has to be able to support multiple tasks running concurrently. The current ISSI software is implemented as several processes that communicate with each other.
- It has to have fast interrupt response. The interface to the ADCs will provide new data every few microseconds. It would be preferable to not have FIFOs built into the hardware interface. Similarly, communication with a central site requires that no bytes are lost on the serial port.
- It has to support the distribution of data across an IEEE-1394 network. This would allow great flexibility in the final configuration of a running system. For example, a particular type of data could be routed to a specialised processing board, without requiring major changes to an existing system.
- It has to do things in realtime. Particularly large seismic events would trigger an emergency response that has to be timeous to be useful.
- It has to be simple to expand and configure. Every customer has specific requirements, and customisation should be easy.

Because of the CPU used, and for cost reasons, the CPU does not provide an MMU. Also, this kernel is for a proprietary system. These two facts make protecting tasks from each other, and protecting the kernel from tasks, less important than on, for example, a desktop system. Obviously such protection would still help for testing and debugging. Also, I use the word 'realtime' to mean that the system will have to respond fast enough to allow data to be collected and processed. Hard realtime response is not required.

This sets various priorities in the design of the kernel:

- It must support synchronisation of multiple tasks. This is a basic requirement for a concurrent system.

- It must try to provide each task with as much CPU time as it requires.
- It does not have to protect tasks from each other, nor protect the kernel from them.
- It must support inter-task communication.
- It must not use much CPU time or memory. In other words, it must be efficient.
- It must support the use of many CPUs communicating via IEEE-1394.
- It must attempt to provide a structure that promotes the re-use of code.

A realtime concurrent system with multiple tasks across multiple CPUs, where the position and number of CPUs can change in the field, has its own special problems (Many of which are beyond the scope of a MSc thesis). A message naming convention has to be agreed upon to allow different tasks to communicate with each other. Each task runs within certain realtime constraints and has to be guaranteed CPU time. This could be done by the user programs (a co-operative system) or by the kernel (a pre-emptive system). A pre-emptive system is more transparent to the programmer.

The kernel has to respond promptly to interrupts, meaning that disabling interrupts within the kernel, for any length of time, should be avoided. A network protocol, and naming system, has to be decided upon, to allow tasks on different CPUs to communicate. This must ensure that the required functions will be provided, even though the configuration or number of CPUs may be changed in the field.

5.2.1 The communication subsystem

I also had to decide how tasks would communicate with each other. The transputer uses a synchronous communication system (as defined by Tannenbaum [Tan92, pg 411]). The current system produced by ISS uses this communication system extensively. However, it is not the only means of communication possible. There are methods based on shared memory [Bur90, pp 185-223] as well as various forms of message passing [Bur90, pp 227-263]. Due to my choice of using a serial bus for communication, passing messages would be more transparent to a network, and I decided to concentrate on these.

Message passing systems can broadly be classified as synchronous or asynchronous [Tan92], [Bur90, pg 228]. Asynchronous systems are ones in which the action of sending a message

does not halt the sending process. This can be more efficient but has other problems - it necessitates having a message queue for messages. Also, if references to buffers or to memory are passed around, the sending process has no idea when it is allowed to interfere with the memory area passed by reference. Because only the action of reception causes a partial synchronisation with another process, more communications are sometimes required to solve a problem. This can make a given problem more complex to solve. Burns and Wellings also state that using asynchronous primitives can make it more difficult to prove the correctness of a complete system, but I am unsure whether this is an indictment of asynchronous message passing or the formal methods available to prove the correctness of concurrent systems [Bur90, pg 229].

Synchronous message passing primitives force a sending process to halt until the message is received by the receiving task. This is a more positive form of synchronisation, forcing both participating processes to synchronise with each other. However, it can force the sending process to halt unnecessarily when sending a message, fragmenting the execution of individual processes, from the point of view of a task scheduler, and possibly incurring a performance penalty. In any situation where asynchronous message passing is required, a special buffer process has to be created to provide this function.

Each approach has its own set of problems for which it is well-matched. However, it can be argued that synchronous message passing primitives can be built up from asynchronous primitives. Thus, an application programmer could build a library of synchronous primitives and use those for the relevant problems. Also, it is my feeling that the number of pending messages in a process' input message queue can provide a good indication to a scheduler of where to allocate CPU time. Preliminary simulations have backed this feeling for simple systems but this is an area for further study.

Another consideration in message passing systems is how processes identify the destination for their messages. In direct systems, processes name each other. In indirect systems, an intermediate name is used, much like a post office box (which explains why such a system is sometimes referred to as a "mailbox" system). While direct naming is simpler to implement, indirect naming has no limitations and provides a useful tool in modularising software [Bur90, pg 231]. Mailboxes can be used as the interface between distinct parts of the system. This advantage is one that I consider important. An example is given in Figure 5.1.

I decided to name my mailboxes by the type of data they would pass. This means that

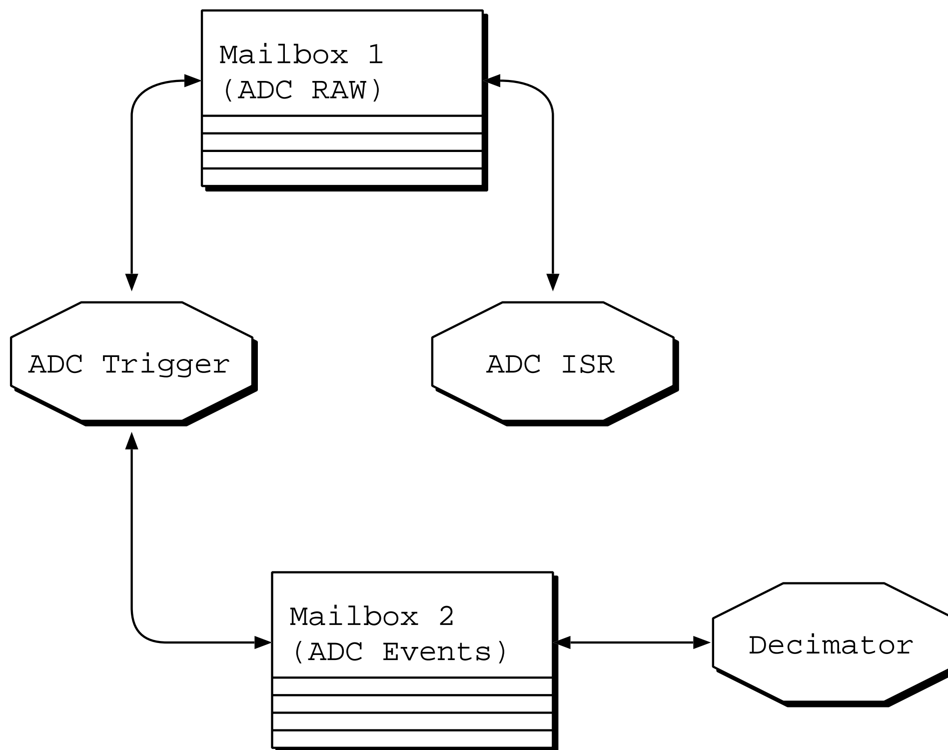


Figure 5.1: A simple example of how processes would communicate via mailboxes. Each process only has to know the name of a mailbox to communicate effectively.

data could, for example, be named as raw ADC input, decimated seismic data, or error log information. The kernel simply has to route the data to a process that indicates that it handles that type of data. This ensures that any process need not know which other process will accept its output or produce its input. Hopefully, this will allow easier integration of processes across multiple CPUs. The same mechanism can be used for synchronising processes and other such applications.

In order to keep the kernel simple, and to avoid problems with priority inversions and the like, I initially decided to implement the kernel task switching using a round-robin, variable time slice, pre-emptive switcher. Each process would be guaranteed some time quantum per scheduler cycle, but the proportion of time given to a process would be adjusted by the kernel. The maximum time used to run all tasks once would have a hard upper limit. Data would pass from process to process via the kernel, which would keep track of all buffers used, and give larger time slices to processes with longer input queues. Eventually, due to time constraints, I decided to implement a simple round-robin, pre-emptive context switcher, in an easily upgradeable manner.

Interrupts are handled by the CPU interrupt hardware and interrupt service routines, which pass data to the kernel, too. Obviously, this was a potential source of problems, as an interrupt might be interrupting a task that is also passing data to the kernel. The challenge was to provide a safe implementation, while using a minimum of interrupt disables.

5.2.2 A suggested programming model for this kernel

I envisage the code run on this system to be split into three types. At the most abstract level are the the applications that process data. They should be fairly immune to changes in the underlying hardware. At an intermediate level are the device handlers. These processes collect data from interrupt routines and pass them to applications, and vice versa. Also, they collect data from these routines, to pass on to applications. These processes should have a limited dependency on the underlying hardware, and should carry most of the burden in providing a non-specific interface to the hardware. The lowest, and final, level consists of the interrupt routines themselves. They are not scheduled in any way, and are executed asynchronously when activated by the CPU's interrupt vector circuitry. However, they are still capable of sending and receiving messages.

5.2.3 The primitives provided

Choosing a minimum set of useful primitives that a kernel should provide is a difficult task. I have tried to implement the bare minimum, while keeping it simple to add extra functionality in the future. How these functions relate to each other is shown in Figure 5.2. So far, I have decided upon, and implemented, the following primitives:

AddProcess(uint Vector, uint Dat, uint TaskSP):

Adds a process to the process list. Returns its PID after having added it to the list. Gives the process a stack, the size of which is a supplied value, and copies one 32-bit parameter to the r0 register. The process code should be re-entrant.

RegisterType(uint MsgID, uint PID):

Marks the given process as being eligible for reception of a particular datatype.

WaitMsg(TMsgHandle **Msg, uint MsgID):

Waits for a message to arrive. If nothing is in the input queue, blocks until something arrives. Can be used by an interrupt. An interrupt executing a blocking wait would be a serious error condition.

CheckMsg(uint MsgID):

Used to determine if there is a message available. Should be used by interrupt routines.

SendMsg(TSMsgHandle **Msg, uint MsgID):

Sends a block of data of a specific type. The block sent is now no longer yours. Can be used by an interrupt.

WaitOnMsgList(uint *MList):

Does a blocking wait on a list of Msg IDs. Returns the offset of an ID with a non-empty message queue.

Yield(void):

Gives up the remainder of your timeslice. This is useful for processes that sit in infinite loops servicing requests, for limiting their CPU usage.

Block(void):

Marks the current task as blocked and then does a Yield().

UnBlock(uint PID):

Marks the indicated task as ready for execution.

EInt(void):

Enables interrupts and returns the previous interrupt state.

DInt(void):

Disables interrupts and returns the previous interrupt state.

SetInt(int ReqIntState):

Sets interrupt status to the desired state and returns previous state. This replaces DInt and EInt but they have not yet been completely removed.

GetPID(void):

Returns the current process ID. Can be used by interrupts and processes, as interrupts also have a process ID.

5.2.4 The kernel structure

The main loop in the kernel is the task switcher. It switches between available tasks and is called by a timer interrupt. High speed IO events are handled by interrupts. This means that the servicing of an IO event is delayed by the handling of an event with a higher interrupt priority, or by the kernel itself disabling interrupts. There is not much that can be done in the first situation, except to minimise the amount of processing done in an interrupt. The second case provides much encouragement for designing the kernel in such a way as to minimise delays due to disabled interrupts. The simple solution for ensuring reliable execution during these cases is to disable all interrupts when servicing an interrupt and during the execution of the critical parts of the kernel. Unfortunately, this can cause unnecessary delays in servicing an interrupt.

I eventually decided on a three-tier approach, as is shown in Figure 5.3. The ARM CPU core provides two layers of interrupts: the FIQ (or Fast Interrupt) and the IRQ (or Interrupt). The FIQ has a higher priority than the IRQ, which is used to trigger a context

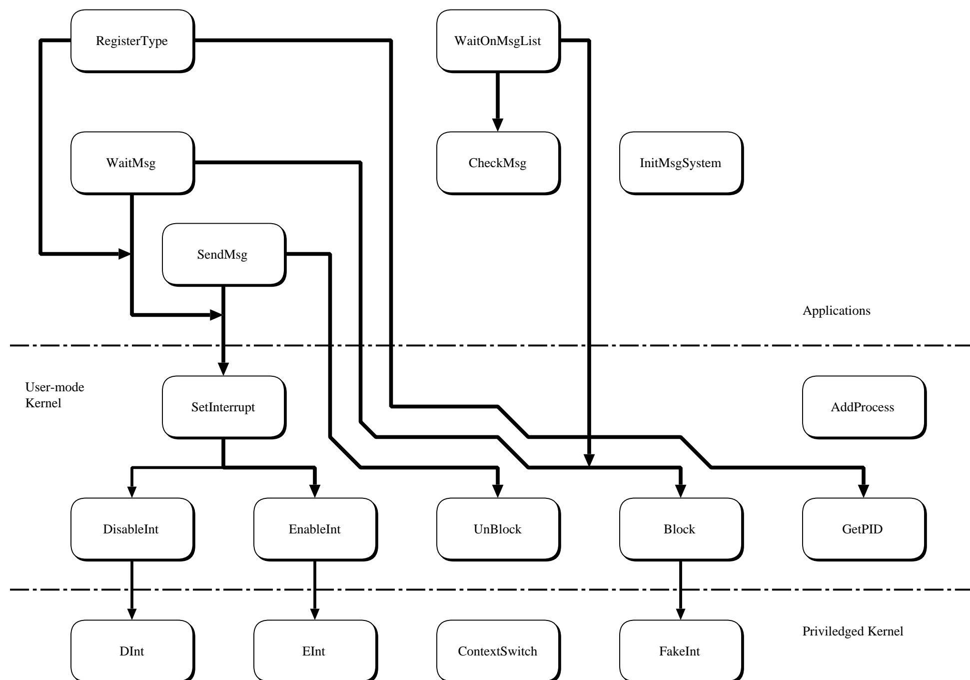


Figure 5.2: The functional dependencies of the functions provided by the kernel and the message passing system.

switch. Thus, temporarily disabling the IRQ ensures that an operation will be atomic at the task level but FIQs will still interrupt the execution of tasks. The point of the FIQ is that it allows fast interrupt response. The ARM core even banks several of its registers when switching into FIQ mode, to save on latency and overhead. Therefore, hardware that requires a fast response should be tied to the FIQ interrupt. Hardware that can tolerate a few microseconds of latency can be connected to the IRQ interrupt. FIQ device handlers cannot use the kernel message passing functions because they could interrupt the kernel in the middle of another message passing operation. However, IRQ device handlers can use the message passing subsystem, as they only ever interrupt the subsystem at convenient moments. This should make it simple to write drivers that abstract the hardware in an elegant manner.

FIQ device handlers can generate IRQ events, meaning that the FIQ handler's sole purpose can be to hide latency in the IRQ handling of events. An external device can generate a FIQ and the FIQ can service the device and update internal buffers. Periodically, the FIQ can generate an IRQ to force a slower handler to fetch a buffer and pass it onto the rest of the system using the message passing subsystem. This guarantees a fast response to peripherals requiring it, and lessens the constraints on the remainder of the system. It is

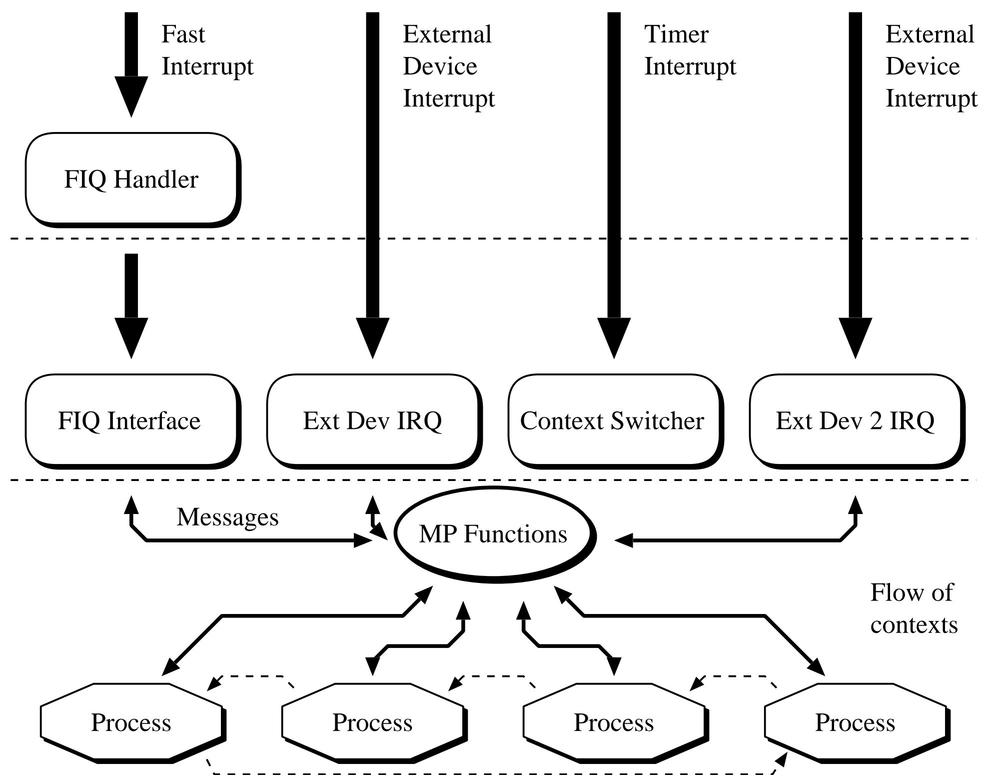


Figure 5.3: The interrupt handling in the kernel and how it fits together with the context switcher and message passing (MP).

worth noting that this system does not often have to process data and provide a fast reply, so for most applications it is acceptable to hide latency using multiple buffers.

5.3 Problems encountered

Once I had built my first prototype, I started the development of the kernel software for the ARM core. Progress was swift while I was still writing routines that would be required later, such as those to add tasks and do context switches. Then came a time when I had to get task switching working. Almost immediately, I was stopped by a very hard problem: I could not determine how to use the interrupt controller. The documentation I had was still preliminary and I had to guess at a few steps in the process of programming the various peripherals and the interrupt controller. This delayed me excessively, as it took me a long time to realise that the problem was not my fault, and I spent much time trying different code combinations to solve the problem.

Interrupt controller

I was able to get the on-board timer to generate an interrupt request. By using the debugger, I could see that the interrupt controller on the MCU (note that this is external to the CPU core) was receiving the interrupt request but was not passing it on to the ARM core. I tried to enable the interrupts but could not. Writes to the interrupt enable mask register were simply ignored. Nowhere in memory could I find an address that would change the value of the mask register. I was completely stalled and contacted Atmel. They were horrified to discover that the mask revision of the beta chips I had been sent was one that they thought had never even left the factory. They rushed me a newer version and it became possible to enable interrupts. They had also fixed the erratic behaviour of the A0 line in the memory interface, mentioned earlier in section 4.2.1.

However, I still had a problem - the interrupt controller would behave as I expected after a power up, but it has an internal stack of pending interrupts, and if certain memory address were strobed in the wrong order, or too late, it could sit in a state where it would never assert the core's IRQ line. I fixed this by disabling all interrupts and then strobing all the correct addresses in a loop until I was sure that the “pending” stack was empty. At this point I had a reliable timer interrupt and could start testing my task switching routines.

Atmel later changed the way in which their interrupt controller worked, to eradicate this and other effects caused by having registers that changed the state of the hardware when read.

Banked registers

At some stage I inadvertently introduced a bug that was intermittent yet fatal. The MCU would run for a few seconds and then would end up executing in some random section of memory. The internal state of the CPU was such that I couldn't see where it had come from. This bug took a long time to find; I eventually traced it down to the last few assembler instructions at the end of my task switch routine.

The ARM core has banked registers to save on register backup operations during interrupts. Inside an interrupt, and especially when switching contexts, it is sometimes necessary to access the banked registers. However, it takes time for a bank switch to occur. In the ARM7TDMI, two instructions should not try to access banked and unbanked registers consecutively, without some sort of intervening instruction. If you do attempt to do so, the bank switch usually works, but is not guaranteed to do so.

My task switch routine was being called at 100Hz and the bank switch was succeeding most of the time. Occasionally, the switch did not complete in time, and an invalid value would be moved into the program counter. This caused the erratic behaviour and, eventually, a data abort exception.

Message passing

My next problem was to start implementing the services that a programmer would expect from the kernel. My primary concern was the message passing subsystem, as it would influence the rest of the system dramatically. I spent by far the greatest amount of time trying to decide how to implement it. I wanted to avoid typecasting pointers whenever I had to deal with a message. A pointer to a void didn't solve my problem - it just forced me to typecast all my pointers, rather than some of them (the ARM compiler is fairly fussy in this regard in its default state). I tried a few different approaches, and after a few false starts, settled on the current implementation. It depends heavily on linked lists, which are conceptually simple, but can be tricky to get right. Most of the development problems

associated with the message passing system were related to edge conditions on the linked lists that I hadn't resolved correctly.

The debug channel

It became clear after a while that it would be useful to have some sort of debug output. One reasons for choosing the Atmel MCU was that it had a full debugging facility, including debug output, built into the core. However, the ARM Software Development Toolkit (SDT) had a particularly bad bug in the software for viewing the debug channel output. The viewer insisted on floating above all other windows, so I eventually closed it in frustration. However, there was no way to get that window open again! The window also disappears when you upgrade from a particular version of the debugger to a slightly later version - something I discovered later. Reinstalling the software did not fix the problem.

Several months later, in desperation, I installed an older copy of the SDT on another machine, and compared Windows registry entries for the two versions of the program against each other. Unfortunately, they were quite different in the areas referring to the debugger and the debug channel viewer. I thought I could see a pattern, though, and after a morning of trying different registry entries and structures, I managed to convince the version I had to open the required window. Soon thereafter, I wrote a small debug message server for my kernel which would check a circular FIFO and would copy any new data found to the ARM debug channel. Finally, I had text output to my screen on the host computer! The debug channel viewer is still somewhat rudimentary, but it is infinitely more useful than no viewer at all.

With the addition of debug output, software development became much easier. I wrote a timer server that accepted messages from other processes and interpreted them as requests for a reply after a delay specified in the message. Thus, another process could send a message requesting a reply in five seconds and then do a wait for that message; the process would then wake up after five seconds to continue processing. This mechanism was also useful for implementing timeouts on a message wait. A process sent a message to the server and then did a WaitOnMsgList (see section 5.2.3) for both the reply and the other message type. The timer does not yet provide an easy way to associate requests and replies.

5.4 A simple test

As a simple test application I implemented a solution to the Dining Philosophers problem [Bur93, pp 93-97]. Five philosophers are given five seats at a round table, but only five chopsticks with which to eat, with each chopstick being placed in the space between adjacent place mats. If each philosopher spends a random amount of time eating and thinking, how should chopsticks be allocated so that no philosopher starves to death? This is a well-known computer science problem, intended to be solved in a concurrent environment, and has a well-known set of solutions. All of these solutions require one or another synchronisation primitive, and some form of concurrency that works. It is a simple, well understood example, with a simple solution, and is very intolerant of mistakes in either its implementation, or the implementation of the underlying primitives it uses. If a chopstick gets lost, our very absentminded philosophers soon starve to death.

I made each philosopher a separate process that registered to receive two types of message. One message type corresponds to the left side of the place mat, the other to the right. Note that no two processes can register to receive the same type of message, so two message types refer to the same chopstick. This applies for every chopstick, and each message type should be associated with the edge of a place mat, rather than a chopstick. A message will be sent to that type if a chopstick crosses the corresponding boundary of that mat.

Here is the C code I used to implement this test:

```
void Ph(int PhNum)
{
  TSMsgHandle *MyLeftChopstick;
  TSMsgHandle *MyRightChopstick;
  TSMsgHandle *MyTemp;

  int LC, RC;   /* Addresses to give chopsticks to */

  char *foo = "Philosopher x is eating.\n";

  /* Figure out where our messages are going */
  if (PhNum > 1) {
    LC = (PhNum*8)-7;
  } else {
    LC = (6*8)-7;
```

```
};

if (PhNum < 5) {
  RC = (PhNum*8)+7;
} else {
  RC = (0*8)+7;
};

/* We have to register for a chopstick from the left and right */
RegisterType((PhNum*8)-1, GetPID());  /* From left */
RegisterType((PhNum*8)+1, GetPID());  /* From right */
RegisterType((PhNum*8)+3, GetPID());  /* Timer reply port */

/* Tell the waiter I'm ready */

WaitSMsg(&MyTemp, ALLOCSMSG);
SendSMsg(&MyTemp, 1);

/* Now we begin */

while (1) {
  /* Think */

  WaitRand((PhNum*8)+3);

  /* Scrabble for chopsticks */

  if (PhNum == 1) {
    WaitSMsg(&MyRightChopstick, (PhNum*8)+1);
    WaitSMsg(&MyLeftChopstick , (PhNum*8)-1);
  } else {
    WaitSMsg(&MyLeftChopstick , (PhNum*8)-1);
    WaitSMsg(&MyRightChopstick, (PhNum*8)+1);
  };


  /* Eat */

  foo[12] = PhNum+48;      /* Quick output hack until I get */
  DebugMsg(foo);           /* sprintf() working             */
  WaitRand((PhNum*8)+3);

  /* give back chopsticks */
```

```
    MyLeftChopstick->MsgPtr->SMsg[0]=PhNum;
    SendSMsg(&MyLeftChopstick , LC);
    MyRightChopstick->MsgPtr->SMsg[0]=PhNum;
    SendSMsg(&MyRightChopstick, RC);
  };
}


void DoPhilosophers(void)
{
  int i;

  TSMsgHandle *MyTemp;

  /* Open a comms channel */
  RegisterType(1, GetPID());

  /* Create 5 philosophers with 1K stack space */
  for (i=1; i<=5; i++) { AddProcess((unsigned int) &Ph, i, 1024); };

  /* Wait for them to say they're ready */
  for (i=1; i<=5; i++) {
    WaitSMsg(&MyTemp,1);
    SendSMsg(&MyTemp,DEALLOCSMSG);  /* Discard the messages */
  };

  /* Set the table */

  WaitSMsg(&MyTemp, ALLOCSMSG);
  SendSMsg(&MyTemp, (1*8)-1);

  WaitSMsg(&MyTemp, ALLOCSMSG);
  SendSMsg(&MyTemp, (1*8)+1);       /* 1st philo gets 2 chopsticks */

  WaitSMsg(&MyTemp, ALLOCSMSG);
  SendSMsg(&MyTemp, (3*8)-1);

  WaitSMsg(&MyTemp, ALLOCSMSG);
  SendSMsg(&MyTemp, (3*8)+1);       /* 3rd philo gets 2 chopsticks */

  WaitSMsg(&MyTemp, ALLOCSMSG);
  SendSMsg(&MyTemp, (4*8)+1);       /* 1 chopstick between 4 and 5 */
```

```
}
```

First, the test initialisation creates the five philosophers. Next, it registers to receive a notification message. The philosopher tasks initialise themselves and send a notification that they are ready. The initialiser then places chopsticks in the correct positions by sending messages of the correct types. Each philosopher thinks for a while and starts waiting on the arrival of a message indicating the availability of a message on a particular side. All philosophers wait for a chopstick, first on one side, then on the other, except for one philosopher who waits in the opposite order. This is to avoid any possible deadlock caused by circular requests for resources [Bur93, pp 84]. Each philosopher waits for one chopstick, then the other, and eats. On completion of eating, the philosopher replaces the chopsticks by sending messages of the correct type. Each philosopher process reports its status using the debug facility mentioned earlier.

The test output from the program was as expected, and the internal process and kernel data structures were consistent with what I required.

It was at this point that I started writing my IEEE-1394 software drivers and protocol stack.

5.5 The IEEE-1394 protocol stack

It was decided early that it would be sufficient for me to demonstrate that my hardware was capable of communicating with other hardware via the IEEE-1394 bus. However, I felt that I could do better than this. Demonstrating that something can operate at the physical layer level might be meaningful but would not be particularly useful. Also, other problems that could occur as a result of a design flaw or some other problem in my kernel would not be revealed. An implementation of a IEEE-1394 stack would be a good test of the other code already written for the hardware.

The stack, at the time of writing, is not fully IEEE-1394 compliant. Certain features have been omitted, simply because they have not been required yet. However, I have attempted to ensure that all the necessary infrastructure is in place. Adding the extra features should be a matter of filling in a few stub routines and perhaps adding a new routine here or there. Further details will be presented in subsequent sections.

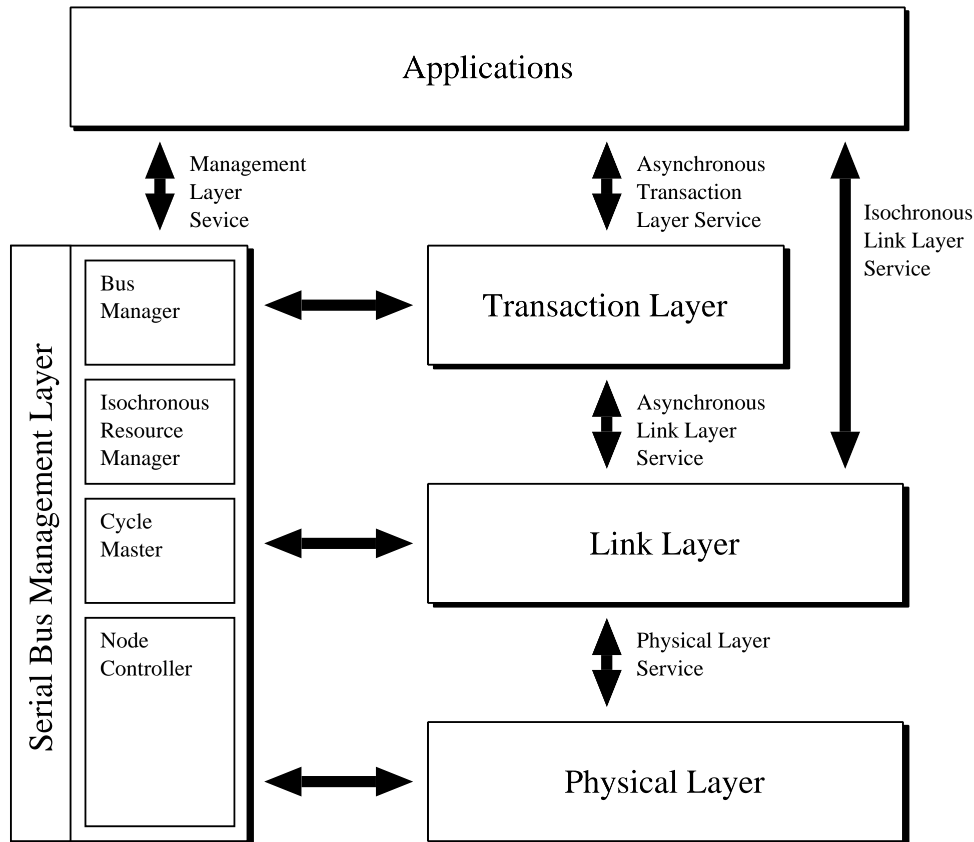


Figure 5.4: The structure of an IEEE-1394 stack. More detailed versions of this diagram can be found in [IEEE1394, pg 22] and [And98, pg 43].

5.5.1 Overview of a generic IEEE-1394 stack

The IEEE-1394 stack does not use the seven layer OSI model. As is shown in Figure 5.4, it uses some parts of the OSI model, but has other additions and modifications to support isochronous data transfers (which are presented at the link layer) and CSR architecture support (which sits "next to" the link and transaction layers, rather than above them) [And98, pg 42].

The protocol layers are:

- Bus Management Layer: provides services to affect all node configurations on the bus, should the current node be chosen bus manager [IEEE1394, 3.8].
- Transaction layer: provides asynchronous services to support CSR architecture operations, such as read, write, and lock. These operations are further explained in

section 5.5.2.

- Link Layer: provides a half-duplex data packet delivery service. Acknowledgements are returned for asynchronous packets, but not for isochronous packets [IEEE1394, 3.6].

- Physical layer: the electrical and mechanical interface required for the transmission to, reception from, and arbitration of, the serial bus [And98, pg 42].

The Bus Management Layer provides various services. Some of them are provided to all the nodes on a bus by a single node. Others are configuration services for the local node that can be accessed remotely or by local applications. Some of these services are an optional part of the IEEE-1394 standard, while others are required for a IEEE-1394 node. In order to be compliant, a minimalist configuration ROM and CSR register implementation is required. The ROM is a section of memory in a particular format, that should be available for reads by other nodes, and supplies information about the node [IEEE1394, 8.3.2.5.2]. The CSR registers are a mixed group of read only, write only, and read/write memory addresses used to control the node, or to read information about the dynamic state of the node or bus [IEEE1394, 8.3.2.2].

The transaction layer has responsibilities that are not immediately obvious. It has to keep track of transactions on behalf of applications and associate any responses received with the outgoing message that prompted them. It has to monitor the age of transactions, and discard them and indicate an error if they should time out. It has to generate unique transaction labels for outgoing requests, for association purposes. It has to be able to handle multithreaded transactions, meaning that a requester may issue multiple requests, with many of them outstanding at any one time, and which can be serviced out of order [IEEE1394, 3.5].

5.5.2 The significance of the CSR Architecture primitives

The CSR architecture is defined in [IEEE1212] and defines a structure and set of primitives that a bus should have in order to be compliant. The idea is that nodes, on multiple busses of different types, will be much easier to control and use if software only has to support one set of primitives. Compliant nodes are expected to provide a configuration ROM and

registers that follow a known format. These facilities are provided in a IEEE-1394 node by the Bus Manager Layer mentioned in section 5.5.1.

A node is expected to be able to perform a read, write or lock transaction. The write is a write to another node's address space. The read is a write requesting a reply containing the contents of a particular address. The lock transaction requests that an indivisible read-modify-write operation be performed in the other node's address space. There are a number of subcommands specifying which modification should be applied, such as compare-and-add or mask-and-swap [IEEE1212, 3.3], or others.

The lock transaction is particularly noteworthy, as atomic transactions are very useful in implementing synchronisation primitives in parallel processing system [Tan92, pg 41]. Seeing as this project is intended to be used in applications where distributed parallel processing is possible, this feature is important. To some extent, the fact that I have implemented a message passing system that can be virtualised over the bus without much effort makes this feature less important. However, I think it likely that it will be used in this system sometime in the future, as it is such a useful primitive.

5.5.3 My 1394 protocol stack implementation

The Texas Instruments chipset that I have used to implement the IEEE-1394 hardware support comes in two parts: a physical layer and link layer chip. The link layer chip, as its name implies, implements most of the IEEE-1394 link layer. It expects to be fed packet data in particular formats [Tex98, pg 5-1]. The format tells the chip whether the packet is a read, write or lock transaction, and whether the data is a response or a request. The different types of packet possible are shown in Table 5.1. The formats expected by the link layer chip, and the formats specified in the IEEE-1394 standard, are similar but not exactly the same (For example, contrast [IEEE1394, 6.2.2] and [Tex98, 5.1.1]). Also, any other chipset which may be used by ISSI in the future could be expected to have different formats, and the method for presenting the data to the chipset will most likely be different.

This indicated to me that I should try to keep the hardware dependent parts of the stack separate from the remainder. I decided to break the IEEE-1394 drivers and stack into different files, as shown in Figure 5.5.

At the lowest level is the file "ilink.s", which is merely an assembler wrapper for the interrupt vector associated with the IEEE-1394 hardware. This should be independent of the chipset

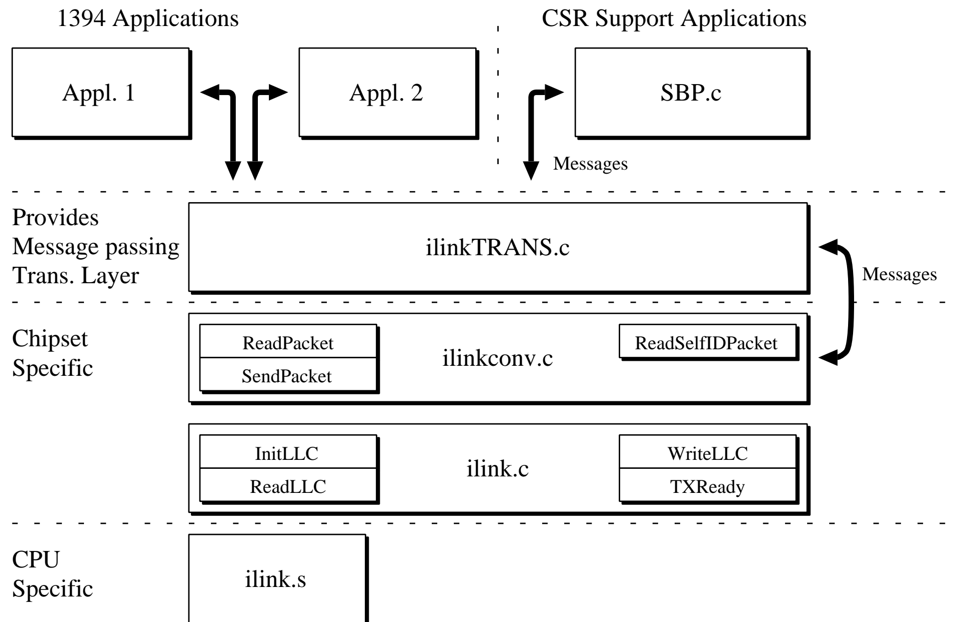


Figure 5.5: An overview of my IEEE-1394 stack implementation.

used but is dependent on the CPU, as it is written in ARM7 assembler. The software layer above that, "ilink.c" which is written in C, is dependent on the the IEEE-1394 chipset. It provides routines that initialise the chipset, read control registers, write control registers, and read and write data from and to hardware FIFOs. A second C file, "ilinkconv.c", which is specific to the data format used by the chipset, provides routines that convert from the IEEE-1394 standard to the chipset packet formats. "ilink.c" and "ilinkconv.c" use functions from each other and can be considered as one functional unit. "ilink.c" communicates with the transaction layer using the kernel message passing mechanism.

The transaction layer has various responsibilities as discussed in section 5.5.1. The need to associate multiple transactions implies a need to have a queue of all outgoing requests. "ilinkTRANS.c" maintains a list of outgoing requests and their transaction labels in order to keep track of them. Incoming packets are automatically queued by the message passing mechanism, through which they are passed to get from the link layer to the transaction layer. Outgoing packets are also queued by the message passing system, so no queuing is done internally to the transaction layer.

The asynchronous message passing paradigm mapped particularly well to this problem, as can be seen in Figure 5.6. It has saved me having to implement multiple input and output queues specifically to keep track of multiple transactions. It has only been necessary to

Transaction Code	Name	Comment
0_{16}	Write request for data quadlet	Quadlet payload
1_{16}	Write request of data block	Block payload
2_{16}	Write response	No payload
4_{16}	Read request for data quadlet	No payload
5_{16}	Read request for data block	Quad payload
6_{16}	Read response for data quadlet	Quad payload
7_{16}	Read response for data block	Block payload
8_{16}	Cycle start packet	Quad payload
9_{16}	Lock request	Block payload
A_{16}	Isochronous data block	Block payload
B_{16}	Lock request	Block payload

Table 5.1: The different possible types of packet available in IEEE-1394, from [IEEE1394, 6.2.4.5]. Unspecified values are reserved.

implement code that literally "tags" transactions and sends them on their way.

The only complication has been keeping track of when a transaction times out. The transaction layer code solves this problem by calculating when the transaction will time out and storing this in the internal list of transaction codes (the "tags" mentioned earlier). It sends a message to the timer server asking for a reply when the oldest message will time out. This message is waited for in parallel with incoming and outgoing transactions, and its reception forces a flush of any out-out-date transactions.

This approach works well because only one timer request is ever issued at a time, and when multiple transactions are successfully conducted, only one message per timeout period is generated, and the timeout period is typically quite long. Other data about the transactions are stored to stop late replies from being associated with new transactions that happen to have the same transaction code.

Above this layer are the applications and CSR address space handlers. Ordinary applications can only initiate transactions and receive replies. Address space handlers are mapped into the CSR address space by the transaction layer. Incoming request packets are delivered to the correct CSR handler depending on their address. At the moment, this is done using a switch (or case) statement. This is good enough for this small system, but it could be easily upgraded to be dynamic by making the transaction layer register a message ID to be used by handlers to assign themselves address spaces.

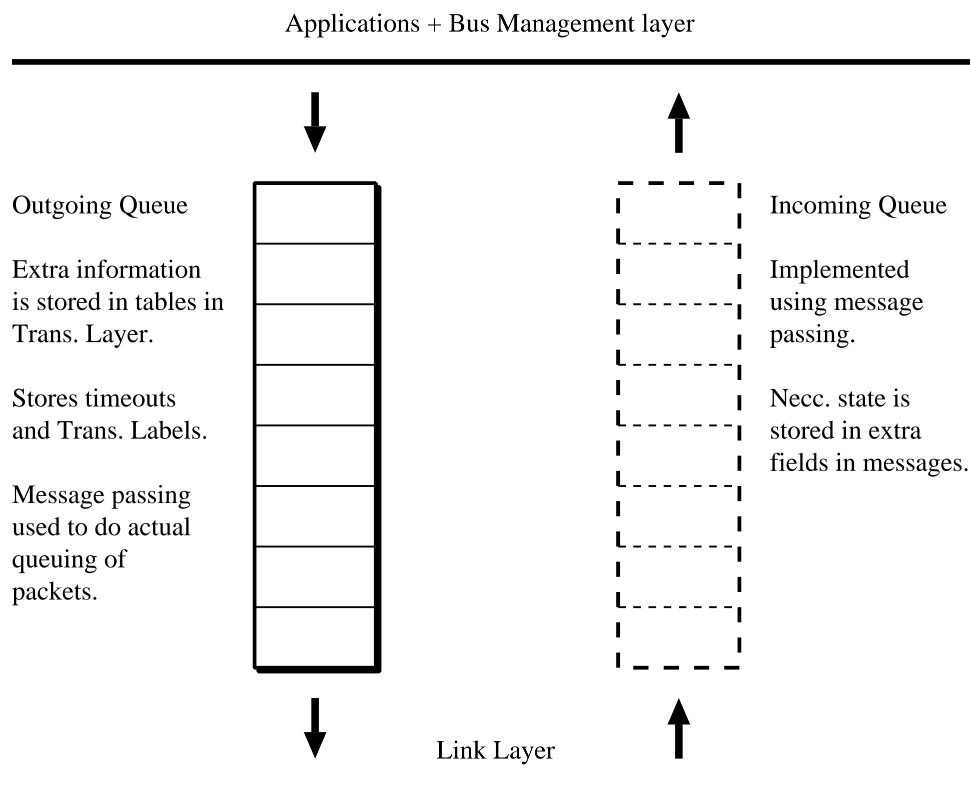


Figure 5.6: The queue structure used to implement the transaction layer functions.

5.6 Testing the 1394 stack

At the time of writing, my implementation of the stack is not fully IEEE-1394 compliant. That is, it doesn't provide all the facilities required in order to meet the minimum requirements [IEEE1394, 1.7]. However, it does provide a sufficient subset of capabilities. The infrastructure is in place and I have tried to provide at least one implementation of any significant class of operation. This infrastructure has been built and tested in stages.

5.6.1 Stages of implementation and testing

My first step was to read from, and write to, the chipset. After solving the problem mentioned in section 4.2.1, I started writing the low-level (bit level) infrastructure to talk to the chipset. Because of the way in which I had wired the 1394 chipset, I could only do a 32-bit register read or write as two separate 16-bit operations. Thus I had to write two small functions that could read or write 32-bit values. This suited me well because I was trying to separate the higher levels of the IEEE-1394 stack from the hardware. Forcing any operation on the chipset to go through those functions has made it simple to modify the stack to work on future boards with different memory layouts. I wrote these functions and tested them by copying the LLC internal state into another section of memory, and by changing various registers. Examining the memory dumps showed that the functions were working as desired.

The next stage was writing an interrupt routine that would examine the LLC status and call any relevant subroutines. Soon I had a section of code that would be called when a packet had arrived. I then started writing "ilinkconv.c", which provides the facilities to read a packet out of the LLC FIFO, and convert them to the IEEE-1394 specification format, which I had decided would be the format used by the link layer. I sent test packets by hand to the board, using a Philips Reference Design Kit (RDK) that I had borrowed for this purpose. I breakpointed the conversion routines and checked their output.

At this point it became possible to start the implementation of the transaction layer. I decided on the method that an application would use for requesting a read packet be sent to another device and wrote a small test application. I then wrote the transaction layer to accept requests from applications and pass them down to the link layer. At the same time, I modified the code I had already implemented at lower layers to send packets to the transaction layer. Eventually, the read request was successfully transferred from the

application to the transaction layer to the chipset and across to the Philips RDK. Using the interface on the Philips board, I could analyse the structure of the packet and convince myself that it was as it should be. Naturally, the RDK also responded correctly to the read request, which meant that I had to implement more infrastructure to get the response back up to the application. It also became necessary to associate incoming responses with a previous outgoing request, which forced me to construct the transaction layer labelling system I mentioned in section 5.5.3. This I tested with single outgoing reads and then, eventually, a program that did multiple reads from the address space of other nodes.

The work I had done allowed applications on the board to query other IEEE-1394 devices, but it still did not let local applications provide services to other devices. An incoming read or write had no obvious target - the transaction layer only knew how to associate incoming responses with previous transaction requests. This also meant that I did not have a mechanism that would provide facilities to implement the Bus Management Layer. I added the code mentioned in section 5.5.3 to route incoming requests to higher layers. I needed this capability to control a IEEE-1394 hard disk drive (HDD) that will be used at ISSI in the future, and the work I have done on the HDD driver has shown that this code behaves as I expect.

5.6.2 Summary

I am confident that there are no fundamental design flaws in my IEEE-1394 implementation. I have successfully communicated with a Philips RDK board and an IEEE-1394 HDD. Adding new facilities as I have needed them has been a comfortable experience, not requiring any major rewrites, and I think I have covered enough of the required functionality to uncover any potential structural flaws. However, I think it is possible that there are still bugs lurking within the implementation, either because I have misunderstood one or more details of the IEEE-1394 standard, or because I have simply done something wrong. I hope that my efforts to make the code easily understandable and readable will mitigate any such problems.

Chapter 6

Consolidation

In section 3.4, I explained what I intended to achieve in this project. So far I have stated what I have done and why I chose various components or methods of implementation. I have listed the problems I had and how I overcame them. I have shown how I tested the sections of the system as I implemented them. However, I have not yet clearly stated the successes, limitations and shortcomings of my current implementation.

6.1 The hardware

In section 3.3.1, I explained my vision for each hardware module in a future system. I have produced the core of one of these modules, and ISSI has started developing hardware and software around it. This collision between a vision and reality has highlighted some problems with my approach.

My initial prototype provided an IDC connector with most of the signals I thought would be used for interfacing to the prototype. In the end, as the availability of components changed and the specifications of the devices that would interface to it changed, it became easier for ISSI to build their own prototype board. They added their own Programmable Logic Device (PLD), to replace some of my discrete logic devices, and added their own logic for interfacing to their devices. Thus, the extra effort I spent manually routing those connections was largely wasted.

As I said in section 4.1, I initially picked the TSB12LV31 Link Layer Controller simply because it was available. It has reasonable power consumption but has a very small on-

board FIFO, severely limiting the size of the packets that it can accept. This could create performance limitations in the future. There is a successor to this chip with much larger FIFOs, but it consumes significantly more power. I think the final solution will be to port the software to another vendor's IEEE-1394 chipset, when it becomes available. Some companies are announcing single chip solutions that contain both the LLC and PHY, and are claiming better power consumption and FIFO sizes. The current implementation uses more power than I would like for the IEEE-1394 chipset and loads the CPU more than it should.

In retrospect, the Atmel AT91M40400 still seems like a good choice of CPU. It is around four times faster than the Transputer CPU it replaces, according to benchmarks done at ISSI. ARM is selling its CPU core to more and more companies, and it looks like there will be a good variety of ARM-based MCUs available in the future. The power consumption of the chip is well within the estimates given by the manufacturers. However, it is already not powerful enough for some of the applications envisioned. I had intended that one ADC be connected to one CPU board. This has already fallen by the wayside, as ISSI have decided it would not be cost effective. However, they are retaining the IEEE-1394 chipset in acknowledgement that it provides a high level of flexibility in expansion.

The choice of IEEE-1394 as an interconnection bus for this system has been vindicated. When I first chose it, there were very few commodity items available that supported it. Since then, it has grown slowly in the audio and video market, and is starting to find its way into some computer products, such as digital cameras and HDDs. The fact that HDDs are becoming available means that no in-house development of extra hardware is required to store large volumes of data. There are laptops available now with built-in IEEE-1394 ports, meaning that if debug data and realtime data are streamed to the bus, technicians in the field have access to a powerful debugging tool. Also, data can be copied off internal HDDs on a standalone system, with little trouble.

The greatest problem with my design is that it uses SRAM for storage. The SRAM we use provides excellent power consumption and acceptable speed but it is expensive on a per megabyte basis. Some applications envisioned by ISSI would require more RAM, yet this would push up the price of the board substantially. The SRAM on the board is currently the most expensive part of the design. Unfortunately, I have not been able to find anything better.

6.2 The software

The work I have done can be broken into three sections. First, I needed to implement a basic kernel. Then I had to implement some form of inter-process communication. Finally, I decided to implement a IEEE-1394 stack.

6.2.1 The kernel

In section 5.2, I have discussed all the requirements for my kernel, and how I decided to approach the problem. I tried to implement only the primitives I would need and I think I have succeeded in this respect. I generally only implemented functions as and when necessary, and I haven't needed to add any more for several months. ISSI has been writing code that runs on this kernel as well, and I have had no requests for improvement to the primitives from them.

My primary concern with the kernel is the context switching and the scheduler. Currently it is a simple round-robin switcher, with only one priority level. The switcher currently uses a simple array of contexts to store the system state. Each context has 128 bytes allocated (but only uses enough for the registers and a few state bits) and the switcher skips through the list until it finds a valid task to execute. This is very fast if the next context is ready to run, taking only 36 ARM opcodes for the complete context switch. Every task that is not ready to run, for whatever reason, takes an extra nine opcodes to skip. This is pleasantly fast, and would be good enough if processes were guaranteed to spend most of their time doing blocking waits, but as soon as intensive processing is done, processes will start using their complete timeslice, and the overall system response time will increase dramatically. The context switcher can comfortably switch at a rate exceeding 1000 contexts per second, but each switch takes a minimum amount of time and the overall efficiency of the system decreases as the switcher is called more often. This means that increasing the number of context switches per second, until the round trip time for all processes is short enough, is unrealistic.

I have tried to remain agnostic as to what scheduler should be used. The legacy ISSI system uses a simple multi-level priority scheduling system: lower priority tasks do not run until all the higher priority tasks are blocked. ISSI have used this very successfully, so I expect that this will be their first choice for a scheduler.

6.2.2 The Inter-Process Communication system

I have discussed my choice of IPC system in section 5.2.1 and the reasons for my choice; I had several false starts before settling on the current implementation. The message passing routines are linked into the final system once and called from whatever process needs them, and include the necessary instructions to serialise execution within themselves. These atomic sections of code are the areas in which I think improvement could be made. They serialise execution by disabling interrupts in the CPU core.

Although code that is expected to respond quickly to outside stimulus should be run from the FIQ interrupt and thus will not be affected by the disabling of the IRQ interrupts, I would still prefer the IRQ response to be as fast as possible. It should be possible to break single large atomic sections in the message passing functions into many smaller atomic sections; this should decrease IRQ interrupt latency.

Another area for improvement is the function to send messages - it does a linear search (i.e. complexity O(n)) to find the linked list corresponding to a message ID. With some modification to other functions, this could be changed to a binary search (which has complexity $O(log_2 n)$). Also, once a particular list is found, the code steps through the list to find the end and adds a message. This could be made O(1) by adding a pointer to the last item in the list to the message passing data structures. I did not add this when I first wrote the code, as I was more interested in correctness than speed. I did make a conscious effort to put any slower code in the "send" functions. This ensures that the producer would always be slower than a consumer. This may not be necessary but negative feedback is more attractive to me than positive feedback.

So far, latencies in the message passing system have not had any noticeable effects.

Another reservation that I have about the message passing system is that it has no way of dealing with low memory conditions. Every process should check that an attempt to allocate a message succeeds but this can complicate its code substantially. Each process should deal with being unable to get a message in a sensible manner, just like ordinary programs should check that a memory allocation has succeeded. I would be much happier if I could provide a general scheme that would allow the entire system to modify its performance under low memory conditions. Perhaps the simplest method would be to provide a system call that reports how many messages are free. I have discussed this problem with the programmers at ISSI that will be using my code, and we have been unable to think of anything more powerful that remains general.

6.2.3 The IEEE-1394 stack

I have already discussed what I have not implemented in the full IEEE-1394 stack, but I have not given my reservations with those facilities which I have implemented.

Parts of the stack use the IPC system to queue messages. The current scheduling scheme means that packets could languish in limbo between processes, causing unnecessary transaction timeouts. Also, I have not yet put full error checking into the stack code that passes messages. Thus it is possible that a message allocation could fail, due to memory constraints. If the available memory is exceeded, it is likely that the entire system will crash anyway, but this is no excuse.

One way of solving this problem would be for the stack to request enough messages for the entire subsystem on bootup. Messages would be passed in a large circuit, from one process to another, and eventually back to the source. This has the advantage that if some part of the stack is losing messages somewhere, the entire subsystem will quickly grind to a halt. Also, this protects the entire system against denial of service attacks on the IEEE-1394 stack. The current implementation could allocate all available messages if it were flooded with enough packets.

6.3 Parts of the implementation that turned out well

There are parts of this project with which I am very satisfied. Some of those parts were as I designed them, and others simply evolved as I implemented them.

6.3.1 The kernel and message passing system

The paradigm of allowing interrupts to use the message passing system has worked excellently. Hardware dependent code can be isolated within an interrupt routine that communicates with a more generic device driver layer running as a process. My experience implementing the IEEE-1394 stack and the timer server has convinced me that this arrangement works well.

I am also very pleased with the minimalism of my kernel. I have provided very few functions but have not felt the need to provide any more while building other systems above it. At one stage I had anticipated the need to add memory management routines; then I realised

that the current ISSI software uses a standard block size for all its data and manages lists of these blocks. It also struck me that allocating a message for use was much the same as allocating memory. I decided that messages could be allocated and used as memory, and that list management would effectively be done by the kernel. This has had the advantage of making it easy to see who owns a particular section of memory. The programmers at ISSI seem to be comfortable with this arrangement, and I have not needed anything more general to date.

My message passing implementation only passes around pointers to static parts of RAM, making it fast, with other advantages: processes are merely stipulating who owns a message by passing it around. As the kernel knows exactly what pointers are valid and over what range they extend, it is trivial to virtualise this process over a network. The current implementation throws a "ERROR_CatchallNotImplementedYet" error when it has to send a message to an unknown destination. This error can be replaced with a call to a function that passes the message to another node. Some form of routing has to be implemented, but this could be as simple as a table manually configured through some custom CSR registers.

6.3.2 The hardware

I think that the current hardware is a good compromise between cost, power consumption and functionality. Testing has shown that it works as expected. It is also of a level of complexity that I had not previously implemented. Although routing the design manually was a frustrating task, it gave me a great sense of accomplishment when I completed it. Not only am I happy that does what it should, I am happy that I successfully built a prototype and programmed it.

6.4 Conclusion

I stated my objectives in section 3.4. I have done the following;

- I have provided hardware to which ISSI have successfully connected external devices.
- I have provided a multi-tasking kernel with a message passing system that provides the same basic functionality as the legacy ISSI hardware, and I have implemented enough code under it to show that it works.

- I have written several device drivers to show how further drivers could be implemented and, more to the point, that it is possible to do so.
- I have implemented an IEEE-1394 stack that has successfully communicated with other IEEE-1394 devices, such as a Reference Design Kit and a HDD.

Furthermore, I have designed a system that is being used by other people and may eventually be part of a commercial product for data acquisition in many environments.

References

[And98] D. Anderson, “FireWire System Architecture, Second Edition”, Addison-Wesley, 1998

[ARM96] ARM Ltd, “The ARM7TDMI Data Sheet”, ref: 0673a 06/96, Advanced RISC Machines Ltd, 1996

[Atm98] Atmel, “AT91M40400 Electrical and Mechanical Characteristics”, ref: doc1078, Atmel Corp., 1998

[Bur90] A. Burns, A. Wellings, “Real-Time Systems and their Programming Languages”, Addison-Wesley, 1990

[Bur93] A. Burns, G. Davies, “Concurrent Programming”, Addison-Wesley, 1993

[Coo96] J.E. Cooling, “Languages for the programming of real-time embedded systems: a survey and comparison”, Microprocessors and Microsystems 20 (1996) 67-77

[IEEE1212] IEEE, Inc., “Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses”, IEEE Std 1212-1994, IEEE, New York, 5 October 1994

[IEEE1394] IEEE Computer Society, “IEEE Standard for a High Performance Serial Bus”, IEEE Std 1394-1995, IEEE, New York, 30 August 1996

[Lau99] R. Laubscher, “An investigation into the use of IEEE-1394 for audio and control data distribution in musical studio environments”, MSc Thesis, Rhodes University, 1999

[Mag96] A.P. Magalhaes, M.Z. Rela, J.G. Silva, “On the nature of deadlines”, Microprocessors and Microsystems 20 (1996) 79-88

[Mot96] Motorola, “DSP56L811 Technical Data Sheet”, ref: DSP56L811/D Rev.2, Motorola Inc., 1996

[Mot98] Motorola, “MCM6323A Technical Data Sheet”,ref: MCM6323A/D, Motorola Inc, 1998

[Tan92] A.S. Tannenbaum, “Modern Operating Systems”, Prentice Hall International Editions, 1992

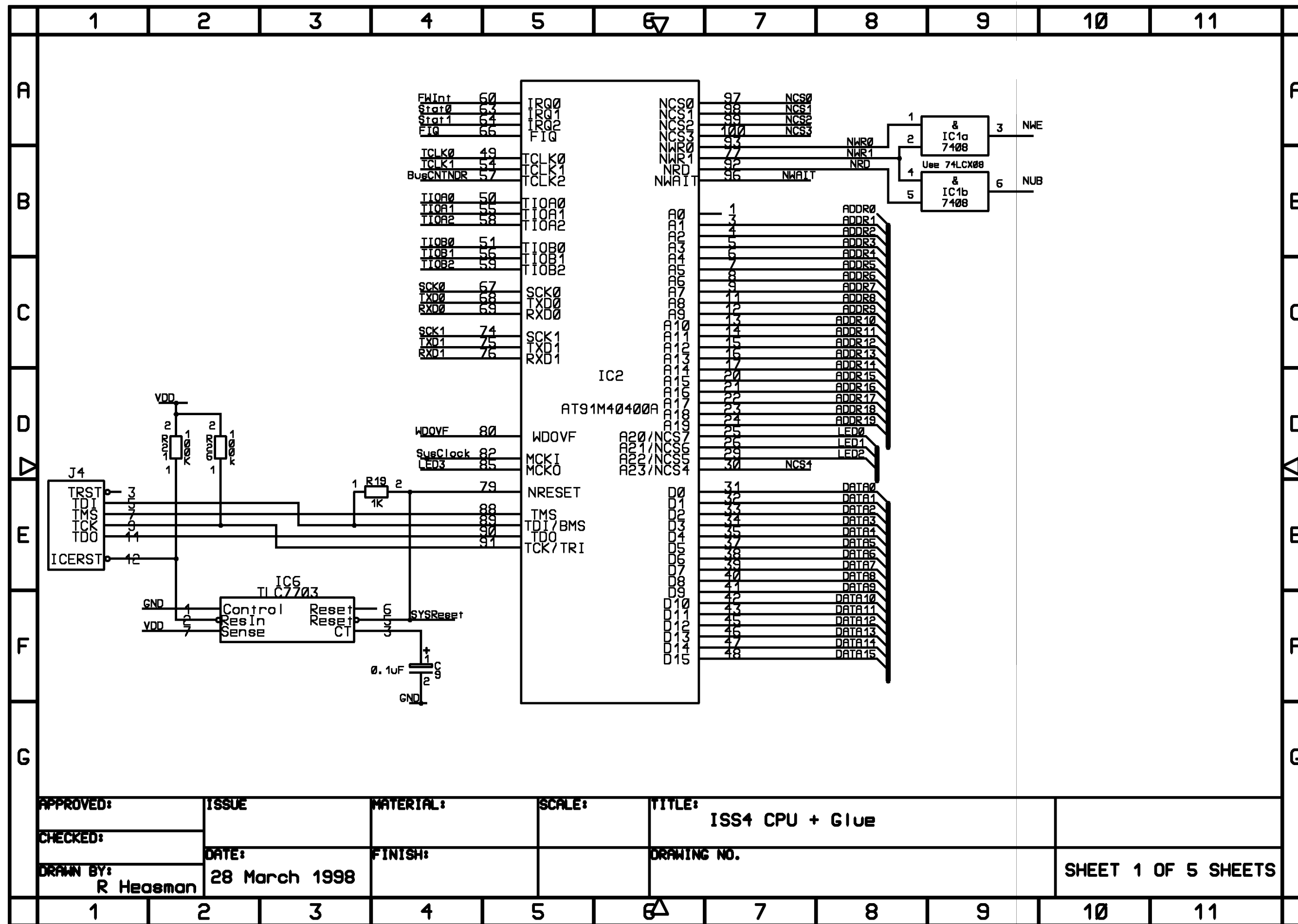
[Tex98] Texas Instruments, “TSB12LV31 Data Manual”, ref: SLLS255A, Texas Instruments, 1998

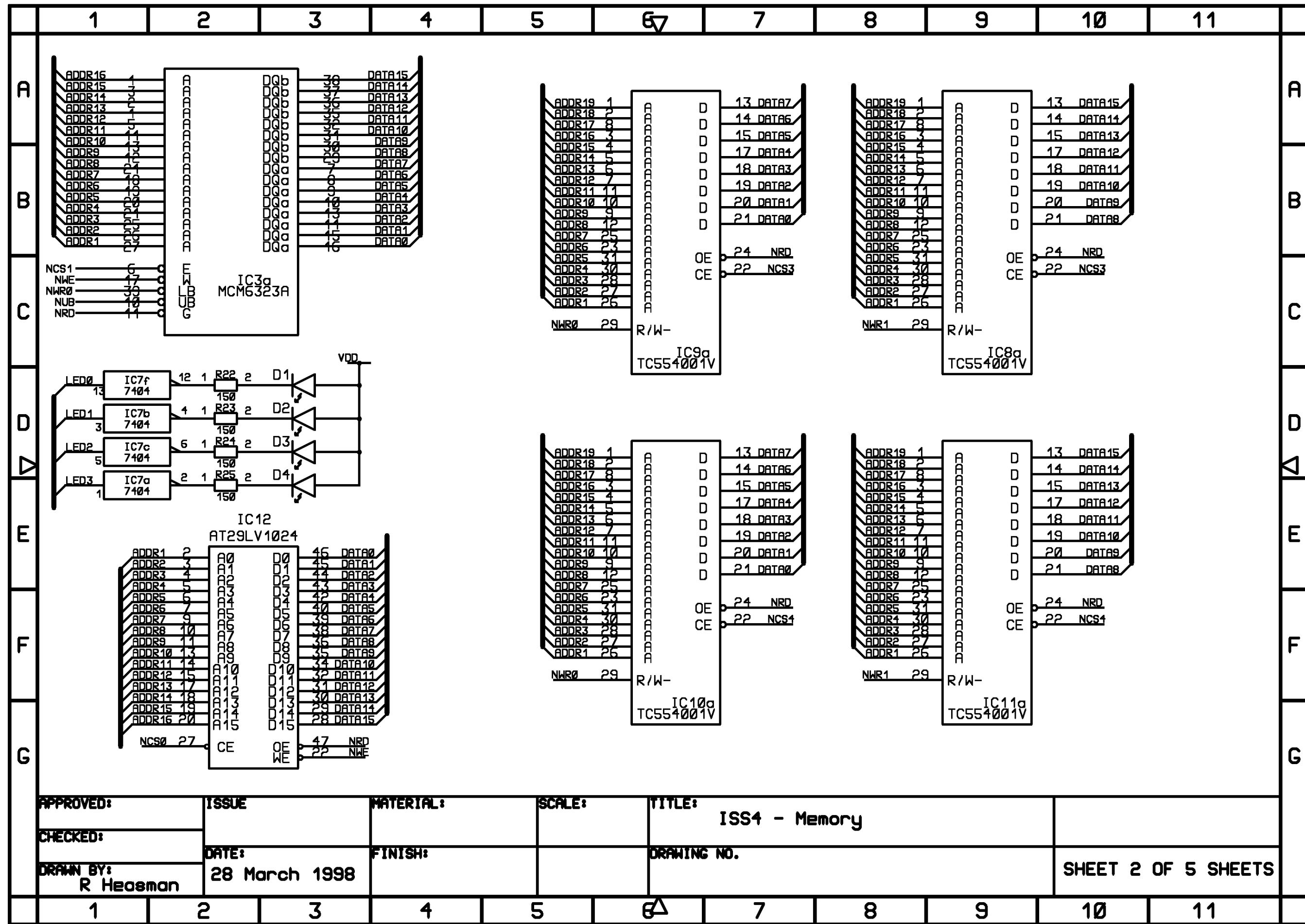
[Tos97] Toshiba, “TC554001FTL-70V,-85V,-10V Data Sheet”, ref: 961001EBA1, Toshiba America Electronics Components, Inc., 1997

Appendix A

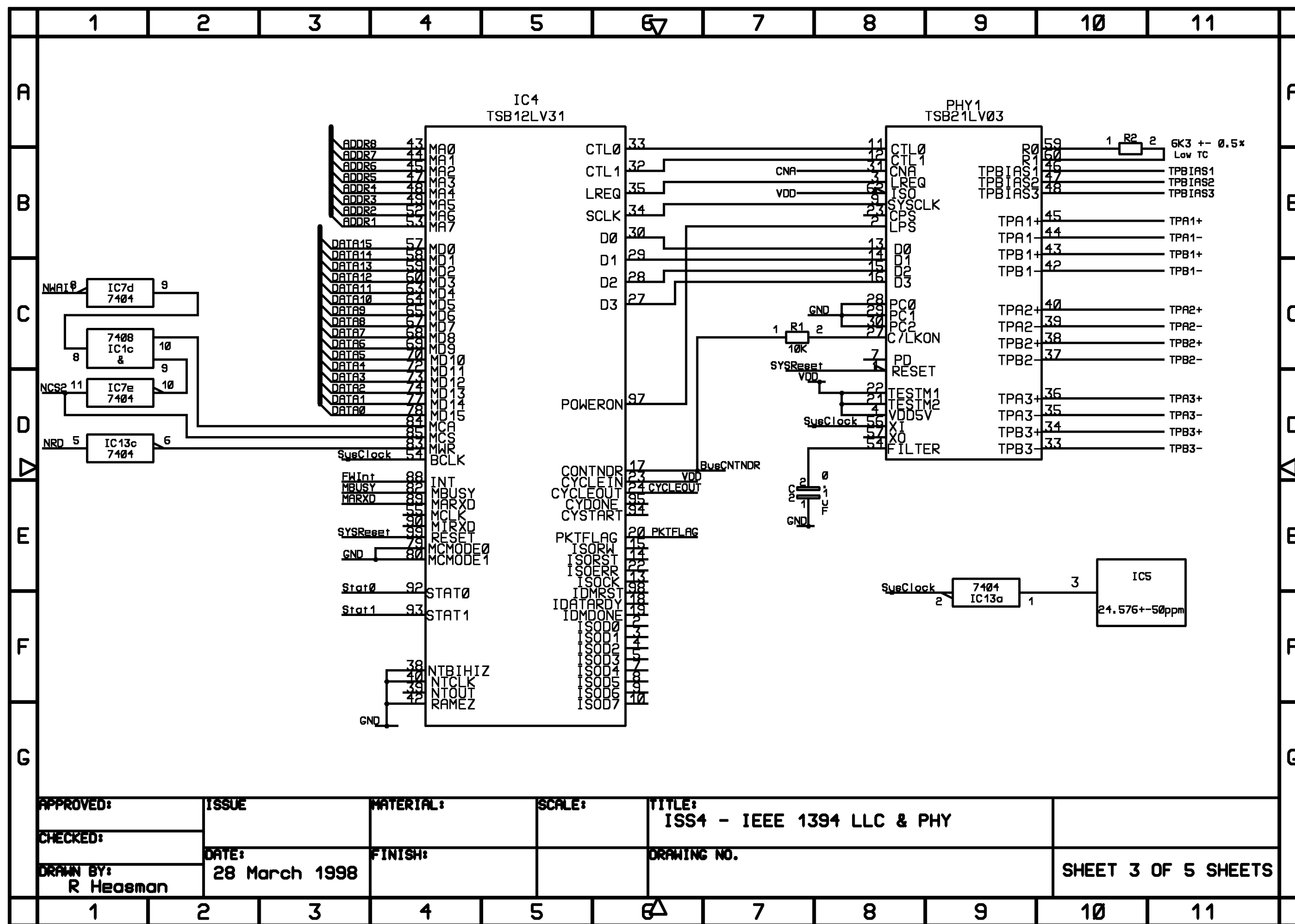
Schematics

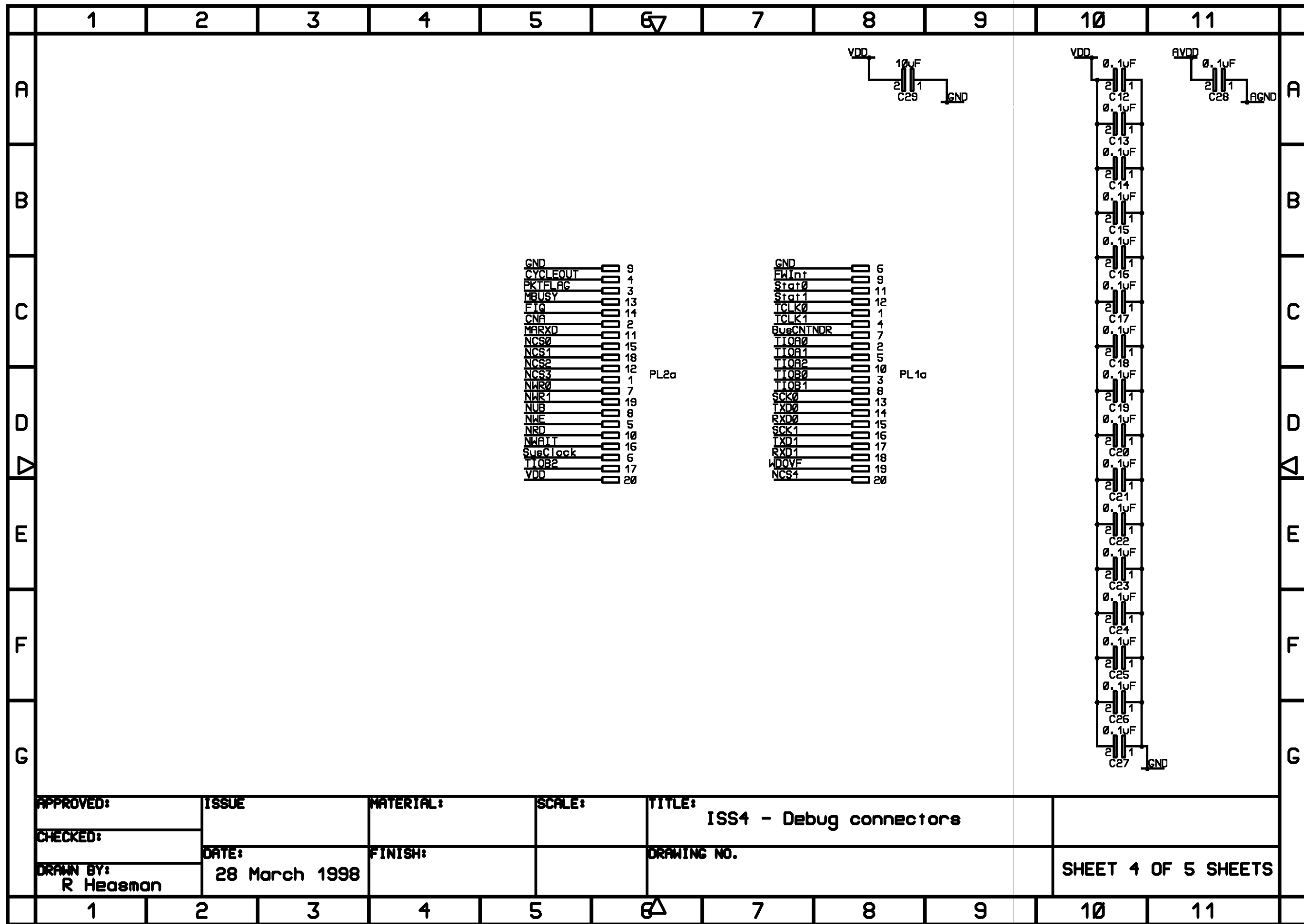
The following five pages contain the schematics for the project hardware.

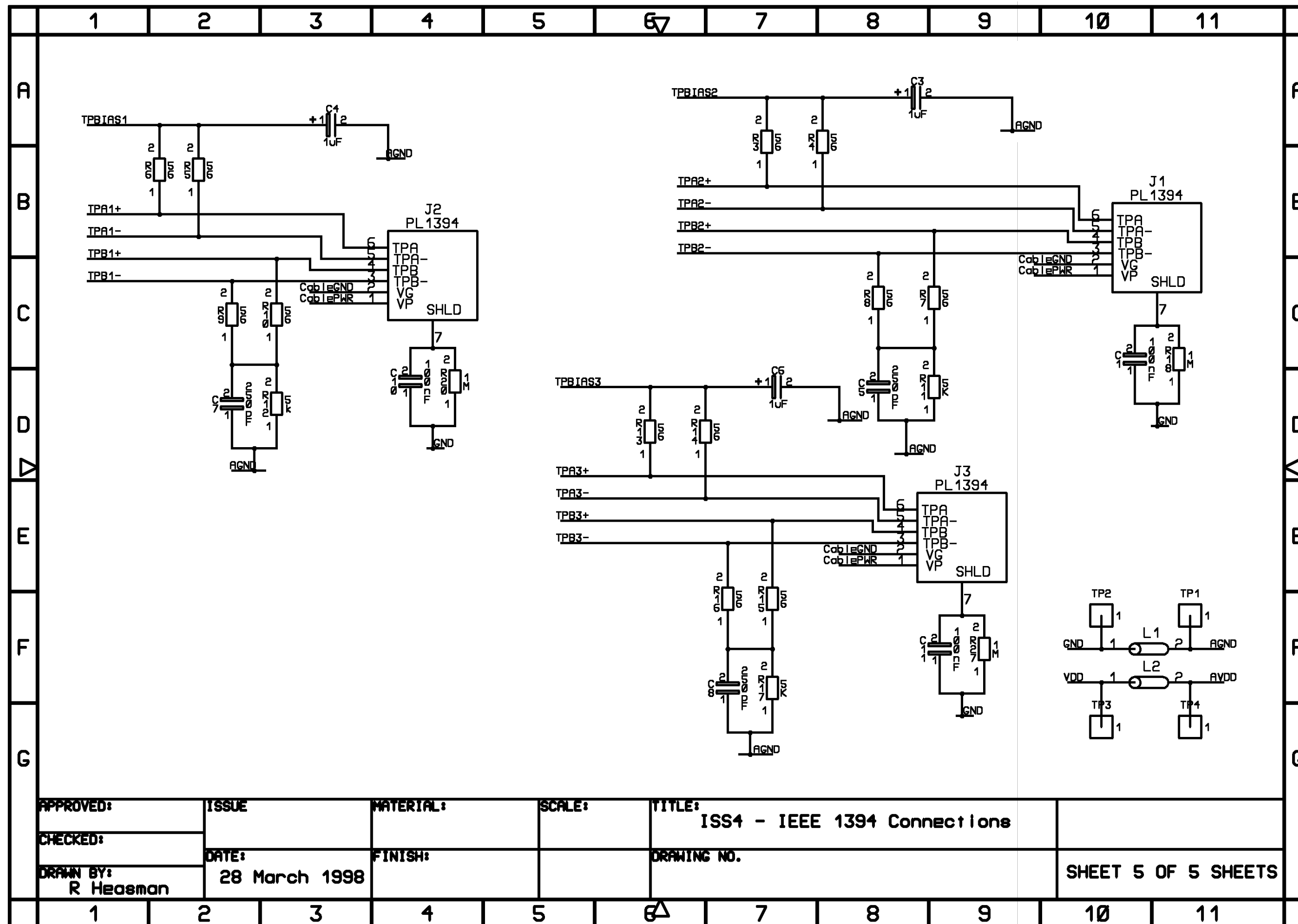

IC2
AT91M40400A
IC1a
7408
IC1b
7408
Use 74LCX08
NWE
NUB
IC6
TLC7703
J4
TRST
TDI
TMS
TCK
TDO
ICERST
R19
1K
0.1uF
VDD
GND
SYSReset
NRESET
WDOVF
MCKI
MCKO
APPROVED:
CHECKED:
DRAWN BY:
R Heasman
ISSUE
DATE:
28 March 1998
MATERIAL:
FINISH:
SCALE:
TITLE:
ISS4 CPU + Glue
DRAWING NO.
SHEET 1 OF 5 SHEETS



IC3a
MCM6323A
IC9a
TC554001V
IC8a
TC554001V
IC10a
TC554001V
IC11a
TC554001V
IC12
AT29LV1024
IC7f 7404
IC7b 7404
IC7c 7404
IC7a 7404
R22 150
R23 150
R24 150
R25 150
D1
D2
D3
D4
VDD
LED0
LED1
LED2
LED3
APPROVED:
CHECKED:
DRAWN BY:
R Heasman
ISSUE
DATE:
28 March 1998
MATERIAL:
FINISH:
SCALE:
TITLE:
ISS4 - Memory
DRAWING NO.
SHEET 2 OF 5 SHEETS


IC4
TSB12LV31
PHY1
TSB21LV03
IC5
24.576+-50ppm
APPROVED:
CHECKED:
DRAWN BY:
R Heasman
ISSUE
DATE:
28 March 1998
MATERIAL:
FINISH:
SCALE:
TITLE:
ISS4 - IEEE 1394 LLC & PHY
DRAWING NO.
SHEET 3 OF 5 SHEETS

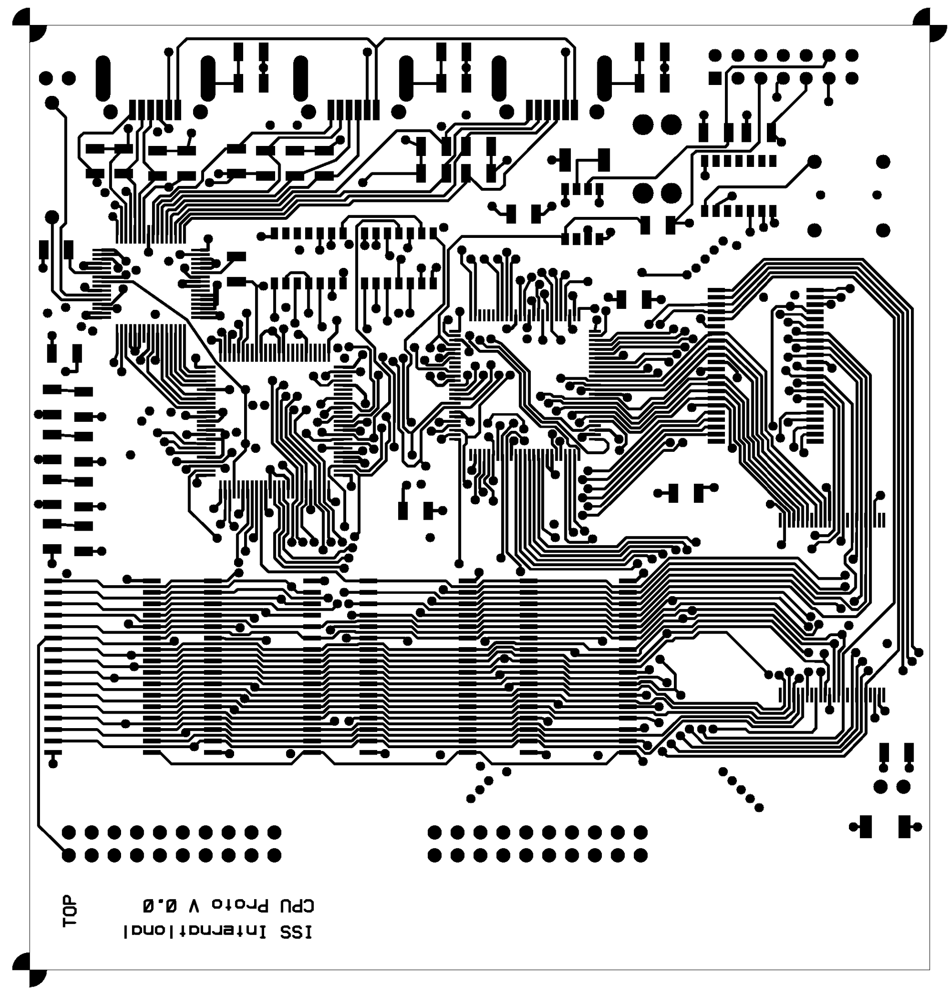

VDD
10uF
C29
GND
VDD
0.1uF
C12
C13
C14
C15
C16
C17
C18
C19
C20
C21
C22
C23
C24
C25
C26
C27
AVDD
0.1uF
C28
AGND
GND
CYCLEOUT
PKTFLAG
MBUSY
FIQ
CNA
MARXD
NCS0
NCS1
NCS2
NCS3
NWR0
NWR1
NUB
NWE
NRD
NWAIT
SysClock
TIOB2
VDD
PL2a
GND
FWInt
Stat0
Stat1
TCLK0
TCLK1
BusCNTNDR
TIOA0
TIOA1
TIOA2
TIOB0
TIOB1
SCK0
TXD0
RXD0
SCK1
TXD1
RXD1
WDOVF
NCS4
PL1a
APPROVED:
CHECKED:
DRAWN BY:
R Heasman
ISSUE
DATE:
28 March 1998
MATERIAL:
FINISH:
SCALE:
TITLE:
ISS4 - Debug connectors
DRAWING NO.
SHEET 4 OF 5 SHEETS

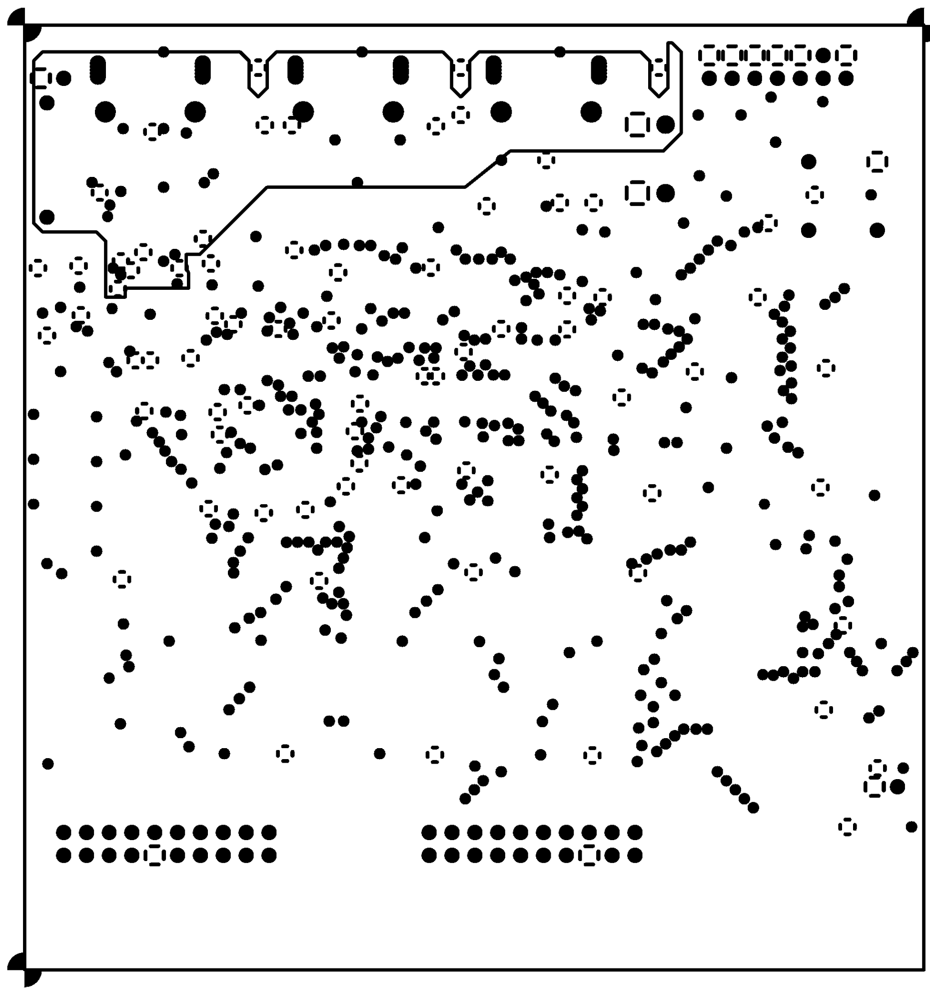

TPBIAS1
C4
1uF
AGND
TPA1+
TPA1-
TPB1+
TPB1-
J2
PL1394
TPA
TPA-
TPB
TPB-
VG
VP
SHLD
CableGND
CablePWR
GND
TPBIAS2
C3
1uF
TPA2+
TPA2-
TPB2+
TPB2-
J1
PL1394
TPBIAS3
C6
1uF
TPA3+
TPA3-
TPB3+
TPB3-
J3
PL1394
TP1
TP2
TP3
TP4
L1
L2
GND
AGND
VDD
AVDD
APPROVED:
CHECKED:
DRAWN BY:
R Heasman
ISSUE
DATE:
28 March 1998
MATERIAL:
FINISH:
SCALE:
TITLE:
ISS4 - IEEE 1394 Connections
DRAWING NO.
SHEET 5 OF 5 SHEETS

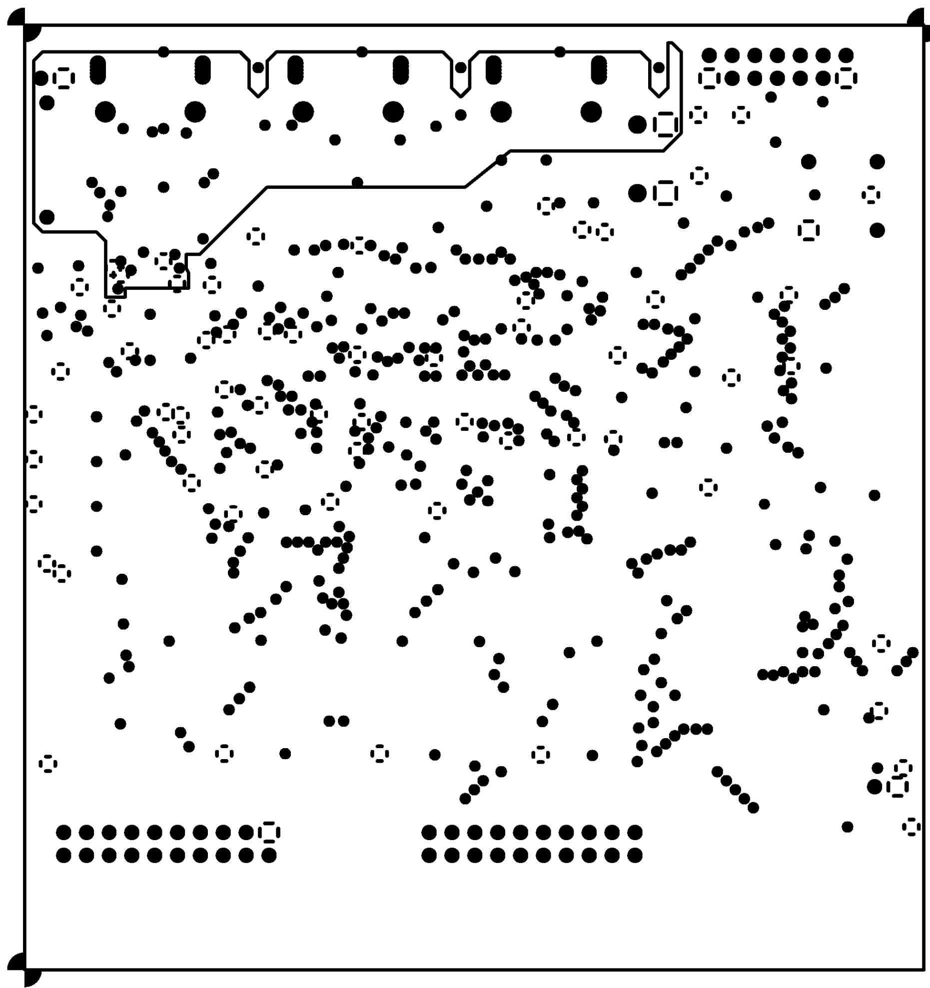
Appendix B

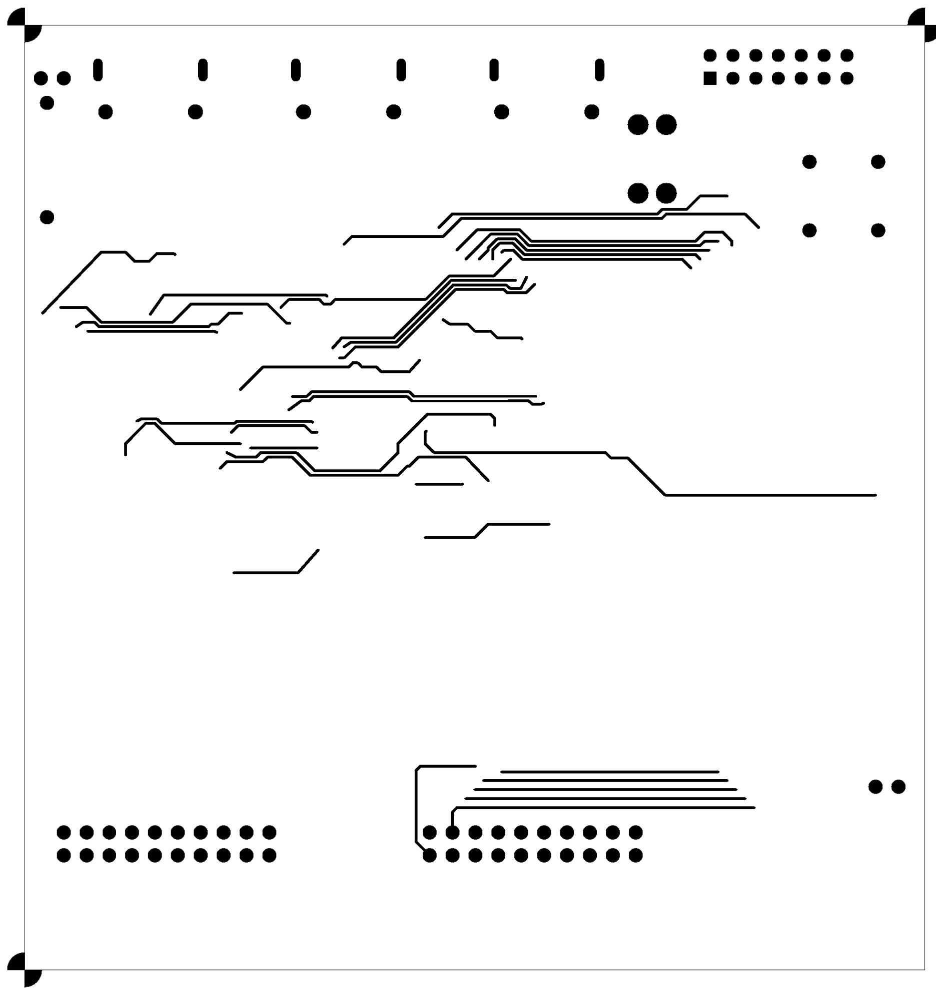
PCB Artwork

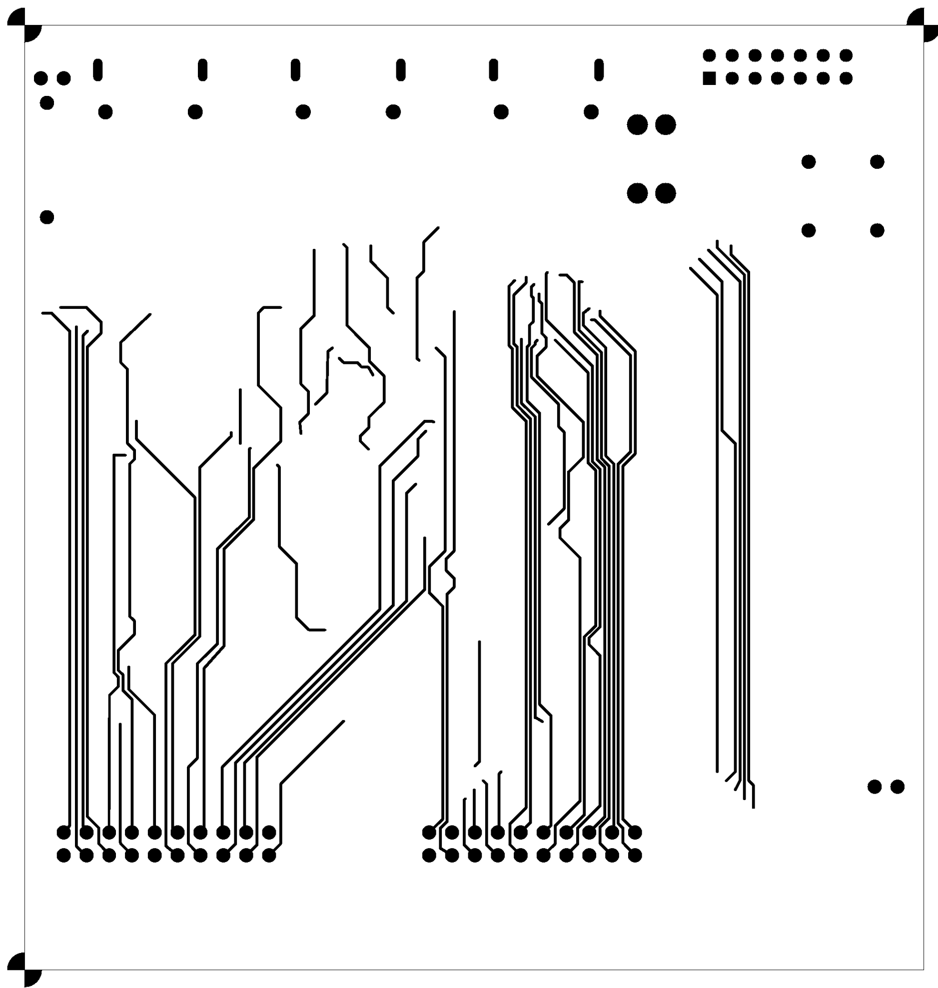
The following pages contain the PCB artwork for the project hardware.

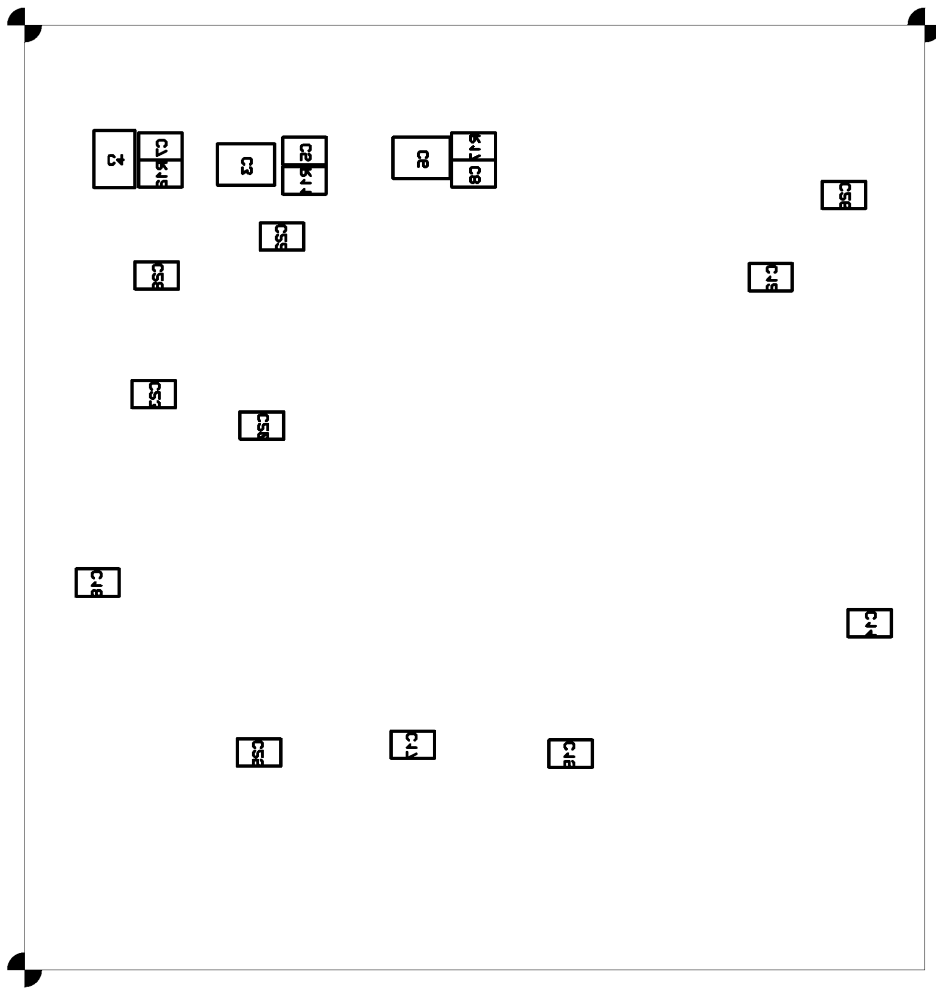

TOP
CPU Proto V 0.0
ISS International

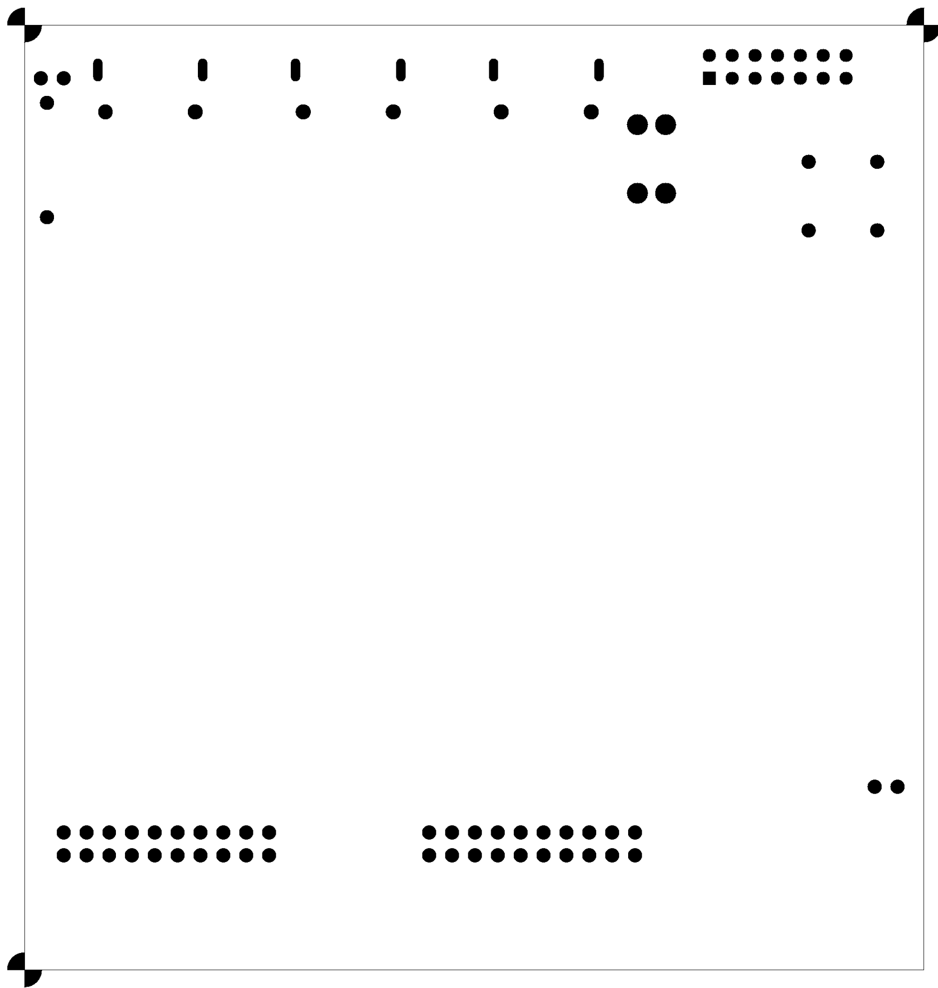


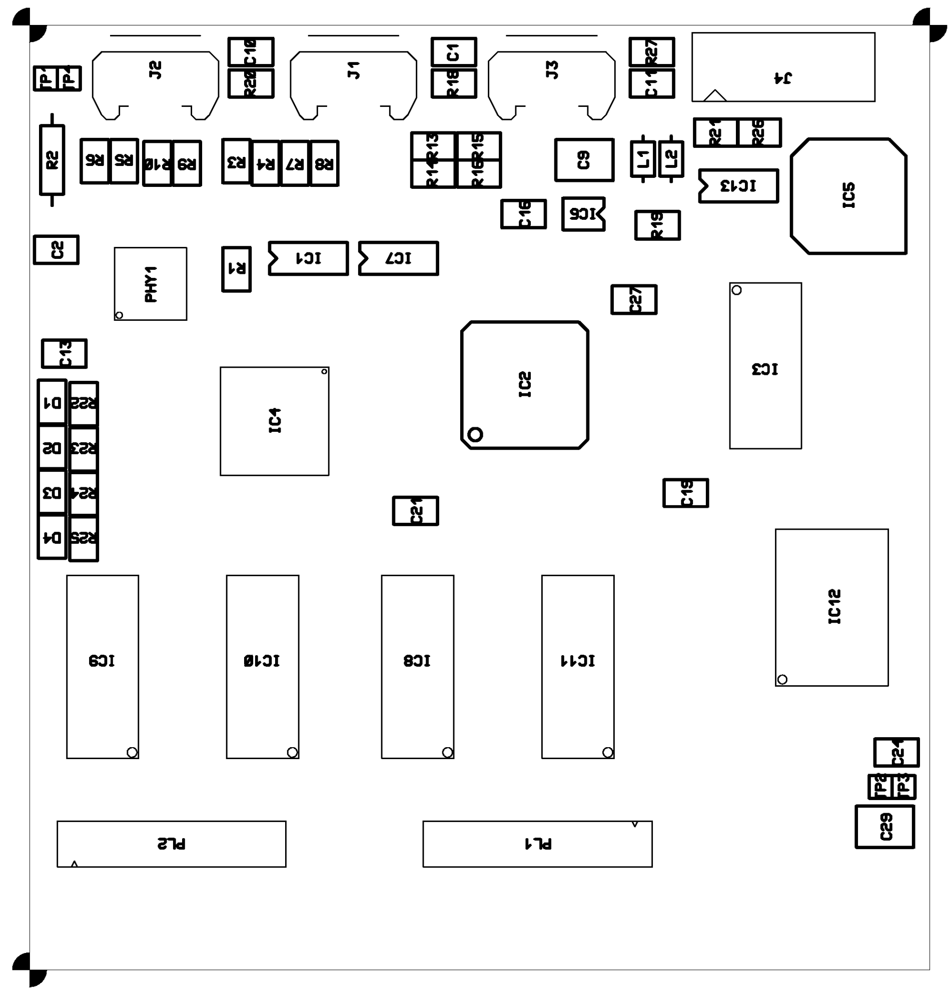






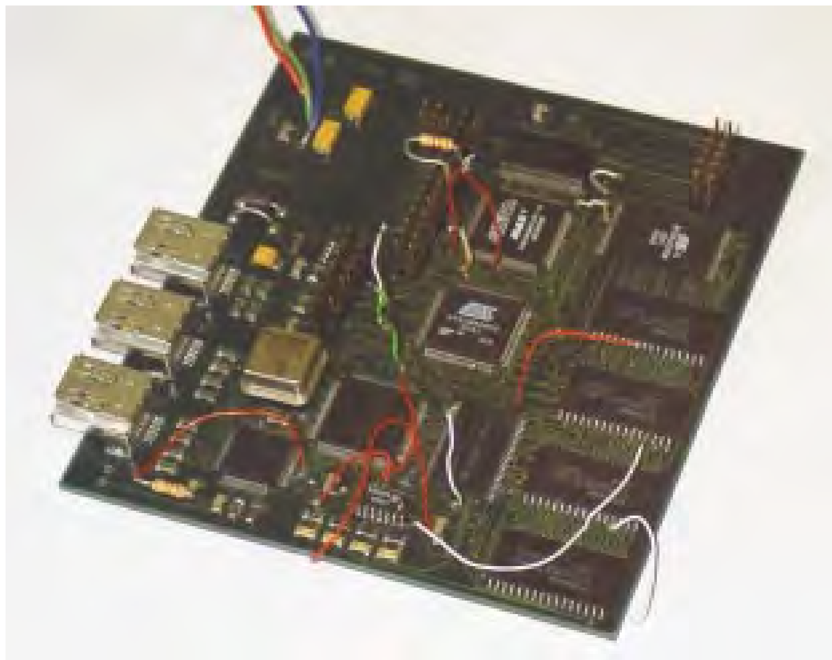





J2
J1
J3
J4
C10
R20
C1
R18
R27
C11
TP1
TP4
R2
R6
R5
R10
R9
R3
R4
R7
R8
R13
R14
R15
R16
C9
L1
L2
R21
R26
IC13
IC5
C16
IC6
R19
C2
PHY1
R1
IC1
IC7
C27
IC3
C13
IC2
IC4
D1
D2
D3
D4
R22
R23
R24
R25
C21
C19
IC12
IC9
IC10
IC8
IC11
C24
TP2
TP3
C29
PL2
PL1

Appendix C

Photograph



This is a picture of one of the later prototype boards, produced after the one mentioned in this thesis. Some logic has been replaced with a PLD, and a 5V to 3V level converter has been added to aid testing software with legacy 5V hardware modules. Some of the wires shown are modifications and the remainder are test points for connection to a logic analyser or oscilloscope.

Appendix D

Source Code

The source code for this project follows. Note that this code is a snapshot of work in progress, as I am adding more facilities for ISSI. However, the message passing and kernel functions have been stable for a while.

AddAlarm 133 (timer.c)
AddProcess...............107 (kernel.c)
AddTime 130 (timer.c)
Block....................105 (kernel.c)
BreakPt 21 (Msgpass.c)
CSRInit....................5 (CSRmain.c)
CSRLoop 4 (CSRmain.c)
CSRROMRead.................3 (CSRmain.c)
CSRROMWriteOrLock 2 (CSRmain.c)
CSRRead....................1 (CSRmain.c)
C_Startup 113 (startup.c)
Check1394IntfOK...........42 (ilink.c)
CheckBMsg 22 (Msgpass.c)
CheckSMsg.................20 (Msgpass.c)
CompareTime 129 (timer.c)
DInt.....................109 (kernel.c)
DebugHandler 34 (debug.c)
DebugMsg..................33 (debug.c)
DelAlarm 135 (timer.c)
DoHDD.....................28 (SBP2.c)
DoPhilosophers 124 (test.c)
DumpMem...................87 (ilinkconv.c)
EInt 108 (kernel.c)
FakeInt..................111 (kernel.c)
GRFNotEmpty 92 (ilinkconv.c)
GetCycTime...............132 (timer.c)
GetMsgID 68 (ilinkTRANS.c)
GetOldestTimeout..........65 (ilinkTRANS.c)
GetPID 103 (kernel.c)
GetTLabel.................61 (ilinkTRANS.c)
HDDLogin 27 (SBP2.c)
HackRand.................121 (test.c)
HandleLkDataInd 76 (ilinkTRANS.c)
HandleLockReq.............80 (ilinkTRANS.c)
HandleRXLockReq 74 (ilinkTRANS.c)
HandleRXLockResp..........75 (ilinkTRANS.c)
HandleRXReadReq 73 (ilinkTRANS.c)
HandleRXReadResp..........70 (ilinkTRANS.c)
HandleRXWriteReq 71 (ilinkTRANS.c)
HandleRXWriteResp.........72 (ilinkTRANS.c)
HandleReadReq 77 (ilinkTRANS.c)
HandleReadResp............79 (ilinkTRANS.c)
HandleTrDataReq 82 (ilinkTRANS.c)
HandleTrDataResp..........81 (ilinkTRANS.c)
HandleTrTimeout 83 (ilinkTRANS.c)
HandleWriteReq............78 (ilinkTRANS.c)
Init1394 59 (ilink.c)
Init1394Ints..............43 (ilink.c)
InitAIC 7 (InitAIC.c)
InitDebugHandler..........35 (debug.c)
InitLLC 58 (ilink.c)
InitMessageSystem.........12 (Msgpass.c)
InitPIO 9 (InitPIO.c)
InitTLayer................67 (ilinkTRANS.c)
IntVec1394C 56 (ilink.c)
LEDOff...................120 (test.c)
LEDOn 118 (test.c)
LEDToggle................119 (test.c)
LLC_AsynchTXStuck 50 (ilink.c)
LLC_CSRResetCommand.......49 (ilink.c)
LLC_CommenceTX 69 (ilinkTRANS.c)
LLC_CycSecond.............54 (ilink.c)
LLC_GRFOverflowed 51 (ilink.c)
LLC_HeaderError...........52 (ilink.c)
LLC_InvalidTCode 53 (ilink.c)
LLC_MCUWriteErr...........55 (ilink.c)
LLC_PhyInt 45 (ilink.c)
LLC_PhyRXData.............46 (ilink.c)
LLC_PhyReset 44 (ilink.c)
LLC_RXedAsynchPacket......48 (ilink.c)

LLC_TXReady 47 (ilink.c)
MsgPassTest..............117 (test.c)
Ph 123 (test.c)
ReadBlockPacket...........90 (ilinkconv.c)
ReadLLC 39 (ilink.c)
ReadNoPayloadPacket.......89 (ilinkconv.c)
ReadPacket 93 (ilinkconv.c)
ReadPhy...................41 (ilink.c)
ReadQuadletPacket 88 (ilinkconv.c)
ReadSelfIDPacket..........91 (ilinkconv.c)
RegisterType 14 (Msgpass.c)
RetireTLabel..............63 (ilinkTRANS.c)
RunTestSuite 126 (test.c)
SBPInit...................32 (SBP2.c)
SBPLoop 31 (SBP2.c)
SBPRead...................29 (SBP2.c)
SBPWrite 30 (SBP2.c)
SendBMsg..................17 (Msgpass.c)
SendBlockPacket 96 (ilinkconv.c)
SendNoPayloadPacket.......95 (ilinkconv.c)
SendPacket 98 (ilinkconv.c)
SendQuadletPacket.........97 (ilinkconv.c)
SendSMsg 15 (Msgpass.c)
SetInt...................110 (kernel.c)
SetTimer 101 (initts.c)
SubTime..................131 (timer.c)
TestILink 115 (test.c)
TimeoutLabel..............64 (ilinkTRANS.c)
TimeoutQNotEmpty 66 (ilinkTRANS.c)
TimerInit................127 (timer.c)
TimerLoop 136 (timer.c)
TimerTest................125 (test.c)
UnBlock 106 (kernel.c)
WaitBMsg..................24 (Msgpass.c)
WaitOnMsgList 23 (Msgpass.c)
WaitRand.................122 (test.c)
WaitSMsg 19 (Msgpass.c)
Wakeup1394................57 (ilink.c)
WriteLLC 37 (ilink.c)
WriteLLCMasked............38 (ilink.c)
WritePhy 40 (ilink.c)
Yield....................104 (kernel.c)
iLinkTransLayer 84 (ilinkTRANS.c)

CSRmain.c **1**
CSRInit....................5
CSRLoop 4
CSRROMRead.................3
CSRROMWriteOrLock 2
CSRRead....................1
InitAIC.c **7**
InitAIC....................7
InitPIO.c **9**
InitPIO....................9
Msgpass.c **11**
BreakPt...................21
CheckBMsg 22
CheckSMsg.................20
InitMessageSystem 12
RegisterType..............14
SendBMsg 17
SendSMsg..................15
WaitBMsg 24
WaitOnMsgList.............23
WaitSMsg 19
SBP2.c **27**
DoHDD 28
HDDLogin..................27
SBPInit 32
SBPLoop...................31
SBPRead 29
SBPWrite..................30
debug.c **33**
DebugHandler..............34
DebugMsg 33
InitDebugHandler..........35
ilink.c **37**
Check1394IntfOK...........42
Init1394 59
Init1394Ints..............43
InitLLC 58
IntVec1394C...............56
LLC_AsynchTXStuck 50
LLC_CSRResetCommand........49
LLC_CycSecond 54
LLC_GRFOverflowed.........51
LLC_HeaderError 52
LLC_InvalidTCode..........53
LLC_MCUWriteErr 55
LLC_PhyInt................45
LLC_PhyRXData 46
LLC_PhyReset..............44
LLC_RXedAsynchPacket 48
LLC_TXReady...............47
ReadLLC 39
ReadPhy...................41
Wakeup1394 57
WriteLLC..................37
WriteLLCMasked 38
WritePhy..................40
ilinkTRANS.c **61**
GetMsgID..................68
GetOldestTimeout 65
GetTLabel.................61
HandleLkDataInd 76
HandleLockReq.............80
HandleRXLockReq 74
HandleRXLockResp..........75
HandleRXReadReq 73
HandleRXReadResp..........70
HandleRXWriteReq 71
HandleRXWriteResp.........72
HandleReadReq 77
HandleReadResp............79

HandleTrDataReq 82
HandleTrDataResp...........81
HandleTrTimeout 83
HandleWriteReq.............78
InitTLayer 67
LLC_CommenceTX.............69
RetireTLabel 63
TimeoutLabel...............64
TimeoutQNotEmpty 66
iLinkTransLayer...........84
ilinkconv.c **87**
DumpMem...................87
GRFNotEmpty 92
ReadBlockPacket...........90
ReadNoPayloadPacket 89
ReadPacket................93
ReadQuadletPacket 88
ReadSelfIDPacket..........91
SendBlockPacket 96
SendNoPayloadPacket........95
SendPacket 98
SendQuadletPacket.........97
initts.c **101**
SetTimer.................101
kernel.c **103**
AddProcess...............107
Block 105
DInt.....................109
EInt 108
FakeInt..................111
GetPID 103
SetInt...................110
UnBlock 106
Yield....................104
startup.c **113**
C_Startup................113
test.c **115**
DoPhilosophers...........124
HackRand 121
LEDOff...................120
LEDOn 118
LEDToggle................119
MsgPassTest 117
Ph.......................123
RunTestSuite 126
TestILink................115
TimerTest 125
WaitRand.................122
timer.c **127**
AddAlarm.................133
AddTime 130
CompareTime..............129
DelAlarm 135
GetCycTime...............132
SubTime 131
TimerInit................127
TimerLoop 136
AT91AIC.h **139**
AT91PIO.h **141**
AT91REVAAIC.h **143**
AT91SF.h **145**
AT91timer.h **147**
ORBstructs.h **149**
TSBLLC.h **151**
errors.h **153**
intIDS.h **155**
miscinc.h **157**
msgtypes.h **161**

```
1       /*******************************************************************
2        * CSRmain.c                                                       *
3        *-----------------------------------------------------------------*
4        *                                                                 *
5        * CSR registers and config ROM handling for 1394                  *
6        *                                                                 *
7        *  December 1999, Ray Heasman                                     *
8        *                                                                 *
9        *******************************************************************/

11      #include "miscinc.h"
12      #include "msgtypes.h"
13      #include <stdio.h>
14      #include "ORBstructs.h"

16      QUAD CSRROMMem[32];     // Make longer as you need it

18      // This process supports reads and writes to the CSR manager and CSR
19      // ROM spaces.

21      void CSRRead(TBMsgHandle *CSRMsg)
22      // Handle a read access to the CSR memory area.
23      // This is not for ROM accesses. Only addresses < (CSRBASEADDR+0x400) are allowed
24  1   {
25  1     TTransReq *CSRReq;

27  1     CSRReq = (TTransReq *) CSRMsg->MsgPtr;
28  1     DebugMsg("CSRRead called");

30  1   // Check the address, and decide what to do...

32  1   //  switch (CsrReq->DestLo & 0x3FF) {
33  1   //    case  :  //
34  1   //              break;
35  1   //    default   :
36  1   //                DebugMsg("(
                                 CSRmain.c) CSRRead: Unhandled CSR read address in transaction.\n");
37  1   //              break;
38  1   //  }

40  1     SendBMsg(&CSRMsg, DEALLOCBMSG);

42      }
```



```
44      void CSRROMWriteOrLock(TBMsgHandle *CSRMsg)
45      // Handle read access to the CSR ROM memory area.
46  1   {
47  1     DebugMsg("(CSRmain.c) CSRWrite: Attempt by external device to modify CSR ROM.\n");
48  1     // FIXME - I suppose I should send some sort of failure notification, rather than letting
49  1     // the other node time out.
50  1     SendBMsg(&CSRMsg, DEALLOCBMSG);
51      }
```

```
53      void CSRROMRead(TBMsgHandle *CSRMsg)
54      // Handle a read access to a CSR ROM memory area
55      // Dont forget to make sure you have defined CSRROMMem large enough for your read/write
56  1   {
57  1     TTransReq *CSRReq;
58  1     QUAD Addr, TmpType, Len;
59  1     int i;

61  1     CSRReq = (TTransReq *) CSRMsg->MsgPtr;

63  1     DebugMsg("CSRROMRead called.\n");

65  1     Addr = (CSRReq->DestLo - CSRROMBASEADDRLo) >> 2;
                                                    // Convert to an offset in the CSRMem array

67  1     TmpType = CSRReq->TrType & 0xFC00;
                                            // Careful not to damage our copy of the Trans Label.
68  1     CSRReq->TrType = TmpType | 8;                     // Read response

70  1     Len = CSRReq->Length;                             // Record length

72  1   // IMPORTANT NOTE
73  1   // Packets to CSR space arrive with their future destination node address copied into the
                                                                                          correct
74  1   // fields from the source identifier. This is because:
75  1   // a) I couldnt find space in the structure anyway.
76  1   // b) You dont need it. If you got it, you were the destination, weren't you?
77  1   // Summary: You dont have to rewrite DestID, DestHi, DestLo, or length.

79  2     if (Len <= 4) {
80  2       CSRReq->Payload[0] = CSRROMMem[Addr];
81  2     } else {
82  2       Len = (Len+3) >> 2;
83  3       for (i=0; i<Len; i++) {
84  3         CSRReq->Payload[i+1] = CSRROMMem[Addr+i];
85  2       };
86  1     };

88  1     // FIXME: I should zero pad the packet here, if it isnt a multiple of four.

90  1     SendBMsg(&CSRMsg, TR_DATARESP);
91      }
```



```
93      void CSRLoop()
94  1   {
95  1     TBMsgHandle *CSRReqMsg;
96  1     TTransReq *CSRReq;

98  1     RegisterType(CSRROMBASEID, GetPID());

100 2     while (1) {
101 2       // Wait for a request (or error, or response) and handle it
102 2       WaitBMsg(&CSRReqMsg, CSRROMBASEID);
103 2       CSRReq = (TTransReq *) CSRReqMsg->MsgPtr;
104 3       switch ((CSRReq->TrType) & 0xFF) {
105 3         case 0 : // A node wishes to read from the CSR ROM space
106 3                  CSRROMRead(CSRReqMsg);
107 3                break;
108 3         case 1 : // A node wishes to write to the CSR ROM space
109 3         case 2 : // or a node wishes to do a lock transaction. Duh, it's _ROM_.
110 3                  CSRROMWriteOrLock(CSRReqMsg);
111 3                break;
112 3         default: // Handle errors by dumping them
113 4                  if ((CSRReq->TrType & 0xFF) >= 32) {
114 4                    SendBMsg(&CSRReqMsg, DEALLOCBMSG);
115 4                  } else {
116 4                  // Moan if I get responses (Should never happen)
117 4                    DebugMsg("(CSRmain.c) CSRLoop: Received a transaction response. Eh? Wot?\n");
118 3                  };
119 3                break;
120 2       }
121 1     }
122     }
```

```
124     void CSRInit(void)
125  1  {
126  1    CSRROMMem[12] = 0x00123456;
                                   // FIXME: FAKE Node vender ID and chip ID. GET A REAL ONE.
127  1    CSRROMMem[13] = 0x00123456;
128  1    AddProcess((unsigned int) &CSRLoop, 0, 1024);
129     }
```

```
 1      /*******************************************************************
 2       * InitAIC.c                                                       *
 3       *-----------------------------------------------------------------*
 4       *                                                                 *
 5       *  This file contains all the necessary routines to set up the    *
 6       *  AIC registers prior to task switching, on an AT91M40400.       *
 7       *                                                                 *
 8       *  October 1998, Ray Heasman                                      *
 9       *                                                                 *
10       *******************************************************************/

12      #include "AT91AIC.h"
13      #include "AT91SF.h"

15      extern unsigned int SpurInt;

17      void InitAIC(void)
18      /* Clear AIC for taskswitching */

20  1   {
21  1     int *Base;
22  1     Base = (int *) AICBase;

24  1     /* First we disable any new interrupts */
25  1     Base[AIC_IDCR]=-1;
26  1     Base[AIC_ICCR]=-1;

28  1     /* Now we furiously clear any pending interrupts until the AIC shuts up */
29  1   //  do {
30  1   //    Base[AIC_EOICR]= Base[AIC_IVR];
31  1   //  } while (Base[AIC_IVR]);
32  2     do {
33  2       Base[AIC_EOICR] = 1;
34  1     } while (Base[AIC_ISR]);

36  1     // set up spurious interrupt vector

38  1     Base[AIC_SIV]= (int) &SpurInt;

40  1     // Put the chip into Protected interrupt mode.

42  1     Base = (int *) SFBase;
43  1     Base[SF_PMR] = 0x27A80020;

45      }
```

```
1      /*******************************************************************
2       * InitPIO.c                                                       *
3       *-----------------------------------------------------------------*
4       *                                                                 *
5       *  This file contains all the necessary routines to set up the    *
6       *  PIO registers prior to task switching, on an AT91M40400.       *
7       *                                                                 *
8       *  October 1998, Ray Heasman                                      *
9       *  Please excuse my accent. I am by habit a Pascal programmer     *
10      *                                                                 *
11      *******************************************************************/

13     #include "AT91PIO.h"

15     void InitPIO(void)
16     /* Clear PIO registers, setting all IOs (exept Chipselects 2,3,4,IRQ0)
17        as being controlled by the PIO, and also set all as inputs,
18        except for the LEDs.
19     */

21  1  {
22  1    int *Base;
23  1    Base = (int *) PIOBase;
24  1    Base[PIO_ODR]= 0x73ffffff;    /* Disable output on pins */
25  1    Base[PIO_PER]= 0x73fffdff;    /* Enable PIO control of those pins */
26  1    Base[PIO_PDR]= 0x8C000200;
27  1  // Use 0x73fffdff for IRQ0 -> AIC
28  1  // Use 0x73ffffff for IRQ0 thru PIO
29  1    Base[PIO_OER]= 0x72080000;    /* Enable output on the LEDs */
30  1    Base[PIO_CODR]=0x00080000;    /* Switch off BUSCONTENDER */
31     }
```

```
/*********************************************************************
 * msgpass.c                                                         *
 *-------------------------------------------------------------------*
 *                                                                   *
 *  This file contains all the necessary routines to implement       *
 *  message passing between processes.                               *
 *                                                                   *
 *  July 1998, Ray Heasman                                           *
 *                                                                   *
 *********************************************************************/

#include "errors.h"
#include "miscinc.h"
#include "msgtypes.h"
#include <stdio.h>

// #define MSGDEBUG

/* Now we define the structures to be used in the message handling routines
   See miscinc.h for the definition of TSmallMsg and TBigMsg              */

/* A Message ID holder - used to store the position of the head of a linked
   list of messages of a particular ID */

typedef struct {
  unsigned int ID;
  int PID;      /* Process ID of task associated with a particular Msg */
  int LLHead;
} TIDHead;

/* Each valid message is pointed to by a node in a linked list. The list is
   pointed to by a message holder. The idea is to have a linked list for
   each message type, hopefully speeding up the search for a new message */

/* All the nodes are stored in an array, allowing me to check limits in
   a high-level fashion, and keeping everything confined to one section
   of memory */

typedef struct {
  int Last;
  int Offset;
  int PID;
  int MyHead;
  int Next;
} TLLNode;

/* The instantiations of the structures to be used */

/* The messages are all stored in contiguous arrays, hopefully making
   bounds checking easy (and high-level) */

TBigMsg   BMsgList[NUM_BIGMSG];
TSmallMsg SMsgList[NUM_SMALLMSG];

/* The message ID headers are stored in their own array */

TIDHead BIDHead[MAXNUM_BID];
TIDHead SIDHead[MAXNUM_SID];

/* The message handles are stored in their own array */

TSMsgHandle SMsgHandle[NUM_SMALLMSG];
TBMsgHandle BMsgHandle[NUM_BIGMSG];

/* The message ID headers need a free slot pointer */
int BFreeIDSlot;
int SFreeIDSlot;

/* The each node in the linked list is just an array element */
```



```
TLLNode SLLNode[NUM_SMALLMSG];
TLLNode BLLNode[NUM_BIGMSG];

/* The next free slot in the array of message ID heads. I am using a plain
   array for this, making Deregister operations expensive (I have to shift
   elements up to fill an empty slot), but that's ok, because Deregister
   should be used infrequently (Hmm. I shouldn't say things like that) */

int BFreeIDSlot;
int SFreeIDSlot;

/* This is the pointer to the linked list of unused nodes
   ie. It points to elements in S/BLLNode
*/

int BHeadFreeLLN;
int SHeadFreeLLN;

/* We also need a pointer to a linked list of used nodes.
   These are nodes that are in use by a process and are
   thus not in any queues
*/

int BHeadUsedLLN;
int SHeadUsedLLN;

/* Code starts here */

#define EIRet(x) { SetInt(LastInt); return(x);}

void InitMessageSystem(void)
/* Initialise the whole message system. All of the message subsystem state
   is reset */
{
  int i;
  int LastInt;


  LastInt = DInt();
  BFreeIDSlot=0;                /* 0 is the first valid slot for a MsgID */
  SFreeIDSlot=0;
  BHeadFreeLLN=0;               /* 0 is the offset of the first node in */
  SHeadFreeLLN=0;               /* the free node list                   */
  BHeadUsedLLN=-1;              /* No used nodes */
  SHeadUsedLLN=-1;

  /* Now initialise the linked lists. All entries are empty, and are linked
     into the "Free Nodes" linked list */
  for (i=0; i<NUM_BIGMSG; i++) {
    BLLNode[i].Last=i-1;        /* -1 is a "NULL" pointer */
    BLLNode[i].Offset=i;        /* Each node points to a message */
    BLLNode[i].PID = -1;
    BLLNode[i].Next=i+1;
  };
  BLLNode[NUM_BIGMSG-1].Next=-1;        /* Ensure last element leads */
                                        /* to nowhere */
  for (i=0; i<NUM_SMALLMSG; i++) {
    SLLNode[i].Last=i-1;
    SLLNode[i].Offset=i;
    SLLNode[i].PID = -1;
    SLLNode[i].Next=i+1;
  };
  SLLNode[NUM_SMALLMSG-1].Next=-1;

  /* Clear all handles */
  for (i=0; i<NUM_SMALLMSG; i++) {
    SMsgHandle[i].Node = i;
  };
  EInt();
  #ifdef MSGDEBUG
```

```
141 1      DebugMsg("msgpass.c: InitMessageSystem completed.\n");
142 1    #endif
143    }
```



```
145     int RegisterType(unsigned int MsgID, unsigned int PID)
146     /* Marks PID as being interested in a particular datatype */
147 1   {
148 1     int chkpos,found;
149 1     int LastInt;

151 1     #ifdef MSGDEBUG
152 1       char DBuf[60];
153 1     #endif

155 1     LastInt = DInt(); /*Disable Interrupts to ensure no clashes */
156 1     /* Check for available space and set pointers to the correct array */
157 2     if (MsgID & SMALLMSGMASK) {
158 2       if (SFreeIDSlot >= MAXNUM_SID) EIRet(ERROR_NoFreeSmallIDSlots);

160 2       chkpos=found=0;
161 2       while ((chkpos < MAXNUM_SID) && (!found))
162 2         found = (SIDHead[chkpos++].ID == MsgID);
163 2       if (found) EIRet(ERROR_MsgTypeAlreadyRegistered);

165 2       SIDHead[SFreeIDSlot].ID=MsgID;
166 2       SIDHead[SFreeIDSlot].PID=PID;
167 2       SIDHead[SFreeIDSlot++].LLHead=-1;
168 2     } else {
169 2       if (BFreeIDSlot >= MAXNUM_BID) EIRet(ERROR_NoFreeBigIDSlots);

171 2       chkpos=found=0;
172 2       while ((chkpos < MAXNUM_BID) && (!found))
173 2         found = (BIDHead[chkpos++].ID == MsgID);
174 2       if (found) EIRet(ERROR_MsgTypeAlreadyRegistered);

176 2       BIDHead[BFreeIDSlot].ID=MsgID;
177 2       BIDHead[BFreeIDSlot].PID=PID;
178 2       BIDHead[BFreeIDSlot++].LLHead=-1;
179 1     };
180 1     #ifdef MSGDEBUG
181 1       sprintf(DBuf,"Registered message %d for PID %d.\n", MsgID, PID);
182 1       DebugMsg(DBuf);
183 1     #endif
184 1     EIRet(0);     /* 0 == success */
185     }
```

```
187     int SendSMsg(TSMsgHandle **MsgH, unsigned int MsgID)
188     /* Send a SmallMsg, of type MsgID */

190 1   {
191 1     int Slot, found;
192 1     int LastInt;

194 1     TSmallMsg *Msg;

196 1     LastInt = DInt();
197 1     Slot=found=0;

199 1     Msg = (*MsgH)->MsgPtr;

201 1     if (!(MsgID & SMALLMSGMASK)) { EIRet(ERROR_NotSmallMsg); }

203 1     /* Check the node value passed to us */
204 2     if (((*MsgH)->Node > NUM_SMALLMSG) || ((*MsgH)->Node < 0)) {
205 2       EIRet(ERROR_InvalidNode);
206 1     }
207 1     /* Do some consistency checks on the passed pointer */
208 2     if (( (int) Msg< (int) &SMsgList[0] ) || ((int) Msg> (int) &SMsgList[NUM_SMALLMSG-1])) {
209 2       EIRet(ERROR_MsgNotInCorrectMemArea);
210 1     }
211 1     /* Make a nasty assumption about memory being in the bottom
212 1        Gig of address space */
213 2     if (((int) Msg - (int) &SMsgList[0])%sizeof(TSmallMsg)) {
214 2       EIRet(ERROR_MsgAddressWrong);
215 1     }

217 1     /* Oh well, that will have to do. Nastier errors will go
218 1        undetected */
219 2     if (MsgID == DEALLOCSMSG) {
220 2       Slot = (*MsgH)->Node;
221 2       SLLNode[SHeadFreeLLN].Last = Slot;
222 2       SLLNode[Slot].Next = SHeadFreeLLN;
223 2       SLLNode[Slot].Last = -1;
224 2       BLLNode[Slot].PID = -1;
225 2       SHeadFreeLLN = Slot;
226 2       EIRet(0);
227 1     };

229 2     while ((Slot < MAXNUM_SID) && (!found)) {
230 2       found = (SIDHead[Slot++].ID==MsgID);
231 1     }
232 1     Slot--;
233 1     if (!found) EIRet(ERROR_CatchallNotImplementedYet);

235 1     SLLNode[(*MsgH)->Node].MyHead = Slot;

237 1     /* Now we change Slot from a ID slot pointer to a pointer to node */

239 2     if (SIDHead[Slot].LLHead == -1) {
240 2     /* Handle special case of empty queue */
241 2       SIDHead[Slot].LLHead = (*MsgH)->Node;
242 2       UnBlock(SIDHead[Slot].PID);         /* Tell task switcher to enable this task */
243 2       Slot = SIDHead[Slot].LLHead;
244 2       SLLNode[Slot].Next = -1;
245 2       SLLNode[Slot].Last = -1;
246 2       EIRet(0);
247 1     };

249 1     UnBlock(SIDHead[Slot].PID);           /* Tell task switcher to enable this task */
250 1     Slot = SIDHead[Slot].LLHead;

252 1     while (SLLNode[Slot].Next != -1)
253 1       Slot = SLLNode[Slot].Next;

255 2     if ((*MsgH)->Node == Slot) {
```



```
256 2       DebugMsg(
              "SendSMsg: FATAL ERROR - Attempt to send another process' message! System unstable.\n");
257 2       EIRet(ERROR_NotYourMessage);
258 1     };

260 1     SLLNode[Slot].Next=(*MsgH)->Node;
261 1     SLLNode[(*MsgH)->Node].Last=Slot;
262 1     EIRet(0); /* Everything Peachy */
263     }
```

```
265     int SendBMsg(TBMsgHandle **MsgH, unsigned int MsgID)
266     /* Send a BigMsg, of type MsgID */

268  1  {
269  1    int Slot, found;
270  1    int LastInt;

272  1    TBigMsg *Msg;
273  1    #ifdef MSGDEBUG
274  1      char DBuf[60];
275  1      sprintf(DBuf, "SendBMsg for %d by %d.\n",MsgID, GetPID());
276  1      DebugMsg(DBuf);
277  1    #endif

279  1    LastInt = DInt();
280  1    Slot=found=0;

282  1    Msg = (*MsgH)->MsgPtr;

284  1    if ((MsgID & SMALLMSGMASK)) { EIRet(ERROR_NotBigMsg); }

286  1    /* Check the node value passed to us */
287  2    if (((*MsgH)->Node > NUM_BIGMSG) || ((*MsgH)->Node < 0)) {
288  2      EIRet(ERROR_InvalidNode);
289  1    }
290  1    /* Do some consistency checks on the passed pointer */
291  2    if (( (int) Msg< (int) &BMsgList[0] ) || ((int) Msg> (int) &BMsgList[NUM_BIGMSG-1])) {
292  2      EIRet(ERROR_MsgNotInCorrectMemArea);
293  1    }
294  1    /* Make a nasty assumption about memory being in the bottom
295  1       Gig of address space */
296  2    if (((int) Msg - (int) &BMsgList[0])%sizeof(TBigMsg)) {
297  2      EIRet(ERROR_MsgAddressWrong);
298  1    }

300  1    /* Oh well, that will have to do. Nastier errors will go
301  1       undetected */

303  2    if (MsgID == DEALLOCBMSG) {
304  2  //    DebugMsg("One big msg deallocated.\n");
305  2      Slot = (*MsgH)->Node;
306  2      BLLNode[BHeadFreeLLN].Last = Slot;

308  2      BLLNode[Slot].Next = BHeadFreeLLN;
309  2      BLLNode[Slot].Last = -1;
310  2      BHeadFreeLLN = Slot;
311  2      EIRet(0);
312  1    };

314  2    while ((Slot < MAXNUM_BID) && (!found)) {
315  2      found = (BIDHead[Slot++].ID==MsgID);
316  1    }
317  1    Slot--;
318  1    if (!found) EIRet(ERROR_CatchallNotImplementedYet);

320  1    BLLNode[(*MsgH)->Node].MyHead = Slot;

322  1    /* Now we change Slot from a ID slot pointer to a pointer to node */

324  2    if (BIDHead[Slot].LLHead == -1) {
325  2    /* Handle special case of empty queue */
326  2      BIDHead[Slot].LLHead = (*MsgH)->Node;
327  2      UnBlock(BIDHead[Slot].PID);         /* Tell task switcher to enable this task */
328  2      Slot = BIDHead[Slot].LLHead;
329  2      BLLNode[Slot].Next = -1;
330  2      BLLNode[Slot].Last = -1;
331  2      EIRet(0);
332  1    };
```



```
334  1    UnBlock(BIDHead[Slot].PID);   /* Tell task switcher to enable this task */
335  1    Slot = BIDHead[Slot].LLHead;

337  1    while (BLLNode[Slot].Next != -1)
338  1      Slot = BLLNode[Slot].Next;

340  2    if ((*MsgH)->Node == Slot) {
341  2      DebugMsg(
          "SendBMsg: FATAL ERROR - Attempt to send another process' message! System unstable.\n");
342  2      EIRet(ERROR_NotYourMessage);
343  1    };
344  1    BLLNode[Slot].Next=(*MsgH)->Node;
345  1    BLLNode[(*MsgH)->Node].Last=Slot;
346  1    EIRet(0); /* Everything Peachy */
347     }
```

```
349     int WaitSMsg(TSMsgHandle **MsgH, unsigned int MsgID)
350     /* Wait for a message of type MsgID. A pointer to the data is returned in Msg.
351        Returns 0 if successful */

353 1   {
354 1     int Slot,found;
355 1     int Node;
356 1     int LastInt;

358 1     LastInt = DInt();

360 2     if (MsgID & SMALLMSGMASK) {
361 3       if (MsgID == ALLOCSMSG) {
362 3         /* Create an empty message for the user */
363 3         if (SHeadFreeLLN == -1) EIRet(ERROR_NoFreeSLLNode);

365 3         /* First, get a free node */
366 3         Slot=SHeadFreeLLN;
367 3         SHeadFreeLLN = SLLNode[SHeadFreeLLN].Next;
368 3         SLLNode[Slot].Last = -1;
369 3         SLLNode[Slot].PID = GetPID();
370 3         SLLNode[Slot].Next = -1;
371 3         SLLNode[SHeadFreeLLN].Last=-1;

373 3         /* Nodes correspond 1:1 with Messages and Handles, so */
374 3         *MsgH = &SMsgHandle[Slot];
375 3         (*MsgH)->MsgPtr = &SMsgList[SLLNode[Slot].Offset];
376 3         (*MsgH)->Node = Slot;
377 3         (*MsgH)->UserID = 0;
378 3         (*MsgH)->PassByRef = 0;
379 3         EIRet(0);
380 3       } else {
381 3         Slot = found = 0;
382 3         /* Fixme MAXNUM_SID is too big (Correct but inefficient) */
383 4         while ((!found) && (Slot < MAXNUM_SID)) {
384 4           found = (SIDHead[Slot++].ID==MsgID);
385 3         }
386 3         Slot--;
387 4         if (!found) {
388 4           EIRet(ERROR_MsgNotRegistered);
389 3         };
390 4         while (SIDHead[Slot].LLHead==-1) {
391 4           /* If there isn't a message, block and there should be one
392 4              when we wake up */
393 4           Block();
394 3         };

396 3         /* If we get here then there's a message in the message queue */
397 3         /* so we unlink the item out of the queue */

399 3         Node = SIDHead[Slot].LLHead;

401 3         SIDHead[Slot].LLHead = SLLNode[Node].Next;
402 4         if (SLLNode[Node].Next != -1) {
403 4           SLLNode[SLLNode[Node].Next].Last = -1;
404 3         };
405 3         SLLNode[Node].Last=-1;
406 3         SLLNode[Node].PID=GetPID();
407 3         SLLNode[Node].Next=-1;
408 3         *MsgH = &SMsgHandle[Node];
409 3         (*MsgH)->MsgPtr = &SMsgList[SLLNode[Node].Offset];
410 3         (*MsgH)->Node = Node;
411 3         EIRet(0);               /* Return success */
412 2       };
413 2       EIRet(ERROR_UnreachableCodeReached);
414 1     };

416 1     EIRet(ERROR_NotSmallMsg);
417     }
```



```
419     int CheckSMsg(unsigned int MsgID)
420     /* Check for a message of type MsgID. Returns 0 if a message exists.
421        However, it doesn't block like WaitSMsg does. If you do use it, remember
422        that busy waiting is evil.
423     */

425 1   {
426 1     int Slot,found;
427 1     int LastInt;

429 1     LastInt = DInt();

431 2     if (MsgID & SMALLMSGMASK) {
432 2       Slot = found = 0;
433 2       /* Fixme MAXNUM_SID is too big (Correct but inefficient) */
434 3       while ((!found) && (Slot < MAXNUM_SID)) {
435 3         found = (SIDHead[Slot++].ID==MsgID);
436 2       }
437 2       Slot--;
438 3       if (!found) {
439 3         EIRet(ERROR_MsgNotRegistered);
440 2       };
441 3       while (SIDHead[Slot].LLHead==-1) {
442 3         EIRet(ERROR_MsgNotHereYet);
443 2       };

445 2       /* If we get here then there's a message in the message queue */
446 2       EIRet(0);                   /* Return success */
447 1     };
448 1     EIRet(ERROR_NotSmallMsg);
449     }
```

```
451     void BreakPt(void)
452  1  {
453     }
```



```
455     int CheckBMsg(unsigned int MsgID)
456     /* Check for a message of type MsgID. Returns 0 if a message exists.
457        However, it doesn't block like WaitBMsg does. If you do use it, remember
458        that busy waiting is evil.
459     */

461  1  {
462  1    int Slot,found;
463  1    int LastInt;

465  1    LastInt = DInt();

467  2    if (!(MsgID & SMALLMSGMASK)) {
468  2      Slot = found = 0;
469  2      /* Fixme MAXNUM_BID is too big (Correct but inefficient) */
470  3      while ((!found) && (Slot < MAXNUM_BID)) {
471  3        found = (BIDHead[Slot++].ID==MsgID);
472  2      }
473  2      Slot--;
474  3      if (!found) {
475  3        EIRet(ERROR_MsgNotRegistered);
476  2      };
477  3      while (BIDHead[Slot].LLHead==-1) {
478  3        EIRet(ERROR_MsgNotHereYet);
479  2      };

481  2      BreakPt();
482  2      /* If we get here then there's a message in the message queue */
483  2      EIRet(0);                   /* Return success */
484  1    };
485  1    EIRet(ERROR_NotBigMsg);
486     }
```

```
488      unsigned int WaitOnMsgList(unsigned int *MList)
489      /* Accept zero terminated list of (max 30) MsgIDs. Block until one of them becomes
490         valid. Scan from first to last and exit early if one becomes valid.
491         Returns offset of the MsgID of the received message in the list.
492         This does not lock out Task Switching whilst accessing the list of MsgIDs,
493         so if you share them between threads, be careful!
494         NB: This also means that it is possible for WaitOnMsgList to return an item
495             half the way down the list even though that msg arrived after another.
496      */
497  1   {
498  1     unsigned int Mask;
499  1     QUAD *StartOfList;
500  1     int Cnt;
501  1     int looking;

503  1     StartOfList = MList;
504  2     while (1) {       /* This should never run more than twice. */
505  2       Cnt = 0; Mask = 1; looking = 1; MList = StartOfList;
506  3       while (looking && (Cnt<30) && (*MList)) {
507  3         if (*MList & SMALLMSGMASK)
508  3           looking = CheckSMsg(*MList);
509  3         else
510  3           looking = CheckBMsg(*MList);
511  3         if (looking) Cnt++;
512  3         MList++;
513  2       };
514  2       if (!looking) return(Cnt);
515  2       Block();        /* Block if we didnt find anything the first time */
516  1     }
517  1     return(0);
518      }
```



```
520      int WaitBMsg(TBMsgHandle **MsgH, unsigned int MsgID)
521      /* Wait for a message of type MsgID. A pointer to the data is returned in Msg. */
522  1   {
523  1     int Slot,found;
524  1     int Node;
525  1     int LastInt;
526  1     #ifdef MSGDEBUG
527  1       char DBuf[60];
528  1       sprintf(DBuf, "WaitBMsg for %d by %d.\n",MsgID, GetPID());
529  1       DebugMsg(DBuf);
530  1     #endif
531  1     LastInt = DInt();

533  2     if (!(MsgID & SMALLMSGMASK)) {
534  3       if (MsgID == ALLOCBMSG) {
535  3   //      DebugMsg("One big msg allocated.\n");
536  3         /* Create an empty message for the user */
537  3         if (BHeadFreeLLN == -1) EIRet(ERROR_NoFreeBLLNode);

539  3         /* First, get a free node */
540  3         Slot=BHeadFreeLLN;
541  3         BHeadFreeLLN = BLLNode[BHeadFreeLLN].Next;
542  3         BLLNode[Slot].Last = -1;
543  3         BLLNode[Slot].PID = GetPID();
544  3         BLLNode[Slot].Next = -1;
545  3         BLLNode[BHeadFreeLLN].Last=-1;

547  3         /* Nodes correspond 1:1 with Messages and Handles, so */
548  3         *MsgH = &BMsgHandle[Slot];
549  3         (*MsgH)->MsgPtr = &BMsgList[BLLNode[Slot].Offset];
550  3         (*MsgH)->Node = Slot;
551  3         (*MsgH)->UserID = 0;
552  3         (*MsgH)->PassByRef = 0;
553  3         EIRet(0);
554  3       } else {
555  3         Slot = found = 0;
556  3         /* Fixme MAXNUM_BID is too big (Correct but inefficient) */
557  4         while ((!found) && (Slot < MAXNUM_BID)) {
558  4           found = (BIDHead[Slot++].ID==MsgID);
559  3         }
560  3         Slot--;
561  4         if (!found) {
562  4           EIRet(ERROR_MsgNotRegistered);
563  3         };
564  4         while (BIDHead[Slot].LLHead==-1) {
565  4           /* If there isn't a message, block and there should be one
566  4              when we wake up */
567  4           Block();
568  3         };

570  3         /* If we get here then there's a message in the message queue */
571  3         /* so we unlink the item out of the queue */

573  3         Node = BIDHead[Slot].LLHead;
574  3         BIDHead[Slot].LLHead = BLLNode[Node].Next;
575  4         if (BLLNode[Node].Next != -1) {
576  4           BLLNode[BLLNode[Node].Next].Last = -1;
577  3         };

579  3         BLLNode[Node].Last=-1;
580  3         BLLNode[Node].PID=GetPID();
581  3         BLLNode[Node].Next=-1;
582  3         *MsgH = &BMsgHandle[Node];
583  3         (*MsgH)->MsgPtr = &BMsgList[BLLNode[Node].Offset];
584  3         (*MsgH)->Node = Node;
585  3         EIRet(0);                /* Return success */
586  2       };
587  2       EIRet(ERROR_UnreachableCodeReached);
588  1     };
```

```
590 1   EIRet(ERROR_NotBigMsg);
591   }
```

```
/*******************************************************************
 * SPB2.c                                                          *
 *-----------------------------------------------------------------*
 *                                                                 *
 * Serial Bus Protocol - 2 driver for 1394                         *
 *                                                                 *
 *  October 1999, Ray Heasman                                      *
 *                                                                 *
 *******************************************************************/

#include "miscinc.h"
#include "msgtypes.h"
#include "intIDs.h"
#include <stdio.h>
#include "ORBstructs.h"

extern QUAD MyNodeID;
extern QUAD NodeCount;

// Look in ORBstructs.h for definitions of all the
// ORB types I use, such as LoginManOrb.

QUAD SBPMem[1024];      // 4k of RAM from FFFF F001 0000 to FFFF F001 1000

void HDDLogin(int Node)
// Login to the HDD
{
  TTransReq *TrReq;
  TBMsgHandle *MyBMsg;
  LoginManORB *LoginOrb;

  // We have to tell the Management agent on the HDD where to find our Login ORB

  WaitBMsg(&MyBMsg, ALLOCBMSG);
  MyBMsg->UserID = HDDREPLY;                    // So transaction layer knows who to reply to.
  TrReq = (TTransReq *) MyBMsg->MsgPtr;

// First set up the Login ORB

  LoginOrb = (LoginManORB *) &(SBPMem[0]);
                                    // Make ourselves the correct structure at the right place
  LoginOrb->Pass.L1 = 0xFFFF;
  LoginOrb->Pass.L2 = 0xF0010020;
  LoginOrb->LoginResp.L1 = 0xFFFF;
  LoginOrb->LoginResp.L2 = 0xF0010000;
  LoginOrb->LUNDat = 0x80000000;
  LoginOrb->Lengths = 0x001C0020;
                                        // 0 len password, so use immediate data. 32 len reply
  LoginOrb->StatusFIFO.L1 = 0xFFFF;
  LoginOrb->StatusFIFO.L2 = 0xF0010020;
                                              // Offset 32 is the status block for operations.

  TrReq->DestID = Node | 0xFFC0;
  TrReq->TrType = 1;                            // Write
  TrReq->Length = 8;
  TrReq->DestHi = 0xFFFF;
  TrReq->DestLo = 0xF0030000;                   // 2 Quad write to MANAGEMENT_AGENT register
  TrReq->Payload[1] = 0xFFFF;
  TrReq->Payload[2] = 0xF0010000;
  SendBMsg(&MyBMsg, TR_DATAREQ);
  WaitBMsg(&MyBMsg, HDDREPLY);
  TrReq = (TTransReq *) MyBMsg->MsgPtr;

}
```



```
void DoHDD(void)
// This functionality maybe shouldnt be here,
                               but will live here till I figure out exactly what I need to do.
{
  int Node,found;
  TBMsgHandle *MyBMsg;
  TTransReq *TrReq;

  char OutBuf[80];

  RegisterType(HDDREPLY, GetPID());

  found = 0;
  for (Node=0; (Node < NodeCount) && (!found); Node++) {
    if (Node != (MyNodeID & 0x3F)) {
      sprintf(OutBuf, "Scanning node %d.\n",Node);
      DebugMsg(OutBuf);
      WaitBMsg(&MyBMsg, ALLOCBMSG);
      MyBMsg->UserID = HDDREPLY;
                                                    // So transaction layer knows who to reply to.
      TrReq = (TTransReq *) MyBMsg->MsgPtr;

      TrReq->DestID = Node | 0xFFC0;
      TrReq->TrType = 0;
      TrReq->Length = 4;
      TrReq->DestHi = 0xFFFF;
      TrReq->DestLo = 0xF000040C;
      SendBMsg(&MyBMsg, TR_DATAREQ);
      WaitBMsg(&MyBMsg, HDDREPLY);
      TrReq = (TTransReq *) MyBMsg->MsgPtr;
      if (TrReq->TrType != 256) {
//        sprintf(OutBuf,"%x -> %p\n", f*4, TrReq->Payload[0]);
        if (
                                                      TrReq->Payload[0] == 0xD00101) {
                                  // FIXME: Dubious recognition hack. Redo when I have a clue
          found = Node;
          sprintf(OutBuf, "VST HDD found on node %d   \n",Node);
          DebugMsg(OutBuf);
        };
      } else {
        sprintf(OutBuf,"Timeout on node %d   \n", Node);
        DebugMsg(OutBuf);
      }
      SendBMsg(&MyBMsg, DEALLOCBMSG);
    } else {
      DebugMsg("Skipping Myself.\n");
    }
  }
  if (!found) {
    DebugMsg("Scan done. No VST hard drives found.     \n");
  } else {
    HDDLogin(found);
  }
}
```

```
114       void SBPRead(TBMsgHandle *CSRMsg)
115       // Handle a read access to an SBP memory area
116   1   {
117   1     TTransReq *CSRReq;
118   1     QUAD Addr, TmpType, Len;
119   1     int i;

121   1     CSRReq = (TTransReq *) CSRMsg->MsgPtr;

123   1     DebugMsg("SBPRead called.\n");

125   1     Addr = (CSRReq->DestLo - SBP2ADDRLo) >> 2;
                                              // Convert to an offset in the SBPMem array

127   1     TmpType = CSRReq->TrType & 0xFC00;
                                        // Careful not to damage our copy of the Trans Label.
128   1     CSRReq->TrType = TmpType | 8;                         // Read response

130   1     Len = CSRReq->Length;                                 // Record length

132   1   // IMPORTANT NOTE
133   1   // Packets to CSR space arrive with their future destination node address copied into the
                                                                                              correct
134   1   // fields from the source identifier. This is because:
135   1   // a) I couldnt find space in the structure anyway.
136   1   // b) You dont need it. If you got it, you were the destination, weren't you?
137   1   // Summary: You dont have to rewrite DestID, DestHi, DestLo, or length.

139   2     if (Len <= 4) {
140   2       CSRReq->Payload[0] = SBPMem[Addr];
141   2     } else {
142   2       Len = (Len+3) >> 2;
143   3       for (i=0; i<Len; i++) {
144   3         CSRReq->Payload[i+1] = SBPMem[Addr+i];
145   2       };
146   1     };

148   1     // FIXME: I should zero pad the packet here, if it isnt a multiple of four.

150   1     SendBMsg(&CSRMsg, TR_DATARESP);
151       }
```



```
153       void SBPWrite(TBMsgHandle *CSRMsg)
154       // Handle a read access to an SBP memory area
155   1   {
156   1     DebugMsg("SBPWrite called.\n");
157   1     SendBMsg(&CSRMsg, DEALLOCBMSG);
158       }
```

```
160     void SBPLoop()
161 1   {
162 1     TBMsgHandle *CSRReqMsg;
163 1     TTransReq *CSRReq;

165 1     RegisterType(CSRSBP2BASEID, GetPID());

167 2     while (1) {
168 2       // Wait for a request (or error, or response) and handle it
169 2       WaitBMsg(&CSRReqMsg, CSRSBP2BASEID);
170 2       CSRReq = (TTransReq *) CSRReqMsg->MsgPtr;
171 3       switch ((CSRReq->TrType) & 0xFF) {
172 3         case 0 : // A node wishes to read from the SBP CSR space
173 3                SBPRead(CSRReqMsg);
174 3              break;
175 3         case 1 : // A node wishes to write to the SBP space
176 3                SBPWrite(CSRReqMsg);
177 3              break;
178 3         case 2 : // A node wishes to do a lock transaction
179 3                // SBPLock(CSRReqMsg);
180 3                DebugMsg("(SBP2.c) SBPLoop: Unhandled CSR lock transaction. (
                                                                    To be implemented)\n");
181 3              break;
182 3         default: // Handle errors by dumping them
183 4                if ((CSRReq->TrType & 0xFF) >= 32) {
184 4                  SendBMsg(&CSRReqMsg, DEALLOCBMSG);
185 4                } else {
186 4                // Moan if I get responses (Should never happen)
187 4                  DebugMsg("(SBP2.c) SBPLoop: Received a transaction response. Eh? Wot?\n");
188 3                };
189 3              break;
190 2       }
191 1     }
192     }
```



```
194     void SBPInit(void)
195 1   {
196 1     AddProcess((unsigned int) &SBPLoop, 0, 1024);
197     }
```

```
1      /*******************************************************************
2       * debug.c                                                         *
3       *-----------------------------------------------------------------*
4       *                                                                 *
5       * Send debug messages out the ARM debug channel                   *
6       *                                                                 *
7       * Hopefully, this will be extended into a production line test    *
8       * suite.                                                          *
9       * July 1999                                                       *
10      *                                                                 *
11      *******************************************************************/

13     #include <stdio.h>
14     #include "miscinc.h"

16     #define DEBUGMSGBUFSIZE 8000
17     #define DChanGetCtrl(x)         __asm { MRC p14, 0, x, c0, c0 };
18     #define DChanSendQuad(x)        __asm { MCR p14, 0, x, c1, c0 };
19     #define DChanGetQuad(x)         __asm { MRC p14, 0, x, c1, c0 };

21     extern QUAD Uptime;
22     char DebugMsgBuf[DEBUGMSGBUFSIZE];
23     QUAD incnt, inptr;
24     QUAD outcnt, outptr;

26     void DebugMsg(char *OutputMsg)
27     // Copy message into internal buffer
28     // It does not use message passing, but does have an internal buffer.
29     // Dont worry about buffer overflows at the moment.
30     // Dont send it strings that resolve to longer than 200 bytes.
31 1   {
32 1     int IntStore;
33 1     char TStrBuf[160];
34 1     char *TStr;

36 1     TStr = (char *) TStrBuf;
37 1     sprintf(TStr,"%d: (%d) %s", Uptime, GetPID(), OutputMsg);

39 1     IntStore = DInt();
40 2     while (*TStr) {
41 2       DebugMsgBuf[inptr] = *TStr++;
42 2       inptr = (inptr + 1) % DEBUGMSGBUFSIZE;
43 2       incnt++;
44 1     };
45 1     SetInt(IntStore);
46     }
```



```
48     void DebugHandler(void)
49     // Init and then sit in a tight loop, passing any messages to the
50     // output device
51 1   {
52 1     QUAD *bufptr;
53 1     QUAD tmp;

55 1     inptr = 0;
56 1     outptr = 0;

58 1     DebugMsg("Debug handler initialised.\n");

60 2     while (1) {
61 2       DChanGetCtrl(tmp);
62 2       if (tmp & 1) DChanGetQuad(tmp);
63 3       while (incnt > (outcnt+4)) {
64 3         DChanGetCtrl(tmp);
65 4         if (!(tmp & 2)) {
66 4           bufptr = (QUAD *) &(DebugMsgBuf[outptr]);
67 4           tmp = *bufptr;
68 4           DChanSendQuad(tmp);
69 4           outcnt += 4;
70 4           outptr = (outptr + 4) % DEBUGMSGBUFSIZE;
71 3         };
72 2       };
73 2       Yield();
74 1     };
75     }
```

```
77     void InitDebugHandler(void)
78     // Start the Debug handler process
79  1  {
80  1    AddProcess((unsigned int) &DebugHandler, 0, 1024);
81     }
```

```
1     /*******************************************************************
2      * ilink.c                                                         *
3      *-----------------------------------------------------------------*
4      *                                                                 *
5      * This contains all the routines for the Texas Instruments IEEE1394 *
6      * chipset.                                                        *
7      *                                                                 *
8      *                                                                 *
9      *  October 1998, Ray Heasman                                      *
10     *                                                                 *
11     *******************************************************************/

13    #include "TSBLLC.h"
14    #include "errors.h"
15    #include "AT91AIC.h"
16    #include "AT91PIO.h"
17    #include "miscinc.h"
18    #include "intIDS.h"
19    #include "msgtypes.h"

21    // #define WRITEDEBUG           // Uncomment to get exact transcript of writes to the LLC
22    #define LLCBase 0x600000
23    extern int IntVec1394Asm;
24    extern int IntID;
25    extern QUAD Uptime;

27    int iLinkError;
28    QUAD MyNodeID;
29    QUAD NodeCount;

31    // The above is here temporarily for convenience.

33    /* The following variables are written to in case of error
34       so that debugger watchpoints can be used to make the debugger
35       halt the CPU in the case of a 1394 error. */

37    int NB1394Error=0;   /* For nasty errors */
38    int Arb1394Error=0;  /* Less fatal errors */

40    QUAD LLCArr[0x5C/4];    // Stores last read/write to chipset

42    void InitLLC(void);

44    void WriteLLC(unsigned int Reg, unsigned int Val)
45    // Writes Val to LLC register Reg, and stores the write in an internal array
46  1 {
47  1   volatile unsigned short *tmp;
48  1   unsigned short lo,hi;
49  1   #ifdef WRITEDEBUG
50  1     char DBuf[40];
51  1     sprintf(DBuf, "W - R:%p V:%p\n",Reg,Val);
52  1     DebugMsg(DBuf);
53  1   #endif

55  1   if (Reg <= 0x5C) LLCArr[Reg >> 2] = Val;
56  1   lo = (unsigned short) (Val & 0xFFFF);
57  1   hi = (unsigned short) ((Val >> 16) & 0xFFFF);
58  1   tmp = (unsigned short *) (LLCBase + (Reg*2));
59  1   tmp[0] = hi;
60  1   tmp[2] = lo;
61    }
```



```
63    void WriteLLCMasked(QUAD Reg, QUAD Mask, QUAD Value)
64    // Mask Value with Mask, merge it with the last write and write the result to the LLC
65    // Update the internal array.
66    // 1s in the mask are the bits that will change
67  1 {
68  1   QUAD t;
69  1   t = LLCArr[Reg >> 2] & (Mask ^ -1);
70  1   WriteLLC(Reg, t | (Value & Mask));
71    }
```

```
73      unsigned int ReadLLC(unsigned int Reg)
74  1   {
75  1     volatile unsigned short *tmp;
76  1     unsigned short lo,hi;

78  1     tmp = (unsigned short *) (LLCBase + (Reg*2));
79  1     hi = tmp[0];
80  1     lo = tmp[2];
81  1     return((hi<<16)+lo);
82      }
```



```
84      void WritePhy(unsigned int reg, unsigned short val)
85  1   {
86  1     QUAD tmp;

88  1     tmp = 0x40000000 | (val << 16) | (reg << 24);
89  1     WriteLLC(LLCPhyAcc, tmp);
90      }
```

```
92     int ReadPhy(unsigned int reg)
93  1  {
94  1    WriteLLC(LLCPhyAcc, 0x80000000 | (reg << 24));
95  1    while (!(ReadLLC(LLCInte) & 0x20000000));
96  1    return(ReadLLC(LLCPhyAcc) & 0xFF);
97     }
```



```
99     int Check1394IntfOK(void)
100    /* Check that the memory interface to the TSB12LV31 is working.
101       Because of the word unstacking operations, you can get left with
102       the chipset returning zeros on reads.
103       Returns zero for failure
104    */

106 1  {
107 1    return(ReadLLC(0)==0xCE021394);
108    }
```

```
110      void Init1394Ints(int Vector)
111  1   {
112  1     int *Base;
113  1     int LastInt;

115  1     LastInt = DInt();             // Save interrupt state and disable

117  1     WriteLLC(LLCInte,  0xFFFFFFFF);
118  1     WriteLLC(LLCIntMas,0x0);

120  1     Base = (int *) AICBase;
121  1   // When attempting to get AIC INT0 to work:
122  1   // Debug 00 - All time. 20 - Never. 40 - Never  60 - Never
123  1   // 20 - responds to SW set
124  1   // 40 - no response
125  1   // 60 - responds
126  1   // Conclusion - 20/60 are edge triggered
127  1   //               00 is low level sensitive
128  1     Base[AIC_IDCR]  = 0x10000;
129  1     Base[AIC_SMR16] = 0x26;
130  1     Base[AIC_SVR16] = Vector;
131  1     Base[AIC_ICCR] = 0x10000;              /* Clear int */
132  1     Base[AIC_IECR] = 0x10000;              /* Enable int */

134  1     LEDOff(1);

136  1     WriteLLC(LLCIntMas,0x930b9800);       // Specify what we are interested in
137  1     WriteLLC(LLCInte,0xFFFFFFFF);         // Clear everything

139  1     SetInt(LastInt);              // Restore interrupt state.
140      }
```



```
142      void LLC_PhyReset(void)
143  1   {

145  1   // Tell LLC that next reset it is allowed to try for cycle master
146  1   //  WriteLLC(LLCCont, 0x06000228);              // Look it up. Pg 3-4, LLC datasheet

148  1     DebugMsg("ilink.c: NOTIFICATION - 1394 PHY reset.\n");

150  1     InitLLC();
151  1   //  WriteLLC(0x80,0x006100e0);
152  1   //  WriteLLC(0x8C,0xff9effff);

154  1     WriteLLC(LLCInte,PhRst);
155      }
```

```
157     void LLC_PhyInt(void)
158  1  {
159  1    WriteLLC(LLCInte,PhInt);
160     }
```



```
162     void LLC_PhyRXData(void)
163  1  {
164  1    WriteLLC(LLCInte,PhRRx);
165     }
```

```
167     void LLC_TXReady(void)
168     /* This is called if the LLC is ready to send a packet.
169        Any pending packets will be sent. */
170  1  {
171  1    TBMsgHandle *OutgoingMsg;
172  1    int LastInt;

174  1    // Check if there are any messages to send
175  2    if (CheckBMsg(LK_DATAREQ) == 0) {
176  2      WaitBMsg(&OutgoingMsg, LK_DATAREQ);
177  2  //    DebugMsg("ilink.c: Packet sent at link layer.\n");
178  2      SendPacket((QUAD *) OutgoingMsg->MsgPtr);
179  2      SendBMsg(&OutgoingMsg, DEALLOCBMSG);
180  1    };
181  2    if (CheckBMsg(LK_DATAREQ) != 0) {
182  2      // If we get here, then there are no more packets to be sent
183  2      // so we should stop any further TXReady interrupts from being sent.
184  2      WriteLLCMasked(LLCIntMas, 0x04000000, 0x00000000); // Kill any further interrupts
185  1    }
186  1    WriteLLC(LLCInte,TxRdy);
187     }
```



```
189     void LLC_RXedAsynchPacket(void)
190     /* The LLC has received a packet. The packet has to be read and
191        sent to either the Transaction layer or the Node controller */
192  1  {
193  1    TBMsgHandle *RXHandle;
194  1    QUAD *PacketPtr;
195  1    int Err;

197  1    LEDToggle(2);
198  2    while (GRFNotEmpty()) {
199  2      Err = WaitBMsg(&RXHandle, ALLOCBMSG);
200  3      if (!Err) {
201  3  //      DebugMsg("ilink.c: Async packet received.\n");
202  3        PacketPtr = (QUAD *) RXHandle->MsgPtr;
203  3        ReadPacket(PacketPtr);
204  3        SendBMsg(&RXHandle, LK_DATAIND);
205  3      } else {
206  3        DebugMsg("ilink.c: Could not allocate msg to send packet to Transaction layer.\n");
207  2      }
208  1    }
209  1    WriteLLC(LLCInte,ARxDta);
210     }
```

```
212     void LLC_CSRResetCommand(void)
213     /* Tell the Node Controller that it must init a Reset */
214 1   {
215 1     DebugMsg("ilink.c: NOTIFICATION - 1394 CSR Reset interrupt request received.\n");
216 1     WriteLLC(LLCInte,CmdRst);
217     }
```



```
219     void LLC_AsynchTXStuck(void)
220     /* This is a nasty error condition that implies that
221        the LCC isnt being fed outgoing packets properly. */
222 1   {
223 1     DebugMsg("ilink.c: ERROR - Firewire Asynch TX stuck interrupt generated.\n");
224 1     iLinkError = ATStk;
225 1     WriteLLC(LLCInte,ATStk);
226     }
```

```
228     void LLC_GRFOverflowed(void)
229     /* This is an error implying that the LLC isnt
230        being serviced soon/fast enough. This is not
231        a good thing, but not as bad as AsynchTXStuck. */
232  1  {
233  1  //  ReadPacket(LLCBuf);
234  1    DebugMsg("ilink.c: WARNING - Firewire General Receive FIFO overflowed.\n");
235  1    iLinkError = SntRj;
236  1    WriteLLC(LLCInte,SntRj);
237     }
```



```
239     void LLC_HeaderError(void)
240     /* A packet arrived that might be for us, but the header
241        was chewed up.*/
242  1  {
243  1    DebugMsg("ilink.c: NOTIFICATION - 1394 packet with chewed up header received.\n");
244  1    WriteLLC(LLCInte,HdrEr);
245     }
```

```
247     void LLC_InvalidTCode(void)
248     /* A packet arrived with an invalid transaction code. */
249 1   {
250 1     DebugMsg("ilink.c: NOTIFICATION - 1394 packet with invalid TCode received.\n");
251 1     WriteLLC(LLCInte,TCErr);
252     }
```



```
254     void LLC_CycSecond(void)
255     /* The Cycle timer seconds counter incremented. */
256 1   {
257 1     LEDToggle(1);
258 1     Uptime++;
259 1     WriteLLC(LLCInte,CySec);
260     }
```

```
262     void LLC_MCUWriteErr(void)
263     /* Nasty error condition. The CPU spoke to the LLC in
264        an invalid manner. */
265 1   {
266 1     DebugMsg("ilink.c: ERROR - MCU-LLC write error interrupt generated.\n");
267 1     iLinkError = McWrEr;
268 1     WriteLLC(LLCInte,McWrEr);
269     }
```



```
271     void IntVec1394C(void)
272     /* This gets called by an assembler wrapper function (IntVec1394Asm)
273        that saves all registers, allowing this function to call
274        other C functions
275     */
276 1   {
277 1     // First, figure out why we were called.
278 1     volatile int IBits;
279 1   //  char Buf[20];

281 1     // Set IntID, so message passing knows who we are
282 1     IntID = MAIN1394Int;

284 1   //  if (GRFNotEmpty()) LLC_RXedAsynchPacket();

286 1     LEDToggle(0);

288 1     IBits = ReadLLC(0xC); // get Interrupt bits set

290 1   //  sprintf(Buf, "IBits: %x\n",IBits);
291 1   //  DebugMsg(Buf);
292 1       // Now we vector on the interrupt bits
293 1     // I'll use multiple IF statments
294 1     // It's icky, but easy to read and should compile to nice
295 1     // asm.

297 1     if (IBits & PhRst ) LLC_PhyReset();
298 1     if (IBits & PhInt ) LLC_PhyInt();
299 1     if (IBits & PhRRx ) LLC_PhyRXData();
300 1     if (IBits & ARxDta) LLC_RXedAsynchPacket();
301 1     if (IBits & CmdRst) LLC_CSRResetCommand();
302 1     if (IBits & ATStk ) LLC_AsynchTXStuck();
303 1     if (IBits & SntRj ) LLC_GRFOverflowed();
304 1     if (IBits & HdrEr ) LLC_HeaderError();
305 1     if (IBits & TCErr ) LLC_InvalidTCode();
306 1     if (IBits & CySec ) LLC_CycSecond();
307 1   //  if (IBits & CArbFl) LLC_CycArbFailure();

309 1     if (IBits & McWrEr) LLC_MCUWriteErr();
310 1     if (IBits & TxRdy ) LLC_TXReady();

312 1     // TODO: Everything
313 1   //  WriteLLC(LLCIntMas,0x970b9800);     // Specify what we are interested in
314 1   //  WriteLLC(LLCIntMas,0x930b9800);     // Specify what we are interested in

316 1     WriteLLC(LLCInte,IBits);

318 1   //  WriteLLC(LLCInte,0x80000000);

320 1     IntID = 0;
321     }
```

```
323     void Wakeup1394(void)
324  1  {
325  1    int *Base;

327  1    Base = (int *) PIOBase;
328  1    Base[PIO_OER]  = 0x20000;
329  1    Base[PIO_CODR] = 0x20000;
330     }
```



```
332     void InitLLC(void)
333  1  {
334  1    unsigned int tmp;

336  1    WriteLLC(LLCCont, 0x40300200);
337  1    WriteLLC(LLCIntMas, 0);               // Disable LLC interrupts for a lil while
338  1    WriteLLC(LLCInte,0xFFFFFFFF);
339  1    WriteLLC(LLCGRFSta,0x1019);           // Empty GRF register
340  1    WriteLLC(LLCATFSta,0x1019);           // Empty ATF register
341  1    WriteLLC(LLCIsoNum,0x00000000);
342  1    WriteLLC(LLCInte,0xFFFFFFFF);

344  1  //  WriteLLC(0x80,0x006100e0);
345  1  //  WriteLLC(0x8C,0xff9effff);

347  1    // Now do bus reset

349  2    do {
350  2      WritePhy(1,0x7f);
351  3      if (ReadLLC(LLCGRFSta) & 0x10000) {
352  3        ReadLLC(0);
353  3        while (ReadLLC(LLCGRFSta) & 0x10000) ReadLLC(0); // Wait for Reset to complete
354  2      };
355  2  // Now Collect SelfID packets
356  3      while (!(ReadLLC(LLCGRFSta) & 0x10000)) {
357  3        ReadLLC(0xC0);
358  2      };
359  1    } while ((ReadLLC(LLCBusRes) & 0x80000000)==0);

361  1  // Now InitPhy

363  1  //  tmp = ReadPhy(0) >> 2;
364  1  //  tmp = (0x3FF << 6) | tmp;
365  1  //  WriteLLC(LLCBusRes, tmp);

367  1    tmp = ReadPhy(0);

369  2    if (tmp & 0x2) {
370  2      WriteLLC(LLCCont, 0x46000a20);
371  2    } else {
372  2      WriteLLC(LLCCont, 0x46000220);
373  1    };

375  1    // Now we set the gap count on the system to 33 (Cos the Adaptec board wont change)

377  1  //  WriteLLC(0x80,0x006100e0);
378  1  //  WriteLLC(0x8C,0xff9effff);

380  1    // Remember our Node ID and how many nodes there are
381  1    MyNodeID = ReadLLC(LLCBusRes) & 0xFFFF;
382  1    NodeCount = (ReadLLC(LLCBusRes) & 0x3f000000) >> 24;

384  1    WriteLLC(LLCIntMas,0x930b9800);       // Specify what we are interested in
385  1    WriteLLC(LLCInte,0xFFFFFFFF);         // Clear everything

387  1    DebugMsg("1394 LLC init complete.\n");
388     }
```

```
390     int Init1394(void)
391  1  {
392  1    int Cnt;

394  1    /* First we make sure the Power Down on the chipset is off */

396  1    iLinkError = 0;
397  1    Wakeup1394();


400  1    Cnt = 0;
401  2    while ((!Check1394IntfOK()) && (Cnt++ < 100)) {
402  2    /* Keep reading LLC until we either give up or it works */
403  2      ReadLLC(LLCInte);
404  1    }
405  1    ReadLLC(LLCInte);
406  2    if (!Check1394IntfOK()) {
407  2      DebugMsg("ilink.c: ERROR - LLC-Mem interface poked.\n");
408  2      return(ERROR_LLCMemInterfacePoked);
409  1    };

411  1    /* Start the Transaction Layer, with 1K of stack space */

413  1    AddProcess((unsigned int) &iLinkTransLayer, 0, 4096);


416  1    /* Now we set up interrupts */

418  1    WriteLLC(LLCDiag, 0x00000300);        // Switch off any diagnostics
419  1    InitLLC();

421  1    Init1394Ints((int) &IntVec1394Asm);

423  1    DebugMsg("Init1394 completed.\n");
424  1    return(0);
425     }
```

```
1     /*******************************************************************
2      * ilinkTRANS.c                                                    *
3      *-----------------------------------------------------------------*
4      *                                                                 *
5      * Transaction layer for 1394                                      *
                                                                                    *
6      *                                                                 *
7      *  April 1999, Ray Heasman                                        *
8      *                                                                 *
9      *******************************************************************/

11    #include "miscinc.h"
12    #include "msgtypes.h"
13    #include "TSBLLC.h"
14    #include "intIDs.h"
15    #include <stdio.h>

17    // #define ILINKTRDEBUG
18    // #define LABELDEBUG

20    typedef unsigned short TLAB;
21    typedef unsigned short ADDR;

23 1  typedef struct {
24 1    TLAB  TCode;
25 1    ADDR  Src;
26 1    ADDR  Dest;
27 1    QUAD  RetID;                 // Return ID
28 1    int   Next;
29 1    int   Last;
30 1    TTime TimeOut;
31 1    int   Retries;
32    } TLabelQ;

34    TTime TrSplitTimeout;
35    extern QUAD MyNodeID;

37    TLabelQ TLQ[64];
38    int TLQFree;
39    int TLQHead;
40    int TLQTail;

42    TLAB GetTLabel(QUAD UID, ADDR Src, ADDR Dest, TTime TimeOut)
43    // Get a free Transaction label
44 1  {
45 1     int tmp;

47 1     #ifdef LABELDEBUG
48 1       char DBuf[60];
49 1     #endif

51 2     if (TLQFree == -1) {
52 2       return(0);
53 1     };

55 2     if (UID == 0) {
56 2       DebugMsg("ilinkTRANS.c (GetTLabel): Caller did not set reply message ID.\n");
57 1     };

59 1     tmp = TLQFree;

61 1     TLQFree = TLQ[TLQFree].Next;

63 1     TLQ[tmp].Last = TLQTail;
64 1     TLQ[tmp].Next = -1;
65 1     TLQ[tmp].Src = Src;
66 1     TLQ[tmp].Dest  = Dest;
67 1     TLQ[tmp].RetID = UID;
68 1     TLQ[tmp].TimeOut = TimeOut;
```



```
70 1     if (TLQTail == -1) TLQHead = tmp;
71 1     TLQTail = tmp;

73 1     #ifdef LABELDEBUG
74 1       sprintf(DBuf, "ilinkTRANS.c (GetTLabel): Got TLabel %d\n",TLQ[tmp].TCode);
75 1       DebugMsg(DBuf);
76 1     #endif

78 1     return(TLQ[tmp].TCode);
79    }
```

```
81      int RetireTLabel(ADDR Src, ADDR Dest, TLAB Label)
82      // Retires a Transaction label
83      // Returns the RepID for success. 0 if the label is illegal
84   1  {
85   1     QUAD ID;
86   1     #ifdef LABELDEBUG
87   1       char DBuf[60];
88   1     #endif

90   2    if ((Label > 63) || (Label < 1)) {
91   2      return(0);
92   1    };

94   1    Label--;

96   2    if ((TLQ[Label].Src != Src) || (TLQ[Label].Dest != Dest)) {
97   2      return(0);
98   1    };

100  1    #ifdef LABELDEBUG
101  1      sprintf(DBuf, "ilinkTRANS.c (RetireTLabel): Retired TLabel %d\n",Label+1);
102  1      DebugMsg(DBuf);
103  1    #endif

105  1    ID = TLQ[Label].RetID;

107  1    if (TLQHead == Label)
108  1      TLQHead = TLQ[Label].Next;
109  1    if (TLQTail == Label)
110  1      TLQTail = TLQ[Label].Last;

112  1    TLQ[TLQ[Label].Last].Next = TLQ[Label].Next;
113  1    TLQ[TLQ[Label].Next].Last = TLQ[Label].Last;

115  1    TLQ[TLQFree].Last = Label;
116  1    TLQ[Label].Next = TLQFree;
117  1    TLQ[Label].Last = -1;
118  1    TLQFree = Label;

120  1    return(ID);
121     }
```



```
123     TLAB TimeoutLabel(TTime TimeThresh, ADDR *Src, ADDR *Dest, QUAD *RetID)
124     // Returns a transaction code that has timed out, or 0
125     // If the transaction code is valid, returns the Src, Dest and RetID of the transaction too.
126  1  {
127  1    #ifdef LABELDEBUG
128  1      char DBuf[60];
129  1    #endif

131  2    if (TLQHead == -1) {
132  2      return(0);
133  1    }
134  2    if (CompareTime(TLQ[TLQHead].TimeOut, TimeThresh) <= 0) {
135  2      #ifdef LABELDEBUG
136  2        sprintf(DBuf, "ilinkTRANS.c (TimeoutLabel): Timeout on TLabel %d\n",TLQ[TLQHead].TCode);
137  2        DebugMsg(DBuf);
138  2      #endif
139  2      *Src = TLQ[TLQHead].Src;
140  2      *Dest = TLQ[TLQHead].Dest;
141  2      *RetID = TLQ[TLQHead].RetID;
142  2      return(TLQ[TLQHead].TCode);
143  1    }

145  1    return(0);
146     }
```

```
148     TTime GetOldestTimeout(void)
149     // Returns the next timeout of any entries in the queue
150 1   {
151 1     TTime Tmp;
152 1     #ifdef LABELDEBUG
153 1       char DBuf[60];
154 1     #endif

156 2     if (TLQHead == -1) {
157 2       #ifdef LABELDEBUG
158 2         sprintf(DBuf, "ilinkTRANS.c (GetOldestTimeout): Got called with empty queue!\n");
159 2         DebugMsg(DBuf);
160 2       #endif

162 2       Tmp.Lo = 0;
163 2       Tmp.Hi = 0;
164 2       return(Tmp);
165 1     }
166 1     #ifdef LABELDEBUG
167 1       sprintf(DBuf, "ilinkTRANS.c (GetOldestTimeout): Returned TLabel %d\n",TLQ[TLQHead].TCode);
168 1       DebugMsg(DBuf);
169 1     #endif

171 1     return(TLQ[TLQHead].TimeOut);
172     }
```



```
174     int TimeoutQNotEmpty(void)
175     // Returns zero if the TLQ is empty
176 1   {
177 1      return(TLQHead != -1);
178     }
```

```
180      void InitTLayer(void)
181      // Initialise Transaction layer.
182  1   {
183  1     int f;
184  1     TrSplitTimeout.Hi = 0;
185  1     TrSplitTimeout.Lo = 800;

187  1   // Initialise Transaction Queue
188  2     for (f=0;f<64;f++) {
189  2       TLQ[f].TimeOut.Hi = 0;
190  2       TLQ[f].TimeOut.Lo = 0;
191  2       TLQ[f].TCode = f+1;
192  2       TLQ[f].Last = f-1;
193  2       TLQ[f].Next = f+1;
194  1     };
195  1     TLQ[63].Next = -1;

197  1     TLQFree = 0;
198  1     TLQHead = -1;
199  1     TLQTail = -1;
200      }
```



```
202      QUAD GetMsgID(TLinkPacket *TlReq)
203      // Reads the address in the packet and associates it with a Message ID or 0
204  1   {
205  1     QUAD Addr;
206  1     QUAD RVal;

208  1     RVal = 0;
209  2     if (TlReq->DBlock.AddrBlock.DestOffHi != 0xFFFF) {
210  2       return(0);
211  1     }
212  1     Addr = TlReq->DBlock.AddrBlock.DestOffLo;

214  1   // Chop memory into 1K chunks (for the moment) and figure out what goes where
215  2     switch (Addr >> 10) {
216  2       case 0x3C0000 :  RVal = CSRBASEID;
217  2                     break;
218  2       case 0x3C0001 :  RVal = CSRROMBASEID;
219  2                     break;
220  2       case 0x3C0040 :  // 4k for the SBP space
221  2       case 0x3C0041 :
222  2       case 0x3C0042 :
223  2       case 0x3C0043 :  RVal = CSRSBP2BASEID;
224  2                     break;
225  2       default       : RVal = 0;
226  2                     break;
227  1      }
228  1      return(RVal);
229      }
```

```
231     void LLC_CommenceTX(void)
232     // Set up the interrupt register on the LLC to allow a TXReady interrupt.
233     // The interrupt will check for outgoing packets and send them.
234     // Hopefully, it will also disable any further interrupts of its kind.
235     // FIXME: This should probably be in ilink.c
236  1  {
237  1    // Now we set the TXReady bit in the LLC Interrupt mask.
238  1    WriteLLCMasked(LLCIntMas, 0x04000000, 0x04000000);
239  1    // Clear the int and the overall int bit, to allow it to trigger
240  1    WriteLLC(LLCInte, 0x84000000);
241     }
```



```
243     void HandleRXReadResp(TBMsgHandle *Msg)
244     // Handle a Read response from another node.
245     // Collects the result, associates it with the relevent read request,
246     // and sends it to the specified reply-to MsgID.
247  1  {
248  1     TTransReq *TrResp;
249  1     TLinkPacket *TlResp;
250  1     QUAD MsgID;

252  1     #ifdef ILINKTRDEBUG
253  1       DebugMsg("HandleRXReadResp called.\n");
254  1     #endif

256  1     TlResp = (TLinkPacket *) Msg->MsgPtr;
257  1     TrResp = (TTransReq *) Msg->MsgPtr;

259  1     // Note the order of Src and Dest is changed for calls on incoming response packets.
260  1     MsgID = RetireTLabel(TlResp->DestID, TlResp->DBlock.AddrBlock.SrcID, (
                                                                  TlResp->tcodes & 0xFC0)>>10);
261  2     if (MsgID != 0) {
262  2       TrResp->DestID = TlResp->DBlock.AddrBlock.SrcID;
                                                   // Src ID is the Dest ID given in the request.
263  2       // Now set length to 4 for quadlet payload or otherwise copy the correct value across.
264  2       if (((TlResp->tcodes) >> 4) & 1)
265  2         TrResp->Length = TlResp->QBlock.Info.DataLen;
266  2       else
267  2         TrResp->Length = 4;
268  2       TrResp->TrType = 8;                // Read response.
269  2       SendBMsg(&Msg, MsgID);
270  2     } else {
271  2       DebugMsg("ilinkTRANS.c (HandleRXReadResp), Unsolicited response received.\n");
                                                                                // FIXME see std
272  1     };
273     }
```

```
276     void HandleRXWriteReq(TBMsgHandle *Msg)
277     // Handle a Write request, by passing it on to the CSR handler
278 1   {
279 1      #ifdef ILINKTRDEBUG
280 1        DebugMsg("HandleRXWriteReq called.\n");
281 1      #endif
282     }
```



```
284     void HandleRXWriteResp(TBMsgHandle *Msg)
285     // Handle a write response, by passing it on to the correct process
286 1   {
287 1      #ifdef ILINKTRDEBUG
288 1        DebugMsg("HandleRXWriteResp called.\n");
289 1      #endif
290     }
```

```
292     void HandleRXReadReq(TBMsgHandle *Msg)
293     // Handle a read request, by passing it to the CSR handler
294 1   {
295 1      TTransReq *TrReq;
296 1      TLinkPacket *TlReq;
297 1      QUAD MsgID;

299 1      #ifdef ILINKTRDEBUG
300 1        DebugMsg("HandleRXReadReq called.\n");
301 1      #endif

303 1      TlReq = (TLinkPacket *) Msg->MsgPtr;
304 1      TrReq = (TTransReq *) Msg->MsgPtr;

306 1      MsgID = GetMsgID(TlReq);
307 2      if (MsgID != 0) {
308 2        TrReq->DestID = TlReq->DBlock.AddrBlock.SrcID;
                                                 // Swap source & dest so CSR knows who reply is for
309 2        if (((TlReq->tcodes) >> 4) & 1)
310 2          TrReq->Length = TlReq->QBlock.Info.DataLen;
311 2        else
312 2          TrReq->Length = 4;
313 2        TrReq->TrType = TlReq->tcodes & 0xFC00;    // Read req. (ie bot byte 0) ORed with tlabel
314 2        SendBMsg(&Msg, MsgID);
315 2      } else {
316 2        DebugMsg("ilinkTRANS.c (HandleRXReadReq): RXed CSR read to unsupported address.");
                                                                                // FIXME see std
317 2        SendBMsg(&Msg, DEALLOCBMSG);
318 1      }
319     }
```



```
321     void HandleRXLockReq(TBMsgHandle *Msg)
322     // Handle a lock request, by passing it to the the CSR handler
323 1   {
324 1      #ifdef ILINKTRDEBUG
325 1        DebugMsg("HandleRXLockReq called.\n");
326 1      #endif
327     }
```

```
329      void HandleRXLockResp(TBMsgHandle *Msg)
330      // Handle a lock response, by passing it to the correct process
331 1    {
332 1       #ifdef ILINKTRDEBUG
333 1         DebugMsg("HandleRXLockReq called.\n");
334 1       #endif
335      }
```



```
337      void HandleLkDataInd(void)
338      // Handles incoming data from the link layer. Dont call
339      // unless you _know_ that said data has arrived.
340 1    {
341 1      TBMsgHandle *TmpBMsg;
342 1      TLinkPacket *TlReq;

344 1      WaitBMsg(&TmpBMsg, LK_DATAIND);
345 1      #ifdef ILINKTRDEBUG
346 1        DebugMsg("1394 TL: LK_DATAIND received.\n");
347 1      #endif

349 1      TlReq = (TLinkPacket *) TmpBMsg->MsgPtr;

351 2      switch ((TlReq->tcodes & 0xf0) >> 4) {
352 2        case 0:  HandleRXWriteReq(TmpBMsg);         // A write request from another node
353 2                 break;
354 2        case 1:  HandleRXWriteReq(TmpBMsg);
355 2                 break;
356 2        case 2:  HandleRXWriteResp(TmpBMsg);
357 2                 break;
358 2        case 4:  HandleRXReadReq(TmpBMsg);
359 2                 break;
360 2        case 5:  HandleRXReadReq(TmpBMsg);
361 2                 break;
362 2        case 6:  HandleRXReadResp(TmpBMsg);
363 2                 break;
364 2        case 7:  HandleRXReadResp(TmpBMsg);
365 2                 break;
366 2        case 9:  HandleRXLockReq(TmpBMsg);
367 2        case 11: HandleRXLockResp(TmpBMsg);
368 2                 break;
369 2        default: // Either an error, or a packet type I have yet to implement.
370 2                 DebugMsg("HandleLkDataInd: Unrecognised packet received.\n");
371 2                 break;
372 1      }
373      }
```

```
375     void HandleReadReq(TBMsgHandle *Msg)
376     // Read request from application, to be sent out.
377 1   {
378 1     TTransReq *TrReq;
379 1     TLinkPacket *TlReq;
380 1     QUAD tl;
381 1     int TLen;

383 1     TrReq = (TTransReq *)   Msg->MsgPtr;
384 1     TlReq = (TLinkPacket *) Msg->MsgPtr;

386 1     tl = GetTLabel(Msg->UserID, MyNodeID, TrReq->DestID, AddTime(GetCycTime(
                                                                    ), TrSplitTimeout)) << 10;

388 2     if (TrReq->Length == 4) {
389 2       // Read request for data quadlet
390 2       tl = tl | (4 << 4);         // set tcode
391 2       TlReq->tcodes = tl;
392 2       TlReq->DBlock.AddrBlock.SrcID = MyNodeID;
393 2       SendBMsg(&Msg, LK_DATAREQ);
394 2       #ifdef ILINKTRDEBUG
395 2         DebugMsg("Read request for data quadlet sent.\n");
396 2       #endif
397 2     } else {
398 2       // Read request for a block
399 2       tl = tl | (5 << 4);
400 2       TlReq->tcodes = tl;
401 2       TLen = TrReq->Length;       // Length and SrcID share the same offset in the message
402 2       TlReq->DBlock.AddrBlock.SrcID = MyNodeID;
403 2       TlReq->QBlock.Info.DataLen = TLen;
404 2       TlReq->QBlock.Info.ExtTCode = 0;
405 2       SendBMsg(&Msg, LK_DATAREQ);
406 2       #ifdef ILINKTRDEBUG
407 2         DebugMsg("Read request for data block sent.\n");
408 2       #endif
409 1     }
410 1   //   LLC_TXReady();
411 1     LLC_CommenceTX();
412     }
```



```
414     void HandleWriteReq(TBMsgHandle *Msg)
415 1   {
416 1     TTransReq *TrReq;
417 1     TLinkPacket *TlReq;
418 1     QUAD tl,TLen;

420 1     TrReq = (TTransReq *)   Msg->MsgPtr;
421 1     TlReq = (TLinkPacket *) Msg->MsgPtr;

423 1     tl = GetTLabel(Msg->UserID, MyNodeID, TrReq->DestID, AddTime(GetCycTime(
                                                                    ), TrSplitTimeout)) << 10;

425 2     if (TrReq->Length == 4) {
426 2       // Write request for data quadlet
427 2       tl = tl | (0 << 4);         // set tcode (yes, redundant, but you can see its right)
428 2       TlReq->tcodes = tl;
429 2       TlReq->DBlock.AddrBlock.SrcID = MyNodeID;
430 2       SendBMsg(&Msg, LK_DATAREQ);
431 2       #ifdef ILINKTRDEBUG
432 2         DebugMsg("Write request for data quadlet sent.\n");
433 2       #endif
434 2     } else {
435 2       // Write request for data block
436 2       tl = tl | (1 << 4);
437 2       TlReq->tcodes = tl;
438 2       TLen = TrReq->Length;       // Length and SrcID share the same offset in the message
439 2       TlReq->DBlock.AddrBlock.SrcID = MyNodeID;
440 2       TlReq->QBlock.Info.DataLen =  TLen;
441 2       TlReq->QBlock.Info.ExtTCode = 0;
442 2       SendBMsg(&Msg, LK_DATAREQ);
443 2       #ifdef ILINKTRDEBUG
444 2         DebugMsg("Write request for data block sent.\n");
445 2       #endif
446 1     }
447 1   //  LLC_TXReady();
448 1     LLC_CommenceTX();
449     }
```

```
451     void HandleReadResp(TBMsgHandle *Msg)
452 1   {
453 1     TTransResp *TrResp;
454 1     TLinkPacket *TlReq;
455 1     QUAD tl,TLen;

457 1     TrResp = (TTransResp *) Msg->MsgPtr;
458 1     TlReq = (TLinkPacket *) Msg->MsgPtr;

460 1     tl = TrResp->TrType & 0xFF00; // Use supplied Transaction label, preshifted

462 2     if (TrResp->Length == 4) {
463 2       // Read response for data quadlet
464 2       tl = tl | (6 << 4);         // set tcode to "read response for data quadlet"
465 2       TlReq->tcodes = tl;
466 2       TlReq->DBlock.AddrBlock.SrcID = MyNodeID;

468 2       TlReq->DBlock.RCodeBlock.RCode = 0x0000;
                                        // RCode of 0 is "alls well". FIXME: Added errors later
469 2       TlReq->DBlock.RCodeBlock.Reserved = 0x00000000;
470 2       SendBMsg(&Msg, LK_DATAREQ);
471 2       #ifdef ILINKTRDEBUG
472 2         DebugMsg("Read respose for data quadlet sent.\n");
473 2       #endif
474 2     } else {
475 2       // Read response for data block
476 2       tl = tl | (7 << 4);
477 2       TlReq->tcodes = tl;
478 2       TLen = TrResp->Length;      // Length and SrcID share the same offset in the message
479 2       TlReq->DBlock.RCodeBlock.SrcID = MyNodeID;
480 2       TlReq->DBlock.RCodeBlock.RCode = 0x0000;
                                        // RCode of 0 is "alls well". FIXME: Added errors later
481 2       TlReq->DBlock.RCodeBlock.Reserved = 0x00000000;

483 2       TlReq->QBlock.Info.DataLen =  TLen;
484 2       TlReq->QBlock.Info.ExtTCode = 0;

486 2       SendBMsg(&Msg, LK_DATAREQ);
487 2       #ifdef ILINKTRDEBUG
488 2         DebugMsg("Read response for data block sent.\n");
489 2       #endif
490 1     }
491 1   //  LLC_TXReady();
492 1     LLC_CommenceTX();
493     }
```



```
495     void HandleLockReq(TBMsgHandle *Msg)
496 1   {
497 1     DebugMsg("ilinkTRANS.c - HandleLockReq called. Unimplemented function.\n");
498 1     SendBMsg(&Msg, DEALLOCBMSG);
499     }
```

```
501      void HandleTrDataResp(void)
502      // Handles incoming read responses from the application later. Dont call this
503      // unless you know that said data has arrived.
504      // This routine is not strictly needed, but is included to maintain a consistent
505      // structure in the program.
506  1   {
507  1     TBMsgHandle *TmpBMsg;

509  1     WaitBMsg(&TmpBMsg, TR_DATARESP);
510  1     #ifdef ILINKTRDEBUG
511  1       DebugMsg("1394 TL: TR_DATARESP received.\n");
512  1     #endif

514  1     HandleReadResp(TmpBMsg);

516  1     // Msg is not deallocated, because it gets rewritten and passed on.
517      }
```



```
519      void HandleTrDataReq(void)
520      // Handles incoming data from the application layer. Dont call this
521      // unless you know that said data has arrived.
522  1   {
523  1     TBMsgHandle *TmpBMsg;
524  1     TTransReq *TrReq;

526  1     WaitBMsg(&TmpBMsg, TR_DATAREQ);
527  1     #ifdef ILINKTRDEBUG
528  1       DebugMsg("1394 TL: TR_DATAREQ received.\n");
529  1     #endif

531  1     TrReq = (TTransReq *) TmpBMsg->MsgPtr;

533  2     switch (TrReq->TrType) {
534  2       case 0 :  // Read request
535  2             HandleReadReq(TmpBMsg);
536  2           break;
537  2       case 1 :  // Write request
538  2             HandleWriteReq(TmpBMsg);
539  2           break;
540  2       case 2 :  // Lock request
541  2             HandleLockReq(TmpBMsg);
542  2           break;
543  2       default:
544  2         DebugMsg("ilinkTRANS.c (
                                     HandleTrDataReq) - ERROR: RXed invalid message from App layer.\n");
545  1     }
546  1     // Msg is not deallocated, because it gets rewritten and passed on.
547      }
```

```
549     void HandleTrTimeout()
550     // Gets called when a transaction has timed out and should be retired
551 1   {
552 1     TLAB Label;
553 1     ADDR Src, Dest;
554 1     QUAD RetID;
555 1     TTime TimeThresh;
556 1     TTransReq *TrResp;
557 1     TBMsgHandle *TmpBMsg;
558 1     #ifdef LABELDEBUG
559 1       char DBuf[80];
560 1     #endif

562 1     TimeThresh = GetCycTime();
563 1     Label = TimeoutLabel(TimeThresh, &Src, &Dest, &RetID);

565 1     #ifdef LABELDEBUG
566 1         sprintf(DBuf, "ilinkTRANS.c (HandleTrTimeout): Label: %hd\n", Label);
567 1         DebugMsg(DBuf);
568 1     #endif

570 1   // Remove this transaction, and any others that may be pending

572 2     while (Label != 0) {
573 2       RetireTLabel(Src, Dest, Label);
574 2       WaitBMsg(&TmpBMsg, ALLOCBMSG);
575 2       TrResp = (TTransReq *) TmpBMsg->MsgPtr;

577 2       TrResp->TrType = TRTIMEOUT; // Signal a Timeout to the calling process
578 2       TrResp->DestID = Dest;
579 2       TrResp->Length = 0xFFFF;
                                // Nice and big and illegal. Hopefully will break bad programs.

581 2       #ifdef LABELDEBUG
582 2         sprintf(DBuf, "ilinkTRANS.c (HandleTrTimeout): Sending to %d\n",RetID);
583 2         DebugMsg(DBuf);
584 2       #endif

586 2       SendBMsg(&TmpBMsg, RetID);
587 2       Label = TimeoutLabel(TimeThresh, &Src, &Dest, &RetID);
588 1     }
589     }
```



```
591     void iLinkTransLayer(void)
592 1   {
594 1     TBMsgHandle *TmpBMsg;
595 1     TSMsgHandle *TmpSMsg;
596 1     QUAD *Msg;
597 1     TTimerReq *TimeReq;
598 1     int MesgNum;
599 1     TTime TmpTime, CurrTime;
600 1     int TOPending;

602 1     QUAD MList[6];

604 1     #ifdef LABELDEBUG
605 1       char DBuf[80];
606 1     #endif

608 1     // First make sure that the interrupt is registered for everything.

610 1     RegisterType(LK_DATAREQ, MAIN1394Int); // Register interrupt for outgoing packets
611 1     RegisterType(LK_DATACONF, GetPID());  // Used to rx confirmation of sending a packet
612 1     RegisterType(LK_DATAIND, GetPID());   // Used to rx a packet. (Big msg)
613 1     RegisterType(TR_DATAREQ, GetPID());   // Request from app to send packet
614 1     RegisterType(TR_DATARESP, GetPID());  // Request from app to send response to read
615 1     RegisterType(TRANS_TIMER, GetPID());  // used to get timer messages from timer server

617 1     MList[0] = LK_DATAIND;
618 1     MList[1] = LK_DATACONF;
619 1     MList[2] = TR_DATAREQ;
620 1     MList[3] = TR_DATARESP;
621 1     MList[4] = TRANS_TIMER;
622 1     MList[5] = 0;

624 1     InitTLayer();
625 1     TOPending = 0;

627 1     DebugMsg("1394 Transaction layer server started.\n");

629 1     WaitSMsg(&TmpSMsg, ALLOCSMSG);
630 1     SendSMsg(&TmpSMsg, INITREPLY);        // Tell the init sequence to carry on.

632 2     while (1){
633 2         MesgNum = WaitOnMsgList(MList);
634 3         switch (MesgNum) {
635 3           case 0 : // Link layer data indication: Data has arrived.
636 3                    HandleLkDataInd();
637 3                  break;
638 3           case 1 : // Link layer data confirmation
639 3   //               HandleLkDataConf();
640 3                  break;
641 3           case 2 : // App layer request to send data
642 3                    HandleTrDataReq();
643 3                  break;
644 3           case 3 : // App layer request to send address space response
645 3                    HandleTrDataResp();
646 3           case 4 : // Transaction has timed out
647 3                    WaitSMsg(&TmpSMsg, TRANS_TIMER);
648 3                    SendSMsg(&TmpSMsg, DEALLOCSMSG);
649 3                    HandleTrTimeout();
650 3                    TOPending = 0;
651 3                  break;
652 3           default:
653 3                    DebugMsg("iLinkTransLayer: ERROR - Unreachable code reached.\n");
654 2         }

656 2         #ifdef LABELDEBUG
657 2           sprintf(DBuf, "ilinkTRANS.c (iLinkTransLayer): TOPending: %d\n", TOPending);
658 2           DebugMsg(DBuf);
659 2           sprintf(DBuf, "ilinkTRANS.c (iLinkTransLayer): TLQT: %d\n", TLQTail);
```

```
660 2         DebugMsg(DBuf);
661 2       #endif

663 3       while ((!TOPending) && (TimeoutQNotEmpty())) {

665 3         // If we get here then there are items in the Q and we should set a timeout
666 3         CurrTime = GetCycTime();
667 3         TmpTime = GetOldestTimeout();
668 4         if (CompareTime(TmpTime, CurrTime) > 0) {
669 4           // If we are still ahead of the clock work out a new tmp time and issue the request
670 4           WaitSMsg(&TmpSMsg, ALLOCSMSG);
671 4           TimeReq = (TTimerReq *) TmpSMsg->MsgPtr;
672 4           TimeReq->ReplyAdd = TRANS_TIMER;

674 4           TmpTime = SubTime(TmpTime, CurrTime);
675 4           TimeReq->Time = TmpTime.Hi*8000+TmpTime.Lo;
676 4           #ifdef LABELDEBUG
677 4              sprintf(DBuf, "ilinkTRANS.c (
                                      iLinkTransLayer): Timeout in %d ticks\n", TimeReq->Time);
678 4              DebugMsg(DBuf);
679 4           #endif
680 4           SendSMsg(&TmpSMsg, TIMERINPORT);
681 4           TOPending = 1;
682 4         } else {
683 4           // The clock caught us, force a Timeout. This is hopefully an optimisation
684 4           HandleTrTimeout();
685 4           TOPending = 0;
686 3         };
687 2       };
688 2 //     WaitSMsg(&TmpSMsg, ALLOCSMSG);
689 2 //     TimeReq = (TTimerReq *) TmpSMsg->MsgPtr;
690 2 //     TimeReq->ReplyAdd = TRANS_TIMER;
691 2 //     TimeReq->Time = 1200;                        // Wait for 1.5s
692 2 //     SendSMsg(&TmpSMsg, TIMERINPORT);
693 2 //     WaitSMsg(&TmpSMsg, TRANS_TIMER);
694 2 //     SendSMsg(&TmpSMsg, DEALLOCSMSG);
695 2 //     WaitBMsg(&TmpBMsg, ALLOCBMSG);
696 2 //     Msg = (QUAD*) TmpBMsg->MsgPtr;
697 2 //     Msg[0]=0xFFC10440;
698 2 //     Msg[1]=0xFFC0FFFF;
699 2 //     Msg[2]=0xF0000000;
700 2 //     SendBMsg(&TmpBMsg, LK_DATAREQ);
701 2 //     DebugMsg("Test packet sent to link layer.\n");
702 1   };
703   }
```

```
1      /*******************************************************************
2       * ilinkconv.c                                                     *
3       *-----------------------------------------------------------------*
4       *                                                                 *
5       * 1394 Read & Conversion routines, from TI LLC to 1394 Standard   *
6       *                                                                 *
7       * April 1999                                                      *
8       *                                                                 *
9       *******************************************************************/

11     #include "miscinc.h"
12     #include "TSBLLC.h"
13     #include "errors.h"
14     #include <stdio.h>

16     extern int iLinkError;

18     #define ILINKRXDEBUG
19     #define ILINKTXDEBUG

21     void DumpMem(QUAD *Mem, QUAD Size)
22  1  {
23  1     char DBuf[40];
24  1     int i;

26  1     DebugMsg("Packet Dump:\n");
27  2     for (i = 0; i < Size; i++) {
28  2       sprintf(DBuf, "   %p\n",*(Mem++));
29  2       DebugMsg(DBuf);
30  1     }
31     }
```



```
33     int ReadQuadletPacket(QUAD First, QUAD *Mem)
34     // Read the quadlet packet into memory in 1394 Std format
35     // P SPECIFIC
36  1  {
37  1    #ifdef ILINKRXDEBUG
38  1      QUAD *DMem;
39  1      DMem = Mem;
40  1    #endif
41  1    *(Mem++) = First;              // Write DestID, tLabel, tCode, pri
42  1    *(Mem++) = ReadLLC(0xC0);      // Read SrcID, reserved/DestOffHi from GRF
43  1    *(Mem++) = ReadLLC(0xC0);      // Read DestOffLo from GRF
44  1    *(Mem++) = ReadLLC(0xC0);
45  1    *(Mem++) = ReadLLC(0xC0);      // Get LLC extra info

47  1    #ifdef ILINKRXDEBUG
48  1      DumpMem(DMem, 5);
49  1    #endif
50  1    return(1);
51     }
```

```
53     int ReadNoPayloadPacket(QUAD First, QUAD *Mem)
54     // Read a packet containing no data payload into memory in 1394 Std format
55  1  {
56  1    #ifdef ILINKRXDEBUG
57  1      QUAD *DMem;
58  1      DMem = Mem;
59  1    #endif

61  1    *(Mem++) = First;
62  1    *(Mem++) = ReadLLC(0xC0);
63  1    *(Mem++) = ReadLLC(0xC0);
64  1    *(Mem++) = ReadLLC(0xC0);     // Get LLC ack info

66  1    #ifdef ILINKRXDEBUG
67  1      DumpMem(DMem, 4);
68  1    #endif
69  1    return(1); // Success
70     }
```



```
72     int ReadBlockPacket(QUAD First, QUAD *Mem)
73     // Read a packet containing a block payload into memory in 1394 Std format
74     // Dont bother with CRC. Define CRC of 0 as "all OK". Possible FIXME
75  1  {
76  1    int tmp;
77  1    int DLen, DLen2;
78  1    *(Mem++) = First;
79  1    *(Mem++) = ReadLLC(0xC0);
80  1    *(Mem++) = ReadLLC(0xC0);
81  1    DLen = ReadLLC(0xC0);
82  1    *(Mem++) = DLen;
83  1    *(Mem++) = 0;          // Write header FAKE CRC to mem

85  1    // Make DLen and DLen2 the number of bytes in the packet

87  1    DLen = DLen2 = (DLen >> 16)&0xFFFF;

89  1    // Read (possibly zero length) zero-padded packet
90  2    while (DLen2 > 0) {
91  2      *(Mem++) = ReadLLC(0xC0);
92  2      DLen2 -= 4;
93  1    };
94  1    tmp = ReadLLC(0xC0);
95  2    if (DLen > 0) {
96  2      // Test for Data RX error or type error.
97  2      // Lump them together as "bad data"
98  3      if (((tmp & 0xF) == 0xD) || (((tmp & 0xF) == 0xE))) {
99  3        *(Mem++) = tmp;   // Write non zero FAKE CRC
100 3      } else {
101 3        *(Mem++) = 0;     // Write zero FAKE CRC. ie. Ok data
102 2      };
103 1    };
104 1    *(Mem++) = tmp;       // Write LLC ack info
105 1    return(1); // Success
106    }
```

```
108     int ReadSelfIDPacket(QUAD First, QUAD *Mem)
109     // Read SelfID packets into memory pointed to by Mem
110  1  {
111  1    QUAD tmp;

113  1    *(Mem)++ = First;
114  2    do {
115  2      tmp = ReadLLC(0xC0);
116  2      *(Mem)++ = tmp;
117  1    } while (tmp & 0xC0000000);
118  1    return(1);
119     }
```



```
121     int GRFNotEmpty(void)
122     // Returns 0 if the GRF is empty
123  1  {
124  1    return(!(ReadLLC(LLCGRFSta) & 0x10000));
125     }
```

```
128      int ReadPacket(QUAD *Mem)
129      // Read a packet of unknown type into memory pointed to by Mem
130      // Returns 0 for failure
131  1   {
132  1     QUAD tmp;

134  1   // Have added a check of the Cd bit in GRF Status Reg.
135  1   // Not sure if it indicates that the next quadlet begins
136  1   // a packet or if the quadlet just read is the beginning of
137  1   // a packet. Anyway, it would allow resynching if something
138  1   // goes wrong. Will poke everything if I got it wrong.

140  1     tmp = ReadLLC(LLCGRFSta);

142  1     if (tmp & 0x10000) return(0); // Empty GRF - Failure

144  2     while (!(tmp & 0x8000)) {
145  2       ReadLLC(0xC0);
146  2       tmp = ReadLLC(LLCGRFSta);
147  1     };

149  1     tmp = ReadLLC(0xC0);  // Get first quadlet

151  1     // Check transaction code and get correct length packet.

153  2     switch ((tmp & 0xF0)>>4) {
154  2       case  0:
155  2               #ifdef ILINKRXDEBUG
156  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Write req. for data quadlet.\n");

157  2               #endif
158  2               ReadQuadletPacket(tmp, Mem);
159  2               break;
160  2       case  1:
161  2               #ifdef ILINKRXDEBUG
162  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Write req. for data block.\n");
163  2               #endif
164  2               ReadBlockPacket(tmp, Mem);
165  2               break;
166  2       case  2:
167  2               #ifdef ILINKRXDEBUG
168  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Write resp. No payload. \n");
169  2               #endif
170  2               ReadNoPayloadPacket(tmp, Mem);
171  2               break;
172  2       case  4:
173  2               #ifdef ILINKRXDEBUG
174  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Read req for data quadlet.\n");
175  2               #endif
176  2               ReadNoPayloadPacket(tmp, Mem);
177  2               break;
178  2       case  5:
179  2               #ifdef ILINKRXDEBUG
180  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Read req for data block.\n");
181  2               #endif
182  2               ReadQuadletPacket(tmp, Mem);
183  2               break;
184  2       case  6:
185  2               #ifdef ILINKRXDEBUG
186  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Read resp for data quadlet.\n");

187  2               #endif
188  2               ReadQuadletPacket(tmp, Mem);
189  2               break;
190  2       case  7:
191  2               #ifdef ILINKRXDEBUG
192  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Read resp for data block.\n");
193  2               #endif
```



```
194  2               ReadBlockPacket(tmp, Mem);
195  2               break;
196  2       case  8:
197  2               #ifdef ILINKRXDEBUG
198  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Cycle start packet.\n");
199  2               #endif
200  2               ReadQuadletPacket(tmp, Mem);
201  2               break;
202  2       case  9:
203  2               #ifdef ILINKRXDEBUG
204  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Lock req. Block payload.\n");
205  2               #endif
206  2               ReadBlockPacket(tmp, Mem);
207  2               break;
208  2       case 10:
209  2               #ifdef ILINKRXDEBUG
210  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Isochronouse data block.\n");
211  2               #endif
212  2               ReadBlockPacket(tmp, Mem);
213  2               break;
214  2       case 11:
215  2               #ifdef ILINKRXDEBUG
216  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Lock response. Block payload.\n");

217  2               #endif
218  2               ReadBlockPacket(tmp, Mem);
219  2               break;
220  2       case 14:
221  2               #ifdef ILINKRXDEBUG
222  2                 DebugMsg("ilinkconv.c: (readpacket) RXed Write req for data quadlet.\n");

223  2               #endif
224  2               ReadSelfIDPacket(tmp, Mem);
225  2               break;
226  2       default:
227  2               #ifdef ILINKRXDEBUG
228  2                 DebugMsg("ilinkconv.c: (readpacket) WARNING - Unknown transaction code.\n");

229  2               #endif
230  2               iLinkError = ERROR_LLCRXedUnknownTCode;
231  2               break;
232  1     };
233  1     return(0);
234      }
```

```
236     int SendNoPayloadPacket(QUAD *Mem)
237     // Send a packet containing no data payload from memory in 1394 Std format
238 1   {
239 1     QUAD tmp;

241 1     #ifdef ILINKTXDEBUG
242 1       QUAD *foo;
243 1       char DBuf[60];
244 1       DumpMem(Mem, 4);
245 1       foo = Mem;
246 1       tmp = *(Mem++);
247 1       sprintf(DBuf,"  :%x\n",(tmp & 0xFFFF) | 0x10000);           // First
248 1       DebugMsg(DBuf);
249 1       sprintf(DBuf,"  :%x\n",(tmp & 0xFFFF0000) |(*(Mem++) & 0xFFFF));    // Body
250 1       DebugMsg(DBuf);
251 1       sprintf(DBuf,"  :%x\n",*(Mem++));                    // Confirm and send
252 1       DebugMsg(DBuf);
253 1       DebugMsg("   \n");
254 1       Mem = foo;
255 1     #endif

257 1     tmp = *(Mem++);
258 1     WriteLLC(0x80,(tmp & 0xFFFF) | 0x10000);                       // First
259 1     WriteLLC(0x84,(tmp & 0xFFFF0000) |(*(Mem++) & 0xFFFF));        // Body
260 1     WriteLLC(0x8C,*(Mem++));                         // Confirm and send

262 1     return(1); // Success
263     }
```



```
265     int SendBlockPacket(QUAD *Mem)
266 1   {
267 1     QUAD tmp;
268 1     int f;

270 1     #ifdef ILINKTXDEBUG
271 1       QUAD *foo;
272 1       char DBuf[60];

274 1       sprintf(DBuf, "ATF status: %p\n",ReadLLC(LLCATFSta));
275 1       DebugMsg(DBuf);
276 1       DumpMem(Mem, 4);
277 1       foo = Mem;
278 1       tmp = *(Mem++);
279 1       sprintf(DBuf,"  :%p\n",(tmp & 0xFFFF) | 0x10000);           // First
280 1       DebugMsg(DBuf);
281 1       sprintf(DBuf,"  :%p\n",(tmp & 0xFFFF0000) |(*(Mem++) & 0xFFFF));    // Body
282 1       DebugMsg(DBuf);
283 1       sprintf(DBuf,"  :%p\n",*(Mem++));                    // DestLo
284 1       DebugMsg(DBuf);
285 1       sprintf(DBuf,"  :%p\n",*(Mem++));                    // Length + ExtTCode
286 1       DebugMsg(DBuf);
287 1       DebugMsg("   \n");
288 1       Mem = foo;
289 1     #endif
290 1     tmp = *(Mem++);
291 1     WriteLLC(0x80,(tmp & 0xFFFF) | 0x10000);
292 1     WriteLLC(0x84,(tmp & 0xFFFF0000) | (*(Mem++) & 0xFFFF));
293 1     WriteLLC(0x84,*(Mem++));      // DestLo
294 1     tmp = (((*Mem & 0xFFFF0000) >> 16)+3)>>2;     // Tmp is length in quads, including padding
295 1     WriteLLC(0x84,*(Mem++));      // Length + Ext TCode
296 1     #ifdef ILINKTXDEBUG
297 1       DumpMem(Mem, tmp);
298 1     #endif
299 2     for (f=0; f<(tmp-1); f++) {
300 2   //    DebugMsg(".");
301 2       WriteLLC(0x84,*(Mem++));    // Write all but last quadlet
302 1     };
303 1     WriteLLC(0x8C,*Mem);
304 1   //  DebugMsg(".    \n");
305 1     return(0);
306     }
```

```
308      int SendQuadletPacket(QUAD *Mem)
309  1   {
310  1     QUAD tmp;

312  1     #ifdef ILINKTXDEBUG
313  1       DumpMem(Mem, 5);
314  1     #endif

316  1     tmp = *(Mem++);
317  1     WriteLLC(0x80,(tmp & 0xFFFF) | 0x10000);                    // First
318  1     WriteLLC(0x84,(tmp & 0xFFFF0000) |(*(Mem++) & 0xFFFF));      // Body
319  1     WriteLLC(0x84,*(Mem++));                        // Data quadlet
320  1     WriteLLC(0x8C,*(Mem++));                        // Confirm and send
321  1     return(1); // Success
322      }
```



```
324      int SendPacket(QUAD *Mem)
325      // Send a packet of unknown type, from memory pointed to by Mem
326      // Returns 0 for failure

328  1   {
329  1     QUAD tmp;

331  1     tmp = ReadLLC(LLCATFSta);

333  2     if (tmp & 0x80000000) {
334  2       #ifdef ILINKTXDEBUG
335  2         DebugMsg("SendPacket: ATF full\n");
336  2       #endif
337  2       return(0); // Full ATF - Failure
338  1     }

340  1     // Check transaction code and get correct length packet.

342  1     tmp = 0;

344  2     switch ((Mem[0] & 0xF0)>>4) {
345  2       case  0:
346  2               #ifdef ILINKTXDEBUG
347  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Write req. for data quadlet.\n");

348  2               #endif
349  2               tmp = SendQuadletPacket(Mem);
350  2               break;
351  2       case  1:
352  2               #ifdef ILINKTXDEBUG
353  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Write req. for data block.\n");
354  2               #endif
355  2               tmp = SendBlockPacket(Mem);
356  2               break;
357  2       case  2:
358  2               #ifdef ILINKTXDEBUG
359  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Write resp. No payload.\n");
360  2               #endif
361  2               tmp = SendNoPayloadPacket(Mem);
362  2               break;
363  2       case  4:
364  2               #ifdef ILINKTXDEBUG
365  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Read req. for data quadlet.\n");

366  2               #endif
367  2               tmp = SendNoPayloadPacket(Mem);
368  2               break;
369  2       case  5:
370  2               #ifdef ILINKTXDEBUG
371  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Read req. for data block.\n");
372  2               #endif
373  2               tmp = SendQuadletPacket(Mem);
374  2               break;
375  2       case  6:
376  2               #ifdef ILINKTXDEBUG
377  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Read resp. for data quadlet.\n");

378  2               #endif
379  2               tmp = SendQuadletPacket(Mem);
380  2               break;
381  2       case  7:
382  2               #ifdef ILINKTXDEBUG
383  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Read resp. for data block.\n");
384  2               #endif
385  2               tmp = SendBlockPacket(Mem);
386  2               break;
387  2       case  8:
388  2               #ifdef ILINKTXDEBUG
389  2                 DebugMsg("ilinkconv.c: (SendPacket) TXed Cycle Start packet.\n");
```

```
390 2             #endif
391 2             tmp = SendQuadletPacket(Mem);
392 2             break;
393 2     case  9:
394 2             #ifdef ILINKTXDEBUG
395 2               DebugMsg("ilinkconv.c: (SendPacket) TXed Lock req.\n");
396 2             #endif
397 2             tmp = SendBlockPacket(Mem);
398 2             break;
399 2     case 10:
400 2             #ifdef ILINKTXDEBUG
401 2               DebugMsg("ilinkconv.c: (SendPacket) TXed Isoch. data block.\n");
402 2             #endif
403 2             tmp = SendBlockPacket(Mem);
404 2             break;
405 2     case 11:
406 2             #ifdef ILINKTXDEBUG
407 2               DebugMsg("ilinkconv.c: (SendPacket) TXed Lock resp.\n");
408 2             #endif
409 2             tmp = SendBlockPacket(Mem);
410 2             break;
411 2     case 14: tmp = 0;
412 2             break;
413 2     default: tmp = 0;
414 2             iLinkError = ERROR_LLCRXedUnknownTCode;
415 2             #ifdef ILINKTXDEBUG
416 2               DebugMsg("ilinkconv.c: (
                                SendPacket) Unknown packet typed received for transmit.\n");
417 2             #endif
418 2             break;
419 1   };
420 1   return(tmp);
421   }
```

```
1     /*******************************************************************
2      * initts.c                                                         *
3      *------------------------------------------------------------------*
4      *                                                                  *
5      *  This file contains all the necessary routines to set up the     *
6      *  timer registers for task switching, on an AT91M40400.           *
7      *                                                                  *
8      *  July 1998, Ray Heasman                                          *
9      *  Please excuse my accent. I am by habit a Pascal programmer      *
10     *                                                                  *
11     *******************************************************************/

13    #include "AT91timer.h"          /* Get Timer related constants and
14                                       addresses */
15    #include "AT91AIC.h"

17    void SetTimer(int TimerVec)
18    /* Set up the timer for taskswitching, and vector it to the given address */

20  1 {
21  1   int *Base;
22  1   Base = (int *) (TCBase+TCChan2);
23  1   Base[TCx_CMR]=0xC004;                /* Lookitup */
24  1   Base[TCx_RC]=24;                     /* Set Register C for 100Hz int */
25  1   Base[TCx_IER]=0x10;                  /* Enable RC compare interrupt */
26  1   Base[TCx_IDR]=0xEF;                  /* Disable anything else */

28  1   Base = (int *) TCBase;
29  1   Base[TC_BMR] =0x15;
30  1   Base[TC_BCR] =0;

32  1   /* Now we tell the Interrupt controller to respond to the interrupt. */
33  1   Base = (int *) AICBase;
34  1   Base[AIC_SMR6] =0x00;
35  1   Base[AIC_SVR6] = TimerVec;
36  1   Base[AIC_ICCR] = 0x40;               /* Clear int */
37  1   Base[AIC_IECR] = 0x40;               /* Enable int */

39  1   Base = (int *) (TCBase+TCChan2);
40  1   Base[TCx_CCR] =0x5;           /* Enable and start clock */

42    }
```

```
1    /*********************************************************************
2     * kernel.c                                                          *
3     *-------------------------------------------------------------------*
4     *                                                                   *
5     * This is the c part of the kernel                                  *
6     *                                                                   *
7     *  October 1998, Ray Heasman                                        *
8     *                                                                   *
9     *********************************************************************/

11   #include "miscinc.h"
12   #include "AT91Timer.h"

14   extern int CurrTask;
15   extern int *TaskList;
16   extern unsigned int *TaskSwitch;
17   extern int AddTask;
18   extern int IntID;

20   int GetPID(void)
21 1 {
22 2   if (IntID > 4096) {
23 2     return(IntID);
24 2   } else {
25 2     return( ((CurrTask - (int) &TaskList) >> 7)+1);
26 1   };
27   }
```



```
29   void Yield(void)
30   // Forces a TaskSwitch
31 1 {
32 1   FakeInt((unsigned int) &TaskSwitch);
33   }
```

```
35      void Block(void)
36      /* This:
37         Marks the current task as blocked.
38         Resets the TaskSwitch timer.
39         Forces a TaskSwitch.
40      */
41  1   {
42  1     int *tmp;
43  1   //  int *Base;

45  1     tmp = (int *) CurrTask;
46  1     *tmp = (*tmp) | 4;
47  1   /*
48  1     Base = (int *) (TCBase+TCChan2);
49  1     Base[TCx_CCR] =0x5;
50  1   */
51  1     FakeInt((unsigned int) &TaskSwitch);
52      }
```



```
54      void UnBlock(unsigned int PID)
55  1   {
56  1     int *tmp;

58  1     tmp = (int *) (((int) &TaskList)+(PID-1)*128);
59  1     *tmp = (*tmp) & 0xFFFFFFFB;
60      }
```

```
62     int AddProcess(unsigned int Vector, unsigned int Dat, unsigned int SPSpace)
63  1  {
64  1    int tmp;
65  2    __asm {
66  2      mov r0,Vector
67  2      mov r1,SPSpace
68  2      mov r4,Dat
69  2      bl  AddTask
70  2      mov tmp,r0
71  1    };
72  1    return(tmp);
73     }
```



```
76     int EInt(void)
77     /* Enable Interrupts, returning previous interrupt state
78     */
79  1  {
80  1    int tmp;

82  2    __asm {
83  2      mrs tmp,CPSR
84  2      swi 0x100
85  1    };

87  1    return(tmp & 0x80); // Return 0 for Ints were enabled.
88     }
```

```
 90    int DInt(void)
 91    /* Disable Interrupts, returning previous interrupt state
 92    */
 93  1 {
 94  1   int tmp;

 96  2   __asm {
 97  2     mrs tmp,CPSR
 98  2     swi 0x101
 99  1   };

101  1   return(tmp & 0x80);
102    }
```



```
104    int SetInt(int ReqInt)
105    /* Sets interrupt state to ReqInt (ReqInt=0 means "Enable"), and
106       returns the previous interrupt state
107    */
108  1 {
109  1   int tmp;
110  1   if (ReqInt)
111  1     tmp = DInt();
112  1   else
113  1     tmp = EInt();
114  1   return(tmp);
115    }
```

```
117      void FakeInt(unsigned int Vector)
118   1  {

120   2    __asm {
121   2      mov r0,Vector
122   2      swi 0x102,{r0}
123   1    };
124      }
```

```
1     /*****************************************************************
2      * startup.
              c                                                          *
3      *---------------------------------------------------------------*
4      *                                                               *
5      * C startup code for the QS                                     *
6      *                                                               *
7      * 12 July R. Heasman                                            *
8      *                                                               *
9      *****************************************************************/

11    #include "miscinc.h"
12    #include "msgtypes.h"

14    void C_Startup(void)
15 1  {
16 1    TSMsgHandle *InitMsg;

18 1    InitDebugHandler();
19 1    InitMessageSystem();
20 1    Init1394();
21 1    RegisterType(INITREPLY, GetPID());  // Will be used to serialise init sequence.
22 1    WaitSMsg(&InitMsg, INITREPLY);        // Hang around till Init1394  done.
23 1    SendSMsg(&InitMsg, DEALLOCSMSG);

25 1    TimerInit();
26 1    WaitSMsg(&InitMsg, INITREPLY);        // Hang around till TimerInit done.
27 1    SendSMsg(&InitMsg, DEALLOCSMSG);

29 1    RunTestSuite();
30    }
```

```
1     /*******************************************************************
2      * test.c                                                          *
3      *-----------------------------------------------------------------*
4      *                                                                 *
5      * Some test routines used during initial debugging                *
6      *                                                                 *
7      * Hopefully, this will be extended into a production line test    *
8      * suite.                                                          *
9      * October 1998                                                    *
10     *                                                                 *
11     *******************************************************************/

13    #include "miscinc.h"
14    #include "msgtypes.h"
15    #include "AT91PIO.h"
16    #include "TSBLLC.h"
17    #include <stdio.h>

19    extern QUAD MyNodeID;
20    extern QUAD NodeCount;

22    void TestILink(void)
23  1 {
24  1   TBMsgHandle *MyBMsg;
25  1   TSMsgHandle *Msg;
26  1   TTransReq *TrReq;

28  1   TTimerReq *TimeReq;
29  1   char OutBuf[40];
30  1   QUAD f, Node;

32  1   RegisterType(ILINKREPLY, GetPID());
33  1   RegisterType(TIMERREPLY, GetPID());
34  1   WaitSMsg(&Msg, ALLOCSMSG);

36  1   TimeReq = (TTimerReq *) Msg->MsgPtr;
37  1   TimeReq->ReplyAdd = TIMERREPLY;
38  1   TimeReq->Time = 12000;                        // Wait for 1.5 seconds
39  1   SendSMsg(&Msg, TIMERINPORT);
40  1   WaitSMsg(&Msg, TIMERREPLY);
41  1   SendSMsg(&Msg, DEALLOCSMSG);

43  1   SBPInit();
44  1   CSRInit();
45  1 // Now we probe the CSR ROM address space
46  1   DoHDD();
47  1 /*
48  1   for (Node=0; Node < NodeCount; Node++) {
49  1     if (Node != (MyNodeID & 0x3F)) {
50  1       sprintf(OutBuf, "Scanning Node: %u   \n", Node);
51  1       DebugMsg(OutBuf);
52  1       for (f=0; f<32; f++) {
53  1         WaitBMsg(&MyBMsg, ALLOCBMSG);

55  1         MyBMsg->UserID = ILINKREPLY;
                                                  // So transaction layer knows who to reply to.
56  1         TrReq = (TTransReq *) MyBMsg->MsgPtr;

58  1         TrReq->DestID = Node | 0xFFC0;
59  1         TrReq->TrType = 0;
60  1         TrReq->Length = 4;
61  1         TrReq->DestHi = 0xFFFF;
62  1         TrReq->DestLo = 0xF0000400+f*4;
63  1         SendBMsg(&MyBMsg, TR_DATAREQ);
64  1         WaitBMsg(&MyBMsg, ILINKREPLY);
65  1         TrReq = (TTransReq *) MyBMsg->MsgPtr;
66  1         if (TrReq->TrType != 256) {
67  1           sprintf(OutBuf,"%x -> %p\n", f*4, TrReq->Payload[0]);
68  1         } else {
69  1           sprintf(OutBuf,"%x -> Timeout!\n", f*4);
```



```
70  1         }
71  1         DebugMsg(OutBuf);
72  1         SendBMsg(&MyBMsg, DEALLOCBMSG);
73  1       }
74  1     } else {
75  1       sprintf(OutBuf, "Skipping Myself.\n");
76  1       DebugMsg(OutBuf);
77  1     }
78  1   }
79  1   DebugMsg("Scan done.      \n");
80  1 */
81    }
```

```
83      void MsgPassTest(void)
84   1  {
85   1    TSMsgHandle   *MySMsg;
86   1  //  TBMsgHandle *MyBMsg;

88   1   //  RegisterType(2);
89   1  //  WaitBMsg(&MyBMsg, ALLOCBMSG);
90   1    RegisterType(3, GetPID());            /* register small message ID */
91   1    WaitSMsg(&MySMsg, ALLOCSMSG); /* get a message to fill in */
92   1    SendSMsg(&MySMsg, 3);
93   1    WaitSMsg(&MySMsg, 3);
94   1    SendSMsg(&MySMsg, DEALLOCSMSG); /* Give it back */
95      }
```



```
97      void LEDOn(int LEDNum)
98      /* Switch on LED 0 through 3 */
99   1  {

101  1  #define LED0 (1<<28)

103  1    int *Base;
104  1    Base = (int *) PIOBase;
105  2    if (LEDNum < 3) {
106  2      Base[PIO_SODR]=LED0 << LEDNum;
107  1    };
108  2    if (LEDNum == 3) {
109  2      Base[PIO_SODR]=0x2000000;
110  1    };
111     }
```

```
113      void LEDToggle(int LEDNum)
114      // Toggle LED 0 through 3
115  1   {
116  1     int *Base;
117  1     QUAD tmp,Lmask;

119  1     Base = (int *) PIOBase;
120  1     if (LEDNum < 3)
121  1       Lmask = LED0 << LEDNum;
122  1     else
123  1       Lmask = 0x2000000;
124  1     tmp = Base[PIO_ODSR] & Lmask;
125  1     if (tmp)
126  1       Base[PIO_CODR] = Lmask;
127  1     else
128  1       Base[PIO_SODR] = Lmask;
129      }
```



```
131      void LEDOff(int LEDNum)
132      /* Switch off LED 0 through 3 */
133  1   {

135  1     int *Base;
136  1     Base = (int *) PIOBase;
137  2     if (LEDNum < 3) {
138  2       Base[PIO_CODR]=LED0 << LEDNum;
139  1     };
140  2     if (LEDNum == 3) {
141  2       Base[PIO_CODR]=0x2000000;
142  1     };
143      }
```

```
145     int HackRand(void)
146     // Return a value from 0-255
147  1  {
148  1    QUAD tmp;

150  1    tmp = ReadLLC(LLCCycTim);

152  1    return((tmp ^ (tmp >> 8)) & 0xFF);
153     }
```



```
155     void WaitRand(int Addr)
156     // Wait for a random time (up to 5 seconds)
157     // Use Addr as the timer reply port
158  1  {
159  1    TTimerReq *TimeReq;
160  1    TSMsgHandle *Msg;

162  1    WaitSMsg(&Msg, ALLOCSMSG);

164  1    TimeReq = (TTimerReq *) Msg->MsgPtr;
165  1    TimeReq->ReplyAdd = Addr;
166  1    TimeReq->Time = 156*HackRand();                // Wait for random time up to 5s
167  1    SendSMsg(&Msg, TIMERINPORT);

169  1    WaitSMsg(&Msg, Addr);
170  1    SendSMsg(&Msg, DEALLOCSMSG);
171     }
```

```
173     void Ph(int PhNum)
174  1  {
175  1    TSMsgHandle *MyLeftChopstick;
176  1    TSMsgHandle *MyRightChopstick;
177  1    TSMsgHandle *MyTemp;

179  1    int LC, RC;   /* Addresses to give chopsticks to */

181  1    char *foo = "Philosopher x is eating.\n";

183  1    /* Figure out where our messages are going to */
184  2    if (PhNum > 1) {
185  2      LC = (PhNum*8)-7;
186  2    } else {
187  2      LC = (6*8)-7;
188  1    };

190  2    if (PhNum < 5) {
191  2      RC = (PhNum*8)+7;
192  2    } else {
193  2      RC = (0*8)+7;
194  1    };

196  1    /* We have to register for a chopstick from the left and right */
197  1    RegisterType((PhNum*8)-1, GetPID());  /* From left */
198  1    RegisterType((PhNum*8)+1, GetPID());  /* From right */
199  1    RegisterType((PhNum*8)+3, GetPID());  /* Timer reply port */

201  1    /* Tell the waiter I'm ready */

203  1    WaitSMsg(&MyTemp, ALLOCSMSG);
204  1    SendSMsg(&MyTemp, 1);

206  1    /* Now we begin */

208  2    while (1) {
209  2      /* Think */

211  2      WaitRand((PhNum*8)+3);

213  2      /* Scrabble for chopsticks */

215  3      if (PhNum == 1) {
216  3        WaitSMsg(&MyRightChopstick, (PhNum*8)+1);
217  3        WaitSMsg(&MyLeftChopstick , (PhNum*8)-1);
218  3      } else {
219  3        WaitSMsg(&MyLeftChopstick , (PhNum*8)-1);
220  3        WaitSMsg(&MyRightChopstick, (PhNum*8)+1);
221  2      };


224  2      /* Eat */

226  2      foo[12] = PhNum+48;
227  2      DebugMsg(foo);
228  2      WaitRand((PhNum*8)+3);

230  2      /* give back chopsticks */

232  2      MyLeftChopstick->MsgPtr->SMsg[0]=PhNum;
233  2      SendSMsg(&MyLeftChopstick , LC);
234  2      MyRightChopstick->MsgPtr->SMsg[0]=PhNum;
235  2      SendSMsg(&MyRightChopstick, RC);
236  1    };
237     }
```



```
240     void DoPhilosophers(void)
241  1  {
242  1    int i;

244  1    TSMsgHandle *MyTemp;

246  1    /* Open a comms channel */
247  1    RegisterType(1, GetPID());

249  1    /* Create 5 philosophers with 1K stack space */
250  1    for (i=1; i<=5; i++) { AddProcess((unsigned int) &Ph, i, 1024); };

252  1    /* Wait for them to say they're ready */
253  2    for (i=1; i<=5; i++) {
254  2      WaitSMsg(&MyTemp,1);
255  2      SendSMsg(&MyTemp,DEALLOCSMSG);      /* Discard the messages */
256  1    };

258  1    /* Set the table */

260  1    WaitSMsg(&MyTemp, ALLOCSMSG);
261  1    SendSMsg(&MyTemp, (1*8)-1);

263  1    WaitSMsg(&MyTemp, ALLOCSMSG);
264  1    SendSMsg(&MyTemp, (1*8)+1);           /* 1st philo gets 2 chopsticks */

266  1    WaitSMsg(&MyTemp, ALLOCSMSG);
267  1    SendSMsg(&MyTemp, (3*8)-1);

269  1    WaitSMsg(&MyTemp, ALLOCSMSG);
270  1    SendSMsg(&MyTemp, (3*8)+1);           /* 3rd philo gets 2 chopsticks */

272  1    WaitSMsg(&MyTemp, ALLOCSMSG);
273  1    SendSMsg(&MyTemp, (4*8)+1);           /* 1 chopstick between 4 and 5 */

275     }
```

```
277     void TimerTest()
278 1   {
279 1     TSMsgHandle *Msg;
280 1     QUAD  t1;
281 1     QUAD  t2;
282 1     QUAD  t3;
283 1     TTimerReq *TimeReq;

285 1     RegisterType(TIMERREPLY, GetPID());
286 1     WaitSMsg(&Msg, ALLOCSMSG);

288 1     TimeReq = (TTimerReq *) Msg->MsgPtr;
289 1     TimeReq->ReplyAdd = TIMERREPLY;
290 1     TimeReq->Time = 80000;                          // Wait for 10 seconds
291 1     SendSMsg(&Msg, TIMERINPORT);
292 1   //  WaitSMsg(&Msg, TIMERREPLY);

294 1     WaitSMsg(&Msg, ALLOCSMSG);
295 1     TimeReq = (TTimerReq *) Msg->MsgPtr;
296 1     TimeReq->ReplyAdd = TIMERREPLY;
297 1     TimeReq->Time = 15000;
298 1     SendSMsg(&Msg, TIMERINPORT);

300 1     WaitSMsg(&Msg, ALLOCSMSG);
301 1     TimeReq = (TTimerReq *) Msg->MsgPtr;
302 1     TimeReq->ReplyAdd = TIMERREPLY;
303 1     TimeReq->Time = 1000;
304 1     t1 = ReadLLC(LLCCycTim) >> 12;
305 1     SendSMsg(&Msg, TIMERINPORT);
306 1     WaitSMsg(&Msg, TIMERREPLY);
307 1     t2 = ReadLLC(LLCCycTim) >> 12;

309 1     SendSMsg(&Msg, DEALLOCSMSG);
310 1     WaitSMsg(&Msg, TIMERREPLY);
311 1     SendSMsg(&Msg, DEALLOCSMSG);

313 1     WaitSMsg(&Msg, TIMERREPLY);
314 1     SendSMsg(&Msg, DEALLOCSMSG);

316 1     t3 = ReadLLC(LLCCycTim) >> 12;
317 1     t3 = t1 + t2 + t3;
318     }
```



```
320     void RunTestSuite(void)
321 1   {
322 1   //  TimerTest();
323 1       TestILink();
324 1   //  MsgPassTest();
325 1   //  DoPhilosophers();
326 2     while (1) {
327 2       LEDOff(0);
328 2       LEDOff(1);
329 2       LEDOff(2);
330 2       LEDOff(3);    /* This cos we have no place to return to */
331 2       Yield();
332 1     };
333     }
```

```
/*********************************************************************
 * timer.c                                                           *
 *-------------------------------------------------------------------*
 *                                                                   *
 *  Timer server                                                     *
 *                                                                   *
 *  May 1998, Ray Heasman                                            *
 *                                                                   *
 *********************************************************************/

#include "AT91timer.h"          /* Get Timer related constants and
                                   addresses */
#include "AT91AIC.h"
#include "miscinc.h"
#include "msgtypes.h"
#include "intIDs.h"
#include "TSBLLC.h"
#include <stdio.h>

// #define      TIMERDEBUG

// I'm trying to standardise things:
// Call TimerInit, and it will start any necessary tasks.
// The main loop is TimerLoop and will run continuously

// How to use it:
// Send a Message to TIMERINPORT, with the following format:
// QUAD ReturnAdd;
// QUAD RelTimeOffset;
// Return Address is the place to send messages to.
// RelTimeOffset is the offset from now to whenever, measured in 8000ths
// of a second.

extern int IntID;
extern int IntVecTimer1Asm;
extern QUAD Uptime;
void TimerLoop(void);

#define MAX_OTRS        100     // Max num outstanding Timer reqs

typedef struct {
  MADDR ReplyAdd;
  TTime Time;
  int   Last;
  int   Next;
} TimerEntry;

TimerEntry TEBlock[100];

int TimerHead;
int TimerFree;
int Ticking;            // Set to 1 if interrupt is just biding its time till
                        // the outstanding time is low enough to fit in a 16
                        // bit counter.

void TimerInit(void)
// Start up any necessary tasks to run a timer loop.
{
  int i;

  Ticking = 0;
  TimerHead = -1;
  TimerFree = 0;

  for (i = 0; i < MAX_OTRS; i++) {
    TEBlock[i].Next = i+1;
    TEBlock[i].Last = i-1;
  };
  TEBlock[MAX_OTRS-1].Next = -1;
```



```
  AddProcess((unsigned int) &TimerLoop, 0, 4096);
}
```

```
 74     int CompareTime(TTime Time1, TTime Time2)
 75     // Returns 0 if equal, -1 if Time1 < Time2, 1 if Time1 > Time2
 76  1  {
 77  1    if (Time1.Hi < Time2.Hi) return(-1);
 78  1    if (Time1.Hi > Time2.Hi) return( 1);

 80  1    // Get here if .Lo is significant
 81  1    if (Time1.Lo < Time2.Lo) return(-1);
 82  1    if (Time1.Lo > Time2.Lo) return( 1);
 83  1    return(0);
 84     }
```



```
 86     TTime AddTime(TTime Time1, TTime Time2)
 87  1  {
 88  1    QUAD Carry;
 89  1    TTime Result;

 91  1    Carry = 0;
 92  1    Result.Lo = Time1.Lo + Time2.Lo;      // Add LSBs
 93  1    // Check for overflow. ie. Carry
 94  2    if (Result.Lo >= 8000) {
 95  2      Carry = 1;
 96  2      Result.Lo = Result.Lo - 8000;
 97  1    };
 98  1    Result.Hi = Time1.Hi + Time2.Hi + Carry;
 99  1    // if the MSBs overflow, you deserve what you get

101  1    return(Result);
102     }
```

```
104     TTime SubTime(TTime Time1, TTime Time2)
105     // Subtract Time2 from Time1
106     // Doesnt work if the result is negative
107  1  {
108  1    QUAD Carry;
109  1    TTime Result;

111  2    if (CompareTime(Time1,Time2) < 0) {
112  2      DebugMsg("SubTime: Attempt to subtract Times resulting in negative value.\n");
113  2      Result.Hi = 0;
114  2      Result.Lo = 0;
115  2      return(Result);
116  1    };

118  1    Carry = 0;

120  1    Result.Lo = Time1.Lo - Time2.Lo;

122  1    // Check for underflow. ie. Carry
123  2    if (Time2.Lo > Time1.Lo) {
124  2      Carry = 1;
125  2      Result.Lo = (Time1.Lo+8000)-Time2.Lo;
126  1    };
127  1    Result.Hi = Time1.Hi - Time2.Hi - Carry;

129  1    return(Result);
130     }
```



```
132     TTime GetCycTime(void)
133     // Fetch the absolute time from the FireWire chipset
134  1  {
135  1    TTime Tmp;
136  1    Tmp.Hi = Uptime;
137  1    Tmp.Lo = (ReadLLC(LLCCycTim)>>12)&0x1fff;
138  2    if (Uptime != Tmp.Hi) {
139  2      // if the timer rolled over, we have to reread everything
140  2      Tmp.Hi = Uptime;
141  2      Tmp.Lo = (ReadLLC(LLCCycTim)>>12)&0x1fff;
142  1    };
143  1    return(Tmp);
144     }
```

```
146      int AddAlarm(MADDR RAddr, QUAD RelAlarm)
147      // Find correct place in Alarm list and insert alarm.
148      // Only ever call this if there are Free slots in the timer queue

150  1   {
151  1     TTime Tmp;
152  1     TTime Alarm;
153  1     int i, lasti;
154  1     int newnode;
155  1     #ifdef TIMERDEBUG
156  1       char DMsgBuf[100];
157  1       char *DMsg;
158  1       DMsg = (char *) DMsgBuf;
159  1     #endif

161  1     if (TimerFree == -1) DebugMsg("AddAlarm: Attempt to add alarm - no free slots.\n");

163  1     Tmp.Hi = RelAlarm/8000;
164  1     Tmp.Lo = RelAlarm%8000;

166  1     Alarm = AddTime(GetCycTime(), Tmp);

168  1     #ifdef TIMERDEBUG
169  1        sprintf(DMsg,"AddAlarm, adding: %u.%u\n", Alarm.Hi, Alarm.Lo);
170  1        DebugMsg(DMsg);
171  1     #endif

173  1     i = TimerHead; lasti = -1;
174  2     while ( (i != -1) && (CompareTime(TEBlock[i].Time, Alarm) < 1) ) {
175  2       // Use '<=' rather than '<' to make requests for the same time FIFO
176  2       lasti = i;
177  2       i = TEBlock[i].Next;
178  1     };
179  1     // Right. Now we can insert the alarm at the correct place (before i)

181  1     // Now follows a hopefully really easy to read list insertion.
182  2     if ((i != TimerHead) && (i != -1)) {
183  2       // Middle of list
184  2       newnode = TimerFree;
185  2       TimerFree = TEBlock[TimerFree].Next;
186  2       if (TimerFree != -1) TEBlock[TimerFree].Last = -1;

188  2       // set up new node
189  2       TEBlock[newnode].Last = lasti;
190  2       TEBlock[newnode].Next = i;
191  2       TEBlock[newnode].Time = Alarm;
192  2       TEBlock[newnode].ReplyAdd = RAddr;

194  2       // link it in
195  2       TEBlock[lasti].Next = newnode;
196  2       TEBlock[i].Last = newnode;
197  2     } else {
198  3       if (lasti == -1) {
199  3         // Head of list
200  3         newnode = TimerFree;
201  3         TimerFree = TEBlock[TimerFree].Next;
202  3         if (TimerFree != -1) TEBlock[TimerFree].Last = -1;


205  3         TEBlock[newnode].Last = -1;
206  3         TEBlock[newnode].Next = TimerHead;
207  3         TEBlock[newnode].Time = Alarm;
208  3         TEBlock[newnode].ReplyAdd = RAddr;

210  3         // link it in
211  3         TimerHead = newnode;
212  3       } else {
213  3         // End of list
214  3         newnode = TimerFree;
```



```
215  3         TimerFree = TEBlock[TimerFree].Next;
216  3         if (TimerFree != -1) TEBlock[TimerFree].Last = -1;

218  3         TEBlock[newnode].Last = lasti;
219  3         TEBlock[newnode].Next = -1;
220  3         TEBlock[newnode].Time = Alarm;
221  3         TEBlock[newnode].ReplyAdd = RAddr;

223  3         TEBlock[lasti].Next = newnode;
224  2       }
225  1     }
226      }
```

```
228     MADDR DelAlarm(TTime CurrTime)
229 1   {
230 1     int tmp;
231 1     MADDR RAddr;
232 1     #ifdef TIMERDEBUG
233 1       char DMsgBuf[100];
234 1       char *DMsg;
235 1       DMsg = (char *) DMsgBuf;
236 1     #endif

238 1     #ifdef TIMERDEBUG
239 1        sprintf(DMsg,"DelAlarm: Current time: %u.%u\n",CurrTime.Hi,CurrTime.Lo);
240 1        DebugMsg(DMsg);
241 1      #endif

243 2     if (CompareTime(TEBlock[TimerHead].Time, CurrTime) == 1) {
244 2       DebugMsg("DelAlarm: DelAlarm called before event is due.\n");
245 2       return(0);
246 1     };

248 2     if (TimerHead == -1) {
249 2       DebugMsg("DelAlarm: Attempt to remove item from empty list.\n");
250 2       return(0);
251 1     };

253 1     RAddr = TEBlock[TimerHead].ReplyAdd;

255 1     tmp = TimerHead;
256 1     TimerHead = TEBlock[TimerHead].Next;
257 1     if (TimerHead != -1) TEBlock[TimerHead].Last = -1;

259 1     TEBlock[TimerFree].Last = tmp;
260 1     TEBlock[tmp].Next = TimerFree;
261 1     TimerFree = tmp;

263 1     return(RAddr);
264     }
```



```
266     void TimerLoop(void)
267 1   {
268 1     int   RepAddr;
269 1     TTime CurrTime;
270 1     TSMsgHandle *TmpSMsg;
271 1     TTimerReq *TimeReq;
272 1     #ifdef TIMERDEBUG
273 1       char DMsgBuf[100];
274 1       char *DMsg;
275 1       TTime DTime;
276 1       DMsg = (char *) DMsgBuf;
277 1     #endif

279 1     RegisterType(TIMERINPORT, GetPID());

281 1     WaitSMsg(&TmpSMsg, ALLOCSMSG);
282 1     SendSMsg(&TmpSMsg, INITREPLY);        // Tell the init sequence to carry on.

284 1     DebugMsg("Timer server started.\n");

286 1     #ifdef TIMERDEBUG
287 1       DebugMsg("Timer debugging active.\n");
288 1     #endif

290 2     while (1) {
291 2       Yield();
292 3       while (CheckSMsg(TIMERINPORT) == 0) {
293 4         if (TimerFree != -1) {
294 4           // Handle request to add alarm to queue
295 4           WaitSMsg(&TmpSMsg, TIMERINPORT);
296 4           TimeReq = (TTimerReq *) TmpSMsg->MsgPtr;
297 4           #ifdef TIMERDEBUG
298 4             DTime = GetCycTime();
299 4             sprintf(DMsg,"Current time is: %u.%u\n", DTime.Hi, DTime.Lo);
300 4             DebugMsg(DMsg);
301 4             sprintf(DMsg,"Adding alarm offset by: %u\n", TimeReq->Time);
302 4             DebugMsg(DMsg);
303 4           #endif
304 4           AddAlarm(TimeReq->ReplyAdd, TimeReq->Time);
305 4           SendSMsg(&TmpSMsg, DEALLOCSMSG);
306 3         };
307 2       };
308 3       if (TimerHead != -1) {
309 3         CurrTime = GetCycTime();
310 4         while ((TimerHead != -1) && (CompareTime(TEBlock[TimerHead].Time, CurrTime) < 1)) {
311 4          // Send a wakeup to a task
312 4           #ifdef TIMERDEBUG
313 4             sprintf(
                   DMsg,"Wakeup at: %u.%u\n", TEBlock[TimerHead].Time.Hi, TEBlock[TimerHead].Time.Lo);
314 4             DebugMsg(DMsg);
315 4           #endif

317 4           CurrTime = GetCycTime();
318 4           // Next =, and not ==, is intentional
319 5           if (RepAddr = DelAlarm(CurrTime)) {
320 5           // DelAlarm will return a non zero value if all is ok
321 5           // otherwise it will generate debug output and refuse to del the event.
322 5             WaitSMsg(&TmpSMsg, ALLOCSMSG);
323 5             SendSMsg(&TmpSMsg, RepAddr);
324 4           };
325 3         };
326 2       };
327 1     };
328     }
```

```
1   /*
2    |  AT91M40400 Advanced Interrupt Controller constants
3    |  Note that all offsets are used as offsets into an integer array in C
4    |  This means you have to MULTIPLY THEM BY FOUR when using them in
5    |  assembler programs.
6   */

8   /* Base address is here */

10  #define AICBase 0xFFFFF000

12  /* Source mode Register Offsets */
13  #define AIC_SMR0 0
14  #define AIC_SMR1 1
15  #define AIC_SMR2 2
16  #define AIC_SMR3 3
17  #define AIC_SMR4 4
18  #define AIC_SMR5 5
19  #define AIC_SMR6 6
20  #define AIC_SMR7 7
21  #define AIC_SMR8 8
22  #define AIC_SMR9 9
23  #define AIC_SMR10 10
24  #define AIC_SMR11 11
25  #define AIC_SMR12 12
26  #define AIC_SMR13 13
27  #define AIC_SMR14 14
28  #define AIC_SMR15 15
29  #define AIC_SMR16 16
30  #define AIC_SMR17 17
31  #define AIC_SMR18 18
32  #define AIC_SMR19 19
33  #define AIC_SMR20 20
34  #define AIC_SMR21 21
35  #define AIC_SMR22 22
36  #define AIC_SMR23 23
37  #define AIC_SMR24 24
38  #define AIC_SMR25 25
39  #define AIC_SMR26 26
40  #define AIC_SMR27 27
41  #define AIC_SMR28 28
42  #define AIC_SMR29 29
43  #define AIC_SMR30 30
44  #define AIC_SMR31 31

46  /* Source Vector Registers */
47  #define AIC_SVR0 32
48  #define AIC_SVR1 33
49  #define AIC_SVR2 34
50  #define AIC_SVR3 35
51  #define AIC_SVR4 36
52  #define AIC_SVR5 37
53  #define AIC_SVR6 38
54  #define AIC_SVR7 39
55  #define AIC_SVR8 40
56  #define AIC_SVR9 41
57  #define AIC_SVR10 42
58  #define AIC_SVR11 43
59  #define AIC_SVR12 44
60  #define AIC_SVR13 45
61  #define AIC_SVR14 46
62  #define AIC_SVR15 47
63  #define AIC_SVR16 48
64  #define AIC_SVR17 49
65  #define AIC_SVR18 50
66  #define AIC_SVR19 51
67  #define AIC_SVR20 52
68  #define AIC_SVR21 53
69  #define AIC_SVR22 54
70  #define AIC_SVR23 55
```



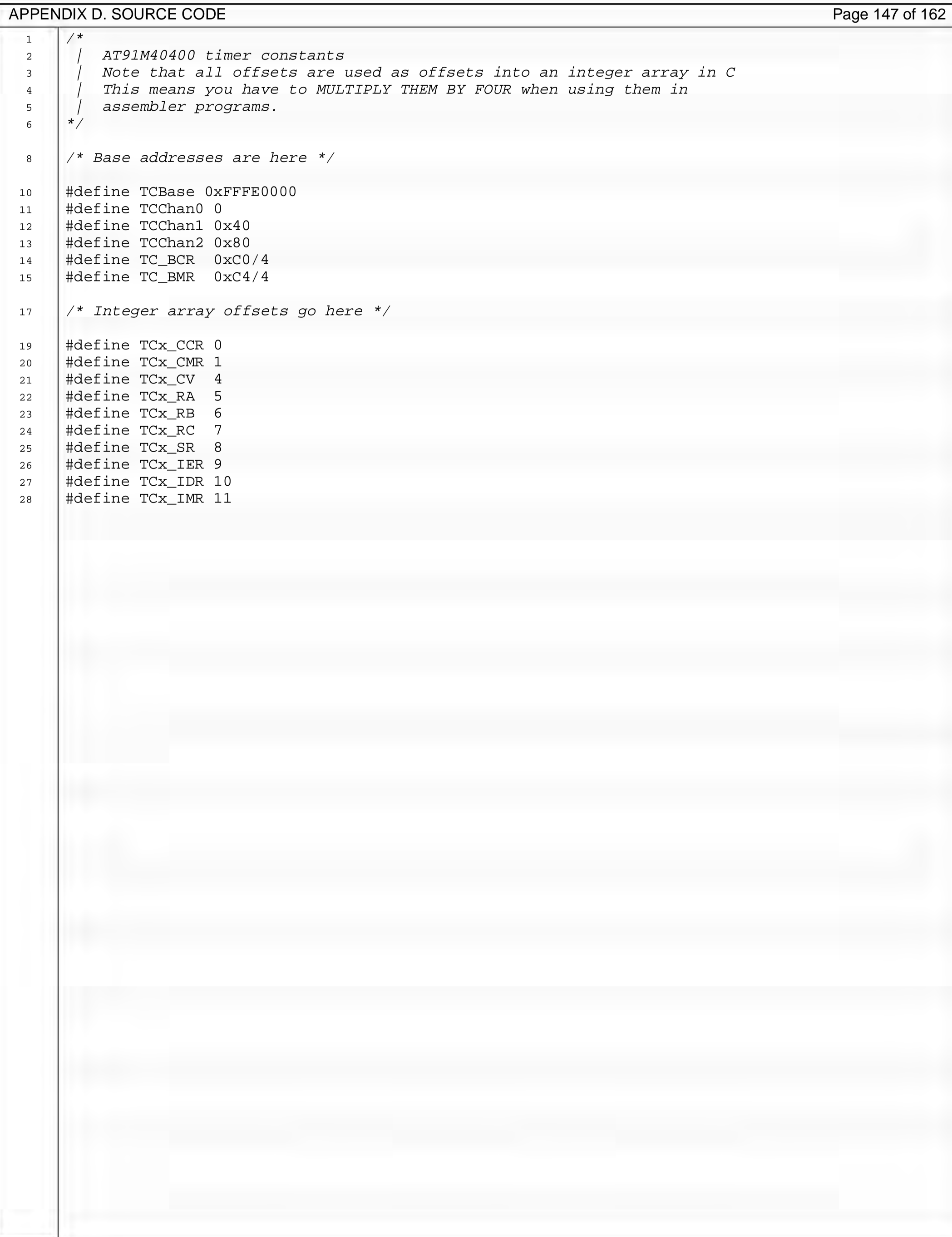
```
71  #define AIC_SVR24 56
72  #define AIC_SVR25 57
73  #define AIC_SVR26 58
74  #define AIC_SVR27 59
75  #define AIC_SVR28 60
76  #define AIC_SVR29 61
77  #define AIC_SVR30 62
78  #define AIC_SVR31 63

80  #define AIC_IVR 64
81  #define AIC_FVR 65
82  #define AIC_ISR 66
83  #define AIC_IPR 67
84  #define AIC_IMR 68
85  #define AIC_CISR 69

87  #define AIC_IECR 72
88  #define AIC_IDCR 73
89  #define AIC_ICCR 74
90  #define AIC_ISCR 75
91  #define AIC_EOICR 76
92  #define AIC_SIV 77
```

```
1  /*
2   |  AT91M40400 Programmable Input/Output constants
3   |  Note that all offsets are used as offsets into an integer array in C
4   |  This means you have to MULTIPLY THEM BY FOUR when using them in
5   |  assembler programs.
6  */

8  /* Base address is here */

10 #define PIOBase 0xFFFF0000

12 /* Source mode Register Offsets */

14 #define PIO_PER         0
15 #define PIO_PDR         1
16 #define PIO_PSR         2

18 #define PIO_OER         4
19 #define PIO_ODR         5
20 #define PIO_OSR         6

22 #define PIO_IFER        8
23 #define PIO_IFDR        9
24 #define PIO_IFSR        10

26 #define PIO_SODR        12
27 #define PIO_CODR        13
28 #define PIO_ODSR        14
29 #define PIO_PDSR        15
30 #define PIO_IER         16
31 #define PIO_IDR         17
32 #define PIO_IMR         18
33 #define PIO_ISR         19
```

```
1  /*
2   |  AT91M40400 Advanced Interrupt Controller constants
3   |  Note that all offsets are used as offsets into an integer array in C
4   |  This means you have to MULTIPLY THEM BY FOUR when using them in
5   |  assembler programs.
6  */

8  /* Base address is here */

10 #define RAAICBase 0xFFFFC000

12 #define IR0_SOU 0
13 #define IR0_ALO 1
14 #define IR0_MRE 2
15 #define IR0_MRD 3
16 #define IR0_SWI 4
17 #define IR0_ICR 5
18 #define IR0_ISR 8
19 #define IR0_IVR 9
20 #define IR0_P00 12
21 #define IR0_P01 13
22 #define IR0_P02 14
23 #define IR0_P03 15
24 #define IR0_P04 16
25 #define IR0_P05 17
26 #define IR0_P06 18
27 #define IR0_P07 19
28 #define IR0_P08 20
29 #define IR0_P09 21
30 #define IR0_P10 22
31 #define IR0_P11 23
32 #define IR0_P16 28
33 #define IR0_P17 29
34 #define IR0_P18 30
```

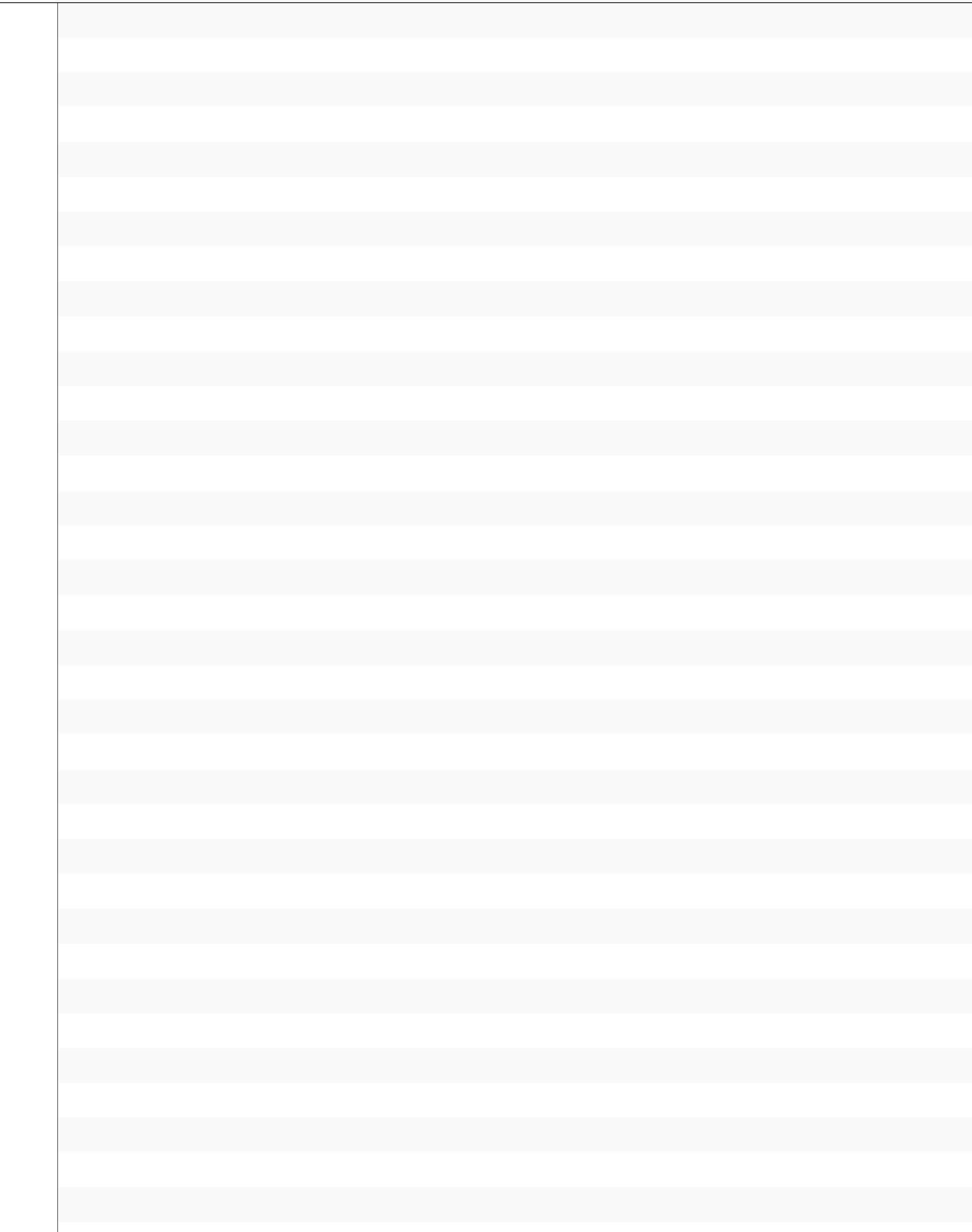
```
1   /*
2    |  AT91M40400 Special Function Controller constants
3    |  Note that all offsets are used as offsets into an integer array in C
4    |  This means you have to MULTIPLY THEM BY FOUR when using them in
5    |  assembler programs.
6   */

8   #define SFBase 0xFFF00000

10  #define SF_CIDR 0
11  #define SF_EXID 1
12  #define SF_RSR  2

14  // Reserved entries here (3-5)

16  #define SF_PMR  6
```

```
/*
 |  AT91M40400 timer constants
 |  Note that all offsets are used as offsets into an integer array in C
 |  This means you have to MULTIPLY THEM BY FOUR when using them in
 |  assembler programs.
*/

/* Base addresses are here */

#define TCBase 0xFFFE0000
#define TCChan0 0
#define TCChan1 0x40
#define TCChan2 0x80
#define TC_BCR  0xC0/4
#define TC_BMR  0xC4/4

/* Integer array offsets go here */

#define TCx_CCR 0
#define TCx_CMR 1
#define TCx_CV  4
#define TCx_RA  5
#define TCx_RB  6
#define TCx_RC  7
#define TCx_SR  8
#define TCx_IER 9
#define TCx_IDR 10
#define TCx_IMR 11
```

```
1     /*********************************************************************
2      * ORBstructs.h                                                      *
3      *-------------------------------------------------------------------*
4      *                                                                   *
5      *  Central list of ORB structures                                   *
6      *                                                                   *
7      *  December 1999, Ray Heasman                                       *
8      *                                                                   *
9      *********************************************************************/
10    #define UCHAR unsigned char

12  1 typedef struct {
13  1   QUAD L1;
14  1   QUAD L2;
15    } SBPAddr;

17  1 typedef struct {
18  1   SBPAddr Pass;
19  1   SBPAddr LoginResp;    // 2 QUADs
20  1   QUAD LUNDat;
21  1   QUAD Lengths;
22  1   SBPAddr StatusFIFO;
23    } LoginManORB;
```

```
/*
 |  TSB12LV31 Link Layer Controller constants
 |
 |  Seeing as TI do not provide their own short names for these registers,
 |  I am using 'LLC' followed by the first 4 letters of the register's name,
 |  if the register has only one word in its name. Otherwise I use the
 |  first three letters of each word appended onto 'LLC', except for
 |  words that are already shortened or are abbreviations, which are not
 |  shortened.
 |
 |  Ray Heasman, Oct 1998
*/

#define LLCVers         0x00
#define LLCMISC         0x04
#define LLCCont         0x08
#define LLCInte         0x0C
#define LLCIntMas       0x10
#define LLCCycTim       0x14
#define LLCIsoNum       0x18
#define LLCDiag         0x20
#define LLCPhyAcc       0x24
#define LLCATFSta       0x30
#define LLCBusRes       0x34
#define LLCSelIDChe     0x38
#define LLCGRFSta       0x3C
#define LLCFIFOSta      0x50
#define LLCIsoCon       0x54
#define LLCIsoMod       0x58
#define LLCIsoHea       0x5C

// The bitmask definitions for the Interrupt register
// These are taken, case and all, from the LLC datasheet

#define IArbFl          0x00000001
#define FrGp            0x00000010
#define ArbGp           0x00000020
#define CArbFl          0x00000040
#define CyLst           0x00000080
#define CyPnd           0x00000100
#define CyDne           0x00000200
#define CySt            0x00000400
#define CySec           0x00000800
#define McWrEr          0x00001000
#define TCErr           0x00008000
#define HdrEr           0x00010000
#define SntRj           0x00020000
#define ATStk           0x00080000
#define ITStk           0x00100000
#define SelfIDEr        0x00200000
#define IRXDta          0x00400000
#define CmdRst          0x01000000
#define ARxDta          0x02000000
#define TxRdy           0x04000000
#define PhRst           0x10000000
#define PhRRx           0x20000000
#define PhInt           0x40000000
#define LLCInt          0x80000000      // Had to change this one (was Int)
```

```
1  /* errors.h */

3  #define ERROR_NoFreeSmallIDSlots              1
4  #define ERROR_NoFreeSLLNode                   2
5  #define ERROR_MsgTypeAlreadyRegistered        3
6  #define ERROR_NoFreeBigIDSlots                4
7  #define ERROR_NoFreeBLLNode                   5
8  #define ERROR_CatchallNotImplementedYet       6
9  #define ERROR_UnreachableCodeReached          7
10 #define ERROR_NotSmallMsg                     8
11 #define ERROR_NotBigMsg                       9
12 #define ERROR_InvalidNode                     10
13 #define ERROR_MsgNotInCorrectMemArea          11
14 #define ERROR_MsgAddressWrong                 12
15 #define ERROR_NoMsgFound                      13
16 #define ERROR_MsgNotRegistered                14
17 #define ERROR_MsgNotHereYet                   15
18 #define ERROR_LLCMemInterfacePoked            16
19 #define ERROR_LLCRXedUnknownTCode             17
20 #define ERROR_NotYourMessage                  18
```

```
1   /*********************************************************************
2    * intIDs.h                                                          *
3    *-------------------------------------------------------------------*
4    *                                                                   *
5    *  Central list of interrupts                                       *
6    *                                                                   *
7    *  April 1999, Ray Heasman                                          *
8    *                                                                   *
9    *********************************************************************/

11  // IntID for 1394 Interrupt

13  #define MAIN1394Int 4097
14  #define Timer1Int   4098        // For timer server
```

```
/*********************************************************************
 * miscinc.h                                                         *
 *-------------------------------------------------------------------*
 *                                                                   *
 * The prototype for all the stuff used in the kernel and            *
 * associated routines.                                              *
 *                                                                   *
 * October 1998                                                      *
 *                                                                   *
 *********************************************************************/

/* Global stuff */
#define IntIDBASE 4096

typedef unsigned int QUAD;
typedef unsigned int MADDR;

/* msgpass.c */

#define GETBMSGMASK 1<<28

#define GETSMSGMASK ((1<<28)+1)

#define NUM_BIGMSG 32           /* Number of Large Message buffers */
#define BIGMSG_SIZE 512         /* Size in bytes */

#define NUM_SMALLMSG  128       /* Number of Small Message buffers */
#define SMALLMSG_SIZE  8        /* Size in bytes */

#define MAXNUM_SID 100          /* Max number of registered small msg IDs */
#define MAXNUM_BID 100          /* Max number of registered large msg IDs */

#define SMALLMSGMASK 1          /* Small messages have bit 0 set in their
                                   message IDs                          */

/* A large Message */
typedef struct {
  char BMsg[BIGMSG_SIZE];
} TBigMsg;

/* A small Message */
typedef struct {
    char SMsg[SMALLMSG_SIZE];
} TSmallMsg;

typedef struct {
  TBigMsg *MsgPtr;      /* Pointer to a system message or a provided message   */
  int Node;
  int UserID;           /* this can be used as a reply address by processes    */
  int PassByRef;        /* If non-zero, then the pointed-to area contains a    */
                        /* pointer and a length                                */
} TBMsgHandle;

typedef struct {
  TSmallMsg *MsgPtr;    /* Pointer to a system message or a provided message   */
  int Node;
  int UserID;           /* this can be used as a reply address by processes    */
  int PassByRef;
} TSMsgHandle;

// Return values are in errors.h. 0 = success
void InitMessageSystem(void);
// Call this once, before using message passing.
int RegisterType(unsigned int MsgID, unsigned int PID);
// Register process PID to receive messages of type MsgID
int WaitSMsg(TSMsgHandle **Msg, unsigned int MsgID);
// Blocking read for message of type MsgID. Call with ALLOCSMSG to create a
// message and init the handle.
int CheckSMsg(unsigned int MsgID);
// Non-blocking check for incoming message. 0 = message waiting
```



```
int CheckBMsg(unsigned int MsgID);
int SendSMsg(TSMsgHandle **Msg, unsigned int MsgID);
// Non-blocking send of message of type MsgID. Use DEALLOCSMSG to bin it.
int SendBMsg(TBMsgHandle **Msg, unsigned int MsgID);
// As above. Use DEALLOCBMSG
int WaitBMsg(TBMsgHandle **Msg, unsigned int MsgID);
unsigned int WaitOnMsgList(unsigned int *MList);
// Blocking wait on zero terminated list of message types (big or small).
// Returns the offset into the array of an arrived message type. Do not
// make assumptions about the relative arrival order of messages.

/* kernel.c */
int  GetPID(void);
// Return PID of current process
void Block(void);
// Sleep until a message arrives for which the current process is registered
void Yield(void);
// Give up remainder of current timeslice
void UnBlock(unsigned int PID);
// UnBlock the designated process (Used by message passing handler)
int  EInt(void);
// Enable interrupts
int  DInt(void);
// Disable interrupts
int  SetInt(int ReqInt);
// Set interrupts to desired value, returning previous state. (0 = enable)
void FakeInt(unsigned int Vector);
// Fake an interrupt. Call Vector in interrupt state
int AddProcess(unsigned int Vector, unsigned int Dat, unsigned int TaskSP);
// Add process at Vector, with 32 bits to be placed in r0 in Dat, and amount of
// stack space in TaskSP

/* ilink.c */
void WriteLLC(unsigned int Reg, unsigned int Val);
// Write Val to Register Reg, on the TI LLC
void WriteLLCMasked(QUAD Reg, QUAD Mask, QUAD Value);
// Write Value to Reg, but only change the bits in Mask. Leave 0's untouched.
unsigned int ReadLLC(unsigned int Reg);
// Read value from a register
int Init1394(void);
// Start 1394 Physical and Link Layer services

/* ilinkTRANS.c */
void iLinkTransLayer(void);
// Start Transaction layer services. See msgtypes.h for message IDs to use.

#define LITTLEENDIAN

#ifdef LITTLEENDIAN
#define ENDSWAP(x,y) y x
#else
#define ENDSWAP(x,y) x y
#endif

typedef union {
    QUAD Quadlet;
    struct {
      ENDSWAP(unsigned short DataLen;, unsigned short ExtTCode;)
    } Info;
} TQBlock; // Used to represent either a data quadlet, or the info fields for payload

typedef union {
    struct {
      ENDSWAP(unsigned short SrcID;, unsigned short RCode;)
      QUAD Reserved;
    } RCodeBlock;
    struct {
      ENDSWAP(unsigned short SrcID;, unsigned short DestOffHi;)
      QUAD DestOffLo;
    } AddrBlock;
```

```
141     } TDBlock;  // Used to represent either the destination address or response code

143     // The TLinkPacket and TTransReq message formats are designed so that a few
144     // data elements can be changed to change one into the other, to avoid
145     // copying any data from one packet to another.

147  1  typedef struct {
148  1    ENDSWAP(unsigned short DestID;, unsigned short tcodes;
                                                        )       // 1394 Destination node address
149  1    TDBlock DBlock;
150  1    TQBlock QBlock;
151  1    QUAD Data[19];
152     } TLinkPacket;

154  1  typedef struct {        // 1394 Destination node address
155  1                          // Transaction type 0 - Read, 1 - Write, 2 - Lock
156  1                          // 32 for error.
157  1    ENDSWAP(unsigned short DestID;, unsigned short TrType;)
158  1                          // Number of bytes to be written or requested.
159  1    ENDSWAP(unsigned short Length;, unsigned short DestHi;)
160  1    QUAD  DestLo;
161  1    QUAD  Payload[20];    // FIXME: For block write transactions, start at 1, not 0
162     } TTransReq;

164     // The following data structure is used by CSR stuff to reply to requests.
165     // I know it looks the same (
                                cept for the TrType thing), but it might not be the same in the future
166     // When in doubt, I am keeping seperate copies of things.

168  1  typedef struct {        // 1394 Destination node address
169  1                          // Transaction type 0 - Read, 1 - Write, 2 - Lock (
                                                     add 8 to mark this as a response)
170  1                          // 32 for error. Transaction code in high byte for incoming reads
171  1    ENDSWAP(unsigned short DestID;, unsigned short TrType;)
172  1                          // Number of bytes to be written or requested.
173  1    ENDSWAP(unsigned short Length;, unsigned short DestHi;)
174  1    QUAD  DestLo;
175  1    QUAD  Payload[20];    // FIXME: For block write transactions, start at 1, not 0
176     } TTransResp;

178     #define TRTIMEOUT       32      // Error - transaction timeout (returned in TrType)
179     /* ilinkconv.c */
180     int ReadPacket(QUAD *Mem);
181     // Read LLC FIFO and format into 1394 std format, minus CRCs
182     int GRFNotEmpty(void);
183     // Returns nonzero if FIFO has stuff in it.
184     int SendPacket(QUAD *Mem);
185     // Put 1394 format packet into FIFO in LLC native format

187     /* debug.c */
188     void InitDebugHandler(void);
189     // Call once only (ever) before using DebugMsg
190     void DebugMsg(char* DebugString);
191     // Dump error message to debug output FIFO

193     /* test.c */
194     void LEDOff(int led);
195     void LEDOn(int led);
196     void LEDToggle(int led);

198     /* timer.c */
199  1  typedef struct {
200  1    MADDR ReplyAdd;
201  1    QUAD  Time;
202     } TTimerReq;

204  1  typedef struct {
205  1    QUAD  Hi;
206  1    QUAD  Lo;             // LSB increments at 8kHz
207     } TTime;
```



```
209     void TimerInit(void);
210     TTime GetCycTime(void); // Get current system time. Synched to 1394 Cyc timer
211     int CompareTime(TTime Time1, TTime Time2); // T1 < T2 => -1, T1=T2 => 0, etc
212     TTime AddTime(TTime Time1, TTime Time2);
213     TTime SubTime(TTime Time1, TTime Time2);        // Subtract Time2 from Time1
214                                                     // Doesnt work if the result is negative

216     /* SBP2.c */

218     #define SBP2ADDRLo 0xF0010000   // Beginning of SBP-2 memory range, in 1394 node space
219     #define SBP2ADDRHi 0xF001FFFF   // End of SBP-2 Memory range
220     void DoHDD(void);
221     void SBPInit(void);

223     /* CSRmain.c */
224     #define CSRROMBASEADDRLo 0xF0000400     // Beginning of CSR ROM memory range,
                                                                                in 1394 node space
225     #define CSRROMBASEADDRHi 0xF0000500     // End of CSR ROM memory range
226     void CSRInit(void);
```

```
1  /*******************************************************************
2   * msgtypes.h                                                       *
3   *------------------------------------------------------------------*
4   *                                                                  *
5   *  Central list of message types                                   *
6   *                                                                  *
7   *  April 1999, Ray Heasman                                         *
8   *                                                                  *
9   *******************************************************************/

11 // Big messages are even numbers. Small messages are odd numbers.

13 #define ALLOCBMSG       1<<28
14 #define ALLOCSMSG       ((1<<28)+1)

16 #define DEALLOCBMSG     1<<28
17 #define DEALLOCSMSG     ((1<<28)+1)

19 // FireWire messages. Chosen to match the 1394 Std state diagrams
20 // as closely as possible.
21 #define LK_CONTROLREQ   101     // Requests LLC action & sets parms
22 #define LK_EVENTIND     103     // Used to signal events on the Link layer
23 #define LK_CONFIGREQ    105     // Used to send a Phy config packet
24 #define LK_DATACONF     107     // Used by LLC to indicate a packet sent
25 #define TRANS_TIMER     109     // Transaction layer timer reply port

27 #define LK_DATAREQ      100     // Requests that LLC send a packet
28 #define LK_DATAIND      102     // Used by LLC to deliver a rxed packet
29 #define LK_CONFIGIND    104     // Used by LLC to report self ID packets etc.
30 #define TR_DATAREQ      106     // Used by Transaction layer to accept requests.
31 #define TR_DATARESP     108     // Used by Transaction layer to accept responses.

33 // Timer server messages

35 #define TIMERINPORT     201     // Timer server listens for requests on this port.

37 // Test program messages
38 #define TIMERREPLY      301
39 #define INITREPLY       303
40 #define ILINKREPLY      300     // Big message for packets received at app layer

42 // SBP2 stuff
43 #define CSRBASEID       400     // Data to the CSR base registers
44                                 // from the trans layer go to this
45 #define CSRROMBASEID    402     // Data to the CSR ROM config handler
46                                 // goes to this address
47 #define CSRSBP2BASEID   404     // Data to the SBP2 handler goes here

49 #define HDDREPLY        406     // Used inside SBP driver for general HDD control
```

core.s 1
ilink.s 5
ilinkasm.s 7
init.s 9
timerasm.s 13

```
1   ; Source code for the ARM kernel core

3           AREA core, CODE, READWRITE
4           CODE32
5           IMPORT SetTimer
6           IMPORT InitPIO
7           IMPORT InitAIC
8           IMPORT C_Startup
9           EXPORT IntID
10          EXPORT CurrTask
11          EXPORT TaskList
12          EXPORT TaskSwitch
13          EXPORT AddTask
14          EXPORT SpurInt
15          EXPORT Uptime
16  ;       EXPORT __main

18  Mode_USR        EQU     0x10
19  Mode_IRQ        EQU     0x12
20  Mode_SVC        EQU     0x01
21  I_Bit           EQU     0x80
22  F_Bit           EQU     0x40
23  AIC_EOICR       EQU     0xFFFFF130
24  AIC_IVR         EQU     0xFFFFF100
25  TC_SR           EQU     0xFFFE00A0
26  TC2_CCR         EQU     0xFFFE0080
27  PIO_SODR        EQU     0xFFFF0030
28  PIO_CODR        EQU     0xFFFF0034
29  PS_CR           EQU     0xFFFF4000

31  MaxTs   EQU     64              ; Maximum number of tasks

33          ENTRY
34  main
35          ldr     r13,=0x7FF800   ; Ram limit - 1024
36          mov     r0,#0xdF
37          msr     CPSR,r0
38          ldr     r0,=TaskList
39          mov     r1,#3           ; Marks the current task as active
40          str     r1,[r0]

42          ldr     r0,=MinSP
43          str     sp,[r0]         ; Store the Stack pointer we can use.
44          bl      InitAIC

46          mov     r1,#0x1000
47          ldr     r0,=C_Startup
48          bl      AddTask

50          bl      InitPIO         ; initialise the PIO
51          ldr     r0,=TaskSwitch
52          bl      SetTimer        ; COMMENT OUT to disable task switching
53          mov     r0,#0x10
54          msr     CPSR,r0

56  ;       bl      RunTestSuite
57          ldr     r0,=PS_CR
58          mov     r1,#1
59  IdleLp
60  ;       str     r1,[r0]         ; Activate CPU sleep
61          nop
62          b       IdleLp          ; Go to sleep again

64  AddTask
65  ; Adds a task to the tasklist
66  ; Entry:
67  ;       r0 = Start address of task to add
68  ;       r1 = Stack Space to use for task
69  ;       r4 = 32 bit value passed to r0 of new task.
70  ; Exit: r0 = PID, or 0 if unsuccessful
```



```
71  ; Scratches:
72  ;       r0-r3

74  ; NBNBNB: Possible race condition if two tasks do an AddTask simultaneously with
75  ;         interrupts enabled

77          ldr     r3,=NumSwitches ; use r3 as base register for accessing data area
78          ldr     r2,[r3,#NumTasks-NumSwitches]
79          cmp     r2,#MaxTs
80          movhs   r0,#0
81          movhs   PC,r14                  ; Scarper if all task slots used

83          ldr     r2,[r3,#MinSP-NumSwitches]
84          sub     r1,r2,r1                ; Calculate new SP
85          str     r1,[r3,#MinSP-NumSwitches]

87          ldr     r2,=TaskList-128        ; Get ready to look through available slots

89  ATLp    add     r2,r2,#128
90          ldr     r3,[r2]
91          tst     r3,#2           ; Start stepping though list, looking for a free task
92          bne     ATLp            ; loop until a non-valid task is found.

94          mov     r3,#3
95          str     r3,[r2]         ; Mark task as used and valid
96          mov     r3,#&10         ; This is the new task's status register
97          str     r3,[r2,#4]      ; store it
98          str     r0,[r2,#8]      ; store the program counter
99          str     r1,[r2,#64]     ; Write SP (68 was wrong offset)
100         str     r4,[r2,#12]     ; Store a data value for destination.

102         ldr     r3,=TaskList
103         sub     r2,r2,r3        ; Get offset of current task from beginning of list
104         mov     r0,r2,lsr #7
105         add     r0,r0,#1        ; Make PID by getting task number and adding 1
106         bx      r14


109 TaskSwitch                      ; Never breakpoint here.
110         ldr     r13,=AIC_IVR
111         str     r13,[r13]
112 BP_TS                           ; Break here
113         ldr     r13,=CurrTask   ; First, save current task
114         ldr     r13,[r13]
115         add     r13,r13,#12     ; Skip area to save PC + SPSR
116         stmia   r13,{r0-r14}^
117         mrs     r0, SPSR
118         sub     LR,LR,#4        ; Adjust LR value, cos IRQ adds 4.
119         stmdb   r13,{r0, LR}    ; Store SPSR and PC

121         ldr     r0,=NumSwitches
122         ldr     r1,[r0]
123         add     r1,r1,#1
124         str     r1,[r0]

126         sub     r13,r13,#12
127 TSLoop
128         add     r13,r13,#128    ; Increment pointer
129         ldr     r0,[r13]
130         tst     r0,#1           ; Check bit zero
131         ldreq   r13,=TaskList   ; Go to beginning of table if end reached (==0)
132         ldreq   r0,[r13]
133         tst     r0,#2           ; Check bit 1
134         beq     TSLoop          ; Go to next entry if task is not valid (==0)
135         tst     r0,#4           ; Check bit 2
136         bne     TSLoop          ; Keep on looking if task is blocked (==1)
137 ; If we get here we are pointing to a valid, unblocked task
138 ; now we switch to it

140         ldr     r1,=TC_SR
```

```
141         ldr     r1,[r1]
142         ldr     r1,=AIC_EOICR
143         str     r1,[r1]

145         ldr     r0,=CurrTask
146         str     r13,[r0]
147         add     r13,r13,#12
148         ldmdb   r13,{r0,r14}
149         msr     SPSR,r0         ; Restore SPSR
150         ldmia   r13,{r0-r14}^   ; Get user registers
151         nop
152 TSEnd   movs    PC,r14
153 ; Now we have the Spurious interrupt vector.
154 SpurInt
155         ldr     r13,=AIC_EOICR
156         str     r13,[r13]
157         subs    pc,r14,#4
158 stop
159         MOV     r0, #0x18           ; angel_SWIreason_ReportException
160         LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
161         SWI     0x123456            ; Angel semihosting ARM SWI

163         LTORG
164         AREA coredata, DATA, READWRITE
165 NumSwitches     DCD 0
166 NumTasks        DCD 0
167 IntID           DCD 0
168 MinSP           DCD 0
169 CurrTask        DCD TaskList
170 TaskList        % 32*4*32
171 EndList         DCD 0           ; So Task Switcher can find end of list
172 Uptime          DCD 0
173 Test            DCD 0
174 Test2           DCD 0

176         END
```

```
1   ;/*******************************************************************
2   ; * ilink.s                                                         *
3   ; *----------------------------------------------------------------*
4   ; *                                                                 *
5   ; * This contains assembler wrappers for IEEE 1394 IRQ handler stuff  *
6   ; *                                                                 *
7   ; *                                                                 *
8   ; *  October 1998, Ray Heasman                                      *
9   ; *                                                                 *
10  ; *******************************************************************/

12          AREA    inlinkasm, CODE, READONLY
13          CODE32
14          EXPORT  IntVec1394Asm
15          IMPORT  IntVec1394C

17  IntVec1394Asm
18          stmfd   sp!,{r0-r12}    ; Back up everything
19          ldr     r0,=IntVec1394C ; Call C vector routine
20          bx      r0
21          ldmfd   sp!,{r0-r12}
22          subs    pc,r14,#4       ; Return

24          END
```

```
1  ;/*******************************************************************
2  ; * ilinkasm.s                                                      *
3  ; *-----------------------------------------------------------------*
4  ; *                                                                 *
5  ; * This contains assembler wrappers for IEEE 1394 IRQ handler stuff  *
6  ; *                                                                 *
7  ; *                                                                 *
8  ; *  October 1998, Ray Heasman                                      *
9  ; *                                                                 *
10 ; *******************************************************************/

12         AREA    ilinkasm, CODE, READONLY
13         CODE32
14         EXPORT  IntVec1394Asm
15         IMPORT  IntVec1394C

17 AIC_EOICR       EQU     0xFFFFF130
18 AIC_ICCR        EQU     0xFFFFF128
19 AIC_IDCR        EQU     0xFFFFF124
20 AIC_IECR        EQU     0xFFFFF120
21 AIC_IVR         EQU     0xFFFFF100
22 PIO_ISR         EQU     0xFFFF004c
23 PIO_IER         EQU     0xFFFF0040
24 PIO_IDR         EQU     0xFFFF0040
25 PIO_PDSR        EQU     0xFFFF003C
26 IRQStack        EQU     0x1000

28 IntVec1394Asm                           ; dont breakpoint here
29         ldr     sp,=AIC_IVR
30         str     sp,[sp]
31 BP_IV1394                               ; break here
32         ldr     sp,=IRQStack
33         stmfd   sp!,{r0-r12,r14}        ; Back up everything

35 ;       ldr     r0,=PIO_ISR
36 ;       ldr     r0,[r0]
37 ;       ldr     r0,=PIO_PDSR
38 ;       ldr     r0,[r0]
39 ;       tst     r0,#0x200
40 ;       beq     Line_Lo         ; Stay on if Line is lo
41 Break_here
42 ;       ldr     r0,=LoCnt       ; For breakpoint
43 ;       ldr     r1,[r0]
44 ;       add     r1,r1,#1
45 ;       str     r1,[r0]
46 ;       b       IVCont

48 Line_Lo
49         ; now comes bit out of 5-15 ARM Ref Guide
50         mov     ip,sp
51 ;       stmfd   sp!,{fp,ip,lr,pc}
52         sub     fp,ip,#4
53         ; end bit

55         ldr     r0,=IntVec1394C ; Call C vector routine
56         ldr     r14,=IVCont
57         bx      r0
58 IVCont
59 ;       ldr     r1,=AIC_IVR
60 ;       str     r1,[r1]
61         ldr     r1,=AIC_EOICR
62         str     r1,[r1]

64 ;       ldmfd   sp!,{fp,ip,lr,pc}
65         ldmfd   sp!,{r0-r12,r14}
66         subs    pc,r14,#4       ; Return
67 LoCnt   DCD     0
68         END
```

```
1   ;
2   ; The AREA must have the attribute READONLY, otherwise the linker will not
3   ; place it in ROM.
4   ;
5   ; The AREA must have the attribute CODE, otherwise the assembler will not
6   ; let us put any code in this AREA
7   ;
8   ; Note the '|' character is used to surround any symbols which contain
9   ; non standard characters like '!'.

11                  AREA    Init, CODE, READONLY

13  ; Now some standard definitions...

15  Mode_USR        EQU     0x10
16  Mode_IRQ        EQU     0x12
17  Mode_SVC        EQU     0x13

19  I_Bit           EQU     0x80
20  F_Bit           EQU     0x40

22  ; Locations of various things in our memory system

24  RAM_Base        EQU     0x700000      ; 1MB RAM at this base
25  RAM_Limit       EQU     0x800000

27  IRQ_Stack       EQU     RAM_Limit       ; 1K IRQ stack at top of memory
28  SVC_Stack       EQU     RAM_Limit-1024  ; followed by SVC stack
29  USR_Stack       EQU SVC_Stack-1024      ; followed by USR stack

31  ; --- Define entry point
32          EXPORT  __main  ; defined to ensure that C runtime system
33  __main                          ; is not linked in
34          ENTRY

36  ; We have to handle two types of reset - warm and cold
37  ; In the cold reset, the ROM maps to 0 until a bit is set in a memory mapped register
38  ; In the warm rest, the vector in internal RAM is followed.

40  ; --- Initialise memory system, if we are cold booting.

42  ;        B       __main
43          LDR     r0,=0xFFE00000  ; Memory controller base addr

45          LDR     r1,=0x40302D
46          STR     r1,[r0]

48          LDR     r1,=0x503001
49          STR     r1,[r0,#4]

51          LDR     r1,=0x603021
52          STR     r1,[r0,#8]

54          LDR     r1,=0x702029
55          STR     r1,[r0,#12]

57          LDR     r1,=0x802029
58          STR     r1,[r0,#16]

60          LDR     r1,=0x90003d
61          STR     r1,[r0,#20]

63          LDR     r1,=0xA0003d
64          STR     r1,[r0,#24]

66          LDR     r1,=0xB0003d
67          STR     r1,[r0,#28]

69          LDR     r1,=0x4
70          STR     r1,[r0,#0x24]
```



```
71  Switch

73  ; Now we copy a sequence of LDR PC instructions over the vectors
74  ; (Note: We copy LDR PC instructions because branch instructions
75  ; could not simply be copied, the offset in the branch instruction
76  ; would have to be modified so that it branched into ROM. Also, a
77  ; branch instructions might not reach if the ROM is at an address
78  ; > 32M).
79          MOV     R8, #0x300000
80          ADR     R9, Vector_Init_Block
81          LDMIA   R9!, {R0-R7}
82          STMIA   R8!, {R0-R7}
83          LDMIA   R9!, {R0-R7}
84          STMIA   R8!, {R0-R7}

86          NOP
87          NOP
88          NOP
89          NOP
90          LDR     r3, Reset_Addr
91          MOV     r1,#1
92          LDR     r0,=0xFFE00020
93          STR     r1,[r0]         ; Slide memory out from under us

95          MOV     PC,r3
96  ; Now fall into the LDR PC, Reset_Addr instruction which will continue
97  ; execution at 'Reset_Handler'

99  Vector_Init_Block
100         LDR     PC, Reset_Addr
101         LDR     PC, Undefined_Addr
102         LDR     PC, SWI_Addr
103         LDR     PC, Prefetch_Addr
104         LDR     PC, Abort_Addr
105         NOP
106         LDR     PC,[PC, #-&F20]
107         LDR     PC, FIQ_Addr

109 Reset_Addr      DCD     Reset_Handler
110 Undefined_Addr  DCD     Undefined_Handler
111 SWI_Addr        DCD     SWI_Handler
112 Prefetch_Addr   DCD     Prefetch_Handler
113 Abort_Addr      DCD     Abort_Handler
114                 DCD     0       ; Reserved vector
115 FIQ_Addr        DCD     FIQ_Handler

117 ; The following handlers do not do anything useful in this example.
118 ;
119 Undefined_Handler
120         B       Undefined_Handler

122 SWI_Handler
123         stmfd   sp!,{r0-r2,lr}
124         mrs     r0,SPSR
125         stmfd   sp!,{r0}
126         ldr     r0,[lr,#-4]
127         bic     r0,r0,#0xFF000000       ; Find out what SWI this is
128         sub     r0,r0,#0x100            ; Start at 100 hex (why not?)
129         cmp     r0,#0x2                 ; Check that we have a valid range
130         bhi     SemiSWI                 ; bug out if not 0x100 to 0x102
131         adr     r1,SWIJumpTable
132         ldr     PC,[r1,r0, LSL #2]      ; Jump to correct pozzy
133 SWIJumpTable
134         DCD     EInt                    ; 0x100 = Enable IRQs
135         DCD     DInt                    ; 0x101 = Disable IRQs
136         DCD     FakeIRQ                 ; 0x102 = Fake an IRQ

138 EndOfSWI
139         ldmfd   sp!,{r0}                ; balance stack
140         ldmfd   sp!,{r0-r2,PC}^         ; return
```

```
141  ; We get here if we are attempting a SemiHosting SWI
142  SemiSWI
143          ldmfd   sp!,{r0}                ; Restore SPSR and other stuff
144          msr     SPSR,r0                 ; so that Semihoster knows what to
145          ldmfd   sp!,{r0-r2,lr}          ; do.
146  SemiCatch
147          movs    pc,lr
148  EInt    mrs     r0,SPSR
149          bic     r0,r0,#128              ; Clear the I bit in SPSR
150          msr     SPSR,r0
151          b       EndOfSWI
152  DInt    mrs     r0,SPSR
153          orr     r0,r0,#128              ; Set the I bit in SPSR
154          msr     SPSR,r0
155          b       EndOfSWI

157  FakeIRQ mov     r2,sp                   ; Store the SVC stack pointer.

159          ldr     r0,[r2,#16]             ; Grab the address to return to.
160          ldr     r1,[r2,#4]              ; Grab IRQ vector address.
161          str     r1,[r2,#16]             ; Write into old LR position.

163          mrs     r1,CPSR                 ; Get ready to change CPU mode.
164          bic     r1,r1,#0x1F             ; Clear all bits except I,F,T.
165          orr     r1,r1,#0x12             ; Mask in IRQ mode.

167          add     sp,sp,#20               ; Balance the SVC stack before
168                                          ; switching modes.

170          msr     CPSR,r1                 ; Pow! We're in IRQ mode.
171          add     lr,r0,#4                ; LR_irq=LR_svc+4
172          ldmfd   r2!,{r1}                ; Grab SPSR_svc.
173          msr     SPSR,r1                 ; SPSR_irq=SPSR_svc
174          ldmfd   r2,{r0-r2,PC}           ; restore r0-r2 and jump to IRQ

176  Prefetch_Handler
177          B       Prefetch_Handler
178  Abort_Handler
179          B       Abort_Handler
180  FIQ_Handler
181          B       FIQ_Handler

183  ; The RESET entry point
184  Reset_Handler

186  ; --- Initialise stack pointer registers
187  ; Enter IRQ mode and set up the IRQ stack pointer
188          MOV     R0, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
189          MSR     CPSR, R0
190          LDR     R13, =IRQ_Stack

192  ; Set up other stack pointers if necessary
193          ; ...

195  ; Set up the SVC stack pointer last and return to SVC mode
196          MOV     R0, #Mode_SVC:OR:I_Bit:OR:F_Bit ; No interrupts
197          MSR     CPSR, R0
198          LDR     R13, =SVC_Stack


201  ; --- Initialise critical IO devices
202          ; ...

204  ; --- Initialise interrupt system variables here
205          ; ...

207  ; --- Enable interrupts
208  ; Now safe to enable interrupts, so do this and remain in SVC mode
209          MOV     R0, #Mode_SVC:OR:F_Bit  ; Only IRQ enabled
210          MSR     CPSR, R0
```



```
212  ; --- Initialise memory required by C code

214          IMPORT  |Image$$RO$$Limit|      ; End of ROM code (=start of ROM data)
215          IMPORT  |Image$$RW$$Base|       ; Base of RAM to initialise
216          IMPORT  |Image$$ZI$$Base|       ; Base and limit of area
217          IMPORT  |Image$$ZI$$Limit|      ; to zero initialise

219          LDR     r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
220          LDR     r1, =|Image$$RW$$Base|  ; and RAM copy
221          LDR     r3, =|Image$$ZI$$Base|  ; Zero init base => top of initialised data
222          CMP     r0, r1                  ; Check that they are different
223          BEQ     %1
224  0       CMP     r1, r3                  ; Copy init data
225          LDRCC   r2, [r0], #4
226          STRCC   r2, [r1], #4
227          BCC     %0
228  1       LDR     r1, =|Image$$ZI$$Limit| ; Top of zero init segment
229          MOV     r2, #0
230  2       CMP     r3, r1                  ; Zero init
231          STRCC   r2, [r3], #4
232          BCC     %2

234  ; --- Now change to user mode and set up user mode stack.

236          MOV     R0, #Mode_USR:OR:I_Bit:OR:F_Bit
237          MSR     CPSR, R0
238          LDR     sp, =USR_Stack

240  ; --- Now we enter the C code

242   ;       IMPORT  C_Entry
243  ;    [ :DEF:THUMB
244  ;               ORR     lr, pc, #1
245  ;               BX      lr
246  ;               CODE16                          ; Next instruction will be Thumb
247  ;    ]
248  ;      BL      C_Entry

250  ; In a real application we wouldn't normally expect to return, however
251  ; in case we do the debug monitor swi is used to halt the application.
252  ;        MOV     r0, #0x18           ; angel_SWIreason_ReportException
253  ;        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
254  ;        [ :DEF: THUMB
255  ;               SWI 0xAB                                ; Angel semihosting Thumb SWI
256  ;               NOP                                           ;
                                                                  If we are in Thumb state here,
257  ;               NOP                                           ;
                                                                 need to pad end of code so that
258  ;                                                             ;
                                                                  literal pool is greater than
259  ;                                                             ; 8 bytes away.
260  ;        |
261  ;               SWI     0x123456            ; Angel semihosting ARM SWI
262  ;        ]
263  ILoop
264          nop
265          B       ILoop
266          END
```

```
1  ;/*******************************************************************
2  ; * timerasm.s                                                      *
3  ; *-----------------------------------------------------------------*
4  ; *                                                                 *
5  ; * This contains assembler wrappers for IEEE 1394 IRQ handler stuff  *
6  ; *                                                                 *
7  ; *                                                                 *
8  ; *  October 1998, Ray Heasman                                      *
9  ; *                                                                 *
10 ; *******************************************************************/

12         AREA    timerasm, CODE, READONLY
13         CODE32
14         EXPORT  IntVecTimer1Asm
15         IMPORT  IntVecTimer1C

17 AIC_EOICR       EQU     0xFFFFF130
18 AIC_ICCR        EQU     0xFFFFF128
19 AIC_IDCR        EQU     0xFFFFF124
20 AIC_IECR        EQU     0xFFFFF120
21 AIC_IVR         EQU     0xFFFFF100
22 PIO_ISR         EQU     0xFFFF004c
23 PIO_IER         EQU     0xFFFF0040
24 PIO_IDR         EQU     0xFFFF0040
25 PIO_PDSR        EQU     0xFFFF003C
26 TC1_SR          EQU     0xFFFE0060

28 IRQStack        EQU     0x1000

30 IntVecTimer1Asm                         ; dont breakpoint here
31         ldr     sp,=AIC_IVR
32         str     sp,[sp]
33 BP_IVT1                                 ; break here
34         ldr     sp,=IRQStack
35         stmfd   sp!,{r0-r12,r14}        ; Back up everything

37 Line_Lo
38         ; now comes bit out of 5-15 ARM Ref Guide
39         mov     ip,sp
40 ;       stmfd   sp!,{fp,ip,lr,pc}
41         sub     fp,ip,#4
42         ; end bit

44         ldr     r0,=IntVecTimer1C       ; Call C vector routine
45         ldr     r14,=IVCont
46         bx      r0
47 IVCont
48         ldr     r1,=TC1_SR
49         ldr     r1,[r1]
50         ldr     r1,=AIC_EOICR
51         str     r1,[r1]

53 ;       ldmfd   sp!,{fp,ip,lr,pc}
54         ldmfd   sp!,{r0-r12,r14}
55         subs    pc,r14,#4       ; Return
56         END
```