

# Investigation of a High-Speed Serial Bus Between Satellite Subsystems

Francois Retief



Thesis presented in partial fulfilment of the requirements for the degree of  
**Masters of Science in Electronic Engineering**  
at the University of Stellenbosch.

Study leader: J. Treurnicht

March 2003

## Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

# Abstract

The aim of this thesis is to investigate the implementation of a high-speed serial bus based on the IEEE Std 1394-1995 specification for use in a microsatellite.

Earth observation microsatellites carry imagers (or cameras) that take photographs of the earth. Each photograph generates a large volume of digital data that has to be transferred to either a storage device, a RF transmission unit or a video processing device. Traditionally, the connection between such systems were dedicated serial bus systems that were custom designed for just that purpose.

This thesis will investigate the the implementation of a generic alternative to such a custom serial bus. The IEEE 1394 serial bus will allow many devices and subsystems to be connected to the serial bus and will allow these different subsystems to exchange data between each other.

As an example implementation, a real-time video link between two points using the IEEE 1394 serial bus will be developed.

# Opsomming

Die doel van hierdie tesis is om ondersoek in te stel na die bou van 'n hoëspoed seriebus vir gebruik in 'n mikrosatelliet gebaseer is op die IEEE Std 1394-1995 spesifikasie.

Aardobservasie-mikrosatelliete bevat kameras wat fotos van die aarde neem. Elke foto genereer groot volumes digitale data wat na óf 'n massastoor, óf 'n RF-sender, óf 'n video-verwerkingseenheid gestuur word. Tradisioneel is elkeen van hierdie verbindings met 'n toegewyde seriebus gedoen wat spesiaal vir daardie doel gemaak is.

Hierdie tesis het dit ten doel om ondersoek in te stel na 'n generiese alternatief vir hierdie toegewyde seriële busse. Die IEEE 1394 seriebus sal toelaat dat verskeie eenhede en substelsels aan mekaar gekoppel kan word en dat hulle data tussen mekaar kan uitruil.

Ter demonstrasie sal 'n intydse videokakel ontwerp word wat die IEEE 1394 seriebus gebruik om data tussen twee punte oor te dra.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Design goals and choices</b>	<b>3</b>
2.1	Video data source . . . . .	3
2.2	Source module . . . . .	4
2.3	Sink module . . . . .	4
2.4	Video data encoder . . . . .	4
2.5	Control module . . . . .	5
2.6	Mass memory store . . . . .	5
<b>3</b>	<b>Serial bus architectures</b>	<b>6</b>
3.1	Background . . . . .	6
3.1.1	Single-ended vs. differential transmission . . . . .	6
3.1.2	Parallel vs. serial transmission . . . . .	8
3.1.3	Point-to-point, multidrop and multipoint topologies . . . . .	9
3.1.4	Half and full-duplex transmission . . . . .	9
3.2	RS232, RS422 and RS485 . . . . .	10
3.3	TIA/EIA-644: LVDS . . . . .	11
3.4	USB . . . . .	12
3.5	IEEE 1394 . . . . .	13
3.5.1	Specifications . . . . .	13
3.5.2	Cable and signals . . . . .	14
3.5.3	Addressing . . . . .	15
3.5.4	Protocol architecture . . . . .	16
3.5.4.1	Physical layer . . . . .	18
3.5.4.2	Link layer . . . . .	18
3.5.4.3	Transaction layer . . . . .	18
3.5.4.4	Serial bus management layer . . . . .	19
3.6	SpaceWire . . . . .	20

3.7	Conclusions . . . . .	22
<b>4</b>	<b>Architecture and component selection</b>	<b>23</b>
4.1	Architecture of the demonstration system . . . . .	23
4.2	Bus interface hardware . . . . .	24
4.2.1	Texas Instruments TSB12LV01B . . . . .	26
4.2.2	Philips PDI1394L11 and PDI1394L40 . . . . .	26
4.2.3	Texas Instruments TSB12LV32 . . . . .	27
4.2.4	Texas Instruments TSB42LV03 . . . . .	29
4.3	Microprocessor . . . . .	29
4.3.1	Motorola Coldfire series of processors . . . . .	30
4.3.2	Hitachi H8/300H series of processors . . . . .	30
4.3.3	ARM Evaluator-7T evaluation board . . . . .	31
4.3.4	Microprocessor selection . . . . .	31
4.4	Device selection for glue logic . . . . .	31
4.5	Video converters . . . . .	32
4.6	Video input block . . . . .	33
4.7	Video output block . . . . .	34
4.8	Power supply . . . . .	35
4.9	Software languages and compiler tools . . . . .	36
4.9.1	C language . . . . .	36
4.9.1.1	ARM Developer's Suite . . . . .	37
4.9.1.2	GNU tools: GCC, Binutils and GDB . . . . .	37
4.9.2	VHDL language . . . . .	38
4.9.2.1	Altera's MaxPlusII and QuartusII . . . . .	38
4.10	Operating system selection . . . . .	38
4.10.1	RTX . . . . .	38
4.10.2	RTEMS . . . . .	39
4.10.3	Linux or $\mu$ Clinux . . . . .	39
4.10.4	eCos . . . . .	40
4.10.5	Conclusion . . . . .	40
4.11	Test environment . . . . .	40
<b>5</b>	<b>Design details</b>	<b>42</b>
5.1	Video source . . . . .	43
5.2	Video converter . . . . .	43
5.3	Video glue logic for the transmitter . . . . .	45

5.4	Data FIFO buffer for the transmitter . . . . .	46
5.5	Data mover glue logic . . . . .	47
5.6	IEEE 1394 bus interface . . . . .	48
5.6.1	TSB43LV03, physical-layer device . . . . .	48
5.6.2	TSB12LV32, link-layer controller . . . . .	49
5.7	Data mover glue logic for the receiver . . . . .	50
5.8	FIFO buffer on the receiver side . . . . .	50
5.9	Video glue logic for the receiver . . . . .	50
5.10	Digital video encoder . . . . .	51
5.11	Video sink . . . . .	52
5.12	Microprocessor . . . . .	52
5.12.1	Microprocessor and LLC host interface glue logic . . . . .	52
5.13	FPGA, configuration EPROM and JTAG interface . . . . .	53
5.14	Power supply . . . . .	55
5.15	Software design . . . . .	56
5.15.1	eCos kernel . . . . .	56
5.15.2	IEEE 1394 core and TSB12LV32 driver . . . . .	56
5.15.2.1	CSR module . . . . .	57
5.15.3	eShell command line . . . . .	59
5.15.4	I <sup>2</sup> C and SAA7111A/SAA7127H driver . . . . .	59
5.15.5	Control application . . . . .	59
<b>6</b>	<b>System implementation, integration and results</b>	<b>61</b>
6.1	Schematics and PCB layouts . . . . .	61
6.1.1	PCB: communications board . . . . .	61
6.1.2	PCB: video converter boards . . . . .	63
6.1.3	PCB: ARM Evaluator-7T evaluation board . . . . .	64
6.2	PCB assembly and initial testing . . . . .	65
6.3	Microprocessor interface and glue logic . . . . .	66
6.4	Initial interrupt and transmission tests . . . . .	67
6.5	Transition to the eCos environment . . . . .	69
6.5.1	Program execution on the Evaluator-7T . . . . .	71
6.6	IEEE 1394 driver . . . . .	72
6.6.1	Asynchronous transactions . . . . .	73
6.6.2	CSR and the configuration ROM . . . . .	74
6.7	I <sup>2</sup> C and SAA7111A driver . . . . .	74
6.8	Isochronous data transfer . . . . .	75

<i>CONTENTS</i>	vii
6.9 Unresolved issues and problems . . . . .	78
6.10 Final system . . . . .	81
<b>7 Conclusions</b>	<b>82</b>
<b>A Schematics and PCB layouts</b>	<b>86</b>
<b>B VHDL glue logic code</b>	<b>93</b>
B.1 Video glue logic for transmission . . . . .	93
B.2 Data mover glue logic for the transmitter . . . . .	96
B.3 Microprocessor and LLC interface glue logic . . . . .	98
B.4 Data mover glue logic for the receiver . . . . .	102
<b>C Miscellaneous source code</b>	<b>103</b>
C.1 Script for automatic program upload . . . . .	103



# List of Figures

1.1	Overview of the subsystem interfaces to the OBC. . . . .	2
2.1	A general overview of the test system. . . . .	4
3.1	Single-ended transmission with termination. . . . .	7
3.2	Differential transmission. . . . .	7
3.3	Parallel vs. serial transmission. . . . .	8
3.4	Point-to-point topology. . . . .	9
3.5	Multidrop topology. . . . .	9
3.6	Multipoint topology. . . . .	10
3.7	The LVDS physical layer. . . . .	11
3.8	Data-strobe encoding and the encoder/decoder circuit. . . . .	15
3.9	Addressing of the IEEE 1394 serial bus. . . . .	16
3.10	Block diagram of the IEEE Std 1394-1995 protocol architecture. . . . .	17
3.11	Simplified model of the lock operation. . . . .	19
3.12	Block diagram of a typical SpaceWire implementation. . . . .	21
4.1	Architecture of the demonstration system. . . . .	24
4.2	Galvanic isolation between PHY and LLC. . . . .	25
4.3	Block diagram of the TSB12LV01B LLC device. . . . .	27
4.4	Block diagram of the PDI1394L11 LLC device. . . . .	28
4.5	Block diagram of the PDI1394L40 LLC device. . . . .	28
4.6	Block diagram of the TSB12LV32 LLC device. . . . .	29
4.7	Block diagram of the Philips SAA7113H video signal converter. . . . .	33
4.8	Block diagram of the Philips SAA7111A video signal converter. . . . .	34
4.9	Block diagram of the Philips SAA7120 video converter. . . . .	35
4.10	Block diagram of the Philips SAA7127h video converter. . . . .	36
4.11	Block diagram of the test environment. . . . .	41
5.1	Overall block diagram of the transmission of the data. . . . .	42
5.2	CCIR-656 8-bit digital video data stream. . . . .	45

5.3	Block diagram of the video and data mover glue logic components. . . . .	46
5.4	Timing diagram of a data mover read cycle. . . . .	47
5.5	The IEEE 1394 cable termination network. . . . .	48
5.6	Block diagram of the receiver's glue logic. . . . .	51
5.7	Diagram of the FPGA configuration setup. . . . .	54
5.8	Block diagram of the software layout. . . . .	56
5.9	Flow diagram of the driver's packet reception and transmission code. . . . .	58
6.1	Photograph of the communications board. . . . .	62
6.2	Photograph of the video encoder board. . . . .	63
6.3	PCB layout of the ARM Evaluator-7T evaluation board. . . . .	64
6.4	Diagram of the IEEE 1394 cable interface. . . . .	68
6.5	Block diagram of the ARM7TDMI core's memory cache. . . . .	70
6.6	Logic analyser plot of an end-to-end transmission. . . . .	77
6.7	Magnified plot of the packet header transfer sequence. . . . .	77
6.8	Magnified plot of the start of the data transmission. . . . .	77
6.9	Logic analyser plot of a normal bus reset sequence. . . . .	79
6.10	Logic analyser plot of a false bus reset (abnormality in transmitter). . . . .	79
6.11	Logic analyser plot of false bus reset (abnormality in receiver). . . . .	79
6.12	Photograph of a complete module. . . . .	81
A.1	Schematic diagram of the communications board, page 1. . . . .	87
A.2	Schematic diagram of the communications board, page 2. . . . .	88
A.3	Schematic diagram of the video decoder board. . . . .	89
A.4	Schematic diagram of the video encoder board. . . . .	90
A.5	PCB layout of the communications board. . . . .	91
A.6	PCB layout of the video decoder board. . . . .	92
A.7	PCB layout of the video encoder board. . . . .	92

# List of Tables

4.1	Link-layer controller and physical-layer devices. . . . .	26
4.2	Video converter devices. . . . .	32

# List of Acronyms

<b>Acronym</b>	<b>Description</b>
<b>ADCS</b>	Attitude Determination and Control Subsystem
<b>API</b>	Application Programming Interface
<b>ATF</b>	Asynchronous Transmit FIFO
<b>AV</b>	Audio/Video
<b>BM</b>	Bus Manager
<b>BNC</b>	Bayonet N-Type Compact
<b>bps</b>	Bits per second
<b>Bps</b>	Bytes per second
<b>BSL</b>	BootStrap Loader
<b>C&amp;DH</b>	Command and Data Handling
<b>CAN</b>	Controller Area Network
<b>CCD</b>	Charge-Coupled Device
<b>CPLD</b>	Complex Programmable Logic Device
<b>CPU</b>	Central Processing Unit
<b>CRC</b>	Cyclic Redundancy Code
<b>CSR</b>	Control and Status Register
<b>CVBS</b>	Composite Video Broadcast Signal
<b>DM</b>	Data Mover
<b>DMA</b>	Direct Memory Access
<b>DS</b>	Data Strobe
<b>eCos</b>	Embedded Configurable Operating System
<b>EEPROM</b>	Electrically Erasable Programmable Read-Only Memory
<b>EMC</b>	Electromagnetic Compatibility
<b>EMI</b>	Electromagnetic Interference
<b>EPROM</b>	Erasable Programmable Read-Only Memory

<b>Acronym</b>	<b>Description</b>
<b>EUI-64</b>	Extended Unique Identifier, 64 bits
<b>FIFO</b>	First-In First-Out
<b>FPGA</b>	Field Programmable Gate-Array
<b>GCC</b>	GNU Compiler Collection
<b>GDB</b>	GNU Debugger
<b>GRF</b>	General Receive FIFO
<b>GPS</b>	Global Positioning System
<b>HDLC</b>	High-level Data Link Control
<b>I<sup>2</sup>C</b>	Inter-IC Control
<b>IC</b>	Integrated Circuit
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>IRM</b>	Isochronous Resource Manager
<b>ISO</b>	Isochronous
<b>JTAG</b>	Joint Test Action Group
<b>LED</b>	Light Emitting Diode
<b>LEO</b>	Low-Earth Orbit
<b>LLC</b>	Link Layer Controller
<b>LPM</b>	Library of Parameterized Modules
<b>LVDS</b>	Low Voltage Differential Signalling
<b>kb</b>	Kilobits
<b>kbps</b>	Kilobits per second
<b>kB</b>	Kilobytes
<b>kBps</b>	Kilobytes per second
<b>MAC</b>	Media Access Controller
<b>Mbps</b>	Megabits per second
<b>MB</b>	Megabytes
<b>MBps</b>	Megabytes per second
<b>MIPS</b>	Million Instructions Per Second
<b>MMU</b>	Memory Management Unit
<b>MPEG</b>	Moving Picture Experts Group
<b>NASA</b>	National Aeronautics and Space Administration
<b>NRZ</b>	Non-Return-To-Zero
<b>NTSC</b>	National Television Systems Committee

<b>Acronym</b>	<b>Description</b>
<b>OBC</b>	On-Board Computer
<b>OHCI</b>	Open Host Controller Interface
<b>OS</b>	Operating System
<b>OSD</b>	On-Screen Display
<b>PAL</b>	Phase Alternation Line
<b>PC</b>	Personal Computer
<b>PCB</b>	Printed Circuit Board
<b>PCI</b>	Peripheral Component Interconnect
<b>PHY</b>	Physical Layer
<b>PLL</b>	Phase-Locked Loop
<b>RAM</b>	Random-Access Memory
<b>RF</b>	Radio Frequency
<b>RGB</b>	Red-Green-Blue
<b>ROM</b>	Read-Only Memory
<b>RTOS</b>	Real-Time Operating System
<b>RTX</b>	Real-Time Executive
<b>SRAM</b>	Static Random Access Memory
<b>SUNSAT</b>	Stellenbosch University Satellite
<b>TB</b>	Terabytes
<b>TV</b>	Television
<b>TTL</b>	Transistor-Transistor Logic
<b>USB</b>	Universal Serial Bus
<b>UART</b>	Universal Asynchronous Receiver-Transmitter
<b>USART</b>	Universal Synchronous/Asynchronous Receiver-Transmitter
<b>VHDL</b>	VHSIC Hardware Description Language
<b>VHSIC</b>	Very High Speed Integrated Circuit
<b>VHS</b>	Vertical Helix Scan
<b>YUV</b>	Luminance-Bandwidth-Chrominance

# Chapter 1

## Introduction

In 1991 the SUNSAT program was launched by the Department of Electrical and Electronic Engineering at the University of Stellenbosch. The main objective of the SUNSAT program was to encourage students to continue with post-graduate studies. Since its inception, more than 50 post-graduate students have participated in the project.

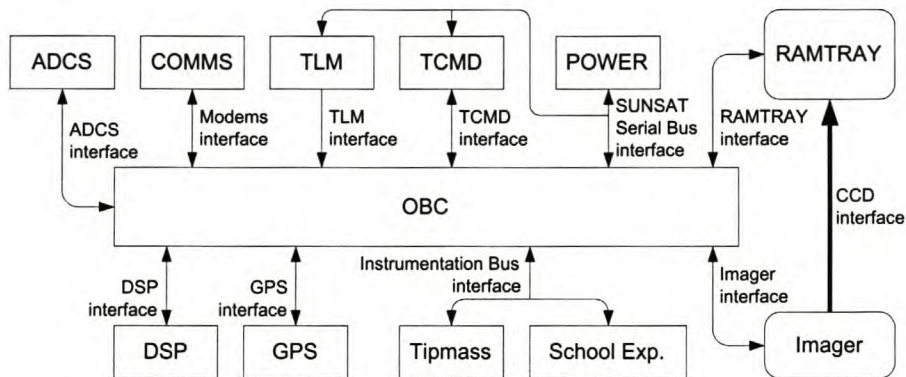
The result of this program was the SUNSAT-1 microsatellite. Launched by the National Aeronautics and Space Administration (NASA) on a Delta 2 launch vehicle on 23 February 1999, SUNSAT-1 operated in space for 696 days until 19 January 2001.

The primary payload of SUNSAT-1 was a high resolution push-boom imager with a spatial resolution of 15 m per pixel. The imager was able to capture images in the green, near-infrared and infrared bands [1] that provided useful information relating to vegetation cover on the earth's surface.

Secondary payloads included a high accuracy global positioning system (GPS) receiver, a scientific magnetometer and a star camera. A number of school experiments and amateur radio services were carried by SUNSAT-1 to support local community involvement in the program.

SUNSAT-1 was constructed as a set of large square trays that were stacked on top of each other. Each tray carried one or more of the satellite's subsystems. As an example, figure 1.1 shows an overview of the communications interfaces to one of the on-board computers (OBCs) and the different subsystems of the microsatellite. Most of these interfaces were parallel interfaces, that required a dedicated set of wires between the OBC tray and the subsystem's tray. All the wires were later assembled into a complex set of cables that interconnected the different trays of the satellite.

In the design of SUNSAT-1, all control functions were located in the two OBCs and the attitude determination and control subsystem (ADCS). The other subsystems had mini-



**Figure 1.1:** *Overview of the subsystem interfaces to the OBC.*

mal intelligence and operated based on commands from one of these controlling systems. Each controlling unit only had a subset of all the subsystem interfaces.

It is easy to see how this mesh of interconnections can become very complex even for a small microsatellite. Also, with the SUNSAT-1 design, inter-subsystem communications were very limited and were mostly used to transfer dedicated types of data, including analog signals. One example was the interdependency between the imager CCD unit and the RAMTRAY subsystem.

An alternative to the SUNSAT-1 mesh architecture, is a pure serial bus architecture. Each subsystem will connect to the microsatellite via a standardised serial bus that allow data and control information to flow seamlessly between systems. Subsystems with non-standard interfaces will use a bridge circuit to connect to the serial bus. For reliability, multiple serial busses can be used in parallel. Redundant busses will decrease the occurrence of single-point failure nodes in the system.

In a previous thesis [2], various serial bus architectures were investigated and a proposal was made for a CAN based serial bus for the command and data handling (C&DH) system of a future SUNSAT microsatellite. The proposal focused mainly on the telemetry and telecommand subsystems. But a CAN based serial bus is not suitable to carry the high-speed data streams generated by most earth observation instruments.

This thesis will investigate the use of the IEEE 1394 serial bus for these high-speed interconnections. A demonstration system will be built that can transfer a digital video stream from one point to another using the IEEE 1394 serial bus as medium.



# Chapter 2

## Design goals and choices

In a typical microsatellite, high volumes of data are captured using an imager or some kind of scientific instrument. This data must then be transmitted to the ground station via a high-bandwidth RF downlink. Thus, inside the satellite, the data must be transferred from the source device to the RF device (also known as a sink device) for transmission.

For purposes of reliability, redundant units are added to the system and there can be more than one RF downlink module in the satellite. There can also be more than one source device (many imagers and science devices) providing data.

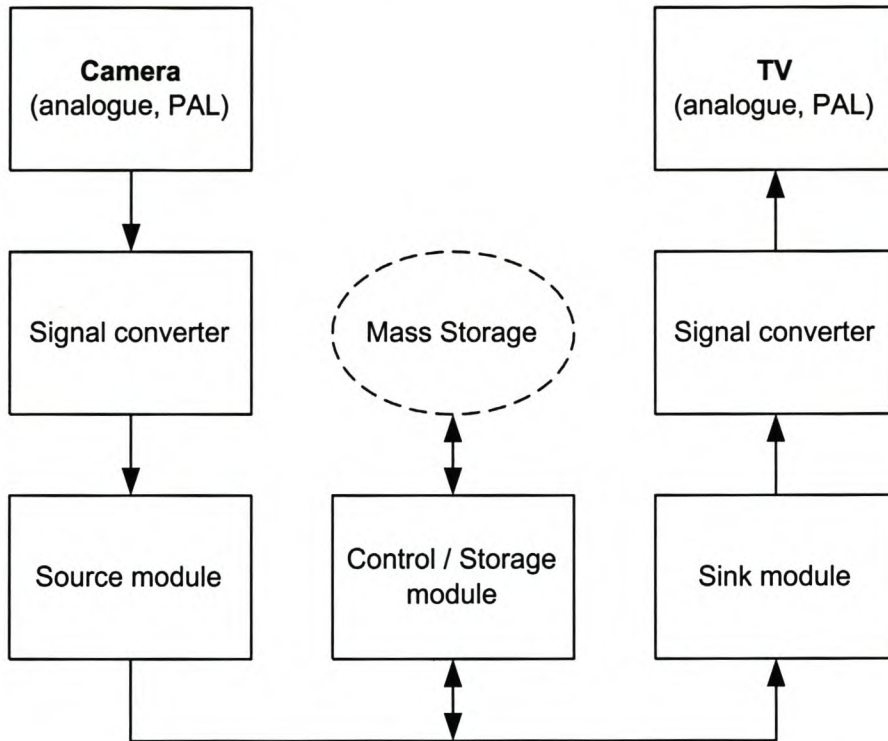
To accommodate these devices, a data bus is needed to transfer the data. For this thesis a simplified version of the scenario described above was selected, as is shown in figure 2.1. The setup consists of an analogue video camera that provides a high volume of data. The television will be the final destination of the data, simulating the radio frequency (RF) downlink. A real-time requirement will be added to ensure that the datalink can sustain the high volume of data.

### 2.1 Video data source

The data source will consist of an ordinary analogue PAL video camera. This analogue signal will then be digitised and prepared for transmission. A raw phase alternation line (PAL) video signal requires about 216 Mbps<sup>1</sup> of transmission bandwidth.

---

<sup>1</sup>CCIR-656: 8-bit data with a 27-MHz clock.



**Figure 2.1:** *A general overview of the test system.*

## 2.2 Source module

The source module will be responsible for breaking the digital data stream into packets for transmission. The packets will then be placed on the serial bus for transmission.

## 2.3 Sink module

The sink modules select one or more data sources and retrieve the data packets from the bus. Error checking and correction (if needed) is performed and packets rearranged into the correct order. The resulting data stream is then presented to a target device.

## 2.4 Video data encoder

The video data encoder take the digital data stream and convert it back into an analogue video signal. The video signal is then sent to the TV for display.

## 2.5 Control module

The control module is responsible for setting up the links between the source and the destination modules, and will start and stop the transmissions. On a microsatellite, commands would be executed by the on-board computer (OBC). The control module will detect and correct any errors that occur in the system.

## 2.6 Mass memory store

On some systems, the capture of the data, and the subsequent transmission to the ground station, do not occur in real time. The captured data is then stored in mass memory. This requires a module that can act as both a data sink and a data store.

# Chapter 3

## Serial bus architectures

This chapter gives an overview of different serial bus architectures that are used (and can possibly be used) on a microsatellite.

### 3.1 Background

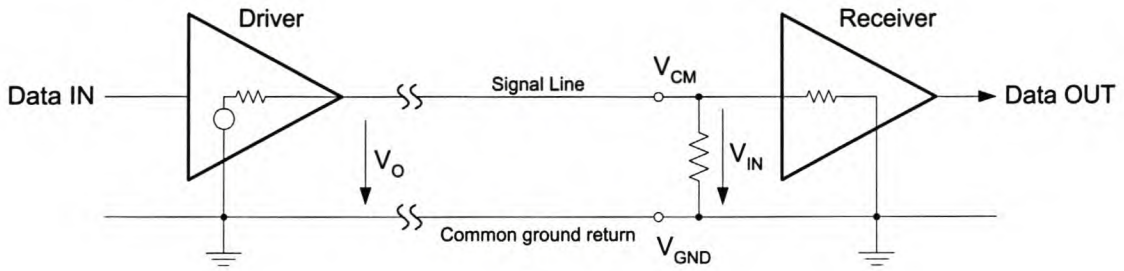
#### 3.1.1 Single-ended vs. differential transmission

All communications standards specify the electrical interface between devices. The two main configurations implemented today are: (i) single-ended (or unbalanced), and (ii) differential (or balanced-data) transmission.

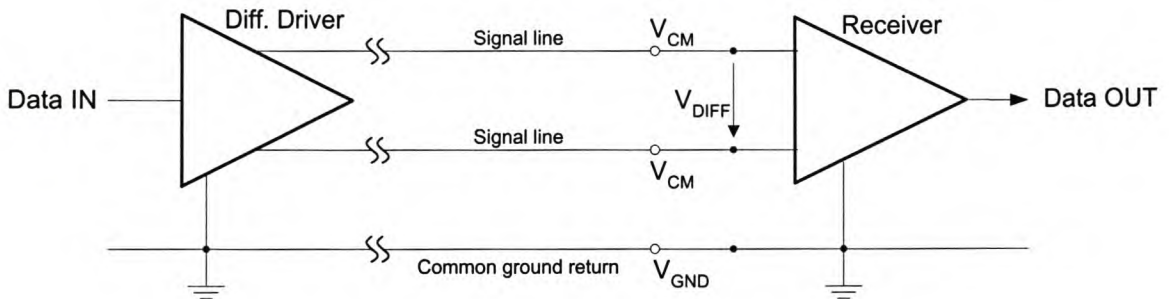
Single-ended transmission requires only one signal line to transfer the logic state. For low-speed devices, a single common ground return path is acceptable. For high-speed communications devices, a ground return path for each signal will be needed to reduce electrical noise. Figure 3.1 shows a circuit diagram of a single-ended transmission system.

The main advantage of a single-ended transmission system is the simple design and the implementation's low cost. Only a single wire is needed per signal, with a single return wire per cable. For longer distances or noisy environments, it would be beneficial to use twisted pairs for the signal wires and to add shielding around the data wires.

A disadvantage of single-ended transmission is its poor noise immunity. The voltage over the signal wire is measured relative to the accompanying ground wire. Shifts in ground potential, transient voltages in nearby systems, and other induced voltages can lead to signal degradation which can lead to false triggering.



**Figure 3.1:** *Single-ended transmission with termination.*



**Figure 3.2:** *Differential transmission.*

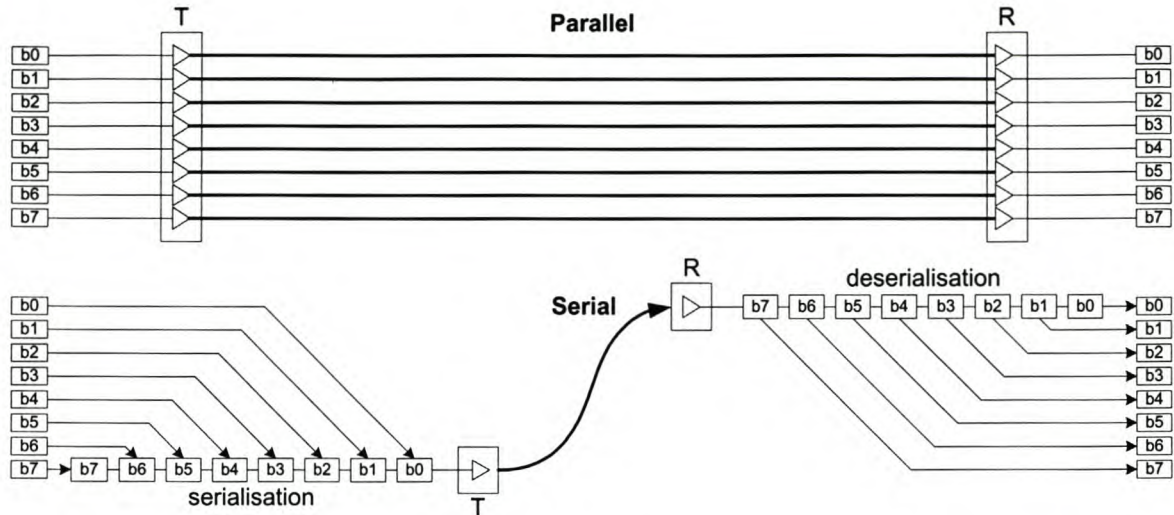
At high frequencies a major concern is crosstalk between signal wires. Crosstalk is generated from capacitive and inductive coupling between signal wires. These problems normally limit the distance and speed of reliable operation of a single-ended transmission path.

An alternative to single-ended transmission is differential transmission, also called balanced data transmission. Figure 3.2 shows a circuit diagram of a differential transmission system with an additional ground return path.

A balanced interface circuit consists of a generator with differential outputs and a receiver with differential inputs. A pair of signal wires is used for each data channel. One wire carries the signal logic level, while the other carries the inverse of the signal. At the receiver, the difference signal between the two wires is taken and the output signal is varied depending on which input line has the higher positive voltage.

The better noise performance of differential transmission stems from the fact that correlated noise is coupled into both wires of the signal pair, and a common noise component occurs in both signals. Due to the common-mode rejection capability of a differential amplifier, such noise will be rejected.

Additionally, since the signal line emits the opposite signal of the adjacent return line, the emissions cancel each other. This is always true for crosstalk from and to the lines of



**Figure 3.3:** *Parallel vs. serial transmission.*

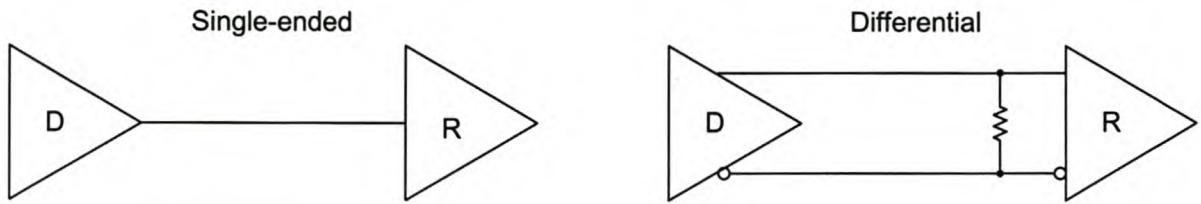
neighbouring signals. It is also true for noise from other sources, as long as the common-mode voltage does not go beyond the common-mode range of the receiver. Since ground noise is also common to both signals, the receiver rejects this noise as well.

Unfortunately, an implementation using differential transmission will be more expensive than a single-ended system because of the extra wires required.

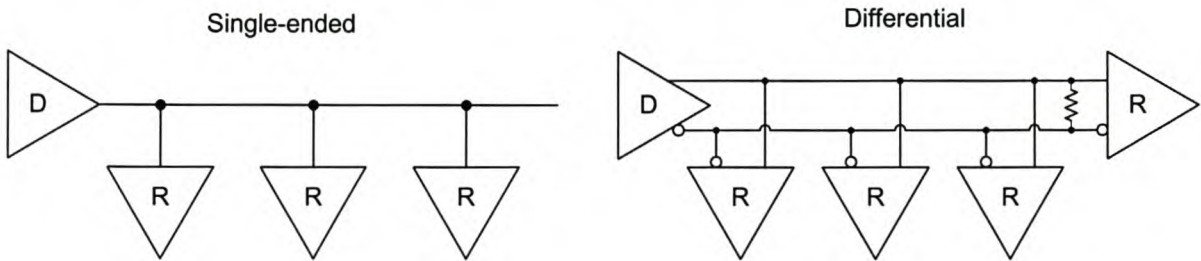
### 3.1.2 Parallel vs. serial transmission

In digital data systems, data is transferred between systems in groups of bits. With parallel communication, all the bits of a group are transferred during one clock cycle. Therefore, each bit requires a transmission path. Parallel interfaces can run at low clock speeds, but are able to transfer large amounts of data. Unfortunately, parallel interfaces require large cables due to the many signal wires. The parallel configuration is normally only used in backplane systems or for very short distances between systems.

In a serial interface, the parallel data must first be converted to a serial stream. This is called serialisation. Figure 3.3 shows the serialisation process in simple terms. The serial data is then transmitted at high speed along a single line to the receiver, where it is deserialised back into the original parallel data. In order for a serial line to be as efficient as parallel lines, the data rate of the serial transmission needs to be at least  $n$  times higher than an  $n$ -bit parallel bus. Due to its simple cable construction, serial interfaces can generally run for much longer lengths (up to 1200 m for TIA/EIA 485) compared to parallel busses (up to 10 m for IEEE Std 1284-1994).



**Figure 3.4:** *Point-to-point topology.*



**Figure 3.5:** *Multidrop topology.*

### 3.1.3 Point-to-point, multidrop and multipoint topologies

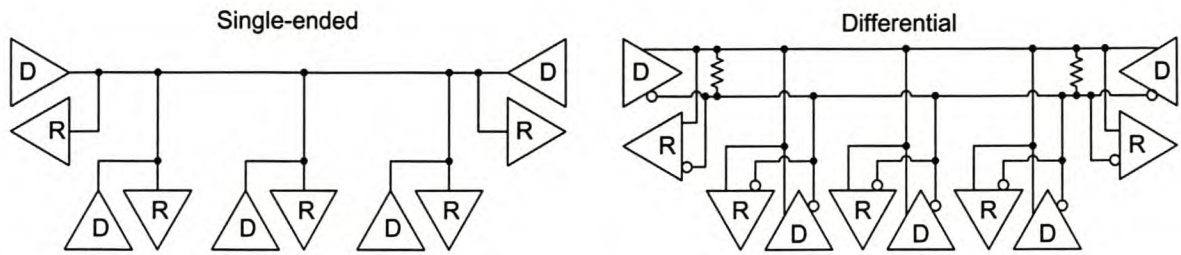
Serial buses can be used in different topologies. The simplest topology is point to point. In this case there is only one transmitter and one receiver. The transmitter is connected directly to the receiver, as shown in figure 3.4.

An alternative is the multidrop topology. With this topology there are one transmitter and many receivers. The data sent by the transmitter is received by all the receivers simultaneously. This can be considered a broadcast topology. Figure 3.5 shows the multidrop topology.

Another alternative is the multipoint topology, consisting of many transmitters and receivers connected to the same serial line. In practice, this solution is normally realised with a transmitter–receiver pair called a transceiver. To prevent many transmitters transmitting at the same time, either a master–slave or a collision detection and prevention protocol must be implemented. Figure 3.6 shows the multipoint topology.

### 3.1.4 Half and full-duplex transmission

Serial interfaces can also be classified as half duplex or full duplex. For half duplex, data flow is only in one direction at any one time. The transmitter must stop transmission and give control over to the receiver, which then becomes the transmitter. With a half-duplex interface there need only be one signal wire pair.



**Figure 3.6:** *Multipoint topology.*

With full-duplex communications, the data flows in both directions simultaneously. Consequently, each unit must have both a transmitter and a receiver. For this mode, the interface needs to have two pairs of signal wires.

Half-duplex transmission must be used for multipoint systems, because only one driver can drive the line at a time. A full-duplex system would consist of two point-to-point connections, one in each direction.

## 3.2 RS232, RS422 and RS485

RS232, RS422 and RS485 are signalling standards that specify the voltages and transmission media for serial communications. No high-level transmission protocols are defined by these standards.

- RS232 is a very popular serial standard. Almost every personal computer (PC) has at least two serial ports that conform to the RS232 standard.

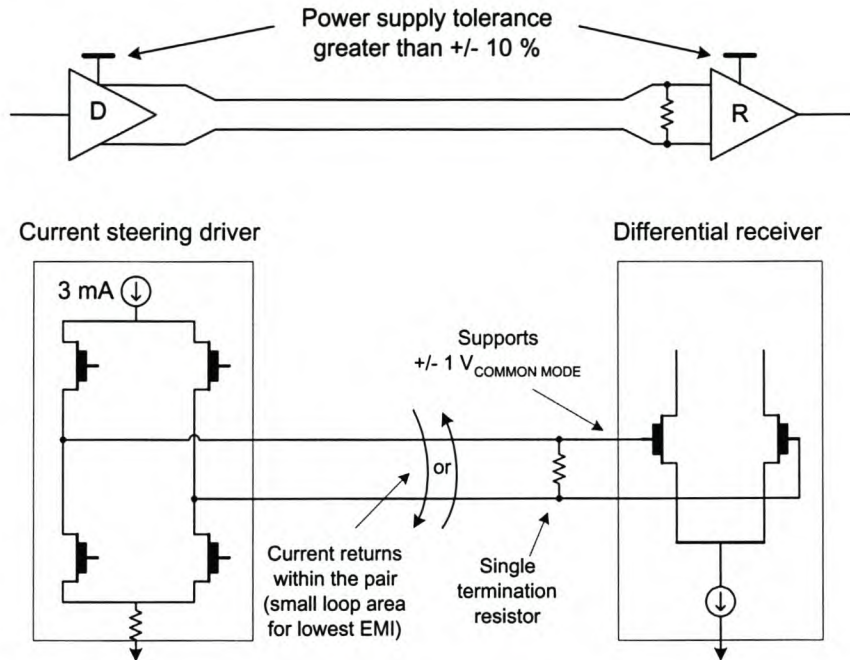
RS232 is a univoltage signalling standard for point-to-point, full-duplex communications between two devices. The data rate is fairly low (up to 115 kbps) with cable lengths of up to 20 m.

The advantage of RS232 is the simplicity of design and the large installed base of applications and tools. Its disadvantage is the slow speed of the serial standard.

- RS422 is a differential-signalling, multidrop communications standard. A single driver can transmit unidirectionally to up to 10 receivers at moderately high speeds (up to 10 Mbps) and with cable lengths of up to 1200 m.

RS422 is very often used in industrial areas due to its robustness. Backplane systems are another application where RS422 is often used.





**Figure 3.7:** *The LVDS physical layer.*

- The RS485 standard uses a differential signalling method similar to RS422, but uses only one pair of wires in a multipoint topology. Due to the multipoint topology, only half-duplex transmission can be achieved. The specification allows for data rates of up to 35 Mbps and cable lengths of up to 1200 m, but both limits cannot be reached at the same time.

Due to the robustness against noise because of the differential transmission mode, and to ground shifts achieved by a large common-mode voltage range, this standard is well suited to industrial applications and other noisy environments. The RS485 standard was also extensively used on the SUNSAT-1 microsatellite [2].

### 3.3 TIA/EIA-644: LVDS

Low-voltage differential signaling (LVDS) is a technique used to achieve higher data rates on commonly used media. Low power consumption was a high priority to the designers of LVDS. A LVDS driver uses a constant current source and switches the current between the two wires. The result is that a LVDS driver always draws a constant supply current.

The receiver is a high-impedance device that detects the differential signal across a 100  $\Omega$  termination resistor. The transmitter drives a constant current of about 2.47 mA to

4.54 mA into the signal wires, giving a voltage difference of 300 mV over the termination resistor. Figure 3.7 shows a diagram of the physical layer of a LVDS connection.

The LVDS standard [3] allows data rates of up to 655 Mbps, but a maximum cable length is not specified. A length of 15 m is given as a recommendation. LVDS is mostly used in the point-to-point topology. A multidrop topology can be constructed, if attention is paid to several conditions. These include stub line length, termination and signalling rate. No high-level protocol is specified by [3].

## 3.4 USB

The universal serial bus (USB) is replacing the RS232 serial port on the common PC as a general-purpose serial bus. USB was designed to be a communications bus for low-cost PC peripherals and had to be easy to use.

USB physically uses a tiered star topology with a hub at the center of each star. Each wire segment is a point-to-point connection between the host and a hub or end device. USB's speed ranges from 1.5 Mbps for slow speed devices, 12 Mbps for full speed devices, and up to 480 Mbps for USB version 2.0 devices.

The low speed version was developed for simple devices (such as mice) to lower power consumption and electromagnetic interference (EMI) of the system. Power is supplied by two of the four wires of a USB cable. The VBUS and GND wires of each segment deliver a nominal 5 V power supply voltage to devices.

The USB has a central host (the PC) with a polled architecture. All bus transfers are initiated by the host. Each transaction begins with a token packet that identifies the type and direction of the transfer, along with the device address and an endpoint number of the destination device.

Next, a data packet is sent by either the host or the device, depending on the transaction direction. Upon receiving the data packet, the other device acknowledges it with a handshake packet.

The USB data transfer model between a source or destination on the host and an endpoint on a device is referred to as a pipe. There are two types of pipes: streams and messages. Stream data has no USB-defined structure, while message data does. Additionally, pipes have associations of data bandwidth, transfer service type, and endpoint characteristics like directionality and buffer sizes. Most pipes come into existence when a USB device

is configured. One message pipe, the default control pipe, always exists once a device is powered, in order to provide access to the device's configuration, status and control information.

The transaction schedule allows flow control for some stream pipes. At the hardware level, this prevents buffers from underrun or overrun situations by using a negative acknowledgment handshake to decrease the data rate. When negatively acknowledged, a transaction is retried when bus time is available. The flow control mechanism permits the construction of flexible schedules that accommodate concurrent servicing of a heterogeneous mix of stream pipes. Thus, multiple stream pipes can be serviced at different intervals and with packets of different sizes.

## 3.5 IEEE 1394

IEEE 1394 is a high-performance serial bus designed to be a generic interface between various types of devices. IEEE 1394 was optimised for guaranteed delivery of time-sensitive data, and enables digital data streams to be transferred in real-time. Multiple independent streams of data can be carried on the bus.

The specification was written for two types of environments, namely backplane and cable environments. The backplane environment operates at data rates of 12.5, 25 or 50 Mbps using a multidrop bus. Possible backplane environments include VME, Futurebus+ and Multibus II.

The cable environment supports data rates of 100, 200 and 400 Mbps using a noncyclic,<sup>1</sup> tree-like network with finite branches and extent. Each network automatically selects a root node. Hot plugging is supported with the cable environment.

### 3.5.1 Specifications

IEEE 1394 was originally developed by Apple Computer as FireWire. In 1995 the serial bus was standardised by the IEEE Computer Society in IEEE Std 1394-1995 [4]. Several companies use the bus under different trade names. Sony uses the trade name *iLink*, Creative Labs uses *SB1394*, and Apple Computer uses the name *FireWire*.

---

<sup>1</sup>Noncyclic means that closed loops are not supported.

The original specification was amended in 2000 to clarify some aspects and to add a few new features. The IEEE Std 1394a-2000 specification [5] also made some optimisation improvements to the serial bus.

The specification was further extended in 2002 with the IEEE Std 1394b-2002 specification [6]. This specification adds support for higher bit rates (800 Mbps, 1.6 Gbps and 3.2 Gbps), a new 8B/10B encoding technique<sup>2</sup> for the data, and changes to the speed signalling method to be more efficient. IEEE 1394b also supports new transmission media, including plastic optical fiber, glass optical fiber and Cat5 twisted-pair cable. The cable run length was also increased to 100 m and allows a speed of 100 Mbps.

The control and status register (CSR) of the serial bus are based on the IEEE 1212r-2001 specification [7]. Various other standards are available to specify how different types of digital content are transferred by the bus.

The greater part of this thesis will describe the workings of the original IEEE 1394-1995 specification, unless stated otherwise.

### 3.5.2 Cable and signals

A standard IEEE 1394 cable consists of two shielded twisted pairs, called TPA and TPB, that carry low-voltage, low-current bidirectional differential signals for the data bits and arbitration signals. Each pair has a common-mode signal impedance of  $33\ \Omega$  and a differential signal impedance of  $110\ \Omega$ .

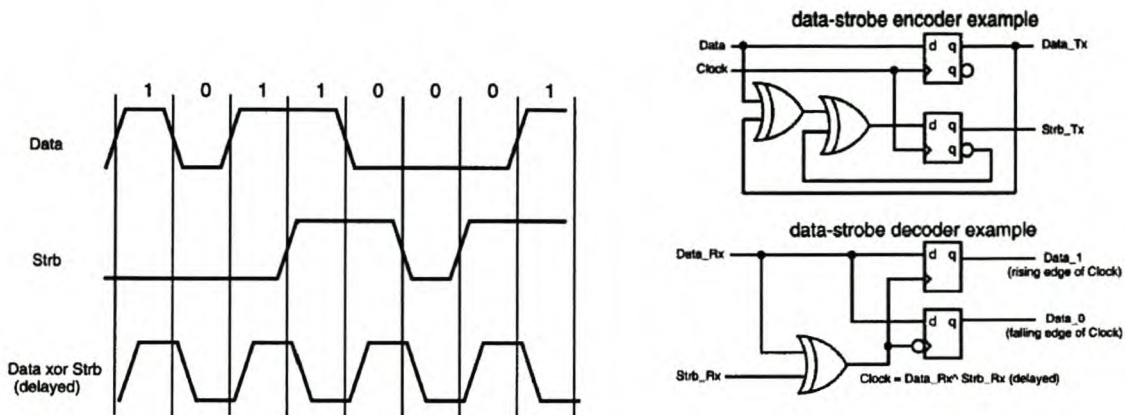
An optional third pair is a power pair that provides current to physical layers to repeat signals. Nodes can either source or sink current. There can be multiple power sources on a bus. A node can provide from  $8\ V_{DC}$  to  $40\ V_{DC}$  at up to  $1.5\ A$  (max.  $15\ W$ ).

The 28-gauge signal cable can have a maximum length of 4.5 m. Cables can be chained together for longer distances, but the two most widely separated nodes must have 16 or fewer cable hops between them. This gives a maximum network length of 72 meters.

The two signals TPA and TPB have three values: '1', '0' and 'Z', where 'Z' represents an undriven or idle condition. Normally only one node will be driving a '1' or a '0' on a pair at a time. If one node is driving a '1' and another is driving a '0' on the pair, the result will be a 'Z', because the two currents will cancel each other out. The node sending a '0'

---

<sup>2</sup>8B/10B is the same encoding technique as that used by Gigabit Ethernet.



**Figure 3.8:** *Data-strobe encoding and the encoder/decoder circuit [4].*

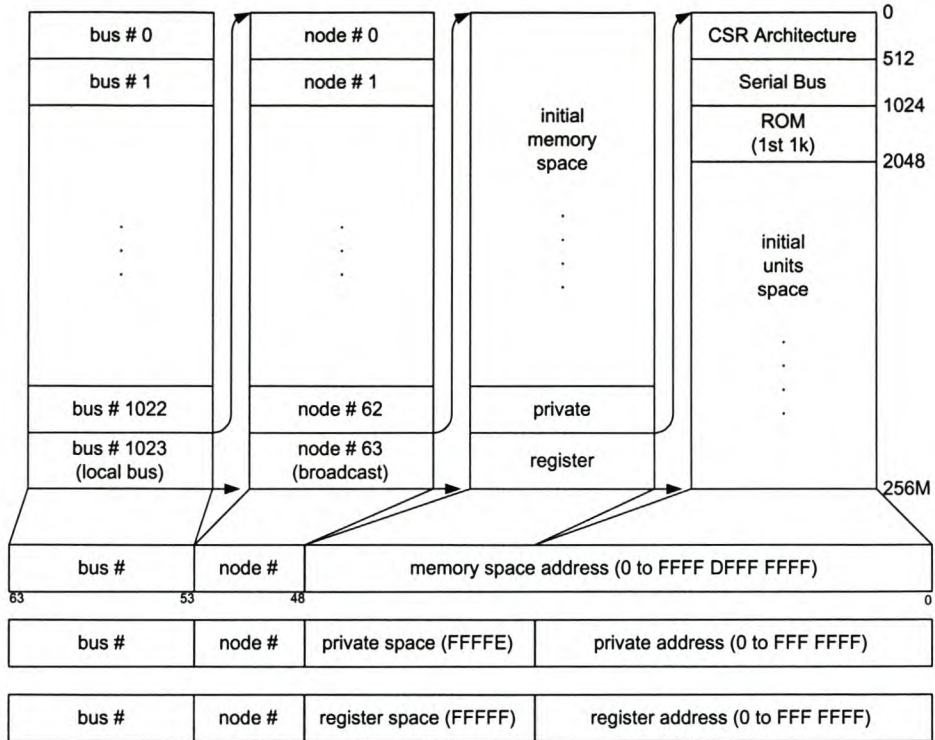
will interpret the receiving ‘Z’ as a ‘1’. The node driving a ‘1’ will always assume that the received signal is a ‘1’. This gives the cable a ‘1’s-dominant signal mode, which is used to detect bus contention and arbitration.

The signals transmitted on both the cable and backplane environments are non-return-to-zero (NRZ) with data-strobe (DS) encoding. The two signal pairs are also known as the data and strobe pairs. The data pair carries the NRZ data signal. The strobe signal changes state whenever two consecutive data bits are the same. This allows reconstruction of the transmission clock from the data and strobe pairs. The DS encoding essentially doubles the jitter tolerance with very little additional circuitry overhead in the hardware. The DS coding is shown in figure 3.8, as well as DS encoder and decoder example circuits.

### 3.5.3 Addressing

The serial bus uses a 64-bit addressing scheme. One IEEE 1394 network can include up to 63 nodes. Each node is identified by a 6-bit physical identification number which is automatically calculated when new devices are added to or removed from the network. Multiple networks can be connected by bridges to form a system with a maximum of 1023 networks. Each network is represented by a 10-bit bus identification number.

The remaining 48 bits are used for memory addresses (256 TB). Memory-based addressing, rather than channel addressing, views resources as registers or memory that can be accessed with processor-to-memory transactions. Figure 3.9 shows an expanded view of the different memory areas in the 64-bit address space.



**Figure 3.9:** Addressing of the IEEE 1394 serial bus.

Within this memory space, a set of CSRs has been predefined for the management and control of the serial bus. Each node has its own set of status and control registers. There is also a configuration ROM in this area that stores identification and node capability information. Nodes advertise their functions and capabilities in this ROM. Other nodes can then search the serial bus to find devices to which they can connect. This allows, for example, a video camera to connect to a television that is able to display the video image being filmed by the camera.

### 3.5.4 Protocol architecture

IEEE 1394 is a transaction-based packet technology for cable and backplane environments. Two types of data transfers are supported by this specification.

**Isochronous** The isochronous format broadcasts data based on a channel number rather than a specific address. Only one isochronous packet can be transmitted per channel during a 125  $\mu$ s period. This period is called the isochronous cycle period. The beginning of this period is marked by a special cycle start packet. The isochronous cycle period guarantee real-time applications that a packet will be transmitted at regular intervals. IEEE

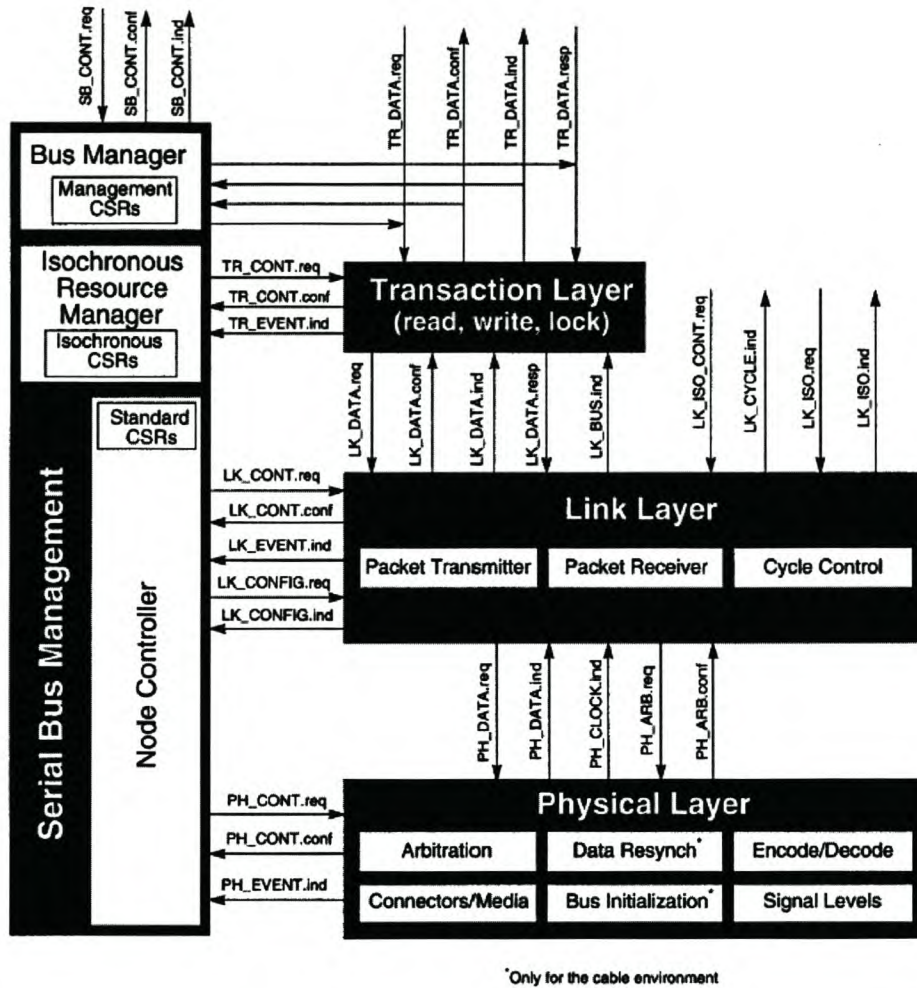


Figure 3.10: Block diagram of the IEEE Std 1394-1995 protocol architecture. [4]

1394 allows the isochronous data to consume a maximum of 80% of the bus bandwidth. Isochronous packets have priority over asynchronous packets.

**Asynchronous** The asynchronous format transfers data and transaction-level information to an explicit address. An asynchronous packet may be transmitted as soon as there is no activity on the bus. A minimum of 20% of the bus bandwidth is reserved for asynchronous packets. All asynchronous packets are guaranteed delivery by an acknowledgment token. An asynchronous packet will be retransmitted if an acknowledgment was negative or missing.

Figure 3.10 shows a block diagram of the overall protocol architecture. The following subsections will discuss each block in more detail.

### 3.5.4.1 Physical layer

The physical layer is responsible for the following three major functions:

- It translates the logical symbols used by the link layer into electrical signals on the different serial bus media.
- It guarantees that only one node at a time is sending data, by providing an arbitration service.
- It defines the mechanical interface for the serial bus.

There are different physical-layer functions for both the cable<sup>3</sup> and backplane environments. The cable physical layer also provides a repeater and a data resynching service and performs automatic bus initialisation when a node is inserted to or removed from the network.

### 3.5.4.2 Link layer

The link layer provides an interface between the physical layer and the application layer, and formats data into packets for transmission over the network. It supports both asynchronous and isochronous data. The link layer provides a half-duplex data packet delivery service.

### 3.5.4.3 Transaction layer

Asynchronous data is transferred between nodes on the serial bus by the following three transaction types:

- Read — reads data from a particular address in a node.
- Write — writes data to a particular address in a node.
- Lock — modifies data in a particular address in a node based on a subcommand. The value of the data before the modification is returned to the sender. Figure 3.11 depicts the lock operation graphically.

The transaction layer is responsible for matching all response packets with their corresponding request packets. The result is returned to the application layer that initiated the transaction. The transaction layer also handles the retransmission of failed packets.

---

<sup>3</sup>The IEEE 1394b-2002 specification extended the cable media even further.



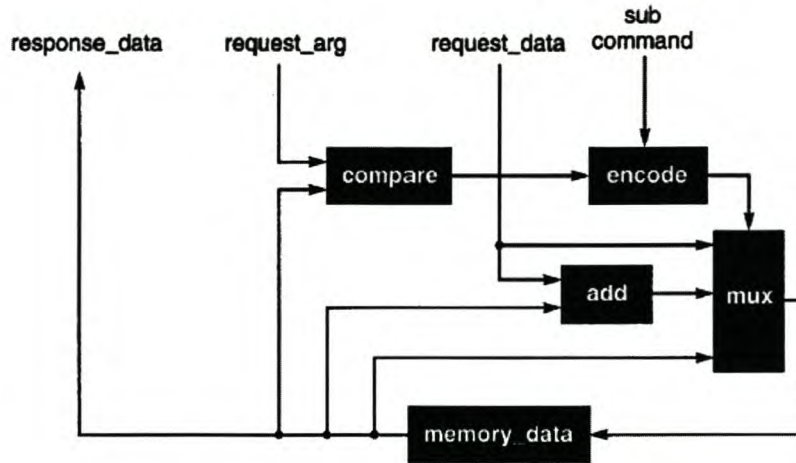


Figure 3.11: *Simplified model of the lock operation [4].*

#### 3.5.4.4 Serial bus management layer

The serial bus management layer consists of the following three units: CSRs, an isochronous resources manager (IRM) and a bus manager (BM). The serial bus has a multimaster (or peer-to-peer) capability. During the bus reset sequence, a single bus master and IRM, node is selected among all nodes that are capable of performing the management functions. This allows a greater degree of redundancy in the system, since a broken device's functionality can easily be transferred to another node.

The bus management layer implements the predefined CSRs that were described in section 3.5.3. These registers can either be implemented in hardware or in the software protocol stack.

**Isochronous resource manager** The IRM is responsible for the allocation of isochronous bandwidth, allocation of channel numbers and the selection of the cycle master. Any node that wants to transmit an isochronous data stream, must request a channel number and bandwidth from the IRM, thus ensuring that only one node can transmit per channel and that there is always enough bandwidth available for real-time isochronous data.

**Bus manager** The BM is responsible for advanced bus power management, maintenance of speed and topological maps, and bus optimisation based on information from the topological map. The power management ensures that there is enough bus power available for all devices drawing current from the cable power supply lines. The speed and topological maps allow nodes to determine the maximum speed between any two nodes in the network.

## 3.6 SpaceWire

SpaceWire [10] is an emerging standard for high-speed data handling which is intended to meet the needs of future high-capability, remote-sensing instruments on spacecraft. SpaceWire is based on two existing standards: IEEE Std 1355-1995 [11] and LVDS [3]. They were combined and strengthened for use on board a spacecraft. Figure 3.12 shows a block diagram of a typical SpaceWire implementation.

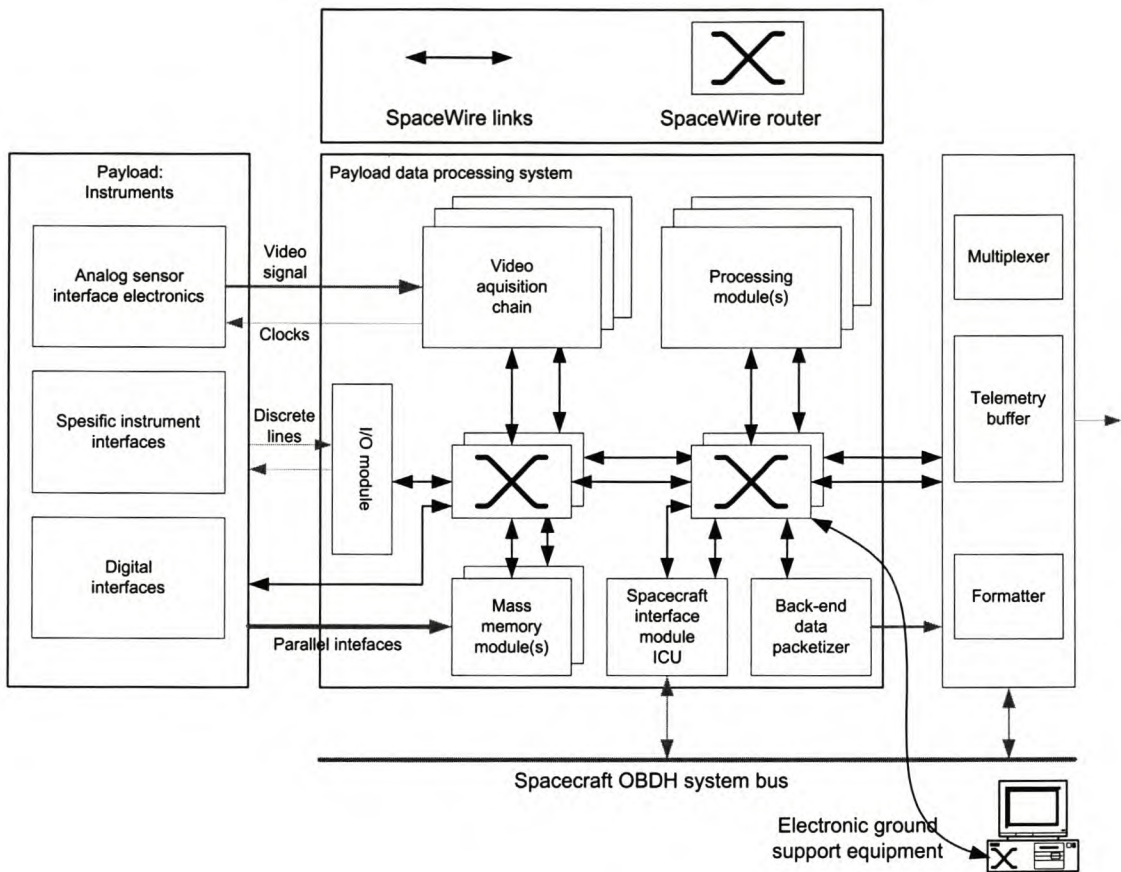
SpaceWire can be summed up as a full-duplex, bidirectional, serial, point-to-point data link. The minimum serial data rate for SpaceWire is specified as 2 Mbps. The maximum data signalling rate is dependent on the signal skew and jitter, and is therefore implementation dependent. Cable length, driver-receiver technology and encoder-decoder design are contributing factors. Data rates of 100 Mbps and more are possible. Cable lengths up to 10 m can be used, and the track layouts for PCB and backplane environments are also specified.

The SpaceWire specification covers the following protocol layers:

- Physical level
- Signal level
- Character level
- Exchange level
- Packet level
- Network level

In the **physical layer**, the cables, connectors, cable assemblies and printed circuit board (PCB) tracks are specified. A 9-pin micro-miniature D-type is specified as the SpaceWire connector because this type of connector has been qualified for use in space. Stringent EMC testing is specified by this layer.

The **signal layer** specifies the signal level, noise margins and data encoding of the bit stream. LVDS is specified as the signalling technique. LVDS has a number of very attractive properties for data handling in space. Some of these include low power consumption, high immunity to induced noise, and low EMI because of the small and opposite currents in the differential signalling lines. SpaceWire uses the same data-strobe (DS) signal encoding technique as the IEEE 1394 standard.



**Figure 3.12:** Block diagram of a typical SpaceWire implementation [12].

In the **character level** of the specification, the data bits are grouped into data characters (with 8 bits of data, and parity and data control bit flags) and control characters. These characters are used by the exchange level to do flow control, link error recovery, detection of parity errors, detection of disconnect errors and to do link initialisation. A state machine is specified to handle all these complex interactions between nodes and routers.

The **packet-level** protocol defines a packet structure with a destination address, a payload and an end-of-packet marker. This level also provides support for packet routing via wormhole routing switches.

A SpaceWire **network** is made up of a number of nodes interconnected by routing switches. Nodes are the interface to the application systems. They are the source and destination devices on the network. Nodes can directly be connected to each other with a cable, or they can be linked together using SpaceWire routing switches. The routing switches have multiple link inputs and use a routing matrix to send the packet to any of the link outputs.

## 3.7 Conclusions

This chapter discussed all the well-known serial bus standards. The first group of standards (RS232, RS422, RS486 and LVDS) consists of signalling standards. These standards will form the physical layer of the seven-layer OSI protocol stack. In a satellite environment, a protocol will have to be developed to communicate between subsystems.

The second group of standards (USB, SpaceWire and IEEE 1394) consists of transmission protocols in addition to the signalling standards. The USB standard, being a host-centric standard, would not easily fit into a system with high redundancy, since the central host is a single point failure node.

SpaceWire was explicitly designed to be used on spacecraft. This makes it the preferred choice for this application. However, the space industry components are manufactured to tolerate high levels of radiation, which makes these devices extremely expensive. Low earth orbit (LEO) satellite projects normally use commercial of-the-shelf products to reduce the cost of the satellite.

The remaining option is IEEE 1394. Since IEEE 1394 devices are designed for consumer products, component costs will be acceptable.

## Chapter 4

# Architecture and component selection

This chapter describes the overall architecture of the demonstration system that will be developed. The basic ideas of the design goals described in chapter 2 will be tested with this demonstration system.

The second part of the chapter describes the component selection that was done to implement the demonstration system, including components that were considered but rejected.

### 4.1 Architecture of the demonstration system

The overall architecture of the demonstration system is shown in figure 4.1. The data flow can easily be followed on the diagram. The video data enters at the camera and flows through the various blocks until it reaches the television (TV) block at the end.

In figure 4.1 on page 24, the dotted lines indicate the partitioning of the system. For the camera and TV blocks, commercial units will be used. The remaining two partitions have to be developed. Strong similarities can be seen between these two partitions. Thus similar blocks can be reused between the two partitions, such as the power supply and bus interface blocks.

In the original design goals of chapter 2, a separate control unit was envisioned. The functions can also be performed by the microprocessor in one of the two partitions. To keep the design simple, it was decided to incorporate the control unit into either the source or sink module.

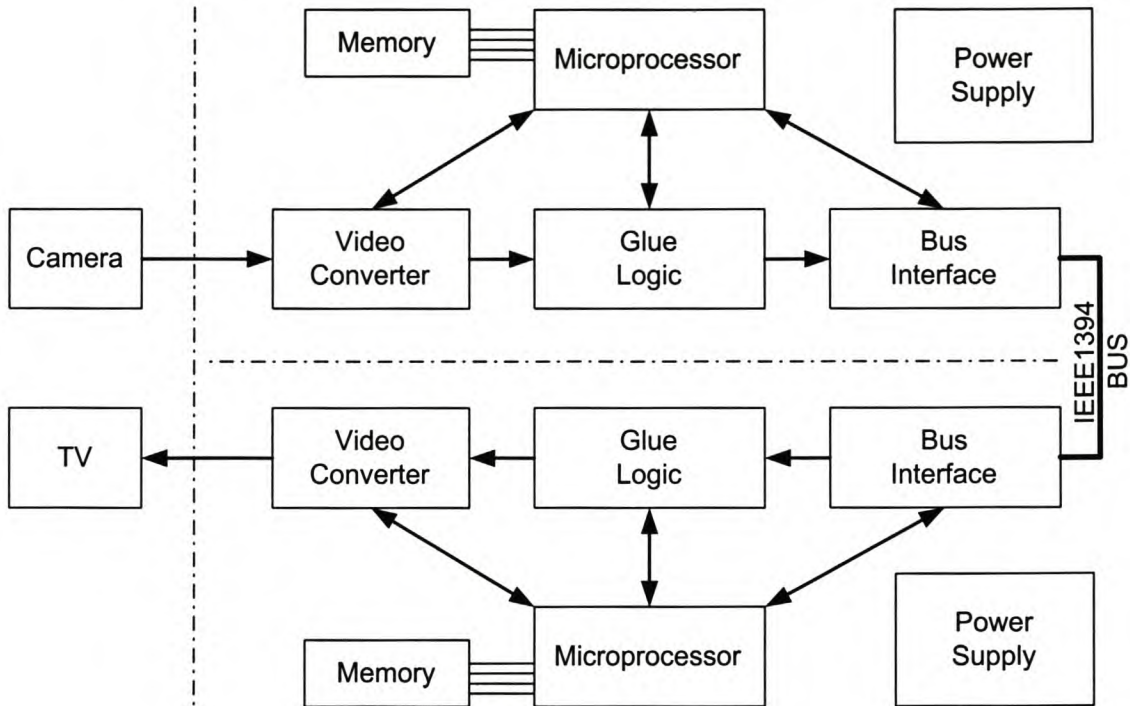


Figure 4.1: *Architecture of the demonstration system.*

The upper and lower partitions will be called the source and sink modules respectively. The implementation of the glue logic will be done in a FPGA device to keep the design flexible.

The software in the microprocessor should only be responsible for the management of the video stream, for example by starting and stopping the stream. The actual data transfer should be handled directly by hardware. The reason for this is to enable the use of a low-power microprocessor. If the microprocessor had to transfer the video data using software, a very fast processor would be needed. An acceptable compromise would be if the video stream data is transferred using a DMA controller on the microprocessor.

## 4.2 Bus interface hardware

This aim of this project is to demonstrate the use of the IEEE 1394 serial bus. An IEEE 1394 interface can be broken down into two devices. The first consists of the physical layer (PHY) devices, and is responsible for connecting to the wire and for handling all the signalling and timing functions of the interface. The PHY device is also responsible for repeating a signal on one port to all other ports on the interface (if there is more than one port on the interface).

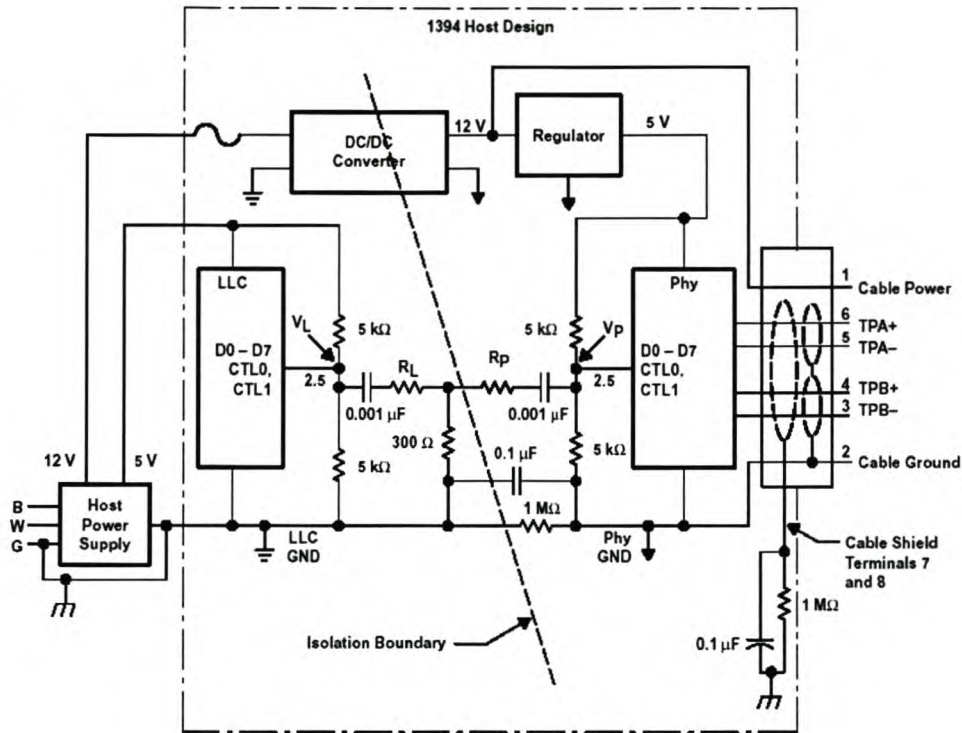


Figure 4.2: Galvanic isolation between PHY and LLC [13].

The second device is the link-layer controller (LLC). This device is responsible for constructing correctly formatted packets and to check the CRC of incoming packets. The LLC provides an interface to the microprocessor.

The main reason for two separate devices is to enable designers to implement galvanic isolation between the bus and the rest of a system. Isolation prevents ground loops between different systems which can cause signal digression. Also, because an IEEE 1394 PHY device acts like a repeater, the PHY cannot be switched off with the rest of the system. With isolation, the PHY can be powered from the IEEE 1394 bus, while the rest of the system, including LLC, can be powered from the power supply of the system. Figure 4.2 shows a typical galvanically isolated bus interface.

In order to simplify the design, galvanic isolation will not be implemented in this thesis. This can be done because there are only two modules that, for demonstration purposes, will be using the same power supply source. Because isolation is not used, integrated PHY/LLC devices can be considered.

Table 4.1 shows a list of LLC and PHY devices that can be used for the bus interface. The list is restricted to devices that are designed for embedded systems. Also, a PCI bus will not be used to connect to the microprocessor. The PCI constraint limited the choice of devices.

Manuf.	Number	Description
Texas	TSB12LV01	General-purpose LLC (32-bit CPU bus)
Texas	TSB12LV32	General-purpose LLC (8/16-bit CPU bus with data mover)
Texas	TSB41LV01	1-port PHY device
Texas	TSB41LV03	3-port PHY device
Philips	PDI1394L11	A/V LLC (8-bit CPU bus with A/V interface)
Philips	PDI1394L40	A/V LLC (8/16-bit CPU bus with full-duplex A/V interf.)
Philips	PDI1394P21	3-port PHY device

**Table 4.1:** *Link-layer controller and physical-layer devices.*

### 4.2.1 Texas Instruments TSB12LV01B

The TSB12LV01B chip was designed to be used by 32-bit embedded processors. A 32-bit data bus provides access to the internal registers and FIFO buffers in the chip. The link layer controller (LLC) is interrupt driven, and it is therefore unnecessary for the program to poll the interface.

Figure 4.3 shows a block diagram of the LLC's functions. The TSB12LV01B is not really suited for this application. The data stream needs to be inserted into an internal FIFO buffer by the microprocessor. Unless the microprocessor has a very good DMA controller, the load from the video stream could be too much for a common embedded processor.

### 4.2.2 Philips PDI1394L11 and PDI1394L40

The PDI1394L11 and PDI1394L40 were designed by Philips for embedded audio/video (AV) equipment. Figure 4.4 shows the block diagram of the PDI1394L11 device. The LLC has an 8-bit microprocessor interface and a separate AV interface unit. The AV interface unit is responsible for accepting 8-bit AV data from various MPEG-2 and DVC codecs. The data is then packaged into the correct format, timestamped (if needed) and sent to the IEEE 1394 bus. The AV unit can also receive packets from the bus, strip headers and present the data to a codec.

The PDI1394L40 chip is similar to the PDI1394L11, but has a 16-bit microprocessor interface and two AV interface units. The block diagram of the PDI1394L40 is shown in figure 4.5.

Compared to the TSB12LV01B, these chips are better suited to the application under consideration. It has a simple microprocessor interface with a second port to allow high-



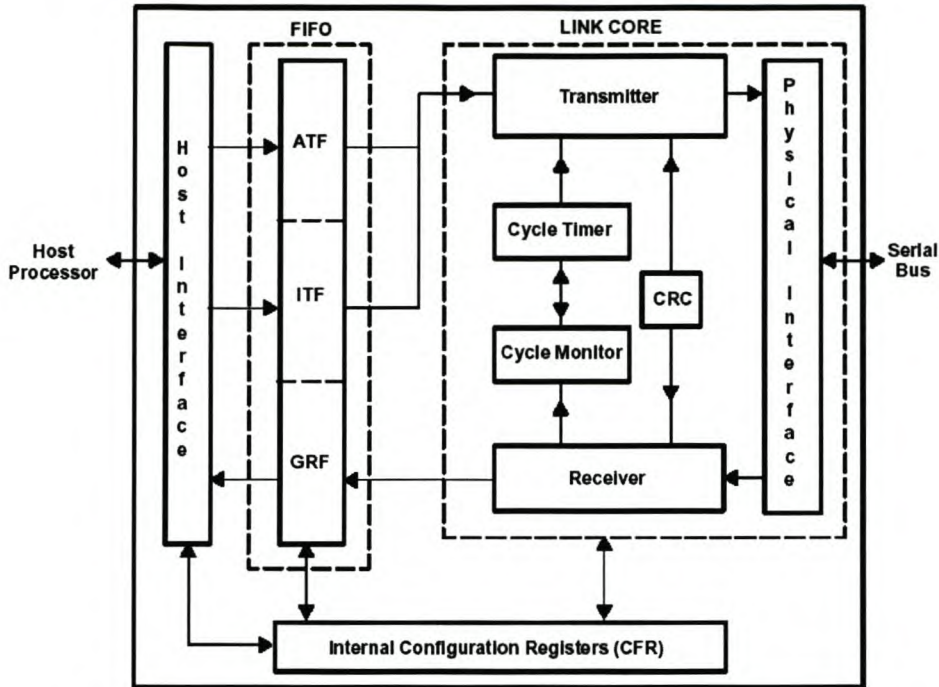


Figure 4.3: Block diagram of the TSB12LV01B LLC device [14].

speed access via external hardware. The microprocessor can thus also be small, because it only has to setup the control registers.

The two Philips chips were designed for a very specific application. On a microsatellite, the data stream will most probably not be a video data stream. Thus a more general-purpose data interface will be required.

### 4.2.3 Texas Instruments TSB12LV32

The TSB12LV32 IC from Texas Instruments is a general-purpose LLC for embedded applications. Figure 4.6 shows the block diagram of the device. The TSB12LV32 is also known as GP2Lynx in [17].

The TSB12LV32 LLC is ideally suited for the current application. The chip has a microprocessor interface (host port) that can connect to either 8-bit or 16-bit microprocessors. A data mover (DM) port is available for the high-speed data. This port accepts either 8-bit or 16-bit data streams.

The DM port can automatically add packet headers to the data before transmission. This means that no glue logic is required to packetise some data streams before transmission. On the receiver side, headers can be stripped before data is presented on the data mover port.

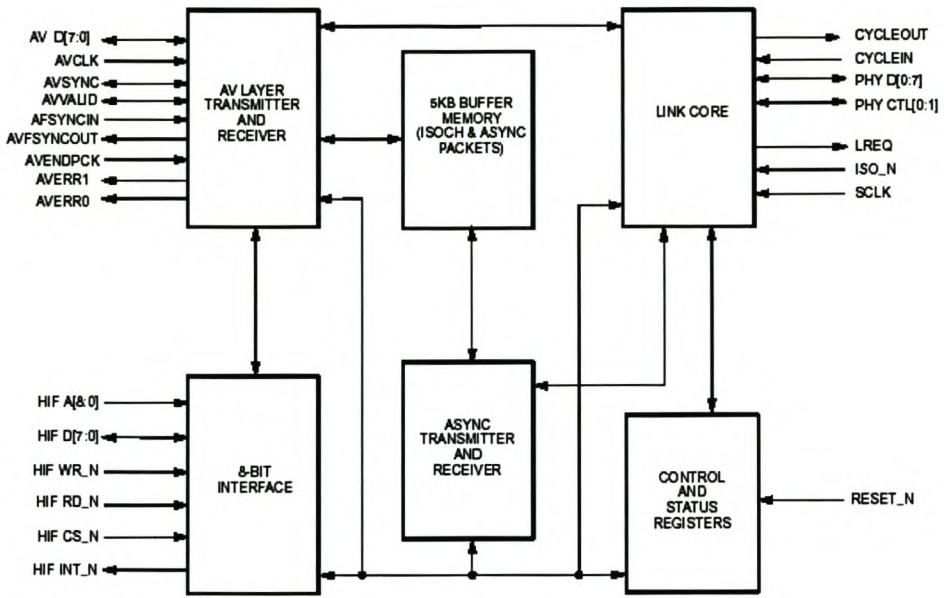


Figure 4.4: Block diagram of the PDI1394L11 LLC device [15].

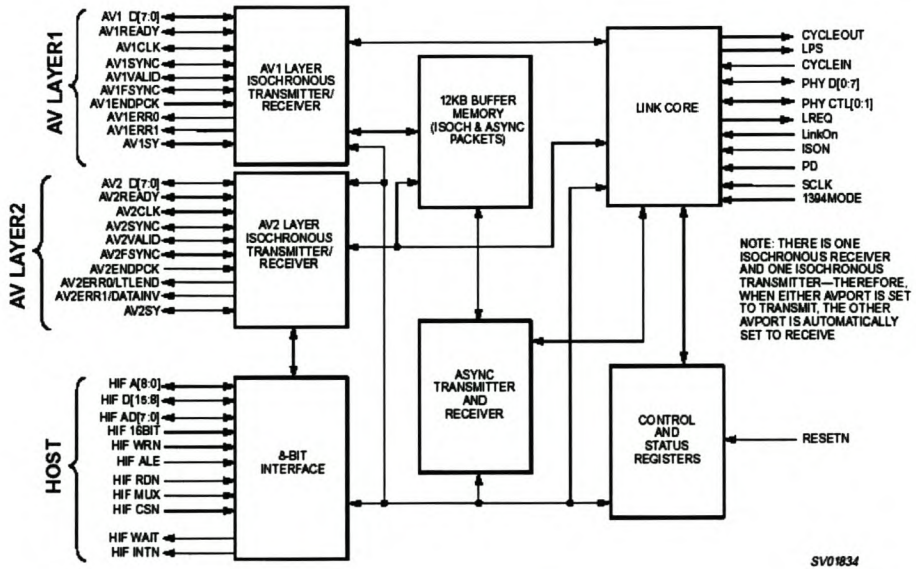


Figure 4.5: Block diagram of the PDI1394L40 LLC device [16].

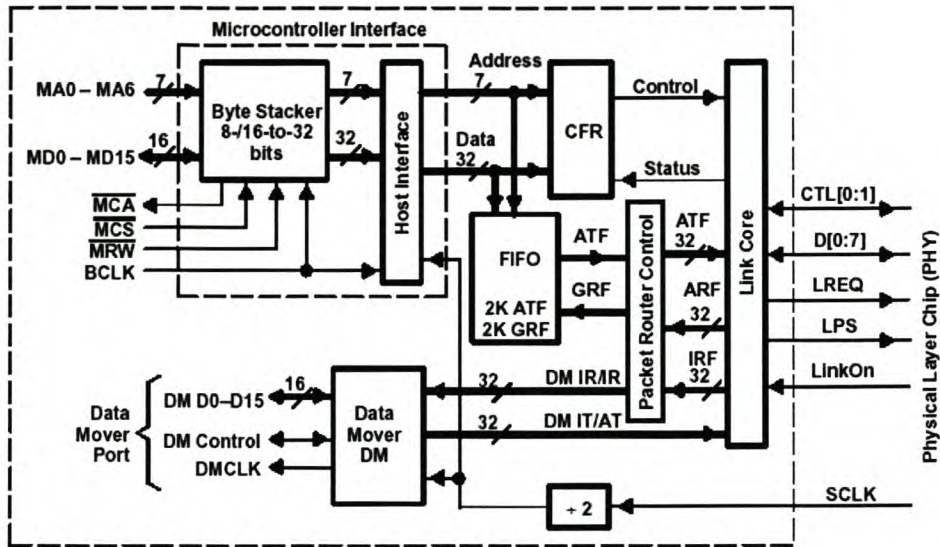


Figure 4.6: Block diagram of the TSB12LV32 LLC device [17].

The TSB12LV32 LLC was eventually selected as the LLC for this project, mainly because it is a general-purpose LLC for embedded systems.

#### 4.2.4 Texas Instruments TSB42LV03

For this project it was decided to have 3 ports on the serial bus module. The TSB42LV03, from Texas Instruments, was selected as physical layer devices. This decision was made primarily for compatibility reasons. A single supplier's devices should interoperate easily.

### 4.3 Microprocessor

The next selection was the choice for the microprocessor of the modules. The processor will be responsible for the overall control of the system.

The IEEE 1394 PHY device provides a 50 MHz clock to the LLC. This frequency was the chosen as the basis for the microprocessor selection. Since a general-purpose module is being designed, the required minimum processor speed (measured in MIPS) is not known beforehand. One possibility that was considered was to use 50 MHz clock from the PHY to also drive the microprocessor.

For this project only the simple on-chip peripherals will be needed: A UART for serial debugging, a direct memory access (DMA) controller to transfer data efficiently, and

an I<sup>2</sup>C controller. A good bus manager that can access different kinds of memory and external devices would also be advantages.

Another selection factor was the development software for the microprocessor. The GNU toolchain (described in section 4.9.1.2) supports a large number of microprocessor architectures, so choosing one with GNU support would not be very difficult.

The microprocessor must also have enough processing resources to run a Linux kernel (or at least a  $\mu$ Clinux kernel). This means that 32-bit processor architectures should be considered.

The TSB12LV32 link-layer controller has support for glueless connection to the Coldfire series of microprocessors from Motorola. This processor was investigated first.

### 4.3.1 Motorola Coldfire series of processors

The Coldfire range of processors from Motorola is designed for embedded applications (i.e. low power consumption) and is based on their popular 68K-series of processors.

In the Coldfire range, the closest processor to the frequency range chosen for this project is the MCF5206e processor. This is a very general-purpose processor with a maximum clock speed of 54 MHz. The processor has a flexible set of on-chip peripherals, including 2 USARTs, a 2-channel DMA controller and an I<sup>2</sup>C controller. All of these peripherals are well suited to this project.

Another popular Coldfire processor is the MCF5272 processor. This processor has a clock frequency of 66 MHz and has advanced on-chip peripherals, for example an Ethernet controller, a HDLC module and a USB 1.1 interface.

### 4.3.2 Hitachi H8/300H series of processors

Hitachi manufactures a large range of embedded processors. The author previously worked with the H8/300H range. This is a 16-bit architecture running at a clock frequency of up to 25 MHz. The 16-bit architecture of these processors will cause technical problems running a Linux kernel. The low clock frequency also makes this microprocessor series unsuitable for this project.

### 4.3.3 ARM Evaluator-7T evaluation board

During one of the author's study courses, the ARM Evaluator-7T evaluation board was used. This board contains a Samsung KS32C50100<sup>1</sup> microprocessor which is based on ARM's ARM7TDMI core.

The microprocessor runs at a clock frequency of 50 MHz. The most prominent on-chip peripherals of this microprocessor include two UARTs, a 2-channel DMA controller, an inter-IC control (I<sup>2</sup>C) controller, two HDLC controllers and a Ethernet interface.

The Evaluator-7T evaluation board contains 512 kB of flash memory and 512 kB of SRAM.

### 4.3.4 Microprocessor selection

Finally, the ARM evaluation board was chosen, primarily because it was already available in the department. The other processors would have required a new PCB design.

The most notable drawback of the evaluation board is the limited amount of RAM. It would have been desirable to run Linux or  $\mu$ Clinux on this processor, but the kernel needs at least 4 MB of RAM. The ARM Evaluator-7T board has expansion headers for all the control and data lines to the microprocessor. Interfacing to other boards only requires a suitable ribbon cable. A memory extension board can also be added if the need arises.

## 4.4 Device selection for glue logic

The glue logic blocks in figure 4.1 will allow the LLC to interface with any type of device or subsystem. For maximum flexibility a field-programmable gate array (FPGA) will be needed to implement the glue logic. With a FPGA, the glue logic can be written in the VHDL language. The compiled output of the VHDL code is stored in an EEPROM device and is loaded into the FPGA when power is applied. As a result, development of the glue logic is simplified and requires no modification to the hardware.

The Altera ACEX 1K series FPGA was chosen for the task. The primary reason for choosing this FPGA is the availability of internal memory blocks that could be used for FIFO or RAM space. Also, the ACEX is the low-cost series of FPGAs from Altera. In

---

<sup>1</sup>Also known as the Samsung S3C4510X01 microprocessor.

Manuf.	Number	Description
Philips	SAA7111A	Enhanced video input processor
Philips	SAA7113H	9-bit video input processor
Philips	SAA7114H	PAL/NTSC/SECAM video decoder
Philips	SAA7120	Digital video encoder
Philips	SAA7187	Digital video encoder

**Table 4.2:** *Video converter devices.*

particular, an EP1K30QC208-3 device was chosen. This is the second smallest device in the ACEX range. All the ACEX devices share the same PCB footprint and pin-out. Thus, if the EP1K30 device proves to be too small, a bigger device can be used, without any modifications to the PCB.

Actel, Atmel and Xilinx products were not considered for this project because a full license for Altera's development software, MaxPlusII and QuartusII, was readily available. If another manufacturer was chosen, a new set of development tools would have had to be bought, which would have increased the cost of the design.

The FPGA configuration is stored in an EEPROM. This type of EEPROM is manufactured by Atmel and Altera. Altera makes the EPC2 device with a 1 695 680 x 1-bit capacity. Atmel makes the AT17C512A device with a 512 k x 1-bit capacity. The EPC2 device is also substantially more expensive than the corresponding Atmel device. The EP1K130 FPGA has a configuration size of 470 000 bits. Eventually the AT17C512A device was chosen as the configuration device for the project, because of the lower price.

## 4.5 Video converters

The demonstration application will use analogue PAL video signals as input and output. However, the IEEE 1394 bus is digital. Consequently, it is necessary to convert the video signal to and from the analogue domain with a converter device. Since the project's priority is the transfer of the data, it is not necessary to be overly concerned with the quality of the video signal. For this reason, a single-chip video converter would be preferred.

Philips manufactures a large range of audio and video processing devices. From this range a few single-chip video converters were evaluated and are summarised in table 4.2. These are all devices without a PCI interface.

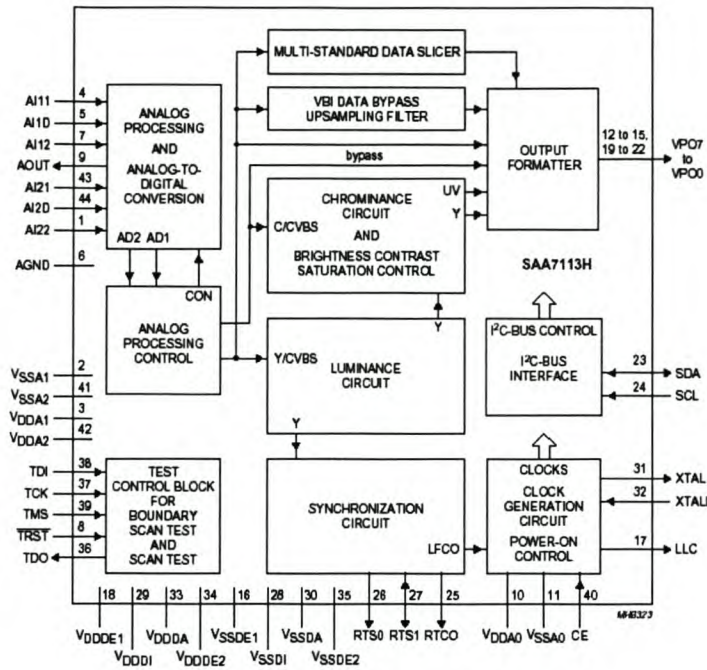


Figure 4.7: Block diagram of the Philips SAA7113H video signal converter [18].

## 4.6 Video input block

The first three devices in table 4.2 convert an analogue signal into a digital video stream. They are all single-chip solutions and can handle both NTSC and PAL video formats.

The SAA7114H is a high-end video converter. This chip can accept up to 6 composite video baseband signals (CVBSs) and outputs an 8-bit or 16-bit digital video stream. The SAA7114H includes various timing output pins, has digital expansion support, an audio clock generator and has an on-chip video scaling function for variable zoom. However, not all these features are useful in the demonstration application. Rather, a chip that can produce an 8-bit CCIR-656 video stream is needed.

The SAA7113H is an excellent example of such a chip. This chip can handle up to two CVBSs and has one digital 8-bit data bus. Figure 4.7 shows a block diagram of this device. Its functionality is similar to that of a typical satellite imager. The control registers for the chip are accessed via an I<sup>2</sup>C bus interface.

The SAA7113H was not commercially available at the time of the selection, so a replacement was chosen. The SAA7111A is very similar, but timing signals accompany the digital outputs. No header and tail identifiers are generated. Thus, with a little glue logic in a FPGA, a CCIR-656 video stream can be generated. Figure 4.8 shows the block diagram

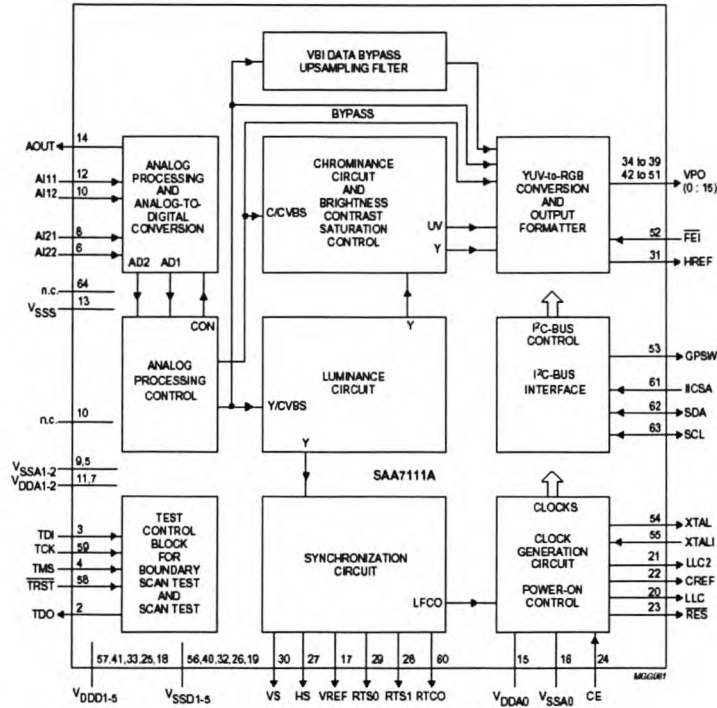


Figure 4.8: Block diagram of the Philips SAA7111A video signal converter [19].

of the SAA7111A device. Compared to the SAA7113, the block diagrams look almost the same. The SAA7111A has a wider data bus (16 bits) and a few more control and status pins. The TSB12LV32 also has a 16-bit data bus for the data mover interface, making it a good match.

The SAA7111A requires very few external components. A crystal is required to generate the clocks. The chip has two video input paths, which are multiplexed inside the chip. The functions of the video converter are controlled via an I<sup>2</sup>C bus interface.

## 4.7 Video output block

The last two devices in table 4.2 convert a digital video stream into an analogue signal. The SAA7120 can be considered to be the counterpart of the SAA7113 chip. It accepts a CCIR-656 byte stream and converts it to a CVBS and S-Video analogue signal. Figure 4.9 shows the block diagram of the SAA7120 chip. The control registers can be accessed via an I<sup>2</sup>C bus.

Again, the SAA7120 was not commercially available at the time of the selection. Another chip in the Philips video converter range was selected, namely the SAA7127H. Like the SAA7113H and SAA7111A devices, the SAA7127H device is functionally similar to



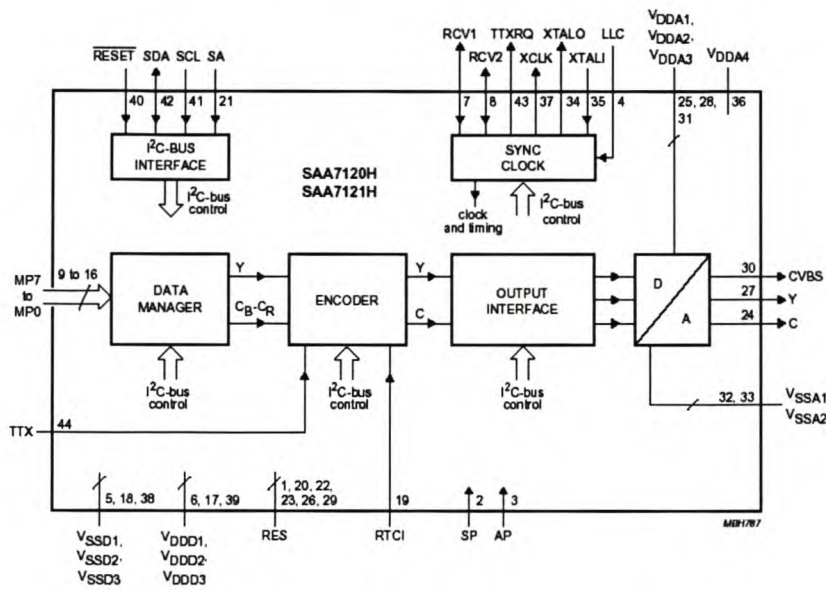


Figure 4.9: Block diagram of the Philips SAA7120 video converter [20].

the SAA7120, but with added features. Figure 4.10 shows the block diagram of the SAA7127H. The similarities with SAA7120 are quite evident.

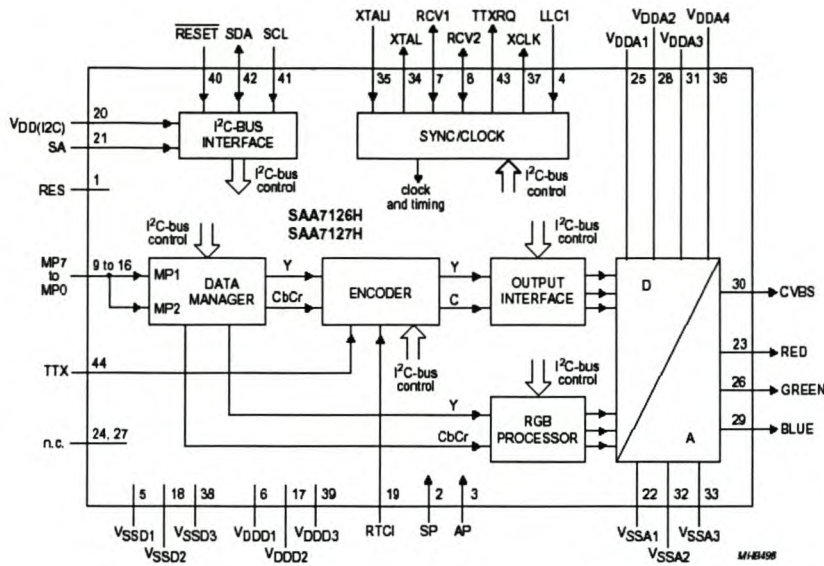
The SAA7127H is capable of handling two video streams in parallel. This allows designers to implement an on-screen display (OSD) functionality. However, this application uses a single video stream. The SAA7127H generates an analogue CVBS, and, unlike the SAA7120 device, RGB video signals. This allows high-fidelity video output. In this application, only the CVBS is required.

## 4.8 Power supply

The IEEE 1394 serial bus has two wires for a bus power supply. The bus power supply can handle voltages from 8 to 40 V (nominally 12 V). Thus the module's power supply must be able to handle that voltage range.

The IEEE 1394 devices are 3.3 V devices. The Altera FPGA requires 3.3 V for the I/O pins and 2.5 V for the internal core. At the time of the initial design, the FPGA configuration device was a 5 V device.

Consequently, three supply voltages were needed. To handle the large input voltages, a switched-mode power supply integrated circuit (IC) that could handle the wide input range was selected. Linear Technologies's LM2594HV switcher was chosen due to its 8–60 V input voltage range. This device delivers a regulated 5 V supply.



**Figure 4.10:** Block diagram of the Philips SAA7127H video converter [21].

The 5 V supply is the primary voltage source. From here, the 3.3 V and 2.5 V supplies are derived using a low-dropout linear regulator. Because the voltage drop between 5 V and 3.3/2.5 V supplies are low, there would not be a significant power loss. Linear regulators are also much simpler to use. The Linear Technologies LT1117 devices were chosen for the task.

## 4.9 Software languages and compiler tools

The two main design choices for software developments are, firstly, what language will be suitable for the project, and, secondly, which compiler for that language will be used.

The above choice has to be compatible with the intended target microprocessor. Thus both the choice of language and compiler are based on the microprocessor. For this design, the ARM7TDMI processor was chosen (see section 4.3).

### 4.9.1 C language

C is a small and very flexible language that is ideally suited for embedded applications. All the kernels that will be examined (see section 4.10) requires the use of the C language, and it was therefore the obvious choice for this project.

#### 4.9.1.1 ARM Developer's Suite

ARM supplies a complete developer's suite for all their processor architectures. The ARM Developer's Suite includes all the tools needed to develop software for the ARM7TDMI processor. It includes an assembler, a C compiler, a debugger and a graphical integrated development environment. Unfortunately, a copy of this suite was not readily available and it is very expensive. As a result it was decided to search for a less expensive set of development tools.

#### 4.9.1.2 GNU tools: GCC, Binutils and GDB

The GNU tool chain is an open-source project that develops a group of software development tools for various processor architectures. These include the ARM, i386, AVR, Alpha, PowerPC, MIPS, 68K and SH architectures, to name but a few.

The compiler, the GNU Compiler Collection (GCC), is able to compile code for multiple languages, including C, C++ and Fortran. The Java and Ada languages have also recently been added to the compiler. For this project, only the C/C++ language is required.

The binutils package includes an assembler and linker tool, and also various tools to view, manipulate and convert the final executable file. For embedded systems, the executable output needs to be converted to either a hex or binary file so that it can be programmed into a flash or EPROM device.

The debugger tool, the GNU Debugger (GDB), is a debugging program that can either simulate a program or execute it directly on a microprocessor. GDB uses a software stub to interface between the program running on the processor and the debugger. GDB is able to upload and run the program, set breakpoints and view variables and memory on the microprocessor.

Other tools in the GNU chain include the make utility that facilitates the automatic building of an executable. For this project, make was used to automate the process of compiling and linking the code.

The GNU tool chain can run on both Windows (in the CYGWIN environment) and Linux. The GNU tools were eventually chosen for this project, because most of the operating systems that were evaluated need the GCC compiler (see 4.10).

## 4.9.2 VHDL language

The glue logic that will be programmed into the FPGA must be written in a hardware description language. There are currently two predominant languages, namely VHSIC Hardware Description Language (VHDL) and Verilog. The VHDL language was chosen for this project because the author has done previous projects in this language.

### 4.9.2.1 Altera's MaxPlusII and QuartusII

The manufacturer of the selected FPGA also produces a software development suite to compile the FPGA code. Two packages are available from Altera, namely MaxPlusII and QuartusII. MaxPlusII is the older of the two suites. QuartusII is newer and has recently added support for the ACEX 1K series. QuartusII is also very resource intensive and slower, but has more advanced features.

For this project the MaxPlusII package was selected. This was because the intended glue logic does not require extensive VHDL coding, and none of the advanced features of the QuartusII package are really needed.

## 4.10 Operating system selection

The control software will be responsible for the management of the communications link between the modules. The software must also be able to do other tasks and control other devices, since the microprocessor might not be dedicated just to the communications link. It may even be one of the satellite's on-board computers. That means that many software threads must run on the microprocessor.

Consequently, an operating system (OS) must be selected for the microprocessor that will be responsible for running the different programs. The OS must be able to run in the limited memory space of this system. Also, the OS must be able to handle the real-time dynamics of an embedded environment. This section will investigate a few OS alternatives.

### 4.10.1 RTX

RTX is a small, non-preemptive kernel that was developed for embedded systems at the University of Stellenbosch [22]. The original kernel was modified and later used in the SUNSAT-1 project for the on-board computer software.

RTX is a non-preemptive, event-driven kernel with a very small memory footprint. This kernel follows a different design philosophy than normal preemptive kernels. As a result, any application written for this kernel is considered a custom program.

### 4.10.2 RTEMS

RTEMS is an OS that was originally developed by OAR Technologies for multiprocessor military systems. RTEMS is now an open-source project with the code available on the internet [23].

RTEMS has support for a broad range of microprocessor architectures and embedded systems, and can be used with the C, C++ and Ada programming languages. RTEMS supports the Motorola 68K and Coldfire processors, while support for the ARM architecture has been added to the current development version of the OS. Unfortunately, RTEMS requires a large memory footprint (about 600 kB), which is too big for the ARM Evaluator-7T board with 512 kB of SRAM. It might be possible to reduce this footprint by removing some of the kernel's features, but this has not been verified.

### 4.10.3 Linux or $\mu$ Clinux

Linux is becoming very prominent in embedded systems. This kernel already has support for the ARM, 68K and other architectures. Linux is a very large and complex OS that runs on platforms ranging from small embedded systems and desktop PCs, up to large IBM mainframes. This means that there is a large base of users for this kernel. Unfortunately, the standard Linux kernel requires a 32-bit processor with a memory management unit (MMU). The selected processor, an ARM7TDMI core, does not have a MMU. The  $\mu$ Clinux project [24] is working on extending the standard Linux kernel to run on processors without a MMU.

The kernel has been ported to many processors, including Motorola's 68K/Coldfire and ARM's ARM7TDMI processors. A benefit of using the Linux kernel would be the use of its IEEE 1394 device driver. This means that a driver does not need to be developed from scratch for this project. Although  $\mu$ Clinux is designed for embedded systems, it still requires more memory resources than is available on the ARM Evaluator-7T board. One solution would be to add more memory to the Evaluator-7T board.

#### 4.10.4 eCos

The Embedded Configurable Operating System (eCos) [25] is a highly modular real-time kernel that was developed by RedHat for embedded systems. The Embedded Configurable Operating System (eCos) allows the user to select only the individual components and features that are required for a particular application. This ensures that eCos uses a minimal amount of memory.

eCos also has support for a very large range of embedded processors, including the 68K/Coldfire, ARM, i386 and SH architectures. eCos has well-documented support for the ARM Evaluator-7T board. This means that eCos can be used without requiring custom-written code for the hardware.

#### 4.10.5 Conclusion

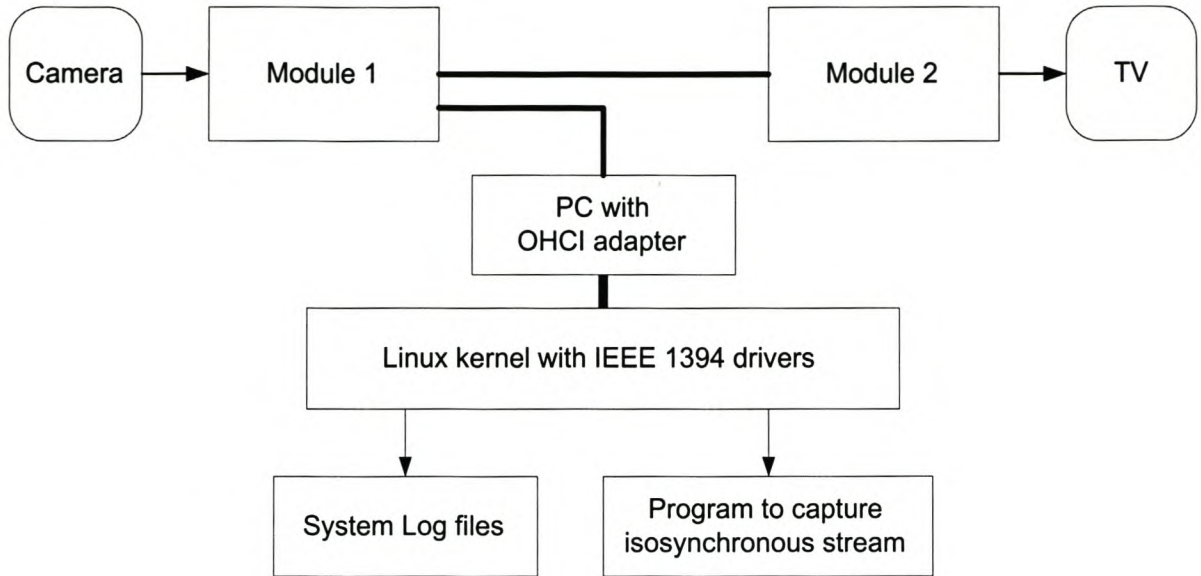
eCos was eventually chosen, because support for the ARM Evaluator-7T board was available and documented, thus facilitating the development of the software for the board. It was also the only kernel (other than RTX) that could easily fit into the small memory footprint of the ARM Evaluator-7T board without modification to the hardware.

### 4.11 Test environment

The goal of this project is to develop two modules that are able to connect to each other and transfer a video stream. To help with this development and to verify the data transfer, a method is needed to capture the traffic that is generated on the IEEE 1394 bus.

The ideal solution would be to use an IEEE 1394 bus analyser. This is a special piece of test equipment that is capable of capturing all packets on the bus, and analyse both the validity of the packets as well as the timing of the transactions. Unfortunately, a bus analyser is very expensive and could not be afforded for this project.

The next option would be to use a packet sniffer program on a PC. To do this, one needs a PCI adapter card that can be placed in promiscuous mode (also known as snoop mode). A sniffer program will then log all packets that pass on the bus, without any timing information. Unfortunately, the normal OHCI cards that are available on the market does not have a snoop mode. The only card that do have a snoop mode, is a PCI adapter card that uses the Texas Instruments PCILynx (TSB12LV21) chipset. Unfortunately,



**Figure 4.11:** *Block diagram of the test environment.*

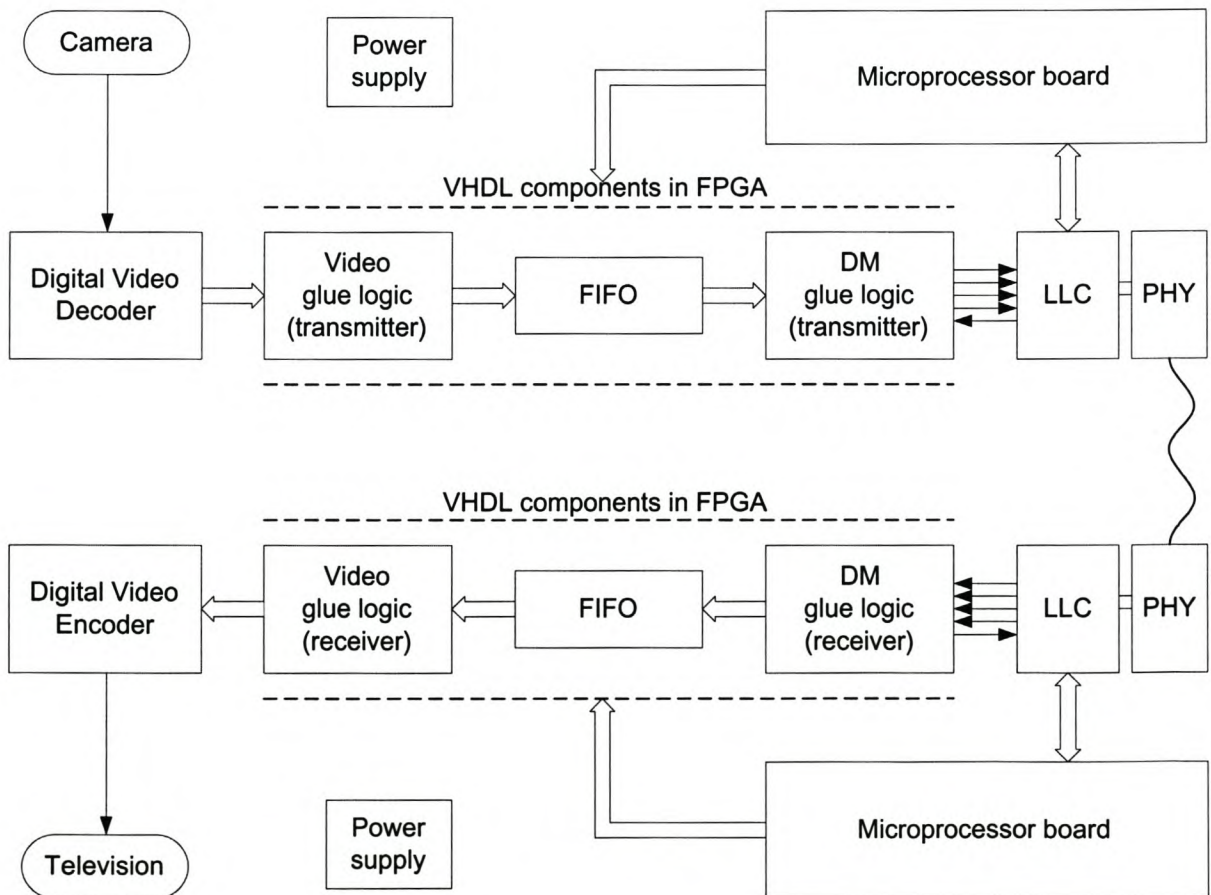
these cards are more expensive than the normal OHCI cards and most companies only stock OHCI cards. They are almost impossible to find.

So, with the absence of a snooping-enabled card, a normal PC with an OHCI card had to be used. The Linux kernel's IEEE 1394 driver has the ability to capture a wealth of information in the system logs. These log entries will be able to reveal what has happened on the bus (e.g. a bus reset occurred). Also, it is possible to listen to the isochronous channels and capture that data stream to disk. This makes it possible to validate the generated data stream.

# Chapter 5

## Design details

Figure 5.1 shows the overall block diagram for the transmission path that the data will take. This video data starts at a normal analogue PAL camera. Next, it is converted into digital form by the video encoder and transferred by the video glue logic into a first-in-first-out (FIFO) buffer.



**Figure 5.1:** Overall block diagram of the transmission of the data.



The DM glue logic lets the LLC know when there is data available for transmission. The LLC then initiates isochronous transfer of the data from the FIFO buffer through the DM to the IEEE 1394 bus. The LLC is connected to the PHY which in turn is responsible for connecting to the cable.

On the receiver side, the PHY connects to the LLC and the LLC-DM connect to the receiver DM logic. This logic is responsible for accepting the data and inserting it into the FIFO buffer. On the other side of the FIFO buffer, the data is extracted and synchronised. The analogue video signal is then reconstructed from the digital data, and fed to a TV for display.

All this is overseen by a pair of microprocessors that are responsible for establishing the link and keeping control of the link.

The following sections describe the design of each of the blocks in figure 5.1.

## 5.1 Video source

The video source for this demonstration application will be any standard analogue PAL video signal. PAL-format video is the standard used in South Africa for television transmissions. More specifically, the application will use the composite video signal from a small, fixed-focus video camera that is normally used in security applications.

This camera requires a 12 V power supply, which will be provided by a bench DC source. The video signal from this camera will go directly to the video converter without any processing or amplification.

## 5.2 Video converter

The video converter is the Philips SAA7111A video encoder. This chip converts an analogue video signal (CVBS or S-Video) to a digital stream. This IC supports various digital video transmission standards, including the CCIR-601 (YUV 4:1:1 and YUV 4:2:2) and RGB video streams.

The video converter was placed on its own PCB to keep the overall design modular. The video converter module can later be replaced by a module that connect to a satellite imager or other high-speed devices.

The SAA7111A is a single-chip solution that requires very few external components. To keep the design simple, the suggested application diagram from the datasheet [19] on page 39 was followed closely.

The SAA7111A chip has a multiplexed front-end that can handle either 4 CVBS or 2 S-Video signals. Most VHS recorders and cameras output at least a CVBS signal. Two RCA sockets were placed on the PCB for easy connection to consumer video devices. A 6x1-pin header with all the input signals was placed on the PCB for future experimentation. The video converter uses a 3.3 V supply voltage. Since this board will be connected to the communications board, it was decided not to add a voltage regulator. The power supply will come from the ribbon cable connected to the communications board. A PCB footprint for a 2.1 mm power socket and polarity diode was added for future experimentation.

The SAA7111A does not require complex peripheral components. The oscillator requires a 24.57 MHz crystal to generate the master clock used to convert the video signal. The oscillator circuit consists of the crystal, decoupling capacitors and an inductor.

The video input lines consist of a simple voltage divider and a coupling capacitor. These values were taken directly from the SAA7111A data sheet.

A 26-pin IDC header is used to connect the video converter to the communications module. On the ribbon cable, lines 0 to 15 are the data lines. The next four lines are used for video control lines. The next two are for the I<sup>2</sup>C bus. The 27 MHz video clock is next to the power supply lines to help prevent crosstalk with the other signal wires. The last wire is the ground line.

The SAA7111A provides quite a few video control lines. All of them were brought to a 16-pin header for future experimentation. Four of these lines were selected to go to the communications module with the data lines, namely the CREF, HREF, RTS0 and VS lines.

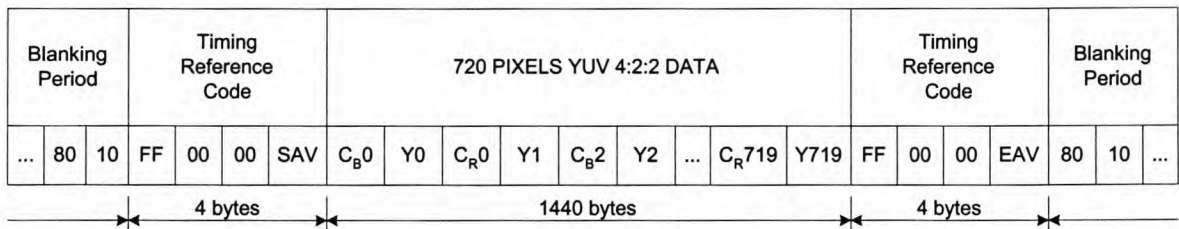
The CREF line is low when the data on the data bus is valid. The HREF line is the horizontal line sync and is high when the scanning position is inside the active video region. The PAL signal is an interleaved video format with an ODD and an EVEN frame. The RTS0 allows the designer to distinguish between the ODD and EVEN frame. The last signal line goes to the VS line, and is the vertical line sync. With these four signal wires it should be possible to construct a CCIR-656 data stream. The next section will describe how this is achieved.

### 5.3 Video glue logic for the transmitter

The SAA7111A's digital data stream does not contain any synchronisation control data. This data is provided by the 4 control lines.

There are two ways to transfer the synchronisation data between the two endpoints using the IEEE 1394 bus. The first option is to create one packet for each horizontal line. An IEEE 1394 isochronous data packet has a maximum data length of 2048 bytes at a speed of 200 Mbps, and each horizontal line has an active region length of 1440 bytes. Synchronisation information can be stored in each packet's header.

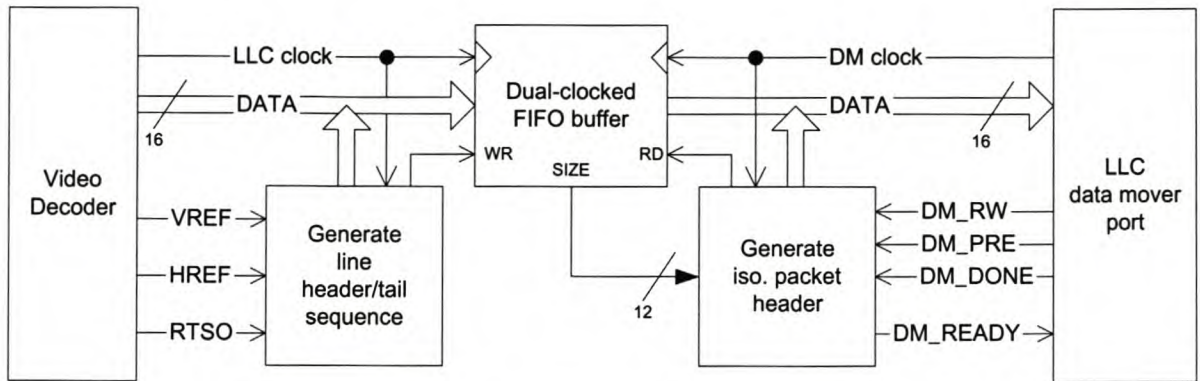
The second option is to use the CCIR-656 standard to embed the synchronisation control data inside the data stream. With this standard, a header and tail is added to each horizontal scan line. The header and tail contain a unique byte pattern that can be used to identify their presence. Thus the beginning and end of a horizontal scan line can be retrieved from the data stream. The header and tail also contain control information to identify the VSYNC and FRAME information. With this extra data, a full video signal can be constructed from the data stream.



**Figure 5.2:** CCIR-656 8-bit digital video data stream.

A choice had to be made between the two transmission systems. The first has the advantage that the line data is packaged nicely in one packet which can easily be tracked. The second has the advantage that the data can be treated as a continuous byte stream without any special controls. Consequently, the data can be inserted into a general FIFO buffer to await transmission. It was finally decided to combine the two methods and send a signal packet per scan line, with embedded CCIR-656 header and tail bytes.

The selected FPGA has only 3 kB of on-chip memory. This means that the maximum FIFO buffer size is 3 kB. The IEEE 1394 bus uses a 125  $\mu$ s isochronous cycle. The video signal arrives at a rate of 27 MBps, which corresponds to 3375 bytes per 125  $\mu$ s cycle. As a result, a full video stream cannot be buffered. One solution would be to transmit only the active video region of a horizontal line, as the blanking period contribute no data to the video signal.



**Figure 5.3:** Block diagram of the video and data mover glue logic components.

The video glue logic is responsible for inserting the header and tail bytes just before and after the horizontal line data. The glue logic then only loads the active video data and the header and tail bytes into the FIFO buffer to await transmission.

The VHDL code has a state machine that monitors the input control lines to determine when the header and tail sequences have to be inserted into the digital stream. A multiplexer is used to feed the data stream to the FIFO buffer. The multiplexer selects between the incoming data and the header-tail register. The incoming data is delayed by two registers to allow the header-tail value to be inserted into the stream. The source code listing for this module is available in appendix B.1.

## 5.4 Data FIFO buffer for the transmitter

Once the data is ready, it is placed in a FIFO buffer to wait for the next IEEE 1394 isochronous cycle to start. An isochronous cycle has a fixed period of  $125\ \mu\text{s}$ . A PAL video signal has a horizontal scan line period of  $64\ \mu\text{s}$ . Thus there will be 2 packets per  $128\ \mu\text{s}$  period.

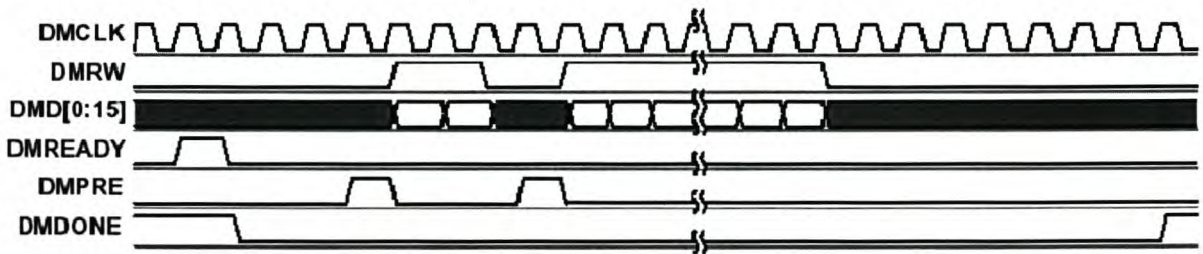
The buffer must have space for at least 3 horizontal lines. Two will wait while the third is being assembled. Unfortunately, the chosen FPGA only has 3 kB of on-chip memory. Three horizontal lines will need more than 3 kB of memory. This means that a complete scan line cannot be sent for each packet. The alternative, which was discussed in the previous section, is to embed the synchronisation data in the data stream and to send variable-length packets.

For this option a FIFO buffer is used to store the data. The IEEE 1394 bus will then be responsible for moving the data without any synchronisation control data in the packet header.

A dual-clock FIFO buffer is used to separate the design into two independent clock domains. The DM domain uses a 25 MHz clock and the video domain uses a 27 MHz clock. A standard Altera LPM component was used for this buffer. The LPM components are optimised to work with the hardware available in the FPGA.

## 5.5 Data mover glue logic

The DM's glue logic is responsible for constructing an isochronous packet header for the LLC's data mover port. The TSB12LV32 has an automatic header insert mode that performs a similar function to this glue logic. However, the automatic header uses only constants, meaning that packets can only have a fixed length and fixed synchronisation data. In this application it must be possible to send variable-length packets and to add custom synchronisation data to the header.



**Figure 5.4:** *Timing diagram of a data mover read cycle [17].*

The glue logic tells the TSB12LV32 that there is data available by driving the DMREADY line high. The LLC responds by driving the DMDONE line low. Next, the DM reads the packet header by pulsing the DMPRE line and reading 4 bytes. When the DMPRE line is pulsed, the glue logic loads the number of bytes waiting in the FIFO buffer into the header. In this way, the LLC knows how many bytes to read from the FIFO buffer when sending the packets. Any data arriving in the FIFO buffer after the header was constructed, has to wait until the next cycle (125  $\mu$ s). Figure 5.4 shows the timing diagram for an isochronous packet transmission cycle by the data mover.

The VHDL code describes a state machine that is used to drive a multiplexer. The multiplexer selects between the data from the FIFO buffer and a register that holds the header value. An IEEE 1394 isochronous packet header consists of 4 bytes. The upper two bytes are the length of the packet. Thus, when the header is read by the DM, the current length of the FIFO buffer is used. The last two bytes hold the packet type and synchronisation data that are programmed into a control register by the microprocessor.

## 5.6 IEEE 1394 bus interface

The IEEE 1394 interface consists of two ICs, the TSB43LV03 and the TSB12LV32, as were selected in section 4.2. The following two sections describe the implementation details of these two chips.

### 5.6.1 TSB43LV03, physical-layer device

Each node in an IEEE 1394 bus has at least one PHY. The PHY is responsible for repeating all data arriving on one port to the other two ports. For this reason, the PHY device is normally powered from the IEEE 1394 bus power line.

The PHY needs a 24.576 MHz crystal to generate the 50 MHz system clock. The oscillator circuit consists of a crystal and two 22 pF coupling capacitors to ground. The crystal is connected to the XI and X0 pins on the chip. A single 0.1  $\mu$ F capacitor is required as a filter capacitor for the internal phase-locked loop (PLL) circuit.

The PHY needs a resistor to set the internal operating current and cable driver output currents to meet the required IEEE Std 1394-1995 specification. The resistor value between terminals R0 and R1 should be  $6.3\text{ k}\Omega \pm 0.5\%$ . Since this is an uncommon resistor value, two resistors in parallel could be used. The one resistor should have a value of  $6.34\text{ k}\Omega \pm 1\%$ , while the other resistor should be  $1\text{ M}\Omega \pm 1\%$ .

The PHY is connected to the IEEE 1394 socket via a termination resistor network. The IEEE 1394 cable has a cable impedance of  $110\ \Omega$ . Figure 5.5 shows the configuration of the termination network. All the resistor tolerances should be within 1% to keep the system tolerances close to the IEEE 1394 specification.



**Figure 5.5:** *The IEEE 1394 cable termination network.*

The TBIAS line provides the 1.86 V nominal bias voltage (common-mode voltage) needed for proper operation of the twisted-pair cable drivers and receivers, and for signalling to

the remote nodes that there is an active cable connection. Inside the IEEE 1394 cable, the TPA and TPB pairs are crossed, which means that on the remote node the TPB pair will be used to detect the 1.86 V common-mode voltage.

The TSB43LV03 has three ports. There are three sets of termination networks connected to three sockets. The termination network should be as close to the PHY chip as possible.

The reset line of the PHY chip must be connected to the CONF\_DONE line of the FPGA. Originally, the INIT\_DONE line of the FPGA was used, but it turns out that the INIT\_DONE line is high at power-on, and only goes low during the configuration phase. For this reason the CONF\_DONE line is used, because this line is low at power-on and stays low until the FPGA configuration has been loaded. In this way, the PHY is kept in reset mode until the FPGA has a valid configuration loaded. This also prevents any unnecessary traffic on the IEEE 1394 bus before the system is ready.

The power capability lines of the TSB43LV03 are attached to jumpers so that these values can be changed during development. The  $\overline{IS0}$  line is pulled high because galvanic isolation is not used between the PHY and the LLC chip. Adding this feature increases system complexity. Also, on a microsatellite all the subsystems are connected to the same power supply system.

### 5.6.2 TSB12LV32, link-layer controller

The TSB12LV32 LLC does not require any external components except a few pull-up resistors and decoupling capacitors on the power supplies. The DIRECT line is pulled high because the galvanic isolation is not used, as was discussed in the previous paragraph.

The 50 MHz system clock generated by the PHY is connected to the LLC's SCLK line. This clock provides the synchronisation between the LLC and PHY devices. The rest of the PHY interface lines should be connected directly to the PHY with the PCB tracks as short as possible.

The rest of the LLC's lines are connected directly to the FPGA, except for the LENDIAN and COLDFIRE lines, which are connected to a set of jumpers. This was done because these lines are not supposed to change during the operation of the system. The microprocessor will work in either big or little endian, but not both. In this way the value of the pin will be set with a jumper to the appropriate value for the microprocessor being used.

The reset line of the LLC is connected to the same line as that of the PHY. Thus, the LLC will be kept in reset mode while the FPGA is loading a new configuration. This

also helps to make sure that the LLC does not transmit something on the IEEE 1394 bus when the system is not fully initialised.

## 5.7 Data mover glue logic for the receiver

When an isochronous packet arrives on the IEEE 1394 bus addressed to the channel from which it is being received, the LLC will route that packet to the DM. The DM will then write the packet to the DM glue logic. This VHDL component will then write the packet directly into the FIFO buffer. The LLC must be configured to automatically remove the packet header.

Alternatively, the LLC can be configured to include the header with the packet data. This would allow synchronisation data to be extracted from the SYNC field in the header.

For this design it will be assumed that the data stream has embedded the CCIR-656 header and tail sequences in the data stream. This will simplify the design by letting the LLC remove the header. The VHDL code in appendix B.4 simply writes the data directly to the FIFO buffer without any modification.

## 5.8 FIFO buffer on the receiver side

The FIFO buffer on the receiver side is responsible for holding the video data until video glue logic is ready to process it. This FIFO buffer is the same size as the transmitter's buffer. The on-chip memory of the FPGA is used for the FIFO buffer. As a result, the buffer's size is limited to 3kB of memory for the Altera ACEX 1K30 FPGA that was selected for this design. The same LPM component is used as the one in the transmitter.

## 5.9 Video glue logic for the receiver

The video glue logic will extract the data stream from the FIFO buffer, determine the video line's header and tail, and reconstruct the correct video stream for the CCIR-656 standard. The transmitter only sends the video data in the active region. The blanking data is not transmitted, thus the glue logic will have to insert data into the data stream during the blanking periods. Figure 5.2 shows the required video data sequence. Finally, the video data is converted into an 8-bit wide data stream for the video encoder, as is described in section 5.10.



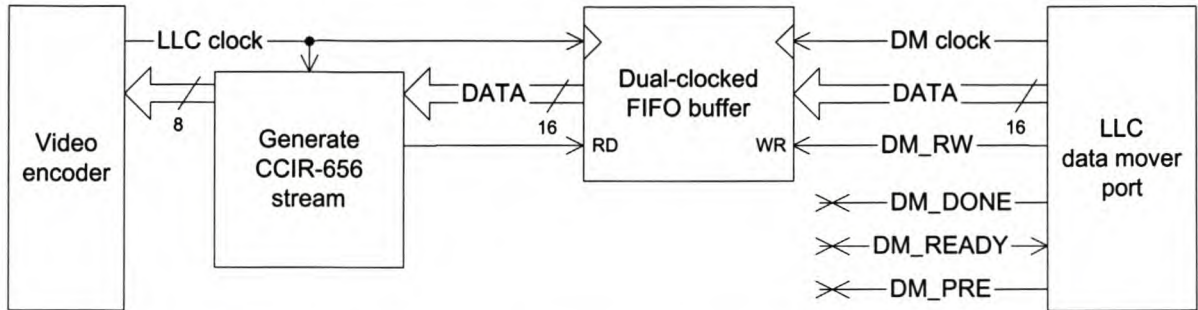


Figure 5.6: Block diagram of the receiver's glue logic.

## 5.10 Digital video encoder

The Philips SAA7127H IC is responsible for creating an analogue PAL video signal from the digital video stream. Figure 4.10 shows a block diagram of the device. An 8-bit wide data input bus accepts the video data from the video glue logic and outputs a CVBS signal along with the RGB signal components.

This IC requires very few external components. For clock generation a 27 MHz crystal, a few capacitors and an inductor is needed to generate the 27 MHz system clock. Internally the clock is multiplied up to 54 MHz to do the conversion.

The video encoder will be placed on a separate PCB to enhance modularity. This board is attached to the FPGA on the communications board through a 26-pin IDC header and ribbon cable. The pin layout is very similar to the video converter in section 5.2. Pins 0 to 7 is the digital video data lines, followed by the control lines. Since the video encoder only uses 8 data bits, there will be a few unused pins on the cable. The I<sup>2</sup>C bus lines followed by the clock and power supply lines is next. The last line is the ground line.

The two I<sup>2</sup>C lines (along with the power supply) is available on a 4-pin header for testing and future experimentation. The I<sup>2</sup>C address line is available on a jumper so that the I<sup>2</sup>C address of the unit can be changed.

No power supply regulators are added to this board, because the module will connect to the power supply of the communications board through the 26-pin ribbon cable. There is a 2-pin header where external power can be connected for development.

The video encoder's output signals are CVBS and RGB component signals. The CVBS signal is taken to an RCA connector. The RCA connector was chosen because this type of socket is almost universally used on television and VHS equipment. An alternative is a BNC coaxial socket that is used mainly on electronic test equipment like oscilloscopes. For this system, a RCA socket would be more useful.

## 5.11 Video sink

The video sink is a normal television or VHS recorder with a CVBS input. The analogue PAL video signal from the video generator is displayed by the television to demonstrate the data reception.

## 5.12 Microprocessor

The supervising microprocessor is responsible for the implementation of the IEEE 1394 protocol stack and the control software for the video link.

Earlier in this document, the Samsung KS32C50100 processor on the ARM Evaluator-7T evaluation board was selected as processor. Consequently, it is not necessary to design a new PCB for the microprocessor. A small interface PCB was designed for a previous project that takes all the data, address and control signals out to headers. All that is needed is to construct a ribbon cable to connect to the communications board.

The microprocessor is connected to the FPGA via two IDC headers. The first 26-pin header contains the 16-bit data bus and an 8-bit address bus. The extra two pins will be used for a 3.3 V power supply and a ground wire. The second 16-pin header contain the various processor control signals, such as chip select, output enable and write select. The I<sup>2</sup>C bus signals and the 50 MHz clock lines follow, with 3.3 V and 5 V supply lines and a ground line.

In the FPGA there is microprocessor interface glue logic that is responsible for the inter-connection between the microprocessor and the TSB12LV32's host port. The glue logic must also implement some control registers to control the functions of the other FPGA components used in the video data stream path.

### 5.12.1 Microprocessor and LLC host interface glue logic

The microprocessor and the LLC host interface VHDL module are responsible to connect the ARM microprocessor's data bus to the LLC's host interface. The ARM bus manager makes it possible to define I/O read and write cycles with a programmable access time. The LLC provides three different types of access cycle methods. The one closest to the ARM's read-write cycle pattern is the handshake mode. The glue logic uses combinational logic to transform the ARM pattern to be usable with the LLC in handshake mode.

The glue logic also implements a number of control registers that are used to control the functions of the FPGA. One register has a read-only value that is used to identify the FPGA and which functions it implements, receiver or transmitter logic. A second register is used mainly for debugging and for testing whether a read-write cycle functions correctly. The third register is a control register that enables the different units in the FPGA.

The last register is used to load parameters for the DM glue logic (i.e. channel number, sync value, etc.).

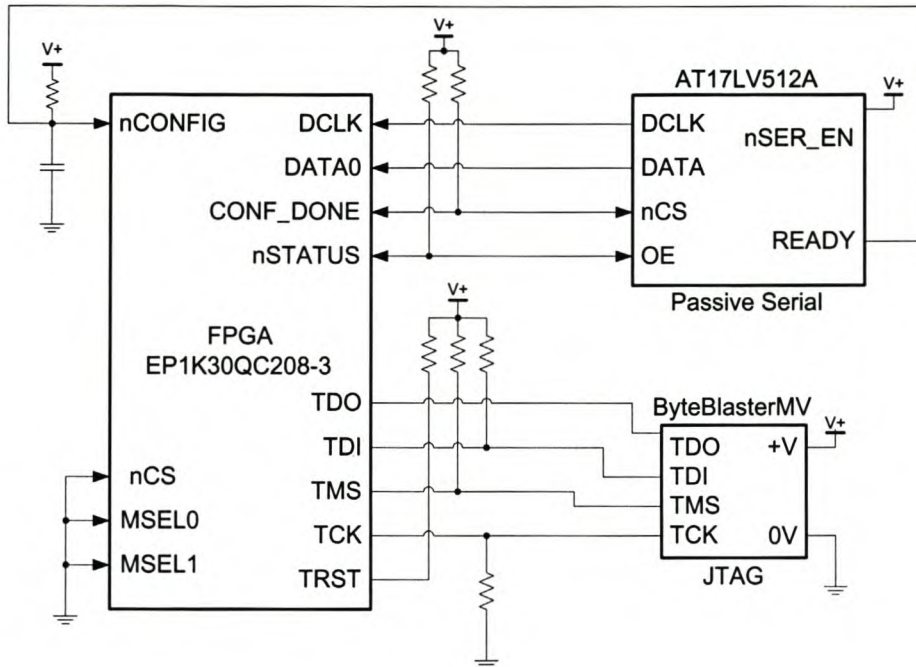
### 5.13 FPGA, configuration EPROM and JTAG interface

All the blocks described in the preceding sections are VHDL components that are implemented on an FPGA. An Altera ACEX 1K30 FPGA was selected in section 4.4 to implement these blocks. The FPGA does not require any external components except for decoupling capacitors for the power supply pins. The FPGA configuration data can be loaded into the device using a variety of modes, but only the passive serial mode and the JTAG programming mode are used in this application. Figure 5.7 shows a diagram of FPGA configuration devices.

The **passive serial** mode uses 5 signal lines. These lines are connected to a configuration EPROM device. The AT17C512A was previously selected for this function. When power is supplied to the system, the configuration EPROM pulls the  $\overline{\text{CONFIG}}$  line low to start the serial data transfer. When all the data bits are loaded into the FPGA, the  $\text{CONFIG\_DONE}$  line is driven high to signal that the configuration has successfully been loaded. The  $\text{CONFIG\_DONE}$  line is also used to reset the TSB12LV32 and TSB43LV03 devices as described earlier.

The second programming method is the **JTAG programming** mode. The FPGA contains a dedicated set of JTAG control lines. These lines are brought to a 10-pin IDC boxed header with pinouts compatible with the Altera ByteBlasterMV interface. The Altera ByteBlasterMV device can be used to download a new configuration into the FPGA from a PC.

Altera's MaxPlusII development environment is used to handle the download of configuration information during development. This means that during development it will be



**Figure 5.7:** Diagram of the FPGA configuration setup.

necessary to program the EPROM each time the VHDL components are changed. Unfortunately, the configuration data loaded via the JTAG interface is only available until the power is removed.

The I/O lines of the FPGA can be divided into 4 groups. The first group consists of the LLC control lines. These are the control lines for the LLC's host interface and data mover interface. The second group of lines go to a 26-pin IDC header that is dedicated for the video interface. These lines will handle the digital video stream.

The third group of I/O lines is dedicated to the microprocessor interface. A 16-bit data bus, 8-bit address bus and 10 control lines are allocated. Two lines are also attached to the I<sup>2</sup>C bus, although this is only for future experimentation with the I<sup>2</sup>C communications bus.

The last group of I/O lines does not have a dedicated function. They were taken to a 26-pin IDC header with a pin layout similar to the video port. This would allow two video interfaces to be attached, if required. This group will be used for debugging purposes during development.

## 5.14 Power supply

On a microsatellite, the power will normally be provided by one power supply system, typically a combination of solar panels and batteries. The IEEE 1394 bus specification requires that the cable carry a nominal 12 V power supply. The cable must be able to deliver 1.5 A of current. Small devices can then draw their current directly from the bus. The bus power is also used to power the PHY chip on systems that go into power save mode. This ensures that the PHY is still able to repeat bus traffic, while the rest of the system's power is switched off.

For this application, a laboratory bench power supply is used to supply the power. A diode is used to allow current to flow only in one direction. The diode will be connected directly to the bus power supply. A 0.75 A fuse is added to protect against short circuits.

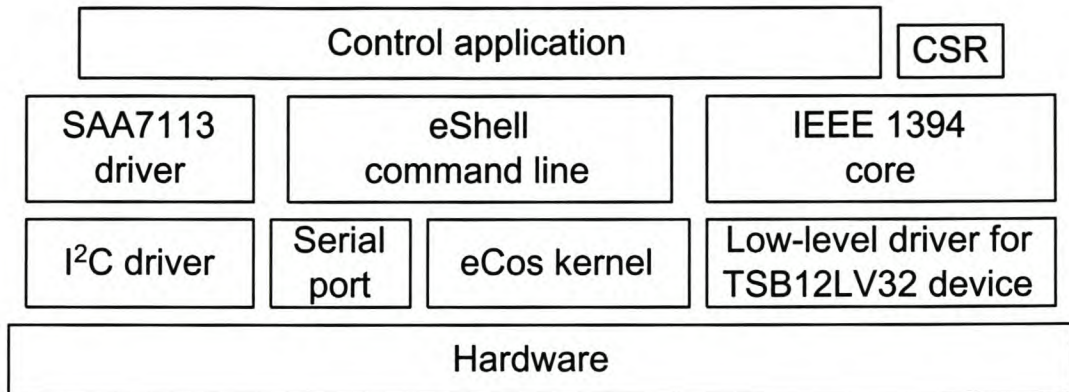
For this application, three regulated supplies are needed. All the ICs need a 3.3 V supply. The FPGA needs a 2.5 V supply as the core voltage. A 5 V supply can be used if TTL devices are to be added to the communications board. The FPGA's I/O ports are all 5 V tolerant.

The IEEE 1394 specification states that the bus power can vary between 8 V and 40 V. Most linear regulators only allow input voltages of up to 30 V. Heat dissipation becomes a problem with linear regulators when there is a large difference between the input and output voltages. For this reason a switched-mode power supply IC was selected in section 4.8 to convert the bus power to 5 V.

The LM2594HVN-5.0 is a monolithic IC that provides all the components for a step-down (buck) switching regulator, capable of driving a 0.5 A load. This chip uses a 150 kHz switching frequency and requires only 4 external components, an input and output capacitor, an inductor and a catch diode.

The application information in the data sheet [26] was followed closely. The data sheet provides excellent component selection tables and graphs. The inductor and output capacitor was chosen for an input voltage of 40 V and a current rating of 0.5 A.

The LT1117T-3.3 and LT1117T-ADJ linear regulators [27] were used to generate 3.3 V and 2.5 V respectively from the 5.0 V power supply. The LT1117 series regulator is a low-drop regulator that can work with a small difference between the input and output voltage. The LT1117T-ADJ device was used for the 2.5 V supply, because the LT1117 series does not have a 2.5 V fixed output voltage component. The adjustable version only requires two extra resistors and a stabilisation capacitor in the feedback path.



**Figure 5.8:** *Block diagram of the software layout.*

A green LED is placed on each power supply terminal to indicate that the voltage supply is functioning correctly. If one of the supplies had a short circuit, the LEDs would simplify the diagnosis of problems. Each LED has a series resistor to act as a current limiter.

## 5.15 Software design

The software on the ARM microprocessor must activate the video link between the two nodes. Most of the software consists of drivers to control the hardware. Figure 5.8 shows a block diagram of the software layout. The following subsections describe the blocks in detail.

### 5.15.1 eCos kernel

The kernel is the most important piece of the software puzzle. The kernel is mainly responsible for the interrupt and thread handling. The eCos kernel that was selected in section 4.10.4 provides a very good hardware abstraction layer for the software. Most of the architecture details are hidden behind the abstraction layer. The eCos environment also provides a debugging and testing infrastructure that is useful during development. The assert and trace packages of the eCos system was used for this purpose.

### 5.15.2 IEEE 1394 core and TSB12LV32 driver

The original idea for this project was to use the Linux kernel and to extend its IEEE 1394 driver to work with a TSB12LV32 device. Unfortunately it was necessary to select the

eCos kernel, which does not have any IEEE 1394 support. Thus, a new IEEE 1394 stack has to be written. The Linux drivers are used as a template for the new driver.

The low-level parts of the IEEE 1394 protocol stack consist of the hardware drivers. This part of the code must implement the interrupt handler and is responsible to load packets into the asynchronous transmit FIFO (ATF) buffer for transmission, and to retrieve the packets from the general receive FIFO (GRF) buffer during reception. Figure 5.9 shows a flow diagram of the driver code. Other hardware-related tasks include the initialisation and clean-up of the driver.

A device control interface is used to implement driver operations. Thus, higher levels of code need only to use a single function to communicate with the driver. This routine implements several functions, including bus reset requests, cycle counter read and writes, driver statistics and data mover control functions. Functions for accessing the PHY registers are also available in the driver.

The IEEE 1394 core is responsible for the management of the IEEE 1394 bus. The core is written to be more or less driver independent. Thus, the low-level API for all drivers are the same. The core only works with packets and knows nothing about the underlying hardware.

The core handles all packet allocation. To speed up embedded systems, fixed-size packets are allocated at initialisation and are stored in a queue. When a packet arrives at the device, it is taken from the queue and used. When the system is finished with the packet, it is replaced on the queue.

Consequently, no memory management has to be done on the embedded system. Statically allocated memory can also be used to allow the system to determine the memory usage of the program during compile time. This can be used to verify that the software will fit in the small memory footprint of the embedded device. The only drawback of this method is that it is easy to run out of allocated packets. The designer will have to tweak the number of packets to suit the application.

The IEEE 1394 core acts like a router for received packets. Applications register an address range during initialisation. The core then send packets that are addressed to that address range to the particular application's packet handling functions. Support functions for the three main transaction types (read, write and lock) are provided by the core.

#### 5.15.2.1 CSR module

The CSR module provides the system registers as specified in IEEE Std 1394-1995. The configuration ROM, as specified in IEEE Std 1212-2000, is handled by this module. The

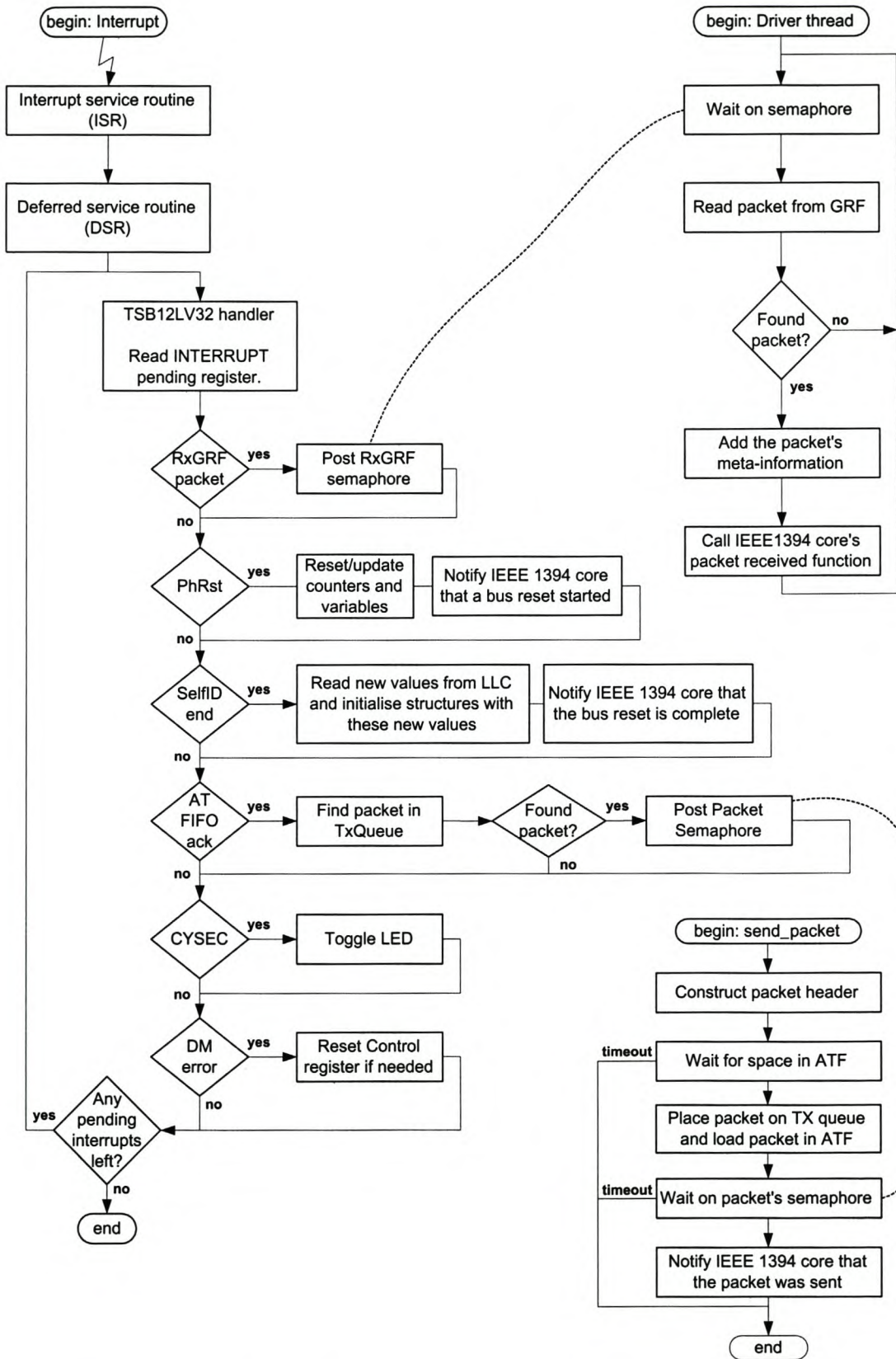


Figure 5.9: Flow diagram of the driver's packet reception and transmission code.



configuration ROM is needed to publish the capabilities of a node and also to identify the node with an extended unique identifier, 64 bits (EUI-64) number. This number must be unique to each node, and is used to identify the node between bus resets.

### 5.15.3 eShell command line

The eShell command line module is an interactive command line unit that communicates with the user over a serial line. A terminal program (on a PC) can connect to this code and allow a user to execute commands on the embedded processor. This module is needed during debugging to execute arbitrary commands on the embedded processor.

Applications register procedures with the shell. Each procedure has an associated command and help string. The help string is printed when the user asks for help on the command line. Each command can also have a number of parameters. The shell will then parse the parameters on the command line, and send them to the procedure when it is called.

Most of the shell code was taken from a similar component in the FireWire Reference Platform [28] that Apple Computer Inc. released. For this project, only a small subset of the features was used.

### 5.15.4 I<sup>2</sup>C and SAA7111A/SAA7127H driver

The Samsung KS32C50100 processor has an on-chip I<sup>2</sup>C unit. A driver for this code was written to send and receive I<sup>2</sup>C data. The Philips SAA7111A driver uses the I<sup>2</sup>C functions to initialise the registers of the video converter. By default, the video converter powers up into standby mode. The device is activated by the first valid I<sup>2</sup>C transaction to this chip.

The configuration of the SAA7127H device is very similar to the SAA7111A chip. Most of the source for the SAA111A can be reused to program the SAA7127H.

### 5.15.5 Control application

The control application is responsible to start and stop the video data stream. When a node is plugged into the IEEE 1394 bus, a bus reset is generated. The control application of the transmitter node should search through all the nodes on the bus and look for a

device that identifies itself as a receiver for the video stream. The transmitter should then start transmitting the video stream. After a bus reset the receiver node should start listening for a video stream.

# Chapter 6

## System implementation, integration and results

The previous chapter described the design of the different blocks of the entire system. This chapter will focus on the implementation and integration of the system. It will discuss all the steps involved in the manufacture and assembly of the system.

While describing the building steps, this chapter will also expand on any problems that were encountered and changes that were made to the original design. In the last part of the chapter, the results of the final system will be shown.

### 6.1 Schematics and PCB layouts

The schematics for the different boards are available in appendix A. The following sections describe the general layout of the components on the PCBs.

#### 6.1.1 PCB: communications board

The PCB layout was done for a 2-layer, through-hole PCB to keep costs down. Since most of the tracks go to the FPGA, track routing was simplified by reassigning the tracks on the FPGA. The LLC and PHY device were also easy to route, because they were designed to work together.

Figure 6.1 shows a photograph of the assembled communications PCB. It shows the placement of all the major components on the PCB.

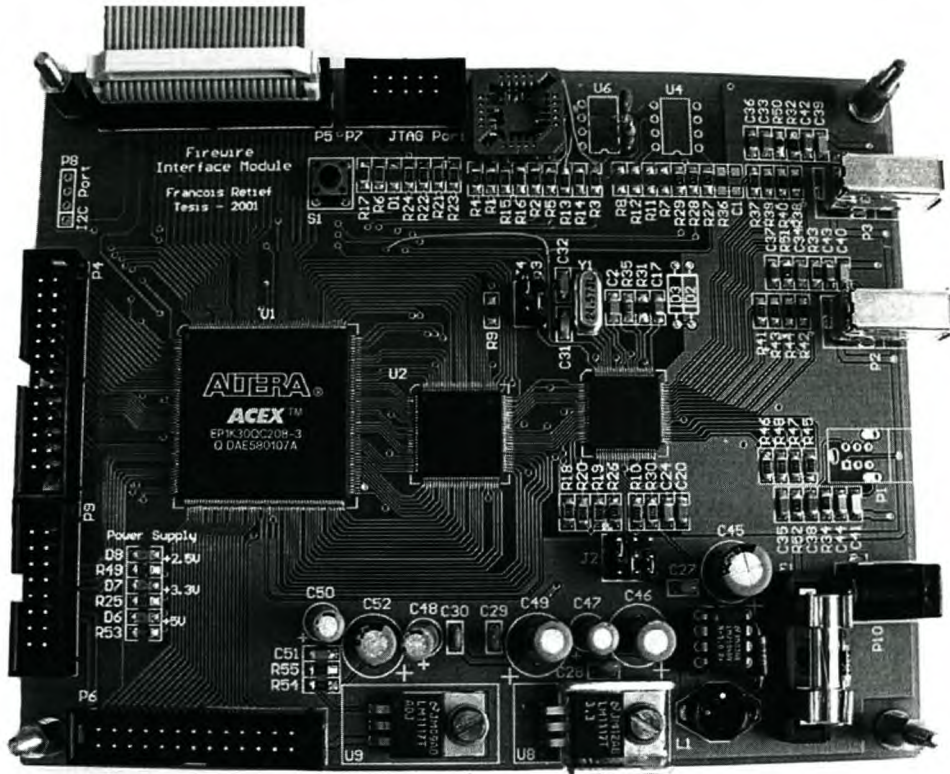


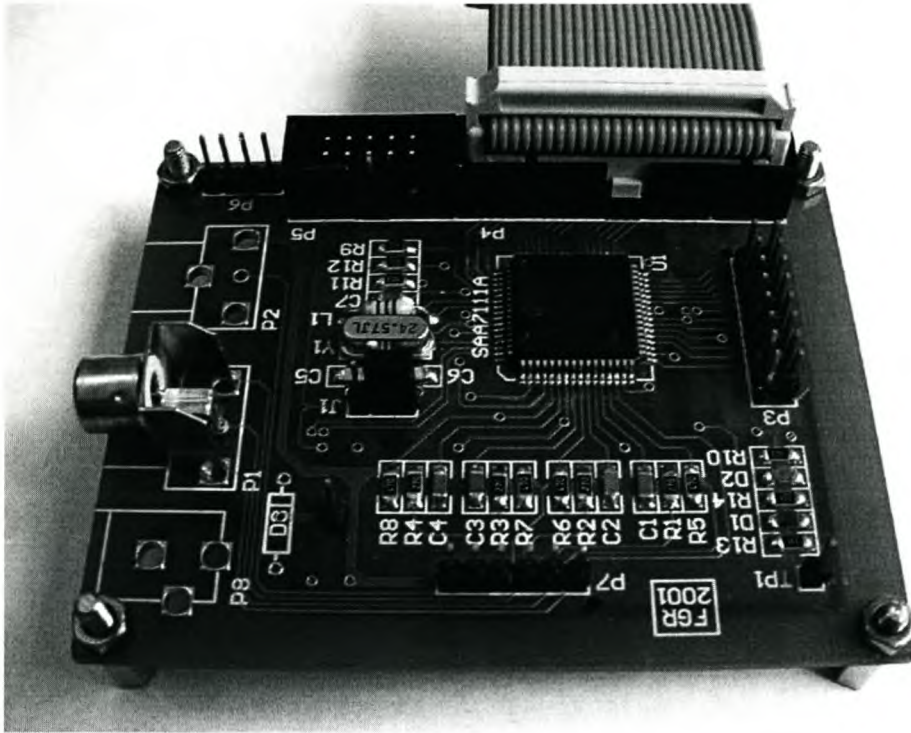
Figure 6.1: Photograph of the communications board.

**Headers and sockets:** The three sets of boxed headers were placed on three edges of the board. Thus, three other modules can be attached without requiring long cables. The IEEE 1394 sockets and the power socket were placed on the remaining edge.

**Power supply:** The power regulator components were kept as close to the power supply socket as possible. Also, they were put right to the edge of the board to minimise the interference from the switched-mode power regulator. A solid ground plane was placed on both the top and bottom layer surrounding the power supply.

The 3.3 V power supply lines for each of the three major components were diverted through a resistor footprint on the bottom layer. This was done so that the current for the three components could be measured. During normal operation, a wire short would be soldered onto the footprint. To test the current consumption, the short can be replaced with a current meter.

**Termination network:** The termination resistors for the IEEE 1394 signal lines were placed as close as possible to the socket. This was later found to be a mistake. The termination resistors should have been placed as close to the TSB41LV03 device as possible.



**Figure 6.2:** *Photograph of the video encoder board.*

Since the tracks between the connector and the socket are relatively short compared to the length of the cable, the effects should not be noticeable.

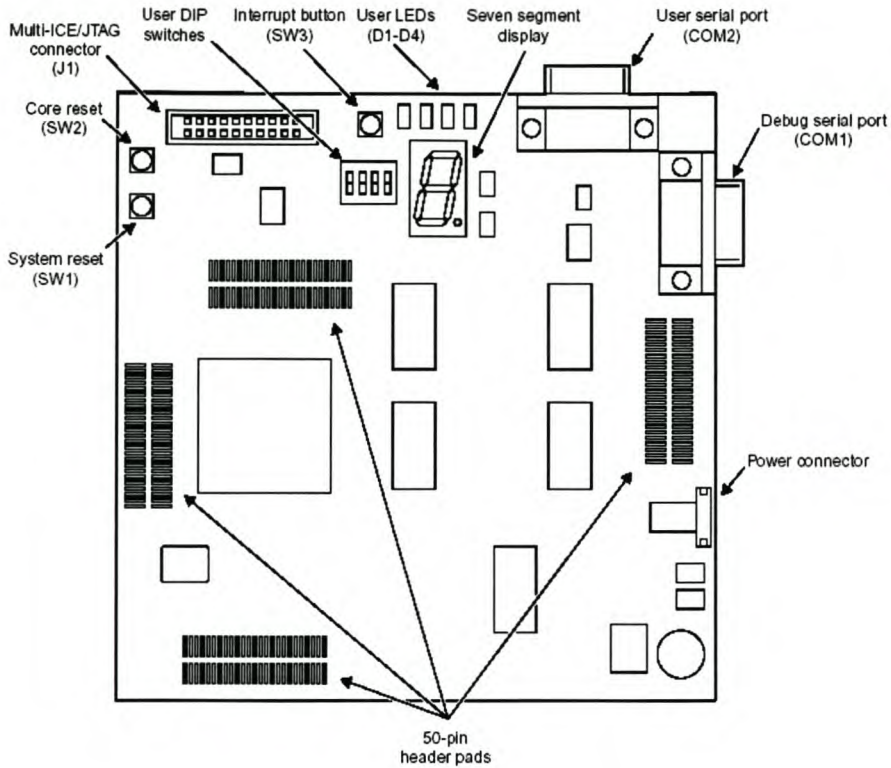
The JTAG socket, FPGA configuration EPROM, switch and LED were all placed on the other edge of the board where there was still space left after all the components were placed. All of them were connected to the FPGA.

### 6.1.2 PCB: video converter boards

A photograph of the assembled video decoder board is shown in figure 6.2. The PCB layout of the video encoder board is available in appendix A.

The two video converter PCBs were designed for easy connection to the communications board. On the video input module, the 26-pin boxed header was placed on the right side, with the video RCA (RCA) sockets on the top side. With this arrangement it is possible to place the module next to the communications board and to use a short ribbon cable to connect the boards. For future systems, a 90-degree angle plug and socket can be used to directly connect the boards together.

The video sockets face in the same direction as the IEEE 1394 and the power cables. This means that all cables will be lying in the same direction. The video input signal header



**Figure 6.3:** *PCB layout of the ARM Evaluator-7T evaluation board [29].*

was placed at the other opposite end of the IDC connector. This provides easy access to test the video signals. The control signal header was placed at the bottom edge, to be close to the communications and microprocessor module in case any of the signal lines are needed in the future.

The video output PCB was designed to be a mirror image of the video input PCB. This will allow both the video input and output modules to be connected to the same communications board, thus simplifying development and testing. In this way, a system can be constructed that both receives and transmits video data.

### 6.1.3 PCB: ARM Evaluator-7T evaluation board

Figure 6.3 shows the PCB layout of the ARM Evaluator-7T evaluation board. All available signals come out onto the four 50-pin header pads. To connect the microprocessor, a ribbon cable was made up to connect between the communications board and the Evaluator-7T. The ribbon cable was quite easy to make because all the pins in the headers are connected to the FPGA.

Consequently, there is no predetermined order in which the signals had to be wired. The VHDL code was written to accept the pinouts of the ribbon cable. In the future, a custom

PCB can be designed to fit the Evaluator-7T board and to add more memory and extra features (e.g. Ethernet) to the microprocessor. In such a system, the signals can be routed to the side of the board to allow a simpler ribbon cable arrangement.

## 6.2 PCB assembly and initial testing

The communications board was assembled first, since this forms the hub of the system. The first test after the PCBs arrived, was to check the ground and power supply nets for short circuits. This would indicate faulty PCB manufacturing. Since this system only uses a simple 2-layer board, there was not much reason for concern.

Next, the LM2594HV switched-mode regulator was soldered in. A 10 V supply voltage was applied to the input port and the 5 V power supply voltage was checked. This was the basis for the rest of the design. Next, the two linear regulators and companion components were added and their power supply voltages were checked. Finally, the power supply light emitting diodes (LEDs) were added. The resistor values were adjusted to ensure that all the LEDs were shining with equal brightness. This allows the tester to quickly verify that all the supply voltages are operating properly.

With the power supply section completed, the next task was to solder the FPGA onto the board. A small piece of VHDL code was written control to the general purpose switch and LED on the board. In the VHDL, code the button was used to invert the state of the LED, since there was no clock on the board at this stage. The FPGA configuration was uploaded with the JTAG interface, since the EPROM was only soldered on later. To test whether the FPGA configuration is loaded correctly, the pushbutton can be pressed. If the LED changes state, the configuration has indeed loaded correctly.

Next came the LLC and the PHY device. The RESET line of these two devices were originally connected to the INIT\_DONE line from the FPGA. The idea was that the LLC would be useless without a valid FPGA configuration, so it could just as well be kept in reset mode until a configuration was loaded. But, unfortunately, the INIT\_DONE line is not low after power-up. It only goes low during configuration and is then asserted high after the configuration has completed.

The RESET line was then moved to the CONF\_DONE line from the FPGA. This line is low after power-up and stays low until the FPGA has been configured. There is a short initialisation period between assertion of the CONF\_DONE signal and when the pins of the FPGA become available. This period is shorter than the time needed for the LLC to

reset itself after **RESET** has been driven high. The reset capacitor from the initial design was removed and only a pull-up resistor was kept, because **CONF\_DONE** is an open-collector output.

At this stage, the three main components of the board were soldered in. The IEEE 1394 sockets were left for a little later after the microprocessor was able to configure the LLC correctly.

### 6.3 Microprocessor interface and glue logic

After the FPGA was working, the microprocessor could be connected. The ribbon cable assembly was added between the ARM Evaluator-7T and the FPGA. The VHDL code was first updated to include control registers. The read-only version register was used to test the read cycle of the processor. Since the version register was programmed with a known value, it was possible to verify that all the data and address lines were working as they should.

The next test was a write-read test. The test register was implemented for this purpose. The microprocessor would write to the register and then read back the value that was written. This test verified the microprocessor's write cycle.

These two tests were included in the final program. During the initialisation phase of the program, these two tests are run to verify that the FPGA is configured with the correct VHDL code. Later, the version register also verifies that the VHDL code is compatible with the microprocessor's program.

After the FPGA was accessible, the glue logic was extended to include the LLC interface. The read-only and write-read tests were repeated with the LLC registers. The first LLC register is a version register with a fixed value. Since all the LLC registers are 32 bits wide, it had to be confirmed that the value is in the correct endian format. The LLC is designed with big-endian registers while the microprocessor is running in little-endian mode.

The LLC's **LENDIAN** pin was set to the correct value by using the jumper. The microprocessor's bus manager was set to a 16-bit interface, and the FPGA was programmed to keep the LLC's **M8BIT** pin low for 16-bit wide access cycles, while the **MDINV** bit was kept high. With all these settings, the microprocessor could now access the LLC registers without endian incompatibilities.



An interesting observation was made at this point. During the initial tests, the 50 MHz system clock from the PHY device was not present. Only when the microprocessor interface was added to the system did the clock appear. This was noted by the fact that a counter was implemented in VHDL using the PHY's clock to toggle the LED.

After a long search the cause was found to be the LPS signal between the LLC and PHY device. The LPS signal is used to tell the PHY device that the LLC device is powered up and active. The LPS toggles at a rate of 1/16 that of the BCLK signal, which in turn is driven by the microprocessor's system clock. Thus, when the microprocessor's clock is disabled, the PHY device will automatically go into low power usage, and disable the PHY clock.

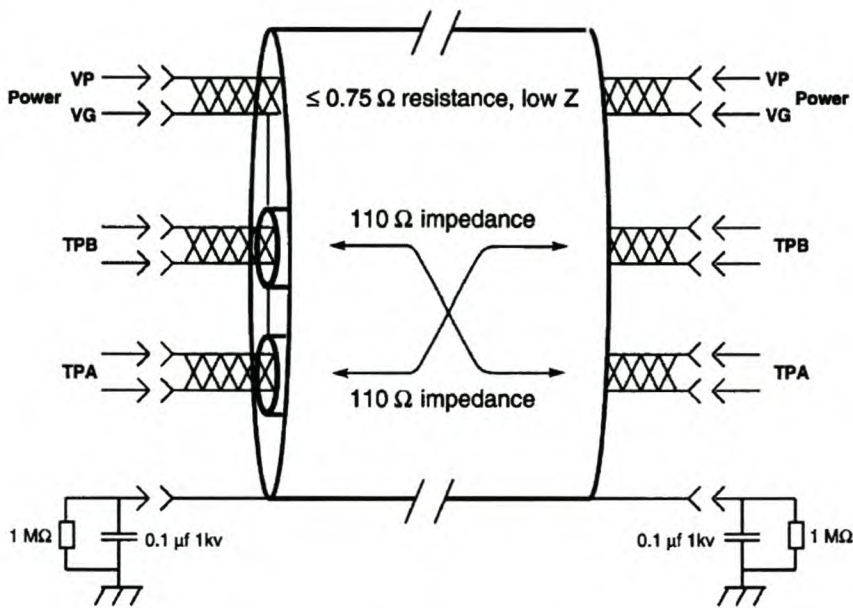
## 6.4 Initial interrupt and transmission tests

The FPGA and LLC device registers can now be accessed from the microprocessor. The next piece of the puzzle is interrupt handling for the LLC device. The VHDL glue logic was extended to connect the LLC\_INT line to the ARM's external interrupt request line. The first type of interrupt of interest is the BUSRESET interrupt, which occurs each time a cable is inserted or removed from an IEEE 1394 socket.

The ARM program was modified to initialise the LLC interrupt mask to allow for the BUSRESET interrupt. A crude interrupt handler had to be invented for the test program. Most of the interrupt handler was written in ARM assembler. For the first tests, the FPGA's button was connected (in VHDL) to the interrupt line. This allowed testing of the program by pressing the button to verify that the handler works.

After the interrupt handler was working correctly, the BUSRESET interrupt was tested. Here the first problem was encountered. The LLC registers were set up correctly, but no interrupt could be generated when unplugging the IEEE 1394 cable. A test was done where a bus reset was initiated via software. This test succeeded, verifying that the BUSRESET interrupt logic and assembler code is working.

The problem was eventually found to be a major PCB design error. The IEEE 1394 cable uses two twisted pairs, called TPA/TPA\* and TPB/TPB\* respectively. Due to a misunderstanding on the designer's part, the IEEE 1394 sockets were wired to a straight cable configuration. In fact the wires had to be crossed in the cable. Thus, TPA/TPA\* on node 1 is connected to TPB/TPB\* on node 2 and vice versa, as shown in figure 6.4. The TPA/TPA\* pair's common-mode voltage is kept at about 1.6 V while the TPB/TPB\*



**Figure 6.4:** *Diagram of the IEEE 1394 cable interface [4].*

pair is treated as an input. A bus reset is detected when the common-mode voltage on the TPB/TPB\* pair changes when a cable is inserted or removed.

Once the PCB tracks were modified, a bus reset interrupt could be generated when a cable was plugged in or removed from the board. The next step was to start transmitting asynchronous packets. The test program was again modified to load a predefined packet into the ATF buffer of the LLC. The Linux test machine was connected to the other side of the cable to catch the packets that will be transmitted. Unfortunately, the first attempts failed.

This was caused by two problems of misinterpretation of the data sheet. The first problem was again a PCB wiring problem with the IEEE 1394 socket. In the IEEE 1394 specification [4], the two signal pairs are identified as TPA/TPA\* and TPB/TPB\* respectively. However, in the TSB12LV32 data sheet [17] the pairs were identified as TPA+/TPA- and TPB+/TPB- respectively. Eventually it was determined that the TPA signal corresponds to TPA- while the TPA\* signal corresponds to TPA+. In the original PCB design the signals were swapped around.

The second problem was with the TXEN bit in the LLC control register. This bit must be set in order to send packets. However, this bit is cleared by hardware during a bus reset event. Thus, after a bus reset, the interrupt handler must again enable the TXEN bit. The original test program only set the bit during initialisation, causing all transmissions to stop after the first bus reset event.

When these two errors were corrected, the Linux PC could capture all the transmitted packets. Thus it was verified that basic communication over the IEEE 1394 bus is possible. Packet reception was not tested until later when more software infrastructure was available.

## 6.5 Transition to the eCos environment

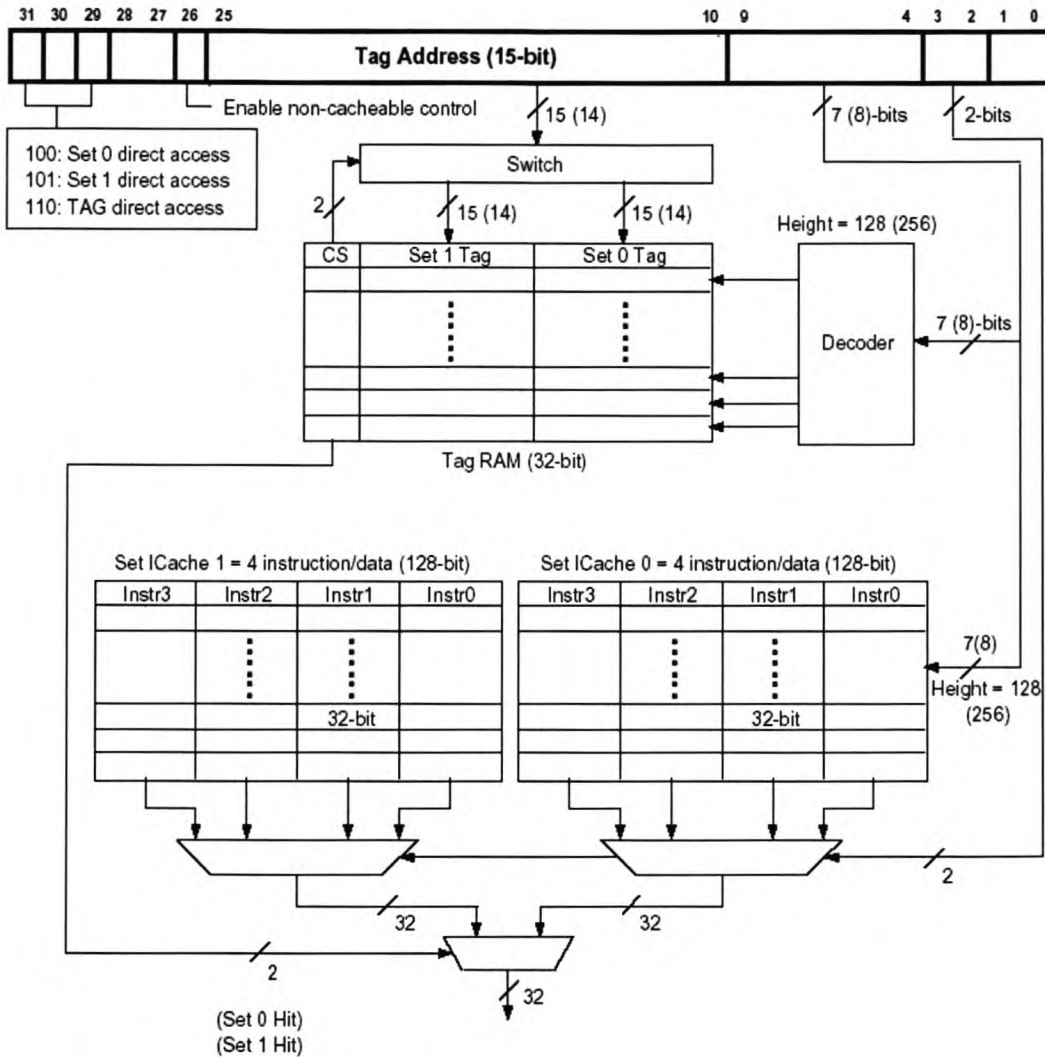
Up to now, the ARM programs were very simplistic and consisted mainly of a single program loop with a few pieces of assembly code. However, this is all implementation-specific and difficult to extend and modify. What was needed was more software infrastructure. The first piece of infrastructure needed is a way to handle interrupts effectively. The second piece of infrastructure needed is the ability to run many threads of code, with the corresponding synchronisation functions, like a semaphore or mutex.

The eCos real-time operating system (RTOS) was selected in section 4.10.4 for this task. eCos has well-documented support for the ARM Evaluator-7T board, allowing the programmer to write minimal pieces of code to implement previous test programs in eCos. eCos implements a hardware abstraction layer for all architecture-dependent features. This makes it possible to port existing code to new processors architectures in the future.

All test programs written thus far were ported to the eCos environment. The low-level hardware initialisation code was moved to the `cyg_Prestart()` function and the main loop was moved to the `main()` function. The interrupt handler was rewritten as a C function and attached to the interrupt using the eCos functions. All the old assembler code was thus obsoleted.

At this point, the eCos test code seemed to work most of the time. However every so often it would give unexpected bus resets, invalid packets or would hang inside the interrupt handler. When an interrupt occurred, the interrupt pending register is read to determine which interrupts were active. After the interrupt had been handled, the interrupt was cleared by writing to the interrupt pending register. However, when reading back from that register, the value was unchanged. From this it could be assumed that the write cycle was not working correctly. But writing to one of the test registers would return the correct results. Also, an analysis with a logic analyser confirmed that the write cycle was indeed correct.

Eventually, the cause was found to be misuse of the ARM7TDMI core's data cache. Normal memory access cycles benefited from the data cache, speeding up program execution.



**Figure 6.5:** Block diagram of the ARM7TDMI core's memory cache [30].

However, a register access should not use the cache. What happened was that the program wrote to a register and read the value of that register shortly afterward. However, the register is modified by hardware and will thus contain a new value. Yet the previously written value is still in the data cache. Thus, when the program read from the register, the value from the cache is used instead of reading the value directly from the register.

Internally, the ARM7TDMI core uses a 26-bit address system, shown in figure 6.5. The upper 6 bits of a standard 32-bit address are used to implement some extra memory control functions. One of these bits is the `NONCACHE` bit which is used to bypass the data cache altogether during a read-write cycle. Thus, to bypass the cache for a register access, `0x40000000` should be added to the register's address to set the `NONCACHE` bit. When this was implemented, most of the erratic behavior was solved, although some sporadic bus resets still occurred without provocation. All addresses in the code that refer to registers

were appended with the `NONCACHE` bit. Normal memory operations still uses the data cache to speed up program execution.

### 6.5.1 Program execution on the Evaluator-7T

One of the main problems with testing the code was uploading the program to the ARM processor and executing it. To do this, a few choices were available. The GNU debugger (gdb) could be used to upload the program and execute the code with the assistance of the Angel debug monitor (from ARM) or the RedBoot bootloader (from eCos). A favourable aspect of this was that the debugger could show where some errors occurred. Unfortunately, if the program crashes, the debugger will be unable to recover, since the serial interrupts might not be serviced.

**JTAG debugging:** What was really needed was JTAG debugging. The ARM7TDMI cores have an in-circuit emulator (called EmbeddedICE) that is able to do debugging without interfering with the software. The EmbeddedICE unit can stop a program by disabling the CPU clock. When the clock is stopped, normal debugging functions like single-stepping, reading of registers and memory, writing values to memory, and setting breakpoints can be done. The EmbeddedICE also has hardware breakpoint facilities to stop program execution.

Unfortunately, to do JTAG debugging an interface device called Multi-ICE is needed to provide the interface between the PC and the ARM core. Such a device was not available, so no JTAG debugging could be performed.

It was eventually decided not to use a debugger at all, but rather to load the program directly into flash memory. This proved to be a much better solution, because, when the power is switched on, the program is immediately available to run, instead of 1 minute 30 seconds later, as is the case when the program must first be uploaded.

**Boot Strap Loader:** To load the program into the flash, a special structure needed by the Evaluator-7T's BootStrap Loader (BSL) had to be added to our program. This structure tells the BSL the program's location in the flash, the starting address of the code, and the name of the program. The structure also has a checksum of the program so that the BSL can verify that the code is correct.

The BSL was designed to work with a simple modem terminal program, such as HyperTerminal (under Windows) or Minicom (under Linux). After power-up, the BSL performs

automatic baud rate detection by sending an *ESC* [5n string to the terminal. The BSL will send this string at different baud rates, until the terminal program (according to the VT100 terminal mode) replies with an *ESC* [0n string. When the baud rate has been established, the BSL waits for 2 seconds for the user to press the enter key before it continues to boot one of the modules in the flash. When the user presses enter, the BSL provide a command-line interface where the user can execute commands to manage the modules in the flash.

**Chat:** To automate this upload process, a UNIX program called *chat* was used. *Chat* is used to automate the login process for modem connections. The basic operation of *chat* is to wait for a given string and then send a reply. This is exactly what was required for this application. Only two modifications to *chat* that were needed, (1) to remove the time delay between characters and (2) the ability to send the contents of a file. This was done to speed up the data transfer.

A chat script was written that handled all the steps in the upload process. Corresponding entries in the makefile was made, so that it was possible to do the whole build process in one step. Thus, to upload a new version of the program, one has to type `make upload` and reboot the ARM board. The script will then automatically upload the file to the flash memory. On the next reboot, the new program will be executed. The whole upload process takes about 55 seconds (the exact time varies with the code size). The chat script used is available in appendix C.1.

## 6.6 IEEE 1394 driver

After all the necessary software infrastructure was in place, packet reception could be tested. The program was extended to handle the `RXGRFPKT` interrupt request. An easy way to generate packets for reception is to use the Linux PC. When the Linux PC is plugged into the system, a bus reset will occur. After the bus reset, the IEEE 1394 stack on the Linux PC will start to read the configuration ROM of the device, by sending asynchronous quadlet read packets to address `0xFFFFF0000400`. The test program in the microprocessor successfully received the packets, verifying that the hardware can receive packets.

After it was established that all the basic communications with the hardware was working, the IEEE 1394 protocol stack had to be developed. Writing a full IEEE 1394 protocol

stack is a big task, so it was decided to implement each feature of the stack as it becomes necessary. As a result the, IEEE 1394 stack would evolve to meet the application's needs.

The stack development can be divided into phases to build the following functions:

- allocating packets
- packet reception (asynchronous and isochronous)
- handling of asynchronous requests
- sending asynchronous responses
- basic CSR and configuration ROM functions
- initiation of asynchronous requests
- handling of asynchronous responses

### 6.6.1 Asynchronous transactions

When a packet arrives, the LLC places the packet into the GRF and triggers the `RXGRFPKT` interrupt. This is a 2 kB FIFO buffer that holds all packets destined for the microprocessor. On the microprocessor, the interrupt is serviced and the packets are retrieved from the GRF and packed into memory. The newly arrived packet is then placed in a receive queue and a semaphore is triggered to signal the main IEEE 1394 thread that a packet is ready for processing. All this processing is hardware specific and is written for the TSB12LV32.

The main IEEE 1394 core thread blocks on a semaphore. When a packet is placed in the receive queue, the semaphore is triggered and the IEEE 1394 core thread retrieves the packet from the queue and starts processing it. The transaction code field of the packet is checked to get the type of packet. The packet is then passed to the corresponding transaction handler. As a first step, only the request transaction handler was implemented. The response and isochronous handlers would be implemented later when needed.

From the request type, the packet can be further subdivided as read, write or lock requests. The packet is then passed to one of these handlers. During startup, an application will register itself to specific address ranges. If a packet with an address in that range arrives, that function will be called. Thus, the IEEE 1394 core looks through the list of functions for one that matches the address of the received packet and then calls it.

The handler that was called does some operations based on the information in the packet. When finished, the handler returns a response code and (if applicable) data to be returned. The IEEE 1394 core encapsulates it in an asynchronous response packet. The packet is given to the TSB12LV32 driver for transmission. The driver places the packet in the ATF buffer when space is available. The driver waits for the acknowledge for the packet before the packet is freed.

### 6.6.2 CSR and the configuration ROM

After asynchronous transactions could be done, the CSR functionality could be implemented. This is a set of registers that is mandated by the IEEE 1394 specification. Their function is mostly bus management and therefore they are required by all devices. Another component of the CSR is the configuration ROM. This is a 1 kB address space that informs other devices on the bus of the capabilities and functionality of this device. It also contains numbers that uniquely identify the device to other devices.

The CSR handling code registers itself with the IEEE 1394 core during initialisation. When an asynchronous request arrives with an address in the CSR region, the CSR handler function is executed. A response is generated and loaded into the buffer space provided by the IEEE 1394 core.

The configuration ROM is generated at compile time and included into the source code as a constant array. This was done to simplify the design. The CSR handler can just copy the data from the configuration ROM array to the response buffer. Since the ROM is static, no management functions have to be implemented.

At this point, the Linux PC can identify the system based on the entries in the configuration ROM. In the near future, a control application can be written to control the system. The control application will use the configuration ROM information to determine whether the system can transmit or receive data.

## 6.7 I<sup>2</sup>C and SAA7111A driver

With the basic IEEE 1394 functionality working, the next step is to get the video stream transmission working. After power-on, the SAA7111A device stays in a low-power mode with the outputs disabled. After the first valid I<sup>2</sup>C transaction, the device is switched on and the video data stream will be available for transmission.



The Samsung KS32C50100 microprocessor has an on-chip I<sup>2</sup>C controller. A software driver for the controller was written to control the I<sup>2</sup>C bus. The driver provides basic read and write operations for a component driver.

The SAA7111A software driver was written to select the correct configuration options. This was done by changing the device registers via the I<sup>2</sup>C port. For simplicity, the driver keeps the settings as close to the defaults values as possible.

## 6.8 Isochronous data transfer

The next major component of the system was to implement the data-mover (DM) port. This port is vital for the video data stream transmission. The transmission and reception of data was tested separately at first. The first pieces of VHDL test code were used to verify the timing diagrams. After that, the integration of the DM glue logic with the FIFO buffer was attempted.

**DM transmission testing:** The VHDL test code consisted of a 16-bit counter. The counter was incremented each time a value is read from the DM port. To verify the data stream, the data can be checked for any break in the sequence of the data. This also helps to determine whether a packet was dropped, because a large chunk of values will be missing.

During testing, the signal timings of the DM-port was also evaluated. This was then incorporated in the VHDL simulations. A typical transmission cycle can be described as follows:

The user asserts the **DMREADY** line low to indicate that there is data ready for transmission. The **DMDONE** flag is pulled low to acknowledge the **DMREADY** line. The **DMPRE** line is asserted for one clock cycle prior to the reading of the packet header. The glue logic will use the **DMPRE** pulse to save the number of bytes waiting in the FIFO buffer. The **DMRW** is asserted for two cycles to read the 4-byte isochronous packet header.

Next, the LLC waits for the next isochronous cycle to start and arbitrates for bus access. The **DMPRE** line is asserted for one clock cycle to indicate the start of the data transmission. Finally, the **DMRW** is asserted high to clock the data onto the IEEE 1394 bus. The number of bytes read depends on the length field in the packet header read previously.

For testing, the transmitted packets were then captured by the Linux PC. The data stream was verified by checking the incrementing values from the counter.

**DM reception testing:** With the transmitter sending a verified data stream, attention was turned to the receiver side. This LLC has a packet routing mechanism whereby certain types of incoming packets can be routed either into the GRF buffer or to the DM port. The first test was to receive the isochronous packets via the GRF buffer. The program can then display the incoming packets (via serial cable) to verify that the payload data is correct. For this test, the transmission rate of the packets was reduced to 3 packets per second to allow the serial cable to keep up.

After the software test, the VHDL code was modified to connect the DM data port to the logic analyser. Both the transmitter and the receiver's data and control lines were connected to the same logic analyser for comparison. The packet routing setup was changed to cause all isochronous packets to be sent to the DM port.

At first the routing setup did not give the correct results. Instead, the `DMERROR` line was pulsed for one clock cycle each time a packet arrives. The cause of this error was eventually found to be in the DM control register. The initial understanding of the meaning of the `DMEN` bit was that it enabled the transmission. The `DMRX` bit would enable the reception. Thus, the understanding was that only one of these bits can be set at any time.

This proved to be incorrect. The `DMEN` bit is used to enable all DM operations, while the `DMRX` bit is used to select between transmission and reception. For transmission only the `DMEN` bit should be set high, while for reception both bits must to be set high.

After the bits were corrected the data arrived at the DM port. The logic analyser plot in figure 6.6 shows the transmission and reception of one packet. The top half of the trace is from the transmitter, while the bottom half is from the receiver. Events A and B are magnified in figures 6.7 and 6.8 respectively.

Event A shows the packet header retrieval sequence described earlier in the section. It should be noted that the `DMRDY` line is high for only one pulse, because inside the FPGA the data-is-available signal is ANDed with the `DMDONE` line. Thus, the `DMRDY` is only high when there is data in the FIFO buffer and the DM is ready to receive data. This was done because older versions of the TSB12LV32 required the `DMRDY` to go low as an acknowledgment signal. By ANDing the signal, it ensures that the `DMDONE` will always be acknowledged.

Event B shows the start of data transmission. In the top traces, the transmitter module places the data on the IEEE 1394 bus. After less than a millisecond, the data is available on the receiver side. The delay should increase as the number of hops between the transmitter and receiver increases. Each PHY device is a repeater and adds some transmission delay at each hop.

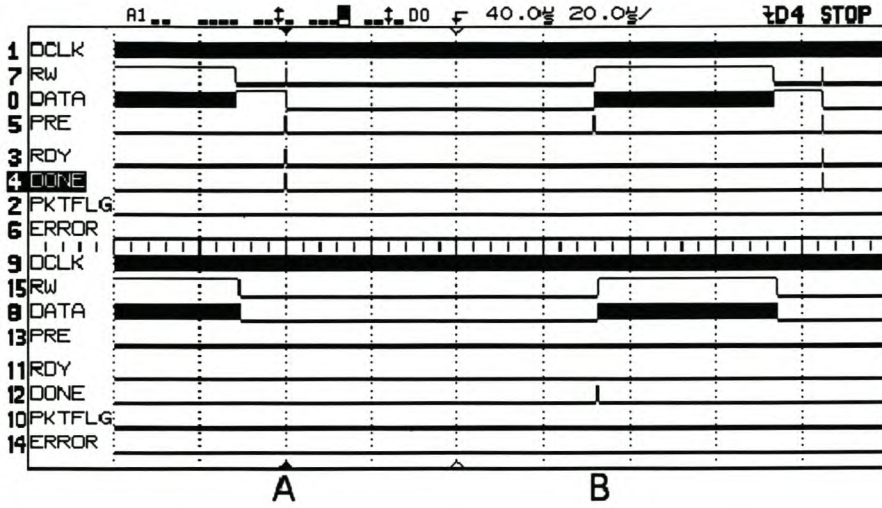


Figure 6.6: Logic analyser plot of an end-to-end transmission.



Figure 6.7: Magnified plot of the packet header transfer sequence.

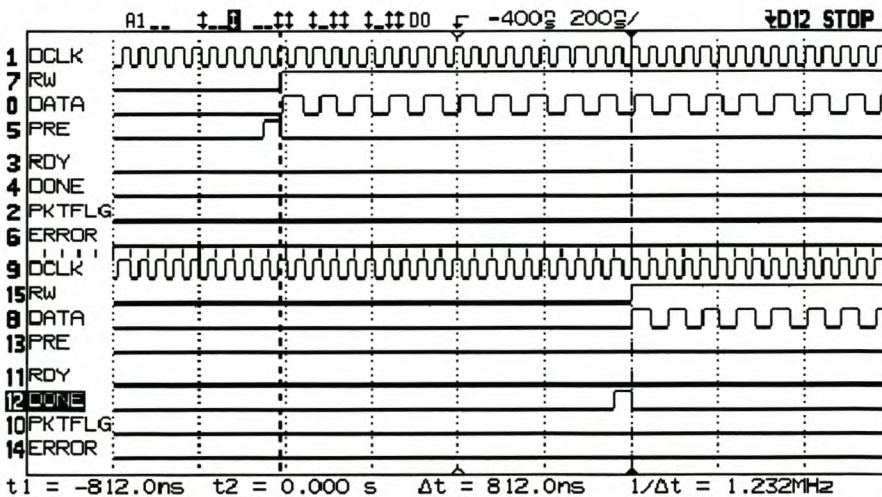


Figure 6.8: Magnified plot of the start of the data transmission.

## 6.9 Unresolved issues and problems

Finally, end-to-end communications over the IEEE 1394 bus was achieved. This was the primary goal of the project. The final step will be to attach the two video converter boards to the communications board and include their respective VHDL glue logic components into the FPGA. The following items still needs to be completed:

- The **SAA7111A board** was only partially tested and was not transmitting the correct video in the optimal format at the time of writing. There were still a some problems with the I<sup>2</sup>C bus software driver that cause either invalid values to be loaded into the registers or the software to crash on the ARM microprocessor. The practical interaction between the I<sup>2</sup>C controller on the microprocessor and the software has to be studied in more detail to understand the problem better. The microprocessor's datasheet [30] is a bit limiting in this regard.
- The **SAA7127H board** has not been assembled or tested. It awaits the completion of the video decoder before it can be tested. The final video reception glue logic has also not been written, and currently all data is sent directly from the receiver's FIFO buffer to the video header socket so that it can be attached to a logic analyser.
- The **control application** is in a limited functional state. The receiver is placed in a default reception mode on power-up, but the transmitter must be activated by a user to start transmission. The control application still has to be extended to allow control via the IEEE 1394 bus. This will allow remote nodes (e.g. an on-board computer on a microsatellite) to initiate or stop the data transmission process.

The following two problems are still hampering the reliability of the system.

**False bus resets:** After a random period of data transmission, the two devices stop transmission without any software command or hardware removal of a cable. The period can be anything from a few seconds to a few minutes.

Figure 6.9 shows a plot of a normal bus reset sequence initiated by software. The top traces are from the transmitter module, while the bottom traces are from the receiver module. Notice that both devices detected the bus reset.

Figure 6.10 shows a plot of an abnormal bus reset. No software commands were issued and the IEEE 1394 cable was not touched. Notice that only one of the two devices has

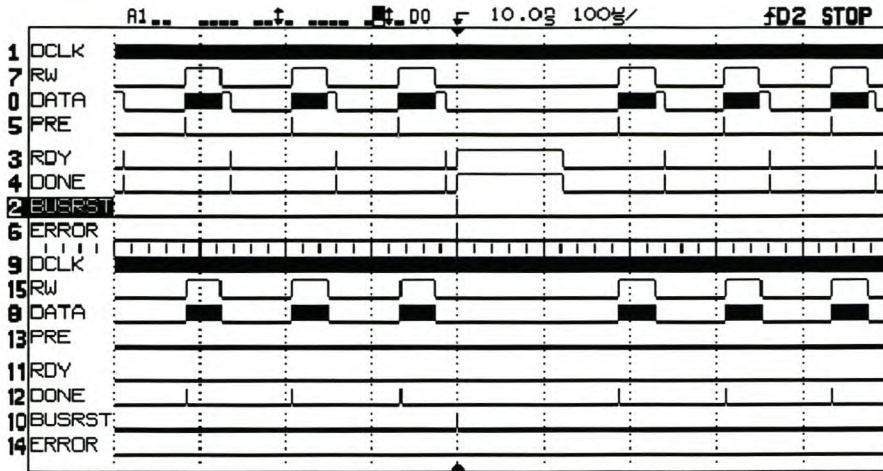


Figure 6.9: Logic analyser plot of a normal bus reset sequence.

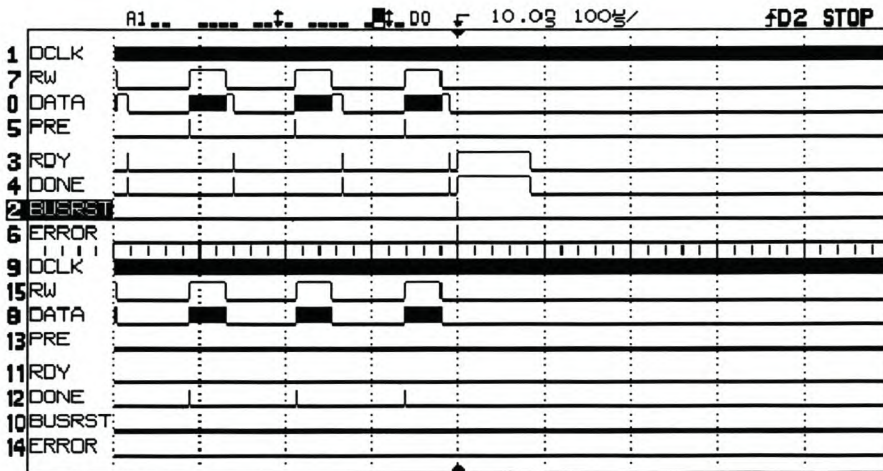


Figure 6.10: Logic analyser plot of a false bus reset (abnormality in transmitter).

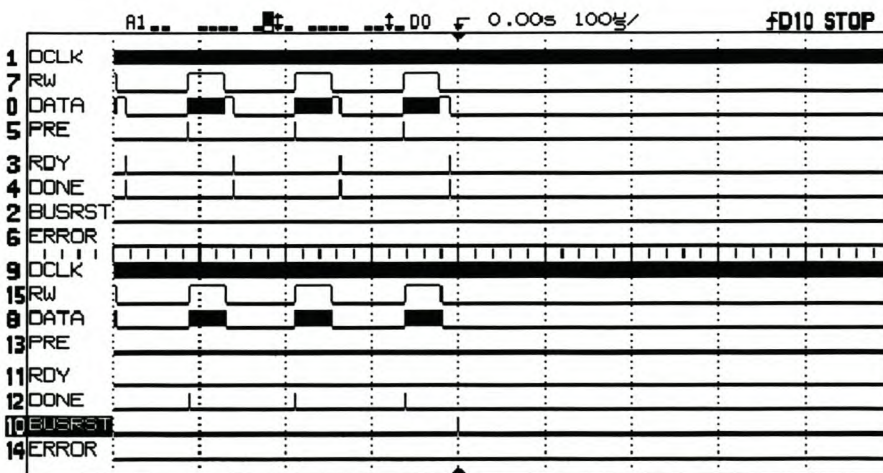


Figure 6.11: Logic analyser plot of false bus reset (abnormality in receiver).

detected a bus reset. Thus the reset signal cannot come from the cable. Figure 6.11 shows a similar plot, but this time the receiver has detected a false bus reset.

From the two plots it can be seen how the data stream stops as soon as a false bus reset occurs. Also note that in the second case, although the transmitter has not detected a bus reset, the transmission of data also stopped. It is suspected that the transmitter would continue to send data, since it has not detected a bus reset. Isochronous data transmission is a broadcast transaction. Thus the transmitter can be sending data to any number of receivers.

Another observation was that after such an event, the PHY device stops responding to software commands. A software bus reset will not work on a device that detected a false bus reset. Physical removal (or insertion) of the IEEE 1394 cable seems to be the only way to restore the device to operation.

One speculation on why the transmission of data stops, is that the false bus reset inhibits the transmission or reception of cycle-start packets. These packets are essential for isochronous data transfer.

Since isochronous transmission can be initiated for short periods of time, the problem was not pursued in great detail. A more detailed analysis should be done to study the reason for the false bus reset.

**Error messages:** Another problem that might be related to the false bus reset problem, is the fact that numerous error messages are received from the LLC for no apparent reason. `HDRERR` indications are received, that are supposed to indicate that a packet was corrupted. However, it was have verified that all packets arrived without any error in the data. Some of these error messages are also received without any traffic on the IEEE 1394 bus.

The Linux PC was attached to the node to see if the errors are related to something on the IEEE 1394 bus. The PC showed no error messages at all and all transmitted packets were received without error or loss. This means that whatever the problem is, it is related to either the LLC or PHY devices used in the design.

Another alternative can be that something in the PCB layout of the communications board is inducing noise into the system, but not onto the IEEE 1394 bus. A very detailed study of the PCB layout will have to be undertaken to understand what this problem is. Again, because it is possible to transmit and receive data traffic (for short periods at a time, 10 seconds to a minute) it was decided to leave the investigation until the full system was working. Enough time was already lost trying to find this problem.

## 6.10 Final system

Figure 6.12 shows the assembled boards of the transmission module. The ARM Evaluator-7T board is on the left, the communications module is bottom right and the video decoder board is top right.

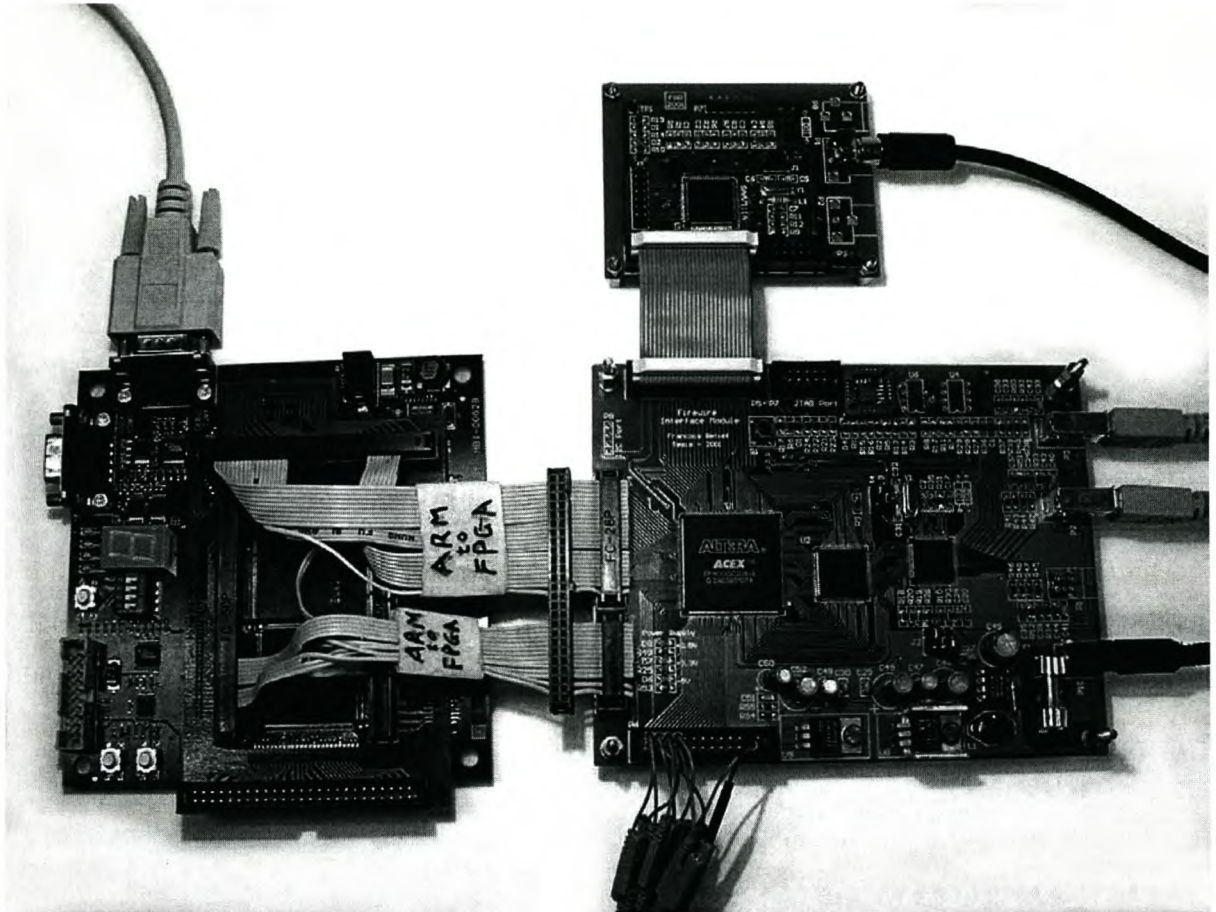


Figure 6.12: *Photograph of a complete module.*

# Chapter 7

## Conclusions

The goal of this thesis is to investigate the implementation of a high-speed serial bus based on the IEEE Std 1394-1995 specification for use in a microsatellite.

A serial communications link between two points was established and a high-speed digital data stream was transferred. Figure 6.6 shows the logic analyser trace of a packet stream running at the 400 Mbps speed between two nodes.

### Overview of the chapters

- Chapter 2 gave a broad overview of the total system. Here all the different parts of the system were described as single blocks without much detail.
- In chapter 3, a brief overview of the concepts related to serial busses were presented. Different kinds of serial busses were then investigated. From the resulting list, only two were found suitable for a high-speed serial bus on a microsatellite, namely SpaceWire and the IEEE 1394 serial bus.
- In chapter 4, the architecture of the system were presented. Then all the hardware and software components were selected for that architecture. These selections were important, because it influenced the design of the system later on. Some of the selections were based upon availability of components while others were performance related.

Limitations such as memory and type of video converter forced the use of alternative methods of achieving the goal. An example is the inability to run the Linux kernel on the Evaluator-7T board, thus forcing the use of the eCos operating system. The use of eCos then forced us to write a IEEE 1394 protocol stack for scratch, instead of using the IEEE 1394 drivers in the Linux kernel.



- The system were designed in chapter 5. Here the details were filled into the blocks that were previously left open. Also added to the design were blocks for the glue logic inside the FPGA. The design process followed the flow of the video data stream through the system.

The choice of video converter devices was largely influenced by availability of components. The design would have been slightly simpler had another device been chosen. That said, the use of the FPGA proved that inefficient interfaces can be attached with the use of glue logic. The flexibility of the glue logic will allow a wide range of subsystems to be connected to the communications module.

- In the final chapter, the system was build. Chapter 6 describes the different steps that were taken during the assembly and testing of the system. Problems that were encountered are discussed and solutions presented.

The **make program** allowed the formation of complex compile and build procedures. Small tools and other *make* scripts can be combined to construct the overall build process. The configuration ROM generator was a good example where the integration of another program's output into the final program proved every useful. Another example was the *chat* program. With a little modification to the *chat* program, it was integrated into the *make* script to automate the uploading of the application into the Evaluator-7T board's flash memory.

The **eCos environment** proved to be a useful tool during the development. Due to the highly modular design of eCos, the application program and the OS were fitted onto the Evaluator-7T board with room to spare. The eCos trace and assertion facilities provided a rich environment to display information about the running application. In particular the trace buffer allowed messages to be collected during interrupt handling. This allowed determining what the system was doing without the use of a bus analyser unit.

The **IEEE 1394 protocol stack** requires a fair bit of microprocessor resources for a full implementation. This project has shown that, barring the bus management functions, a small stack can be written for deeply embedded systems. The bus management functions can be handled by the OBCs, which normally has the largest microprocessors in a typical microsatellite. Subsystems can each have a smaller processor and will function as a slave device to the OBC.

This project has shown that an IEEE 1394 serial bus can be constructed with **limited resources**. In particular the unavailability of a bus analyser has hampered the debugging and testing, but using a Linux PC's system logs have proved sufficient to implement the

basic functionality. Building the system in small steps that can easily be tested and understood, helps to keep the problem manageable.

## Recommendations

After the experience of designing and building this system, the following set of suggestions was formulated to improve the system.

**Improved reliability:** The source of the false bus reset problem, described in section 6.9, must be found and the design fixed accordingly. The final design must be able to start a transmission and maintain the data transfer for long periods of times (hours, even days).

Fault tolerance can be increased by using two separate LLC devices. The two networks would run in parallel. If one network should fail, the software can fall back to the other network and continue transmission.

**Larger FIFO buffer:** The limited size of the on-chip memory of the FPGA placed a severe restraint on the system's ability to send real-time data. Isochronous data is transferred at a rate of one packet every  $125\ \mu\text{s}$ . Thus the FIFO buffer must be big enough to hold all the data for at least one complete packet.

During a bus reset event, packet transmission is suspended for a short period of time. The FIFO buffer must ideally have enough space to prevent loss of any data during a bus reset event.

Upgrading the EP1K30QC208-3 FPGA with a larger version, such as the 1K50 or 1K100, would solve the problem. Another alternative would be to add a memory interface to the FPGA and to add external memory.

**IEEE 1394 protocol stack:** During the later stages of this project, Apple released their FireWire Reference Platform source code. This is a full protocol stack implementation of FireWire with driver support for the TSB12LV32 chip. As for future work, the Apple drivers can be tested on the developed platform, since they have much more functionality than the limited stack that was written during this thesis.

**SpaceWire:** A full investigation of the SpaceWire specification would also be recommended. Future thesis projects could include the design of a SpaceWire node or router in a FPGA. Modern FPGA devices have built-in LVDS support. Thus a SpaceWire node can be designed without any extra external hardware. Having a SpaceWire VHDL core can lead to a very efficient subsystem interface design.

### Advantages of IEEE Std 1394-1995

- The speed of the IEEE 1394 bus is much higher than the implementations used in SUNSAT-1. This leave ample room for future extensions. The new IEEE 1394b specification push that speed limit even higher.
- The IEEE 1394 bus is by design able to handle concurrent data streams. This allow the subsystems to operate concurrently. An good example is an imager that generates a video stream. This video stream can then be compressed by another subsystem. The compressed data stream is then sent to a RF transmission unit. In this setup there will be two concurrent streams on the IEEE 1394 serial bus.
- The peer-to-peer nature of the IEEE 1394 implementation fits well into a microsatellite use of redundant components and subsystems. Thus if one node fail, the bus management operations can be transfered to another subsystem with minimum effort.
- The asynchronous transaction layer exposes a remote node as a set of memory registers. The transaction layer provide read, write and lock operations to the application layer. From an application perspective, a remote node appears as a set of memory registers like in most hardware devices. This will make it easier for a programmer to control the remote node.

Regarding the fact that the IEEE 1394 serial bus has more advantages over previous implementations, and seeing that a working prototype has been constructed, this serial bus is a viable candidate for inclusion in the next generations of SUNSAT microsatellites.

# **Appendix A**

## **Schematics and PCB layouts**

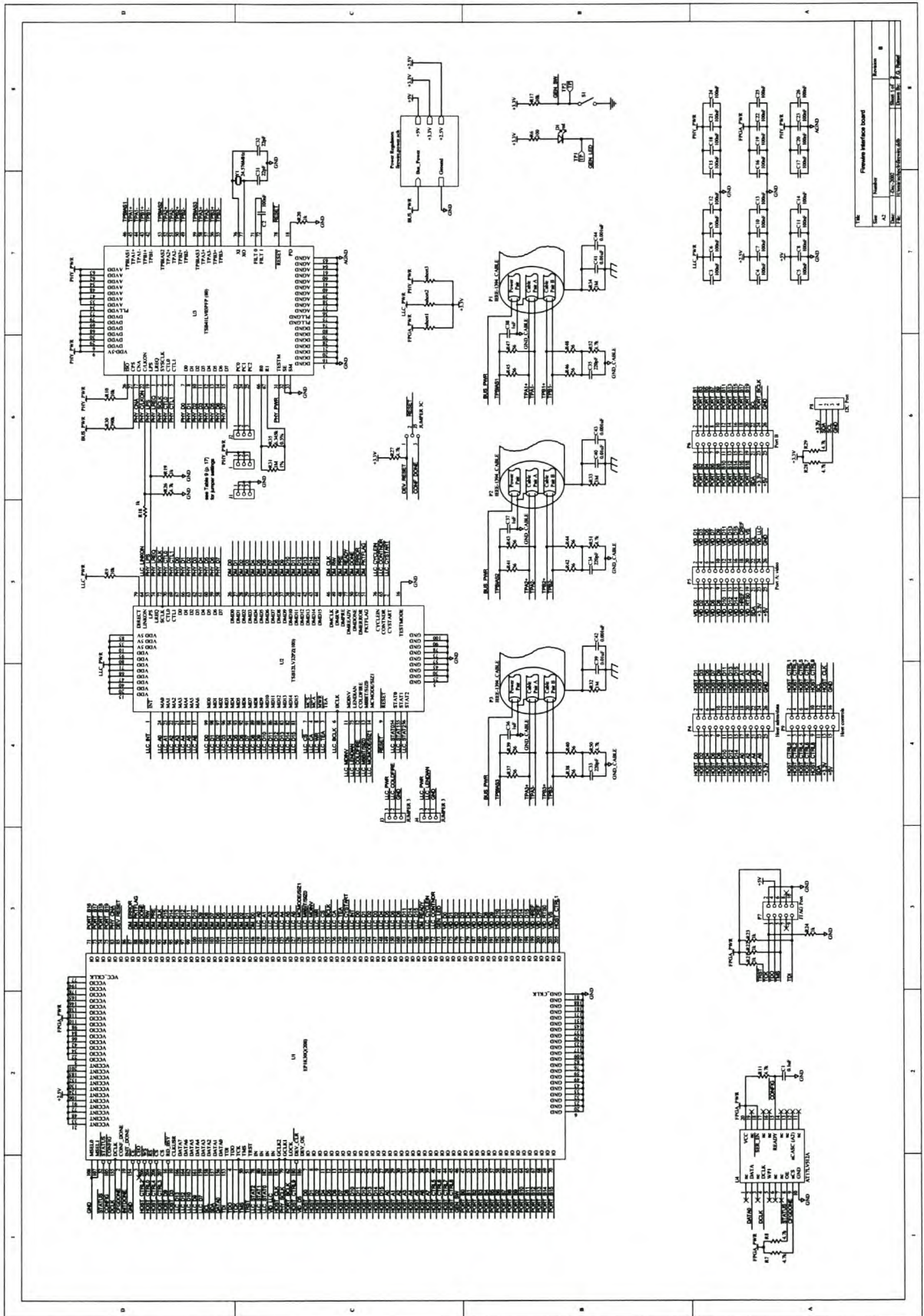


Figure A.1: Schematic diagram of the communications board (rev. B, page 1).

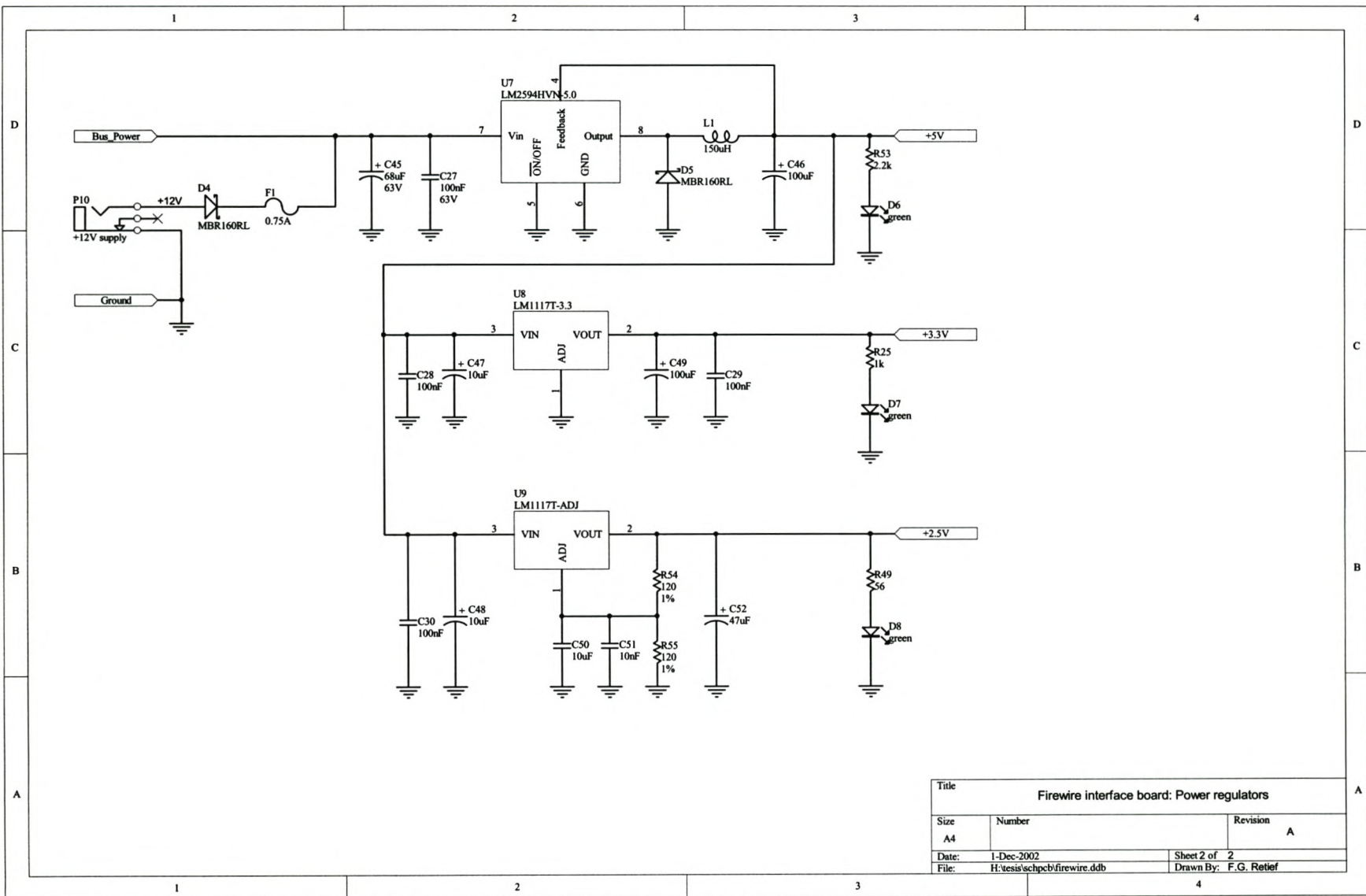


Figure A.2: Schematic diagram of the communications board (rev. B, page 2).

Figure A.3: Schematic diagram of the video decoder board.

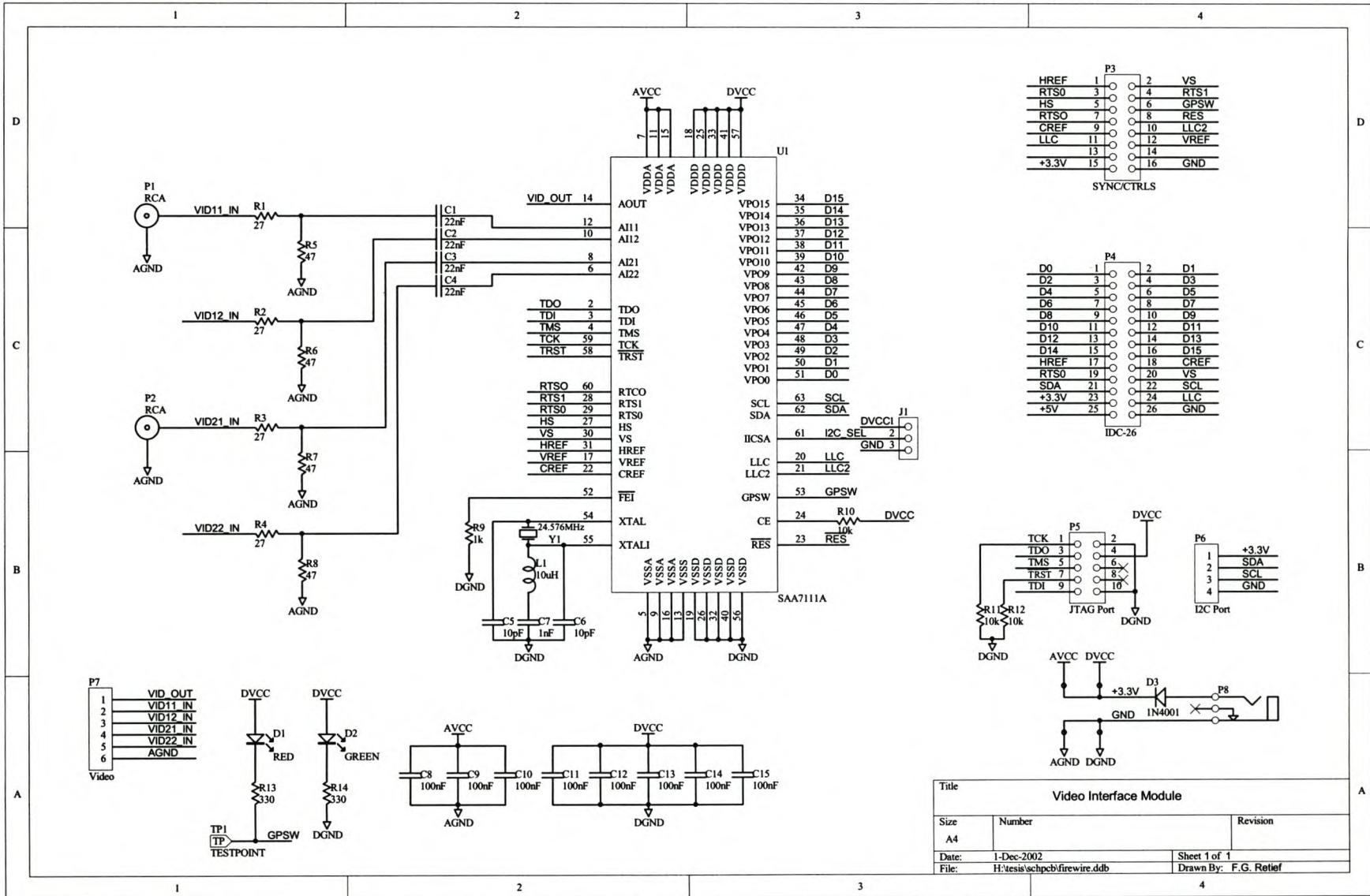
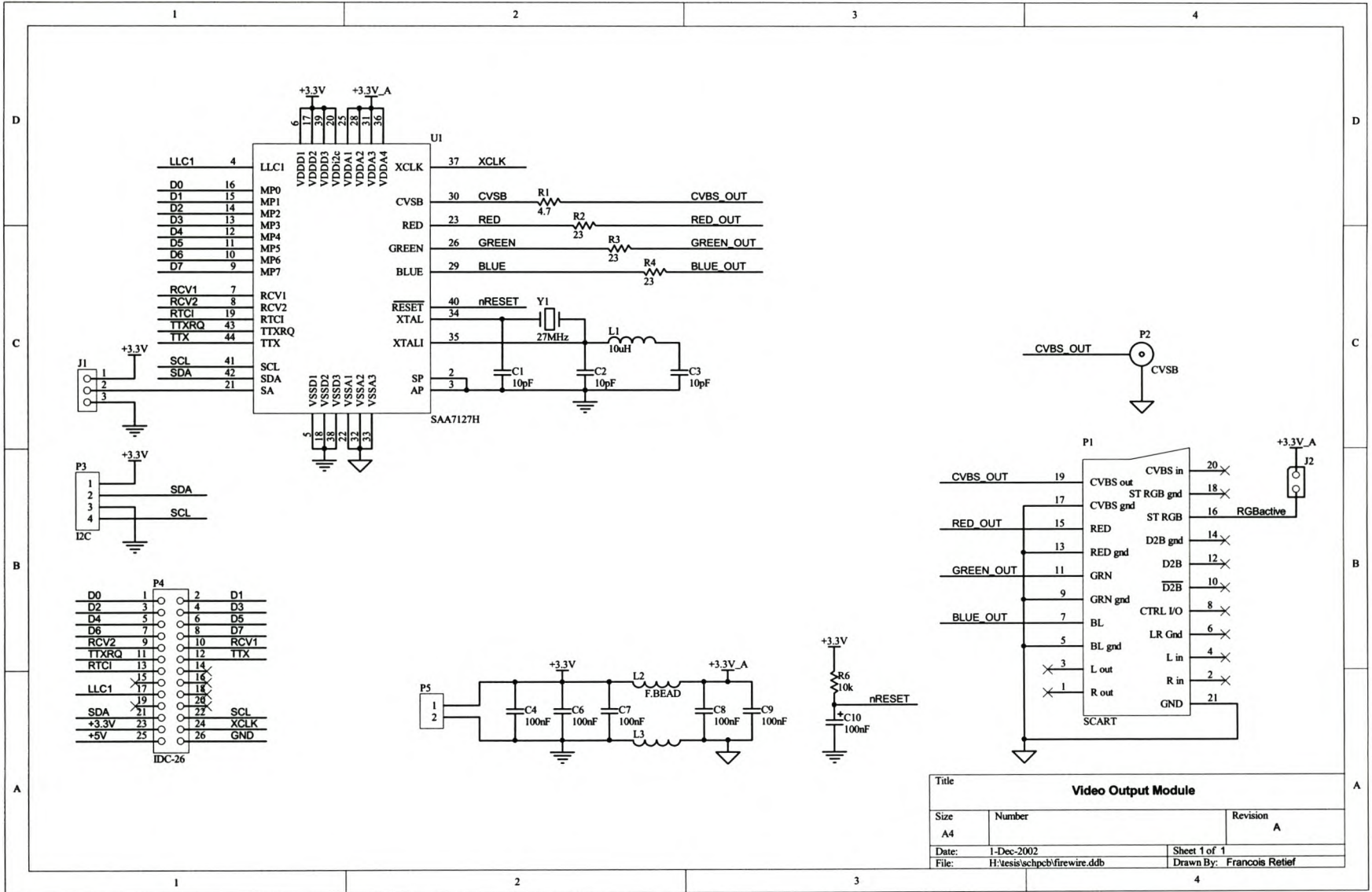
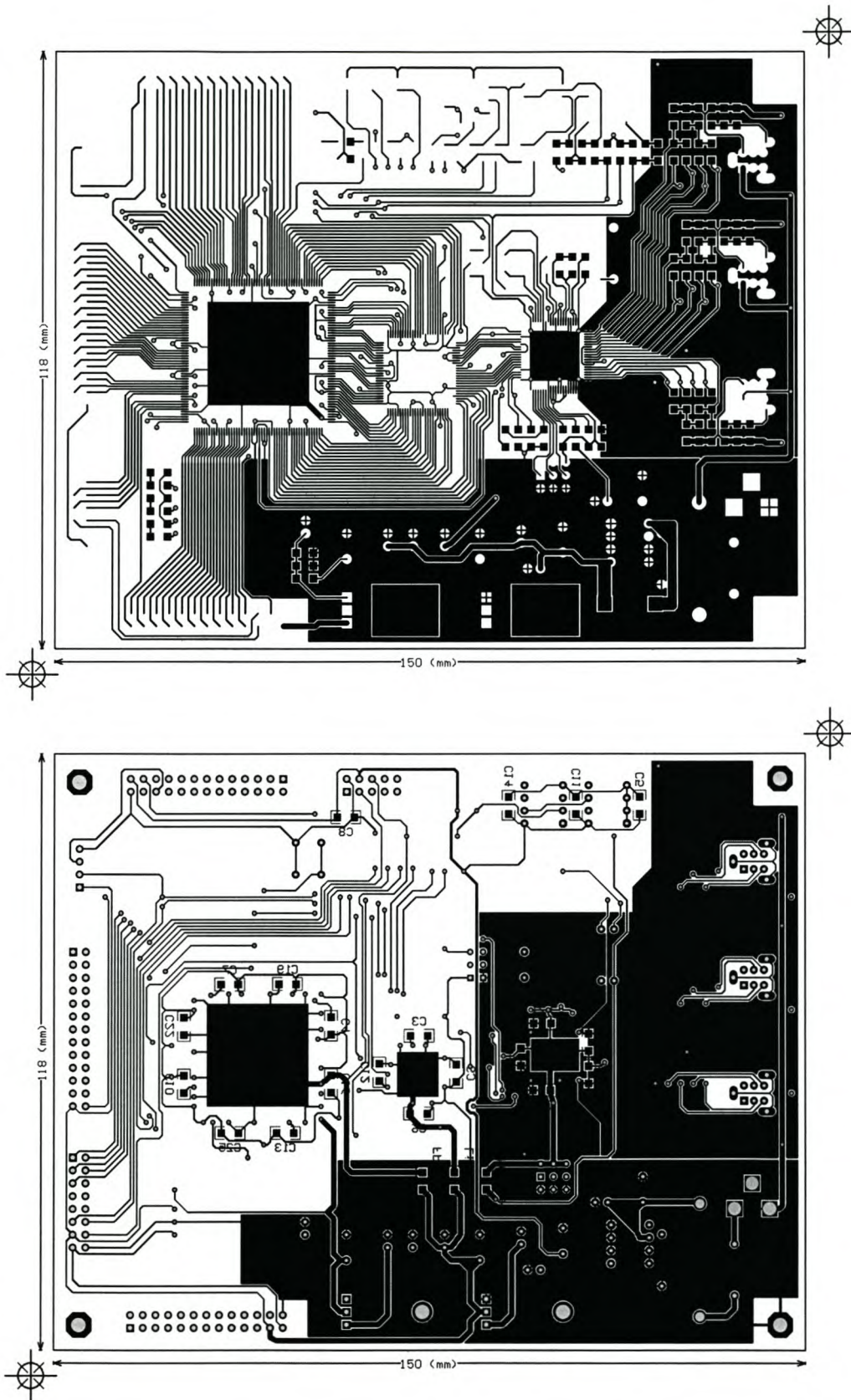


Figure A.4: Schematic diagram of the video encoder board.







**Figure A.5:** *PCB layout of the communications board (rev. A).*

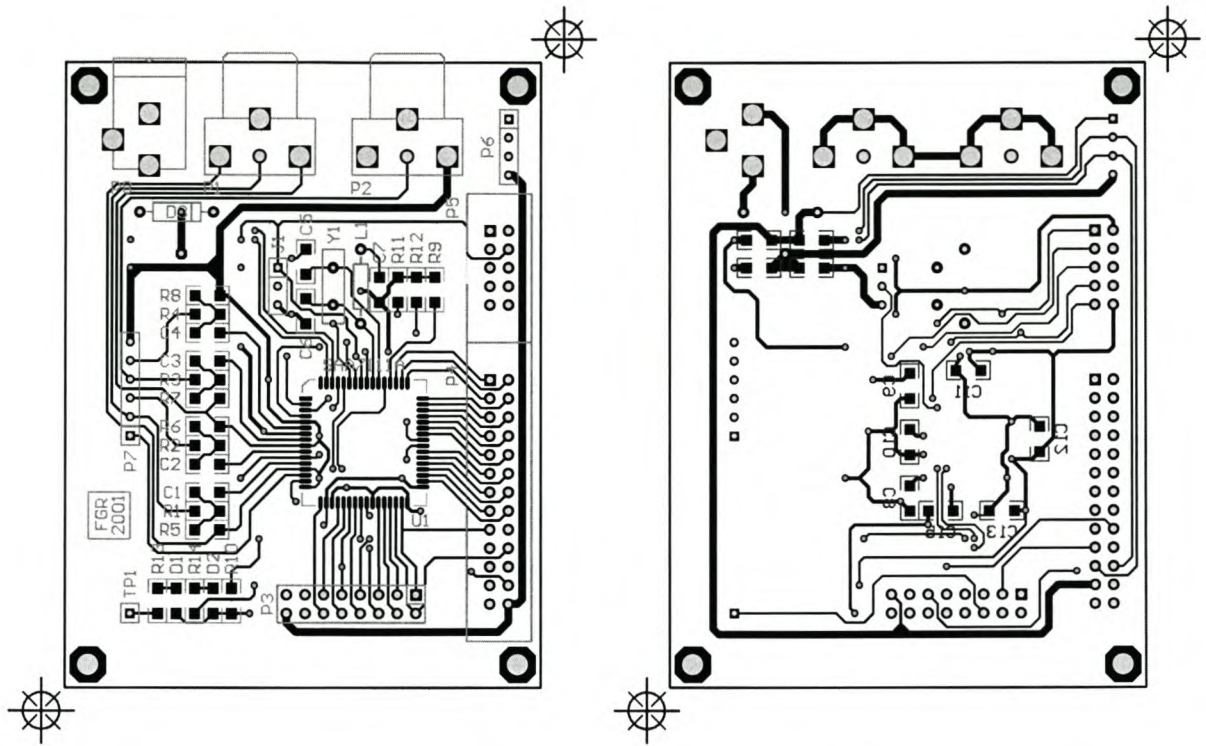


Figure A.6: PCB layout of the video decoder board.

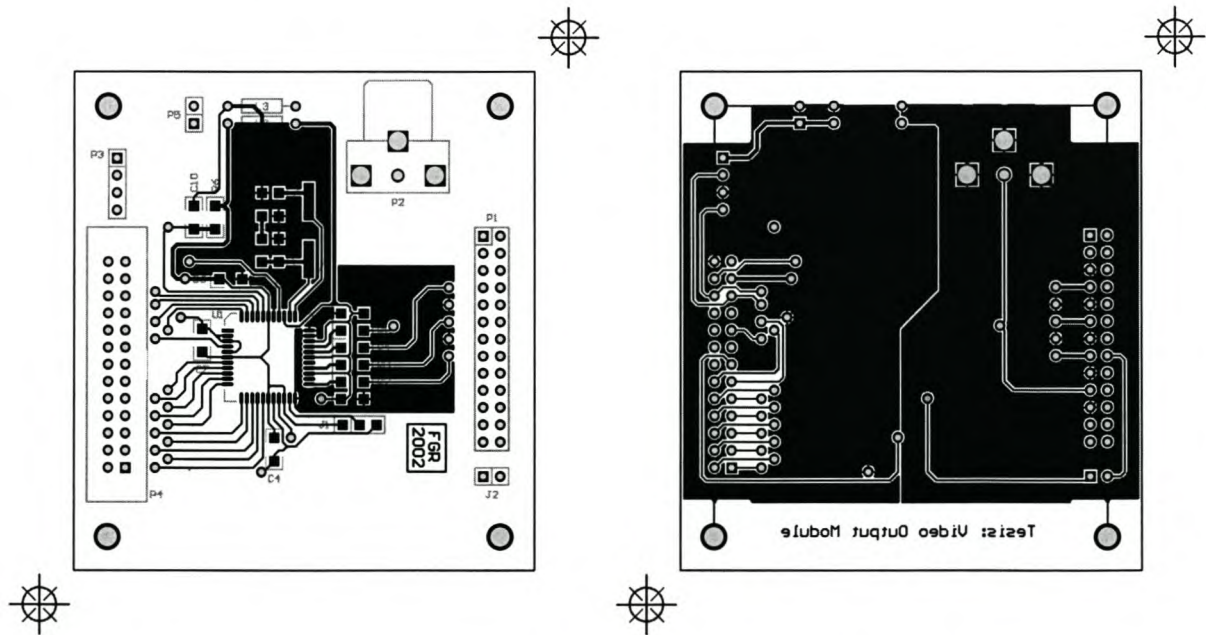


Figure A.7: PCB layout of the video encoder board.

# Appendix B

## VHDL glue logic code

### B.1 Video glue logic for transmission

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY glue_video IS
  PORT (
    -- SAA7111A: Video Port --
    video_llc : IN STD_LOGIC;
    video_data : IN STD_LOGIC_VECTOR(15 downto 0);
    video_href : IN STD_LOGIC;
    video_cref : IN STD_LOGIC;
    video_rts0 : IN STD_LOGIC;
    video_vs : IN STD_LOGIC;

    -- Digital data stream --
    fifo_data : OUT STD_LOGIC_VECTOR(15 downto 0);
    fifo_rw : OUT STD_LOGIC;
    fifo_full : IN STD_LOGIC;

    -- Other control signals --
    glue_clk : IN STD_LOGIC;
    glue_sw : IN STD_LOGIC;
    glue_enable : IN STD_LOGIC
  );
END glue_video;

ARCHITECTURE arch1 OF glue_video IS
  signal video_header : std_logic_vector(31 downto 0);
  alias video_sav_h : std_logic is video_header(28);
  alias video_sav_v : std_logic is video_header(29);
  alias video_sav_f : std_logic is video_header(30);

  alias video_vref : std_logic is video_vs;
  signal video_cref_q, video_cref_q1, video_cref_q2 : std_logic;
  signal video_href_q, video_href_q1, video_href_q2 : std_logic;
  signal video_vref_q, video_vref_q1, video_vref_q2 : std_logic;

```

```

signal video_rts0_q, video_rts0_q1, video_rts0_q2 : std_logic;
signal video_data_q, video_data_q1,
      video_data_q2 : std_logic_vector(15 downto 0);
signal video_counter : std_logic_vector(15 downto 0);

signal state, next_state: integer range 0 to 7;
signal fifo_rw_out : std_logic;
BEGIN
video_header(7 downto 0) <= "11111111"; -- preamble
video_header(15 downto 8) <= "00000000"; -- preamble
video_header(23 downto 16) <= "00000000"; -- preamble

video_header(31) <= '1'; -- status word
video_sav_h <= '1' when (state = 5) or (state = 6);
video_sav_v <= not video_vref_q2;
video_sav_f <= video_rts0_q2;
video_header(27) <= video_sav_v xor video_sav_h;
video_header(26) <= video_sav_f xor video_sav_h;
video_header(25) <= video_sav_f xor video_sav_v;
video_header(24) <= video_sav_f xor video_sav_v xor video_sav_h;

fifo_data <= video_header(15 downto 0) when (state = 2) or (state = 5) else
      video_header(31 downto 16) when (state = 3) or (state = 6) else
      video_data_q2;
fifo_rw <= fifo_rw_out when (fifo_full = '0') else '0';

PROCESS (video_llc)
BEGIN
if (glue_enable = '0') then
      fifo_rw_out <= '0';
elsif rising_edge(video_llc) then
      if (next_state = 2) or (next_state = 3) or
      (next_state = 5) or (next_state = 6) then
            fifo_rw_out <= '1';
      else
            fifo_rw_out <= video_cref_q2 and video_href_q2;
      end if;
end if;
END PROCESS;

PROCESS (video_llc)
BEGIN
if rising_edge(video_llc) then
      video_cref_q <= video_cref; -- synchronize with llc
      video_href_q <= video_href; -- ditto
      video_vref_q <= video_vref; -- ditto
      video_rts0_q <= video_rts0; -- ditto
      video_data_q <= video_data; -- ditto

      if (video_cref = '1') and (video_href = '1') then
            video_data_q1 <= video_data_q;
      end if;
      video_cref_q1 <= video_cref_q;
      video_href_q1 <= video_href_q;
      video_vref_q1 <= video_vref_q;
      video_rts0_q1 <= video_rts0_q;

```

```

        if (video_cref_q1 = '1') and (video_href_q1 = '1') then
            video_data_q2 <= video_data_q1;
        end if;
        video_cref_q2 <= video_cref_q1;
        video_href_q2 <= video_href_q1;
        video_vref_q2 <= video_vref_q1;
        video_rts0_q2 <= video_rts0_q1;
    end if;
END PROCESS;

PROCESS (glue_enable, video_llc, video_cref_q,
         video_href_q, video_cref_q1, video_href_q1)
BEGIN
    case state is
    when 1 =>
        if (video_cref_q = '1') and (video_href_q = '1') then
            next_state <= 2;
        else
            next_state <= 1;
        end if;
    when 2 =>
        next_state <= 3;
    when 3 =>
        next_state <= 4;
    when 4 =>
        if (video_cref_q1 = '1') and (video_href_q1 = '0') then
            next_state <= 5;
        else
            next_state <= 4;
        end if;
    when 5 =>
        next_state <= 6;
    when 6 =>
        next_state <= 1;
    when 7 =>
        if (video_href_q = '1') then
            next_state <= 7;
        else
            next_state <= 1;
        end if;
    when others =>
        if (video_href_q = '1') then
            next_state <= 7;
        else
            next_state <= 1;
        end if;
    end case;

    if (glue_enable = '0') then
        state <= 0;
    elsif rising_edge(video_llc) then
        state <= next_state;
    end if;
END PROCESS;
END arch1;

```

## B.2 Data mover glue logic for the transmitter

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY glue_dm IS
  PORT (
    -- TSB12LV32: Data Mover interface --
    dm_clk : in std_logic;
    dm_data : inout std_logic_vector(0 to 15);
    dm_ready : out std_logic;
    dm_error : in std_logic;
    dm_pktflag : in std_logic;
    dm_done : in std_logic;
    dm_rw : in std_logic;
    dm_pre : in std_logic;

    -- Digital data stream --
    fifo_data : IN STD_LOGIC_VECTOR(15 downto 0);
    fifo_rw : OUT STD_LOGIC;
    fifo_usedw : IN STD_LOGIC_VECTOR(9 downto 0);

    param_channel: in std_logic_vector(5 downto 0);
    param_header: in std_logic;

    -- Other control signals --
    glue_enable : IN STD_LOGIC;
    glue_pktrdy : IN STD_LOGIC
  );
END glue_dm;

ARCHITECTURE arch OF glue_dm IS
  signal pktrdy_d, pktrdy_q: std_logic;
  signal dm_counter: std_logic_vector(15 downto 0);

  signal dm_data_out, dm_data_in: std_logic_vector(0 to 15);
  signal dm_data_tri: std_logic;
  signal dm_ready_out : std_logic;
  signal dm_header : std_logic_vector(0 to 31);
  signal dm_state: integer range 0 to 2;

  signal iso_header: std_logic_vector(0 to 31);
  alias isohdr_dataalen: std_logic_vector(15 downto 0) is iso_header(0 to 15);
  alias isohdr_tag      : std_logic_vector(1 downto 0) is iso_header(16 to 17);
  alias isohdr_channel: std_logic_vector(5 downto 0) is iso_header(18 to 23);
  alias isohdr_tcode  : std_logic_vector(3 downto 0) is iso_header(24 to 27);
  alias isohdr_sy      : std_logic_vector(3 downto 0) is iso_header(28 to 31);
BEGIN
  isohdr_tcode <= "1010";
  isohdr_sy <= "0000";
  isohdr_channel <= param_channel; --"100000"; -- channel 32
  isohdr_tag <= "00"; -- 00 = formatted, 01-11 = reserved
  isohdr_dataalen <= ("0000" & fifo_usedw & "0");

  dm_ready <= dm_ready_out when (glue_enable = '1') else '0';

```

```
dm_data_tri <= not glue_enable;
dm_data <= dm_data_out when (dm_data_tri = '0') else (others => 'Z');

dm_ready_out <= glue_pktrdy when (dm_done = '1') else '0';
dm_data_out <= fifo_data when (dm_state = 2) else dm_header(0 to 15);
fifo_rw <= dm_rw when (dm_state = 2) else '0';

PROCESS (dm_clk, dm_pre)
BEGIN
if rising_edge(dm_clk) then
    if (dm_state = 0) and (dm_pre = '1') then
        dm_header <= iso_header;
    elsif (dm_state = 1) and (dm_rw = '1') then
        dm_header(0 to 15) <= dm_header(16 to 31);
    end if;
end if;
END PROCESS;

PROCESS (dm_clk, glue_enable)
BEGIN
if (glue_enable = '0') then
    dm_state <= 0;
elsif rising_edge(dm_clk) then
    if (dm_done = '1') then
        dm_state <= 0;      -- idle state
    elsif (dm_pre = '1') then
        if (dm_state = 0) and (param_header = '1') then
            dm_state <= 1; -- header state
        else
            dm_state <= 2; -- data state
        end if;
    end if;
end if;
END PROCESS;
END arch;
```

## B.3 Microprocessor and LLC interface glue logic

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY glue_armllc IS
  PORT (
    -- ARM interface --
    arm_mclk : IN STD_LOGIC;
    arm_data : INOUT STD_LOGIC_VECTOR(15 downto 0);
    arm_addr : IN STD_LOGIC_VECTOR(7 downto 0);
    arm_nWBE : IN STD_LOGIC_VECTOR(1 downto 0);
    arm_nOE : IN STD_LOGIC;
    arm_nECS : IN STD_LOGIC;
    arm_nEWAIT : OUT STD_LOGIC;
    arm_nXINTREQ : OUT STD_LOGIC;

    -- TSB12LV32: physical & link-layer interface --
    phy_sclk : in std_logic;
    phy_cna : in std_logic;
    llc_data : inout std_logic_vector(0 to 15);
    -- Note: bit 0 is MSB bit of data
    llc_addr : out std_logic_vector(0 to 6);
    -- Note: bit 0 is MSB bit of address

    llc_wr : out std_logic;
    llc_cs : out std_logic;
    llc_ca : in std_logic;
    llc_tea : in std_logic;
    llc_bclk : out std_logic;
    llc_mcmode : out std_logic;
    llc_m8bit : out std_logic;
    llc_mdinv : out std_logic;
    llc_int : in std_logic;
    llc_cyclein : out std_logic;
    llc_contndr : inout std_logic;
    llc_reset : out std_logic;
    llc_cystart : in std_logic;
    llc_stat : in std_logic_vector(2 downto 0);
    -- Note: for ARM set coldfire pin low (jumper J3)

    -- Other pins
    glue_fpga_enable: out std_logic;
    glue_dm_enable: out std_logic;
    glue_dm_channel: out std_logic_vector(5 downto 0);
    glue_dm_header: out std_logic;
    glue_video_enable: out std_logic;
    glue_video_simdata : out std_logic;
    glue_pkt_interval : out std_logic_vector(3 downto 0);
    glue_portB_enable: out std_logic
  );
END glue_armllc;

ARCHITECTURE arch1 OF glue_armllc IS
  -- Constants for auto detecting the availability of the FPGA --
  constant VERSION_MAJOR : std_logic_vector(15 downto 0) := X"1394";

```



```

constant VERSION_MINOR : std_logic_vector(15 downto 0) := X"A006";

-- Signals for the ARM interface --
signal arm_data_out, arm_data_in: std_logic_vector(15 downto 0);
signal arm_data_tri: std_logic;

-- Signals for the Link Layer Controller (LLC) --
signal llc_data_out, llc_data_in: std_logic_vector(0 to 15);
signal llc_data_tri: std_logic;
signal llc_wait : std_logic;
signal llc_cs_out, llc_wr_out : std_logic;
alias llc_siz1 : std_logic is llc_mcmode;
alias llc_siz0 : std_logic is llc_m8bit;

-- Clocking, debugging and general purpose signals --
alias clk50MHz : std_logic is arm_mclk;
signal test_reg: std_logic_vector(15 downto 0);

signal control_reg: std_logic_vector(15 downto 0);
alias fpga_enable  : std_logic is control_reg(0);
alias llc_reset_out : std_logic is control_reg(1);
alias dm_enable    : std_logic is control_reg(2);
alias video_enable : std_logic is control_reg(3);
alias portB_enable : std_logic is control_reg(4);
alias video_simdata : std_logic is control_reg(8);

signal reg_wb0 : std_logic;
signal reg_wb1 : std_logic;
signal dm_param_reg : std_logic_vector(15 downto 0);

alias dmp_header : std_logic is dm_param_reg(8);
alias dmp_channel: std_logic_vector(5 downto 0)
  is dm_param_reg(5 downto 0);
alias dmp_pkt_interval : std_logic_vector(3 downto 0)
  is dm_param_reg(15 downto 12);
BEGIN
reg_wb0 <= arm_nWBE(0) or arm_nECS;
reg_wb1 <= arm_nWBE(1) or arm_nECS;

with (arm_addr) select
arm_data_out <=
  dm_param_reg when X"FA",
  ("0000000000000000" & llc_int & llc_ca) when X"FB",
  VERSION_MAJOR when X"FC",
  VERSION_MINOR when X"FD",
  test_reg      when X"FE",
  control_reg   when X"FF",
  llc_data_in   when others;

arm_data_in <= arm_data;
arm_data_tri <= arm_nOE or arm_nECS;
arm_data <= arm_data_out when (arm_data_tri = '0') else (others => 'Z');
arm_nEWAIT <= llc_wait or arm_nECS;
arm_nXINTREQ <= llc_int when (llc_reset_out = '1') else '1';

reg_low_byte:
PROCESS (reg_wb0)

```

```

BEGIN
---- Low Byte ----
if rising_edge(reg_wb0) then
  case conv_integer(arm_addr) is
    when 16#FA# =>
      dm_param_reg(7 downto 0) <= arm_data_in(7 downto 0);
    when 16#FE# =>
      test_reg(7 downto 0) <= arm_data_in(7 downto 0);
    when 16#FF# =>
      control_reg(7 downto 0) <= arm_data_in(7 downto 0);
    when others =>
      null;
  end case;
end if;
END PROCESS;

reg_high_byte:
PROCESS (reg_wb1)
BEGIN
---- High Byte ----
if rising_edge(reg_wb1) then
  case conv_integer(arm_addr) is
    when 16#FA# =>
      dm_param_reg(15 downto 8) <= arm_data_in(15 downto 8);
    when 16#FE# =>
      test_reg(15 downto 8) <= arm_data_in(15 downto 8);
    when 16#FF# =>
      control_reg(15 downto 8) <= arm_data_in(15 downto 8);
    when others =>
      null;
  end case;
end if;
END PROCESS;

-----
--- Link Layer Controller(LLC) to/from ARM processor glue logic:
--- This logic is responsible for connecting the ARM to the LLC,
--- and to generate the correct control signals between the two.
--- Only 16 bit write transfers is allowed (8-bit is ignored)
-----

--- Generate address for LLC ---
llc_data_in <= llc_data;
llc_data_out <= arm_data_in;
llc_data_tri <= llc_wr_out or llc_cs_out;
llc_data <= llc_data_out when (llc_data_tri = '0') else (others => 'Z');
llc_addr(0 to 6) <= arm_addr(5 downto 0) & "0";
llc_m8bit <= '0'; -- 16-bit data mode only
llc_mcmode <= '1'; -- handshake mode
llc_mdinv <= '1'; -- data invariance mode
llc_wr_out <= '0' when (arm_nWBE & arm_nECS) = "000" else '1';
llc_cs_out <= '0' when (arm_nWBE & arm_nECS & arm_addr(7 downto 6)) = "00000" else
  '0' when (arm_nOE & arm_nECS & arm_addr(7 downto 6)) = "0000" else
  '1';
llc_wait <= not llc_ca when llc_cs_out = '0' else '1'; -- Chip select acknowledge
llc_cs <= llc_cs_out; -- Chip select line

```

```
llc_wr <= llc_wr_out; -- Write enable line
llc_bclk <= arm_mclk;
llc_cyclein <= '1'; -- "... should be tied high when not used."
llc_contndr <= '0'; -- "... we are NOT an root contender."
llc_reset <= '0' when (llc_reset_out = '0') else 'Z';
--- Note for ARM program: The external IO bus minimum timing
--- must be selected as follows:
---      Tcos = 0, Tacs = 1, Tcoh = 1, Tacc = 4 (minimum values)

glue_fpga_enable <= fpga_enable;
glue_dm_enable <= dm_enable;
glue_video_enable <= video_enable;
glue_portB_enable <= portB_enable;
glue_video_simdata <= video_simdata;
glue_dm_channel <= dmp_channel;
glue_dm_header <= dmp_header;
glue_pkt_interval <= dmp_pkt_interval;
END arch1;

-- Assumptions and simplifications:
--- To simplify the design, we are only going to do 16-bit access to the LLC
-- (all 8-bit accesses will be ignored) Write software accordingly!
```

## B.4 Data mover glue logic for the receiver

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY glue_dm_rx IS
  PORT (
    -- TSB12LV32: Data Mover interface --
    dm_clk : in std_logic;
    dm_data : in std_logic_vector(0 to 15);
    dm_ready : out std_logic;
    dm_error : in std_logic;
    dm_pktflag : in std_logic;
    dm_done : in std_logic;
    dm_rw : in std_logic;
    dm_pre : in std_logic;

    -- Digital data stream --
    fifo_data : OUT STD_LOGIC_VECTOR(15 downto 0);
    fifo_rw : OUT STD_LOGIC;

    -- Other control signals --
    glue_enable : IN STD_LOGIC
  );
END glue_dm_rx;

ARCHITECTURE arch1 OF glue_dm_rx IS
BEGIN
  dm_ready <= '0'; -- Must be low when DM is in receive mode.

  process (dm_clk, glue_enable)
  begin
    if (glue_enable = '0') then
      fifo_data <= (others => '0');
      fifo_rw <= '0';
    elsif rising_edge(dm_clk) then
      fifo_data <= dm_data;
      fifo_rw <= dm_rw and not dm_pktflag;
    end if;
  end process;
END arch1;
```

# Appendix C

## Miscellaneous source code

### C.1 Script for automatic program upload

This script is run by a modified version of the CHAT program. A uuencoded program file is uploaded to the ARM Evaluator-7T board and programmed into the flash memory.

```
####
# Uploads an UU encoded file to the ARM E7T board
# and program the flash. Then it executes the program.
#
# Assumptions:
# o The program is loaded at 0x01830000
# (make sure linker script is the same)
# o The uue file name is "upload.uue"
# o The program name is "thesis"
# o The autoboot flag is set (when compiled) see BSL table
#
# Todo:
# - Error checking has not been implemented
####
SAY "Waiting for ARM to boot...\n"
ABORT "RedBoot>"
ECHO ON
^[[5n "^[[0n\c"
autoboot "\d"
SAY "\nARM booted, sending upload command ..."

oot: "\pflashload 1830000 "

file. "\d"
SAY "\n==> Uploading the FILE"
"" @upload.uue
SAY "\n==> Finished uploading"

oot: "\prommodules"
oot: "\pboot"
oot: "\ptesis"

# end of file #
```

# Bibliography

- [1] N. L. Steenkamp, "Development of the on board computer flight software for SUNSAT 1," Master's thesis, University of Stellenbosch, 1999.
- [2] J.-A. Koekemoer, "Investigation of a command and data handling architecture for the SUNSAT-2 micro satellite." Master's thesis, University of Stellenbosch, 1999.
- [3] *Electrical Characteristics of Low Voltage Differential Signaling (LVDS) Interface Circuits*, Telecommunications Industry Association, Std. ANSI/TIA/EIA-644, Mar. 1996.
- [4] *IEEE Standard for a High Performance Serial Bus*, IEEE Std. 1394, 1995.
- [5] *IEEE Standard for a High Performance Serial Bus - Amendment 1*, IEEE Std. 1394a, 2000.
- [6] *IEEE Standard for a Higher Performance Serial Bus - Amendment 2*, IEEE Std. 1394b, 2002.
- [7] *IEEE Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses*, IEEE Std. 1212, 2000.
- [8] K. Jack, *Video demystified: a handbook for the digital engineer*, 3rd ed., ser. Demystifying technology series. LLH Techonology Publishing, Eagle Rock, VA 24085: LLH Techonology Publishing, 2001, no. ISBN 1-878707-56-6.
- [9] F. Aliche, F. Bartholdy, S. Blozis, F. Dehmelt, F. Forstner, N. Holland, and J. Huchzermeier, "Application report: Comparing bus solutions," Texas Instruments, Mar. 2000.
- [10] *SpaceWire - Links, nodes, routers and networks*, European Cooperation for Space Standardization (ECSS) Std. ECSS-E-50-12A (Draft), Nov. 2002.
- [11] *IEEE Standard for Heterogeneous InterConnect (HIC)*, IEEE Std. 1355, 1995.
- [12] (2002, Nov.) Spacewire home page. [Online]. Available: <http://www.estec.esa.nl/tech/spacewire/>
- [13] B. Henehan, D. Yaklin, B. Gugel, and J. Akgul, *Galvanic Isolation of the IEEE 1394-1995 Serial Bus*, Texas Instruments, Oct. 1997.
- [14] *TSB12LV01B: IEEE 1394-1995 High-Speed Serial-Bus Link-Layer Controller*, Texas Instruments, June 2000.

- [15] *PDI1394L11: 1394 AV link layer controller*, Philips, Oct. 1997.
- [16] *PDI1394L40: 1394 enhanced AV link layer controller*, Philips, May 2000.
- [17] *TSB12LV32, TSB12LV32I: IEEE 1394-1995 and P1394a Compliant, General-Purpose Link-Layer Controller*, Texas Instruments, Apr. 2000.
- [18] *SAA7113H: 9-bit video input processor*, Philips, July 1999.
- [19] *SAA7111A Enhanced Video Input Processor (EVIP)*, Philips, May 1998.
- [20] *SAA7120H; SAA7121H: Digital video encoder*, Philips, Oct. 2002.
- [21] *SAA7126H; SAA7127H: Digital video encoder*, Philips, May 1999.
- [22] M. Ackerman, "Kernel support for embedded reactive systems," Master's thesis, Department of Computer Science, University of Stellenbosch, 1993.
- [23] (2002) On-line applications research (oar) corporation's home page. [Online]. Available: <http://www.rtems.com/>
- [24] (2002, Nov.)  $\mu$ clinux home page. [Online]. Available: <http://www.uclinux.org/>
- [25] (2002) ecos home page. [Online]. Available: <http://sources.redhat.com/ecos/>
- [26] *LM2594/LM2594HV SIMPLE SWITCHER Power Converter 150 kHz 0.5A Step-Down Voltage Regulator*, National Semiconductor, Dec. 1999.
- [27] *LT1117: 800mA Low Dropout Positive Regulators Adjustable and Fixed 2.85V, 3.3V, 5V*, Linear Technology, 2000.
- [28] (2002) Firewire reference platform 1.0. [Online]. Available: <http://developer.apple.com/firewire/platform.html>
- [29] *ARM Evaluator-7T Board User Guide*, ARM, Aug. 2000.
- [30] *KS32C50100 RISC Microcontroller, Revision 1*, Samsung Electronics, 1999.

# Index

- ACEX, 53
- ADCS, 1
- amateur radio, 1
- Angel debug monitor, 71
- architecture, 23
- ARM7TDMI, 71
- assert, 56
- asynchronous
  - quadlet, 72
  - transactions, 73
- ATF, 57, 68
  
- balanced data transmission, 7
- boot strap loader, 71
- BSL, 71
- bus
  - manager, 19, 52
  - reset, 59, 67
- ByteBlasterMV, 53
  
- cache, 70
- CAN, 2
- CCIR-656, 50
- C&DH, 2
- chat, 72
- command line, 59
- common-mode voltage, 8
- configuration
  - FPGA, 53
  - ROM, 59, 74
- control
  - application, 59, 74, 78
  - registers, 52
- crosstalk, 6
- CSR, 57, 74
- CVBS, 51
- cycle counter, 57
- cycle-start packets, 80
  
- data mover port, 75
- data-strobe, 15
  
- debugging, 71
- differential transmission, 7
- differential voltage, 8
- DMA, 26
- driver, 56
  
- eCos, 56, 69, 71
- embedded system, 57
- eShell, 59
- EUI-64, 57
- Evaluator-7T, 52, 64, 81
  
- false bus reset, 78
- FIFO, 26, 46, 50
- FIFO buffer, 73, 76
- FireWire, 13
- flash memory, 71
- FPGA, 52
- full duplex, 9
  
- glue logic, 50, 66
- GNU debugger, 71
- GPS, 1
- GRF, 57, 73
- ground station, 3
  
- half duplex, 9
- hardware abstraction, 56
- HDRERR, 80
- header, 62
- help string, 59
- hot plug, 13
- HyperTerminal, 71
  
- I<sup>2</sup>C, 51, 54, 59, 75, 78
- IEEE 1394, 13, 56
  - cable, 67
  - driver, 72
- iLink, 13
- infrastructure, 56
- integration, 61
- interactive, 59



*INDEX*

- interrupt, 56, 67
- IRM, 19
- isochronous cycle, 75
- JTAG, 53, 71
- kernel, 56
- KS32C50100, 52, 59, 78
- LED, 56
- link layer, 18
- Linux, 56, 68, 72, 74
- LLC, 67, 68, 73
- logic analyser, 76
- LPM component, 47
- LVDS, 11
- MaxPlusII, 54
- memory management, 57
- mesh, 2
- microprocessor, 64
- Minicom, 71
- Multi-ICE, 71
- multidrop, 9
- multipoint, 9
- non-zero return, 15
- noncyclic, 13
- NRZ, 15
- OBC, 1
- packet
  - allocation, 57
  - header, 76
- PAL, 46
- parallel, 8
- passive serial, 53
- PCB, 52
- PCB layouts, 61
- point-to-point, 9
- power supply, 55, 62
- queue, 73
- read-write
  - cycle, 52
  - test, 66
- receiver, 59
- RedBoot, 71
- registers, 66
- regulator
  - linear, 55
  - switched, 55, 65
- repeater, 76
- RF downlink, 3
- RGB, 51
- ribbon cable, 51, 64
- routing setup, 76
- RS232, 10
- RS422, 10
- RS485, 11
- SAA7111A, 59
  - board, 78
  - software driver, 75
- SAA7127H, 51, 59
  - board, 78
- SB1394, 13
- schematics, 61
- script, 72
- semaphore, 69, 73
- single-ended transmission, 6
- software driver, 56
- SpaceWire, 20
- stack, 56, 73
- SUNSAT, 1
- terminal, 71
- termination, 62
- termination network, 48
- thread, 56, 73
- trace, 56
- transaction layer, 18
- transmitter, 59
- TSB12LV32, 49, 56, 73, 76
- TSB41LV03, 62
- TSB43LV03, 48
- twisted-pair, 14
- USB, 12
- VHS, 51
- video
  - converter, 43
  - decoder, 81
  - glue logic, 45
  - signal, 4
  - source, 43
  - stream, 59