Multiprotocol Control of Networked Home Entertainment Devices

A thesis submitted in fulfilment of the
requirements for the degree of

Master of Science

of

Rhodes University

by

David Robert Siebörger

February 2004

Abstract

Networks will soon connect a wide range of computing devices within the home. Amongst those devices will be home entertainment devices. Remote control over the network will be a key application for networked entertainment devices, and requires a protocol for communication understood by both controller and controlled device. Devices capable of communication using multiple control protocols will be compatible with a wider range of controllers than those which implement only one control protocol. This work examines home networks and a number of control protocols. The implementations of the UPnP and AV/C protocols for an AV receiver are described. The issues involved in the concurrent use of multiple control protocols to control a device are considered, possible methods of concurrent control discussed, and a solution which simulates virtual copies of the device is implemented and tested.

Contents

1 Introduction **1**

2 Home Entertainment Networks **4**
2.1 The advent of home networks . . . 4
2.2 Applications that utilise home entertainment networks . . . 6
2.3 Requirements of home entertainment networks . . . 8
2.4 The IEEE 1394 bus . . . 9
2.5 Summary . . . 14

3 Control Protocols **17**
3.1 Universal Plug and Play . . . 17
3.2 Audio Visual Control . . . 20
3.3 Home Audio/Video Interoperability . . . 26
3.4 Common aspects . . . 30
3.5 Summary . . . 33

4 An Example Home Entertainment Device **35**
4.1 The Yamaha RX-V1000 . . . 35
4.2 RX-V1000 serial control protocol . . . 36
4.3 A state variable model of the RX-V1000 . . . 37
4.4 Controlling the RX-V1000 . . . 38
4.5 Summary . . . 39

5 Implementing Universal Plug and Play **40**
5.1 Initial feasibility study . . . 40
5.2 Design of the implementation . . . 41
5.3 The chapters of the UPnP Device Architecture . . . 43
5.4 UPnP chapter 0: Addressing . . . 43
5.5 UPnP chapter 1: Discovery . . . 45
5.6 UPnP chapter 2: Description . . . 49
5.7 UPnP chapter 3: Control . . . 52
5.8 UPnP chapter 4: Eventing . . . 57
5.9 UPnP chapter 5: Presentation . . . 67
5.10 Coordinating the UPnP control server: the *rxvupnp* program . . . 70
5.11 Summary . . . 71

6 Implementing AV/C **72**
6.1 Design of the implementation . . . 72
6.2 Function Control Protocol . . . 72
6.3 AV/C . . . 75
6.4 Unit . . . 76
6.5 Audio subunit . . . 77
6.6 Communicating with the device . . . 81
6.7 Notify commands . . . 81
6.8 An AV/C control point . . . 83
6.9 Performance of the AV/C control server . . . 84
6.10 Summary . . . 85

7 Synchronisation **87**
7.1 The need for synchronisation . . . 87
7.2 A model of the problem . . . 88
7.3 Desired attributes of possible solutions . . . 89
7.4 Possible approaches to synchronisation . . . 90
7.5 Implementing a virtual device simulator . . . 93
7.6 Testing the system . . . 96
7.7 Summary . . . 98

8 Conclusion **101**

A State Variable Model of the RX-V1000 **106**

B UPnP Root Device Description **109**

C UPnP Service Description **110**

D Performance Test Data **114**
D.1 UPnP SOAP tests . . . 114
D.2 UPnP GENA tests . . . 115
D.3 AV/C status tests . . . 116
D.4 AV/C notify tests . . . 116

E Contents of the CD-ROM **117**

List of References **119**

List of Tables

3.1 Possible response codes for each AV/C command type 24

5.1 Results of *soaphttpd* performance testing 56
5.2 Results of *genapub* performance testing 66

6.1 Results of *rxvavc* performance testing 84

7.2 *librxv* functions 95

List of Figures

1.1 The intended multiprotocol network control system . . . 3

2.1 IEEE 1394 four-layer model . . . 10
2.2 IEEE 1394 sockets types . . . 10
2.3 A home with an IEEE 1394 network . . . 15

3.1 UPnP protocol stack . . . 19
3.2 UPnP device and service model . . . 20
3.3 AV/C unit, subunit and function block model . . . 22
3.4 AV/C descriptor hierarchy, adapted from [28] . . . 25
3.5 HAVi device and functional component model . . . 27
3.6 HAVi software elements, taken from [32] . . . 29

4.1 The Yamaha RX-V1000 AV Receiver . . . 36
4.2 *rxvqt* user interface . . . 38

5.1 Components of the UPnP control server and its interactions with a client . . . 42
5.2 SSDP announcement and search processes . . . 46
5.3 SSDP device and service data structure . . . 48
5.4 HTTP retrieval of root and service descriptions . . . 49
5.5 Current and proposed means of indicating the type of an argument . . . 51
5.6 SOAP request data structure . . . 53
5.7 SOAP service designs . . . 55
5.8 Communication between subscriber and publisher . . . 58
5.9 Examples of state variable change messages . . . 59
5.10 One producer and queue; *n* consumer threads . . . 60
5.11 The mailbox data structure (adapted from [57, p 54]) . . . 61
5.12 One producer; *n* consumer threads and queues . . . 61
5.13 genapub data structure . . . 64
5.14 *genapub* performance testing experiment . . . 65
5.15 Presentation page for the RX-V1000 . . . 69
5.16 *rxvupnp* data flows . . . 70

6.1 Components of the AV/C control server and client . . . 73
6.2 An FCP frame in an IEEE 1394 asynchronous write packet, adapted from [24] . . . 73
6.3 Linux IEEE 1394 and *raw1394* driver stack . . . 74

6.4 Stream-based communication between AV/C layers 75
6.5 AV/C frame format, based on [23] 75
6.6 RX-V1000 AV/C unit 78
6.7 Function block command frame format, based on [25] 80
6.8 Example of an AV/C frame containing a selector function block command (byte values are in hexadecimal) 80
6.9 Example of an AV/C frame containing a feature function block command (byte values are in hexadecimal) 81
6.10 Notifications data structure 82
6.11 *avcqt* user interface 83

7.1 Components involved in the control of a device using two protocols 88
7.2 Using a protocol-conversion bridge for multiprotocol control 92
7.3 *splitrxv* data flows 93
7.4 Control server testing system 97
7.5 Test results from *verify.pl* 99

8.1 Software components of the control servers 103
8.2 A home entertainment scenario made possible by multiprotocol control 104

Acknowledgements

My grateful thanks are due to:

- my supervisor, Richard Foss, who plotted the course of this work, and encouraged and assisted me along the way;
- Guy Antony Halse and Barry Irwin, who entertained my arguments about algorithms and protocols;
- Guy, Barry, Ingrid Brandt and my parents, all of whom urged me on to complete this thesis and helped proofread it once I had;
- Digital Harmony Technologies, Inc., who provided funding and the DHIVA system and development software;
- Bradley Klinkradt, whose IP-over-1394 stack was used on the DHIVA; and
- the Centre of Excellence in Distributed Multimedia and its sponsors, who provided equipment and funding.

-drs

Chapter 1

Introduction

Home entertainment systems in today's homes usually consist of a TV, with an amplifier and speakers, and devices such as VCRs, CD and DVD players, TV decoders, and a wide range of other components. All these devices are connected to each other in order to pass audio and video, using many different analogue and digital connection types. This results in consumer confusion, loss of signal quality and an unsightly tangle of cables. The user interface of each device in the system is, typically, a small LCD display and buttons on the front panel, and an infra-red remote control. The front panel interfaces can be difficult to use, and the number of IR remote controls for a home entertainment system grows as devices are added.

Computer networks have entered many homes as a means of connecting PC's to share Internet access, files and printers, amongst other applications. In the future, it is likely that the networks that connect the PCs in the home will expand to connect devices all over the house into a single network. Since most home entertainment devices already include embedded computer systems to manage the functions of the device, and the computing power of those systems is increasing, those devices will also be connected to the network. A network which includes home entertainment devices is considered a home entertainment network. Once connected to a home entertainment network, home entertainment devices will be able to use the network to stream digital media from one device to another, rather than the current analogue and digital cable connections.

Another application that the home entertainment network will make possible is the remote control of one device from another. In an example scenario, a digital TV set could display a user interface for a VCR, and the user would interact with the controls on the interface using the TV's remote control. The instructions that the user issues would be passed from the TV to the VCR over the network, and the VCR would carry them out, then update the interface on the TV to reflect the results of the operation. A hand-held PDA with a wireless network connection could also be used as the controller, and since it

would be able to control multiple devices, it would be able to replace a number of IR remote controls for devices with a single controller.

Control of a device over a network requires that both the controlled device and the controller are connected to the same network and use a common control protocol which specifies the manner in which control instructions and responses are sent. Various alliances of companies in the home entertainment and computing industries have specified a range of such protocols. While those protocols share common goals, they are all designed in different ways, and thus have differing strengths and weaknesses. At present, there is no single protocol implemented by all devices, and each manufacturer selects a protocol for use in their products.

In the market for network-attachable home entertainment devices, a product can gain a competitive advantage by supporting multiple control protocols. This widens the range of other products that it can communicate with, increasing the chances that it will be compatible with devices that are already present on the purchaser's home network. However, there are technical challenges to be overcome in the development of a product that uses multiple control products.

This thesis describes work that has been done to meet these challenges. In order to do this, the thesis first considers, in general terms, the concepts of the home entertainment network and control protocols.

Chapter 2 examines concepts concerning the home entertainment network. The IEEE 1394 serial bus is introduced and proposed as a suitable network technology upon which home entertainment networks can be built.

Chapter 3 describes three control protocols which can be used to control devices on an IEEE 1394 home network: UPnP, AV/C and HAVi. The models and concepts which the protocols employ are described in general, and various aspects that they share are identified and compared.

Since there are few devices available today that are ready for the vision of the home entertainment network described above, the remaining chapters cover the design and development of a prototype IEEE 1394-connected, multiprotocol-controlled home entertainment device that could form part of such a home entertainment network. The prototype is intended to test the feasibility of concepts involved in the vision, including the use of the IEEE 1394 bus for home entertainment control applications, the use of UPnP or AV/C to control a home entertainment device, and the simultaneous use of both control protocols. The intended system is diagrammed in figure 1.1.

Chapter 4 looks at a Yamaha AV receiver which is typical of devices found in today's home entertainment systems. It supports a number of means for remote control, and the RS-232 protocol it provides is discussed.

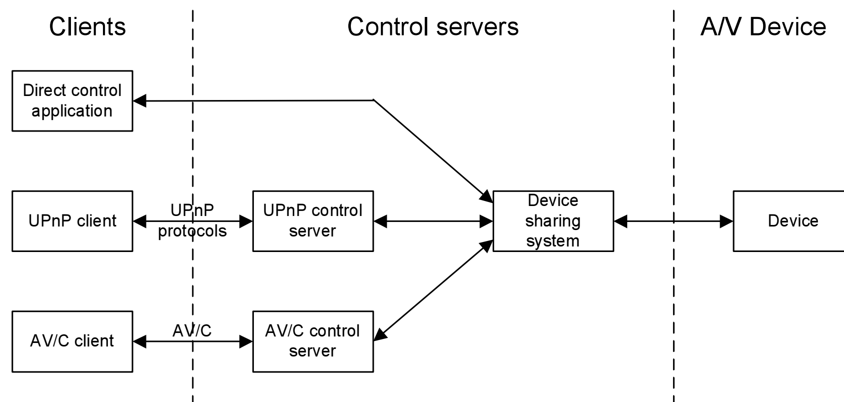

Figure 1.1: The intended multiprotocol network control system

Chapter 5 studies each chapter of the UPnP specification in detail and considers possible methods of implementing the specification. It describes how a UPnP control server (the software which runs on the device and handles control communications on the network) for the AV receiver was designed, developed and tested.

Chapter 6 examines the AV/C protocol and how an AV/C control server was designed for the AVR, and alternative ways in which it could have been designed. The development and testing of the control server and a corresponding control point application are discussed.

Chapter 7 considers the need for devices to support more than one control protocol simultaneously, the issues involved in doing so, and some possible solutions to those problems. One of the methods identified was implemented in the prototype system, and this implementation is described. Automated testing verified its ability to support both UPnP and AV/C control servers at the same time.

Chapter 2

Home Entertainment Networks

This chapter examines how computer networks will cover the home, and defines home entertainment networks within those networks. The IEEE 1394 bus is proposed and evaluated as a network well-suited for building home networks.

2.1 The advent of home networks

The Digital Home Working Group (DHWG), a group of computer and consumer electronics companies promoting interoperability amongst devices on home networks, has identified three islands of computing devices in the home: the PC and Internet world, including PCs and their peripherals such as printers and digital cameras; the mobile device world, including notebook PCs, cellular phones and PDAs; and the consumer electronics, home entertainment and broadcast world.

Home entertainment systems are found in most living rooms today. They are used for recreation and access to news and information by playing broadcast media, such as terrestrial, satellite and cable TV, and radio, as well as recorded media such as compact discs, video tapes and DVD discs. A home entertainment system is comprised of a number of consumer electronics devices, connected to pass media between them in analogue and simple digital formats. They include devices which are sources of broadcast media, such as radio and TV tuners, satellite and cable receivers; players for recorded media, for example, CD players, VCRs, digital video recorders, camcorders and DVD players; media outputs, such as TV sets and speakers; and a range of intermediate devices, including amplifiers, AV receivers, surround sound decoders, and set-top pay-TV decoders. Many of these devices now include increasingly sophisticated computing capabilities used for controlling the operation of the devices, media processing (including tasks such as audio and video effect processing, digital video decoding and surround sound

decoding) and user interfaces. These embedded computers could potentially be connected to a computer network.

Personal computers are present in many homes. Where there is more than one PC in a home, many of those PCs have been connected by networks, to allow file and printer sharing originally, and more recently, multi-player gaming and sharing of broadband Internet access, amongst other applications.

Communications systems exist for the devices within the three islands, such as Ethernet networks connecting PCs to each other and to broadband Internet access, Bluetooth networks linking notebook PCs to cellular phones, and the wide range of analogue and digital connections which home entertainment systems use. However, the three islands remain largely isolated. The DHWG envisage that the islands will converge to a single network reaching throughout the home. [1, p 3]

This view is echoed by Rose and Scherf who characterise consumers' current definition of home networking as "sharing a broadband connection between and among multiple computers in the home" – this only recognises networking in the PC realm. However, they expect that a wider range of products will be connected, leading to their prediction that, in future, home networks "will distribute movies and television broadcasts, music, information, enable online gaming, and send control signals throughout the home." [2, p 1] Thus the network will be used to carry the audio and video signals in streams of digitised media, instead of the variety of connections present in current home entertainment systems.

O'Driscoll presents a wider, more generalised approach to the topic. Recognising that there are processors embedded in devices around the home, he envisages a future in which "pervasive computing" will connect all devices in the home onto a network, and those networks onto the Internet, to allow communication between people at any time and place. He describes a model in which Information Appliance Networks (IANs) connect Electrodomestic Network Devices (ENDs) around the home to provide an information infrastructure that delivers Digital Media Commodities (DMCs) as well as brokering, integrated solutions and customer relationship management services to their users. DMCs are the products that people will make use of for communication, information and entertainment through ENDs on an IAN. The entertainment DMCs that he lists include TV programs, video on demand, streaming audio, gaming, text and hypermedia. [3]

In another paper, Scherf classifies entertainment networks into the following categories:

- point-to-point, a single-purpose network connection between two devices for transferring audio or video;
- distributed or multi-room, allowing AV signals to be transmitted from a device to a TV or stereo elsewhere in the house; and

- cluster, allowing devices to announce themselves on a network, configure themselves, and communicate to share processing and storage. This enables devices that coordinate control of other devices. [4, p 1]

It is quite possible that a home entertainment network will evolve and grow through these three phases to reach the home-wide network described above.

Entertainment devices will be present on home networks in rooms around the house. Scherf and Parks define a network-capable entertainment product as “non-PC device purchased by a consumer that leverages the Internet, processing power, and perhaps a level of hard drive storage to play video, stream audio (music), play games, and perform other similar functions.” [4, p 4] Therefore, home entertainment networks are considered to be those portions of a home network focused on entertainment, and are the area of the home network that will be the focus of this thesis.

2.2 Applications that utilise home entertainment networks

The installation of home networks will be driven by the advantages they offer and new applications that they will make possible. Rose and Scherf divide home networking applications into four categories and offer examples:

1. *data-centric:* sharing broadband Internet access, PC LAN applications, telecommunications, access to information platforms

2. *multimedia-centric:* multi-person gaming, personalised content, stored/streamed multimedia, on-demand content

3. *home management:* temperature and lighting controls, energy management, security systems, remote control

4. *value-added services:* voice, protection, upgrades, communities [2, p 5]

Network-connected home entertainment devices will be primarily involved in applications that fall into the multimedia-centric category, but must exist in the context of other applications. Specific advantages of the home network in entertainment systems include:

1. *Reducing cable clutter:* a plethora of different cabling types in a home entertainment system frustrates and confuses the consumer. With a single connection to the home entertainment network, the devices will be able to send and receive all the signals that would previously be connected with multiple connections.

2. *Digital media:* analogue signals degrade with every length of cable and connection between them. A home entertainment network will transfer media in digital formats, so it will not suffer degradation. Content providers such as movie studios and record labels wish to protect the content they produce from unauthorised duplication, and copy protection mechanisms can be developed for digital networks.

3. *Allowing communication between different classes of device:* a home entertainment network will allow media to move freely between devices that would have been on separate islands. For instance, video stored on a PC could be displayed on a TV across the house.

4. *Access to external media:* allows access to digital media streamed via a broadband network connection, leading to applications such as video-on-demand libraries.

5. *Control and automation:* devices can be controlled by other devices via the network. The most obvious advantage is that this can eliminate a multitude of remote controls; further applications include remote control from outside the home. It can also allow co-operation amongst devices.

This work focuses on protocols that can be used to provide networked control, and their implementation in entertainment devices, rather than media transfer applications.

Networked control occurs when one device, the controller, issues commands to another device, the controlled device, using the network as the means of communication. The controlled device carries out the controller's instructions and returns a response to the controller, and can also inform the controller as its status changes. A control protocol defines a format in which commands, responses and notifications are represented for transmission across the network. The control protocol is the language of communication between controller and controlled device. In this work, the software that implements networked control services on the controlled device is referred to as a control server, while software which allows a user to send commands from a controller device is called a control point.

A number of control protocols suitable for use in home entertainment networks have been specified, usually by groups of consumer electronics, computer or network equipment companies. Those protocols all share the goal of allowing remote control of devices, but since they have been designed by different groups of companies, with different market objectives, the nature of the protocols varies. At present, these control protocols compete for manufacturer and consumer acceptance, in hopes that the protocol will achieve critical mass and become the industry standard.

Consumers will purchase network-controllable home entertainment products to add to their home entertainment networks, which, in many cases, will already contain one or more controller devices. They

will want to select new devices which are compatible with their existing controllers to maintain their investment in those controllers. Further, O'Driscoll points out, in a section on issues relating to home network architectures, that, "because no two households will own the exact make, model and quantity of ENDs, the superstructure [including controllers] will vary greatly across the demographics of the home user market." He draws attention to the need for robustness and flexibility in the lower layers to accommodate this. [3, p 17]

When manufacturers develop new controllable devices, they might try to design their products to be compatible with as many controllers as possible, so that their products appeal to as large a market of purchasers as possible. Selecting the most popular or widely-deployed control protocol will achieve compatibility with a wide range of controllers, but devices which support more than two control protocols will necessarily be compatible with a still wider range of controllers. This work focuses on how controllable devices and the control server software run on those devices can implement support for multiple control protocols.

2.3 Requirements of home entertainment networks

There are obstacles to the widespread use of home entertainment networks, and demands that must be met in order for them to achieve their full potential.

Scherf identifies some of the challenges facing home entertainment networks: [4, p 6]

1. *a lack of a clear standard:* though he believes that there will be a diversity of network standards employed, he suggests that manufacturers might not be willing to commit to producing products until standards have settled.

2. *muddy business models:* content providers have yet to find successful business models for the distribution of content that take advantage of the opportunities presented by home entertainment networks.

3. *digital rights management:* there is a conflict between content producers, who wish to guard their content from unauthorised duplication, and consumers, who wish to make fair use of the content they purchase. Scherf highlights the critical importance of a digital rights management system, but points out that no clear choice exists, and warns of government intervention which might impose a solution on industry.

The network must provide sufficient bandwidth for high quality streaming media. For instance, DVD discs contain video in MPEG-2 format at bitrates between 4 and 12 Mbps, while HDTV is distributed

over digital cable networks at 19 Mbps. Since multiple streams might need to be delivered to multiple locations in the home, the network must be capable of carrying a number of streams simultaneously. The network must also provide sufficient bandwidth for all other applications that will be used on the network at the same time – for instance, a file transfer between two PCs should not interrupt or degrade video playback on a TV. Since the ear is sensitive to skips in audio playback and the eye to breaks in picture, streaming media must be delivered regularly.

Home networks must be able to connect clusters of devices around the home to create a single, unified network.

Since home entertainment systems are used by all members of the family, all aspects of the system should be easy to use. Networked devices, in particular, should make connecting devices simple and require little configuration in order to solve the cable clutter problem, and should operate reliably.

A network that fulfils these requirements and provides a platform for the focus of this thesis is the IEEE 1394 bus.

2.4 The IEEE 1394 bus

The IEEE 1394 standard defines a high-speed ergonomic peripheral bus, designed to provide low-cost, scalable, high-speed connection between computing devices, referred to as nodes on the bus. A key feature of the bus is its ability to carry both asynchronous data packets and isochronous digital media streams, making it well-suited to multimedia applications. The bus is also known as "FireWire," an Apple Computer trademark, or "i.LINK," Sony's branding which is now in common use in Japan.

The FireWire bus was first developed at Apple starting in 1986. It became an IEEE standard, IEEE 1394-1995 [5] in 1996, and has been amended by IEEE 1394a-2000 [6], including improvements to increase efficiency, and IEEE 1394b-2002 [7], which allows new network media types and longer distances. The IEEE P1394.1 working group is preparing a specification which provides a mechanism for bridging between buses.

The bus is based upon a four-layer model, which is shown in figure 2.1 and which Anderson examines in detail. [8] Below is a summary of aspects of those layers and more recent developments.

Physical layer

The physical layer defines the electrical and mechanical characteristics of the bus. Data is transmitted serially at a range of speeds referred to as S100, S200, S400, S800, S1600 and S3200, named after

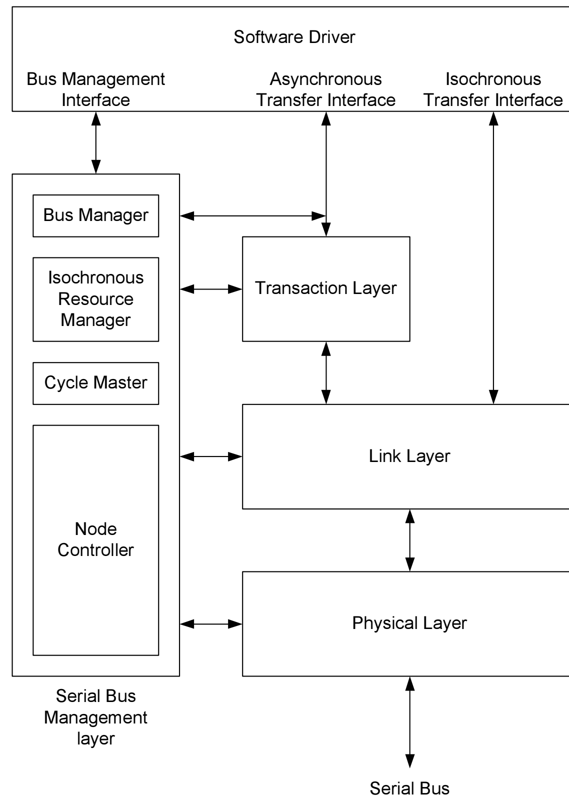

Figure 2.1: IEEE 1394 four-layer model

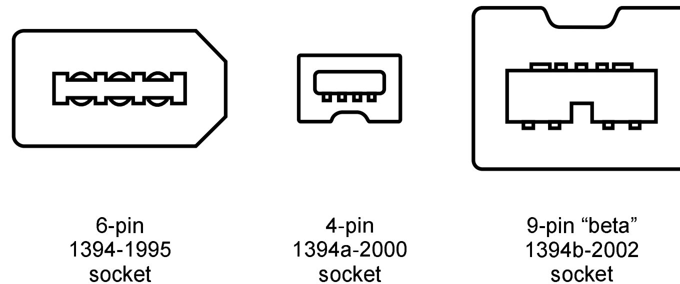

Figure 2.2: IEEE 1394 sockets types

their approximate speeds in megabits per second. IEEE 1394-1995 specified that each node would be connected to another using shielded twisted pair (STP) cables of up to 4.5 m at S100 to S400 speeds.

IEEE 1394b allows the use of a variety of network media to connect nodes, each with different speed capabilities and range limitations. STP can be used at S400 to S3200 up to 4.5 m. Category 5 unshielded twisted pair (UTP) cables can be used at S100 for distances less than 100 m. Multi-mode glass optic fibre can be used at the same speeds and at distances of up to 100 m. Cheaper hard polymer clad fibre and plastic optic fibre allow S200 at up to 100 m and 50 m respectively. [9]

The connectors used on the cables (shown in figure 2.2) are standardised for compatibility and are of a simple design for reliability. Originally a six-contact connector was used on the STP ca-

bling. IEEE 1394a introduced a smaller four-contact connector intended for hand-held devices, and IEEE 1394b-compliant nodes use the nine-contact "beta" connector. UTP cabling uses the same RJ45 connectors used in UTP Ethernet networks, while the fibre optic options use a variety of connectors. The six- and nine-contact connectors incorporate a power supply, which can eliminate the need for separate power supplies for low-current devices. Connections are "hot-pluggable," meaning that the user can safely unplug a node without having to turn it off first.

The topology of the bus is unstructured, so nodes can be connected to each other in any way, such as a tree or a star. No terminators are needed, unlike, for example, the SCSI bus or 10base2 Ethernet. Loops in the bus are forbidden by the IEEE 1394-1995 standard, but loop-detection and elimination facilities in IEEE 1394b-compliant buses automatically disable links which create loops. Nodes of different maximum speeds can be connected to the bus and communication will occur at the highest speed supported by each node. As many as 63 nodes can be connected on a bus, and up to 1023 buses can be connected by bridges.

Since the bus supports peer-to-peer communication between nodes, an arbitration process ensures that only one node attempts to transmit at a time. The physical layers of every node that wishes to transmit compete in the arbitration process, and the winner is able to transmit one packet. Arbitration ensures fair access to the bus for all nodes.

The bus is reset every time a node is connected or disconnected. All nodes discard their bus configuration information and perform tree identification to determine the bus topology and identify a root node. The self-identification process follows, during which unique identification numbers are allocated deterministically to all nodes, and each node notifies the others of its bus capabilities. Thus nodes do not need to be manually configured.

A standard for wireless IEEE 1394 is being finalised by the 1394 Trade Association Wireless Working Group. A protocol adaptation layer has been developed to allow IEEE 1394 applications to use the IEEE 802.15.3 standard for wireless personal-area networks. IEEE 802.15.3 is designed to operate at speeds between 11 and 55 Mbps, and will incorporate Ultra Wideband wireless technologies for speeds between 110 and 480 Mbps. Bridges can be designed to connect a wireless IEEE 802.15.3 network to a wired IEEE 1394 bus, based on the IEEE P1394.1 bridging design. [10]

Link layer

The link layer translates transaction requests from the upper layers into packets on the bus, and decodes the address and channel numbers of asynchronous and isochronous packets received from the bus.

Isochronous transactions offer predictable latency and guaranteed on-time delivery of streaming media, where the rate of transfer is more important than guaranteed delivery. Up to 64 independent isochronous channels can be carried on a bus, each of which can contain logical audio and video streams. A bus cycle starts every 125 μs when the bus cycle master (an isochronous-capable node selected after self identification) sends the cycle-start marker. The link layer of each node transmitting a stream then transmits a block of its stream in each cycle, addressed to the channel number. The link layer of those nodes that receive that channel decode the packet and pass it up to the application. Isochronous transactions are unidirectional, in that the receiving nodes provide no confirmation of delivery.

Asynchronous transactions can occur when isochronous streams are not being transmitted, though at least 20% of each cycle is reserved for asynchronous use. The link layer implements simple subactions as directed by the transaction layer.

Transaction layer

The transaction layer supports asynchronous transactions, which are initiated by a requester and answered by a responder. Asynchronous transactions can be broadcast, or are addressed to a single node given by bus number, node number and memory address within that node. There are three types of asynchronous transactions: reads from a given memory address, writes to a memory address, and locks, used to ensure that an operation is atomic. Transactions consist of two subactions: the request subaction transfers the command and data to the responder; the response subaction returns a status in write transactions, and data in read and lock transactions. Packets are CRC-checked and acknowledged by the recipient, thus delivery can be guaranteed.

Bus management layer

The Bus Management layer implements a variety of services to nodes on the bus. All nodes must be capable of automatic bus configuration, and there are three additional management roles which can be performed: the Cycle Master, the Isochronous Resource Manager and the Bus Manager. Each node may be capable of fulfilling one or more of those roles, depending on its capabilities. Thus, the set of bus management services available on the bus depends on the set of nodes present on the bus. The nodes that will perform those roles are chosen every time the bus is reset, after self identification.

The Cycle Master is responsible for sending cycle start packets to begin each new cycle. This role must be performed by the root node.

The Isochronous Resource Manager handles allocation of isochronous channel numbers and bandwidth to those nodes which wish to send isochronous data.

The Bus Manager provides services for all nodes, such as topology and speed maps of the bus, management of the supply and utilisation of power by nodes on the bus, optimisation of traffic on the bus, and enables the cycle master.

Application protocols

Above the four layers described, the bus is capable of carrying a range of higher level protocols, allowing it to be used in many applications. These protocols include IEC 61883-2 audio, IEC 61883-6 video, TCP/IP networking and SBP2-encapsulated SCSI.

Digital Transmission Content Protection

Digital Transmission Content Protection (DTCP) is a standardised means of protecting commercial digital content being carried on an IEEE 1394 bus from unauthorised copying, interception or modification. It was developed by a group of five electronics companies and has been endorsed by US movie studios.

Each DTCP-protected media stream has Copy Control Information (CCI) which indicates whether it can be copied. Four levels of copy permission are defined: “copy freely” allows the content to be copied at will, “copy once” allows a single copy of the stream, “copy no more” marks copies of copy once content to prevent further duplication, and “copy never” prohibits any copying of the content. The CCI is given in the headers of the isochronous transactions that the content is carried in, and also embedded in the content.

Before a device sends a DTCP stream to a recipient, it verifies the authenticity of that device based on its identification certificate. The Digital Transmission Licensing Administrator (DTLA) is the body which provides those certificates to device manufacturers. The content streams are encrypted using a public-private key encryption scheme so that only devices which are certified secure can read it. The DTLA can issue System Renewability Messages in new devices and content, which can revoke device certificates should the security of the device be compromised. [11]

How IEEE 1394 meets the demands of home entertainment networks

IEEE 1394 has many attributes which make it eminently suitable as the foundation of home entertainment networks, amongst them are the following:

- Wide industry support from consumer electronics, computer hardware and software companies.

- Designed with streaming media in mind, as it provides isochronous transactions which guarantee bandwidth and regular delivery.
- Ability to support all applications by layering other protocols above it, including TCP/IP.
- Abundant bandwidth, particularly since IEEE 1394b has standardised speeds from S800 to S3200.
- Scalable to small embedded devices, by accommodating low-speed devices, allowing minimal implementations without unnecessary functions, and providing power over the same cable.
- Wireless capability using the IEEE 802.15.3 protocol adaptation layer.
- Ability to use a variety of cable types, which provide different lengths between nodes at different costs. Longer range cabling can be used to connect different rooms in the house.
- Ease of use, since the cabling uses simple connector types and the bus allows hot-plug connections, and requires no configuration.
- Ability to protect copyrighted content, while allowing legitimate copying using DTCP.

Figure 2.3 gives a hypothetical example of how an IEEE 1394 bus could be deployed in a home and devices connected to it. The islands identified by the DHWG which would have existed before the IEEE 1394 bus connected them are shaded.

Though one cannot predict which network technology will win in the marketplace, IEEE 1394 is a viable solution and able to meet the requirements of home entertainment networks. This has been confirmed by Scherf, who suggests that network-capable entertainment devices are likely to be connected with IEEE 1394 or Ethernet [4, p 4], and the DHWG, who acknowledge its value in the home entertainment cluster [1, p 7]. This work considers home entertainment devices which are attached to an IEEE 1394 bus.

2.5 Summary

This chapter has introduced home networks and the IEEE 1394 bus.

From today's home entertainment systems, as well as PC and Internet systems, unified home networks will develop to include all devices. Entertainment devices will use the home network to transfer audio and video media and control, in place of the range of connection types currently in use, and are considered to form home entertainment networks.

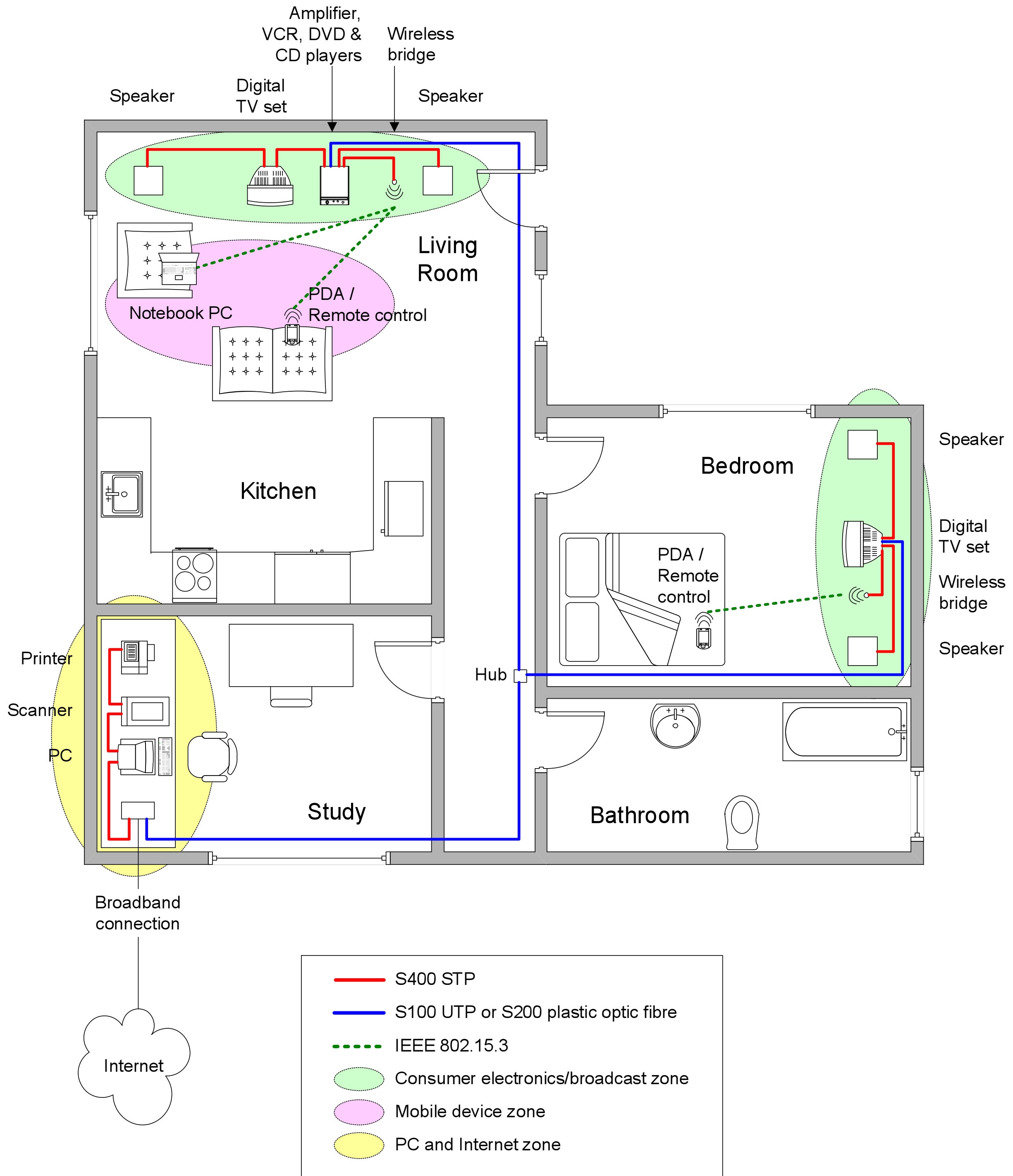

Figure 2.3: A home with an IEEE 1394 network

Home network applications can be classified into data-centric, multimedia-centric, home management and value-added service categories, and entertainment systems fall into the multimedia-centric category. Advantages of attaching entertainment systems to home networks include reduction of the amount of cabling, ability to use digital media, access to other classes of devices and external media, and remote control, which is the focus of the remaining chapters. Networked control involves a controller, running control point software, and a controlled device, running control server software, using a common control protocol for communication. The use of multiple control protocols on a controllable device increases the number of potential purchasers of that device.

Obstacles to the widespread deployment of home entertainment networks, requirements for successful deployment of those networks, and market trends, have been identified.

The IEEE 1394 bus is a high-speed networking technology which offers data rates between 100 Mbps and 3.2 Gbps, and can span distances of up to 100 m between nodes. Bus bandwidth is allocated fairly amongst nodes waiting to transmit. Processes following bus reset identify the tree topology and all nodes on the bus. Applications designed for the wired bus will also be able to be used on wireless IEEE 802.15.3 networks. Isochronous transactions provide a means of broadcasting multimedia streams with guaranteed latency. Asynchronous transactions allow reading and writing from and to memory on a given node. The bus can transport a range of application protocols, and can use DTCP to prevent unauthorised copying. For these reasons, the bus makes an ideal foundation for a home network.

The following chapter provides a general overview of three control protocols.

Chapter 3

Control Protocols

In this chapter, three different protocols which allow for the control of devices across a network are introduced, their underlying conceptual models explained, and their common aspects considered. The Universal Plug and Play, Audio Visual Control and Home Audio/Video Interoperability protocols have been selected for study, as they employ a range of different approaches to control. Other control protocols that could have been examined include Jini from Sun Microsystems, and Apple's Rendezvous zero configuration networking.

3.1 Universal Plug and Play

Universal Plug and Play (UPnP) is a design for making the connection, configuration and control of networked devices easy. It aims to make zero-configuration networking a reality in homes and businesses. UPnP uses a protocol stack based on existing open Internet standards, such as HTTP and XML, and further extending them. This makes UPnP implementation possible on any hardware platform, operating system, programming language and network protocol able to carry TCP/IP. [12]

Origin

UPnP was first conceived of by Microsoft, but currently UPnP standards are developed by the UPnP Forum, a non-profit grouping of more than 600 computer and consumer electronics companies. The Forum is governed by an elected steering committee which casts a final approval vote for all UPnP standards. The Forum is made up of various working committees that are developing UPnP standards in different areas, a technical committee and a compliance task force.

The UPnP Implementers Corporation (UIC) is comprised of UPnP Forum member companies producing UPnP-compliant devices. It designs and executes compliance-testing procedures for UPnP stan-

dards, with the aim of ensuring inter-vendor device compatibility, thus establishing consumer confidence in UPnP-compliant products. The UIC issues the UPnP certification mark for use on compliant devices. [13]

Products

At present, most available UPnP-compliant products are in areas other than home entertainment. The most common UPnP devices are home-to-Internet gateway devices, such as DSL and cable modem routers from manufacturers such as D-Link and Linksys, or Windows XP's Internet Connection Sharing service. UPnP NAT Traversal [14], which works around the disadvantages inherent in IP Network Address Translation in a more convenient manner than any other method, has been key to its popularity in this market.

However, the goals of the standard make it naturally suitable for use in home multimedia networks. Since the UPnP AV architecture has been standardised, BridgeCo has announced a wireless audio adapter development package which allows audio transmission from a PC to a home entertainment system via IEEE 802.11b wireless Ethernet [15]. Such devices might mark UPnP's entry into the multimedia networking market.

Specifications

The Universal Plug and Play Device Architecture [16] defines the device model, the network protocols used by devices and the control points that control them, and the XML documents used during communication and description of devices. It lays out the template language with which specific classes of device are expressed. A variety of Device Control Protocols (DCPs) have been standardised to extend the Device Architecture and allow multi-vendor-compatible control of certain classes of devices, including Internet gateway devices, media servers and renderers, printers and scanners.

Protocol stack

The UPnP Device Architecture specification includes six chapters, which start at network address discovery, and range to device control and presentation. Each chapter is examined in closer detail in chapter 5 of this work, and listed briefly below:

- Chapter 0 concerns the addressing of devices on the network, and allows devices to allocate IP addresses without manual configuration.

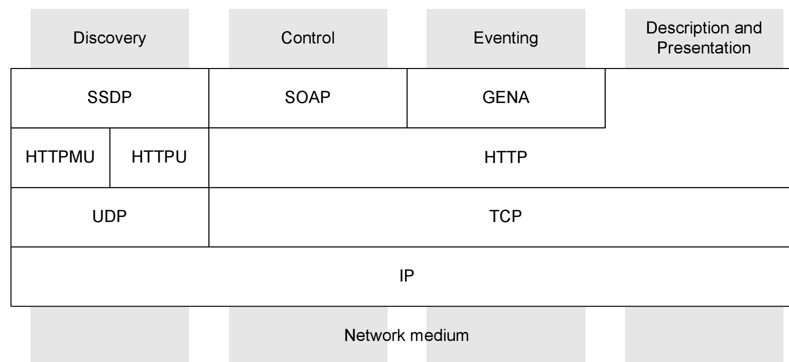

Figure 3.1: UPnP protocol stack

- Chapter 1 describes how devices announce their presence to other devices on the network using the Simple Service Discovery Protocol (SSDP) [17], itself based on the multicast HTTPU and HTTPMU protocols, which are lightweight versions of HTTP for passing simple messages [18].

- Chapter 2 concerns device descriptions, which are marked up in an XML schema and include information about the device, its sub-devices and services. The descriptions are served via HTTP.

- Chapter 3 describes how devices are controlled with the Simple Object Access Protocol (SOAP) [19]. SOAP makes use of XML to format function calls and responses.

- Chapter 4 specifies how events on a UPnP device are reported using the Generic Event Notification Architecture (GENA) [20].

- Chapter 5 details how UPnP devices may provide presentation information to the control point in HTML format via HTTP.

The interactions of the protocols mentioned above are shown together in figure 3.1, where each column represents a UPnP chapter and the protocol stack utilised in that chapter.

Device and service model

Each UPnP-compliant network-attached device is represented as a whole by a root device. A root device may contain a number of embedded devices, which each logically represent a portion of the physical device. Both root and embedded devices can provide a number of services. A service provides for control of the device and notification of state-changing events on the device. The UPnP device publishes an XML description for the root device, and for every service, to allow control points to discover them and make use of their facilities. A diagram of the model of a UPnP device, integrating devices, services and descriptions, is shown in figure 3.2.

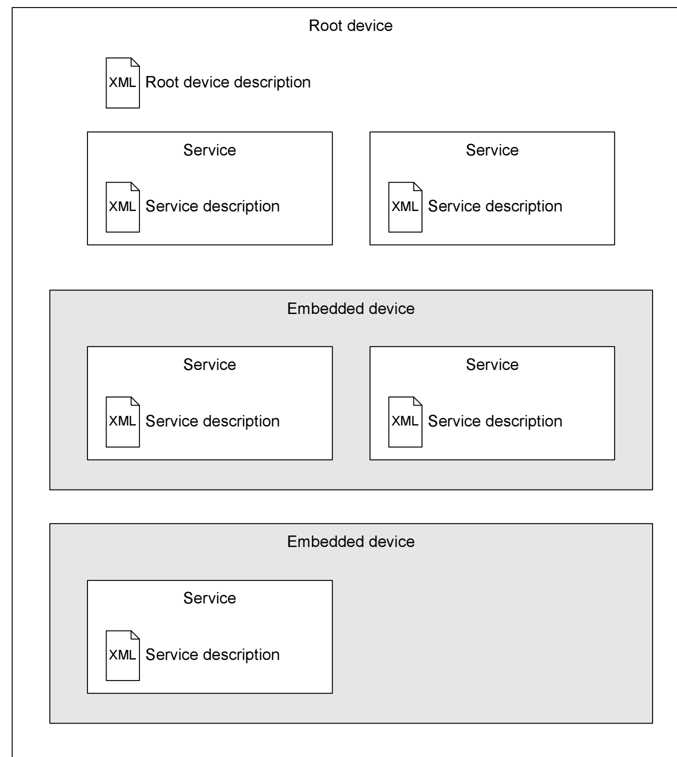

Figure 3.2: UPnP device and service model

Control model

Control of devices is achieved through the services that it offers. The description for each service provides a detailed list of state variables, each of which represents an aspect of the state of the device, and actions which the device can perform with certain parameters. The root device description specifies the URLs through which each service is controlled. Thus control of the device is achieved by using HTTP and SOAP to carry an action request, or state variable value query, to the control URL for that service.

3.2 Audio Visual Control

The Audio Visual Control (AV/C) protocol is designed for the control of consumer electronics equipment via an IEEE 1394 bus.

Origin

The AV/C standards are published by the 1394 Trade Association, a non-profit organisation whose intent is to develop the market for IEEE 1394-compliant products. More than 170 companies (mostly in the consumer and audio-visual electronics industries) are members of the association. Development of the

standards is done in working groups of the association, which are comprised of employees of member companies.

Products

An example of an AV/C-compliant home entertainment product is the Sony LISSA hi-fi system, which consists of CD, MiniDisc and amplifier components connected by 1394 cables. The AV/C audio subunit (explained on page 23) has been implemented in the components, and commands received from the remote control and front panels are passed between the components.

Crest Audio developed a prototype break-out box, the FireBoB, for connecting analogue and digital audio sources to an IEEE 1394 bus in professional studios. The FireBoB was controlled exclusively by AV/C. [21]

BridgeCo provides off-the-shelf AV/C-compliant controller systems for integration into IEEE 1394-attached multimedia devices, making it easy for manufacturers to incorporate AV/C control into their products. [22]

Specifications

The AV/C standards are based on the AV/C Digital Interface Command Set General Specification [23]. The general specification describes how a device is modelled using the AV/C unit and subunit model, covered in detail below; and basic commands which apply to all AV/C units. The specification is itself based on the Function Control Protocol, defined in IEC 61883-1 [24], for the transport of commands and responses across the network, which in turn is based on the IEEE 1394 serial bus standard [5].

The general specification is extended by subunit specifications, which define how certain classes of functionality within a device are modelled and controlled. Examples include the audio subunit [25], which models consumer audio equipment; the panel subunit [26], for modelling a GUI display and the controls on it; and the disc subunit [27], which provides the basis for disc media models.

Unit model

The AV/C standards define a logical model of units, subunits and function blocks, which represent aspects of a physical consumer electronics device that is a node on an IEEE 1394 bus. The model of function blocks within subunits within an AV/C unit and the connections between them is explained below and illustrated in figure 3.3.

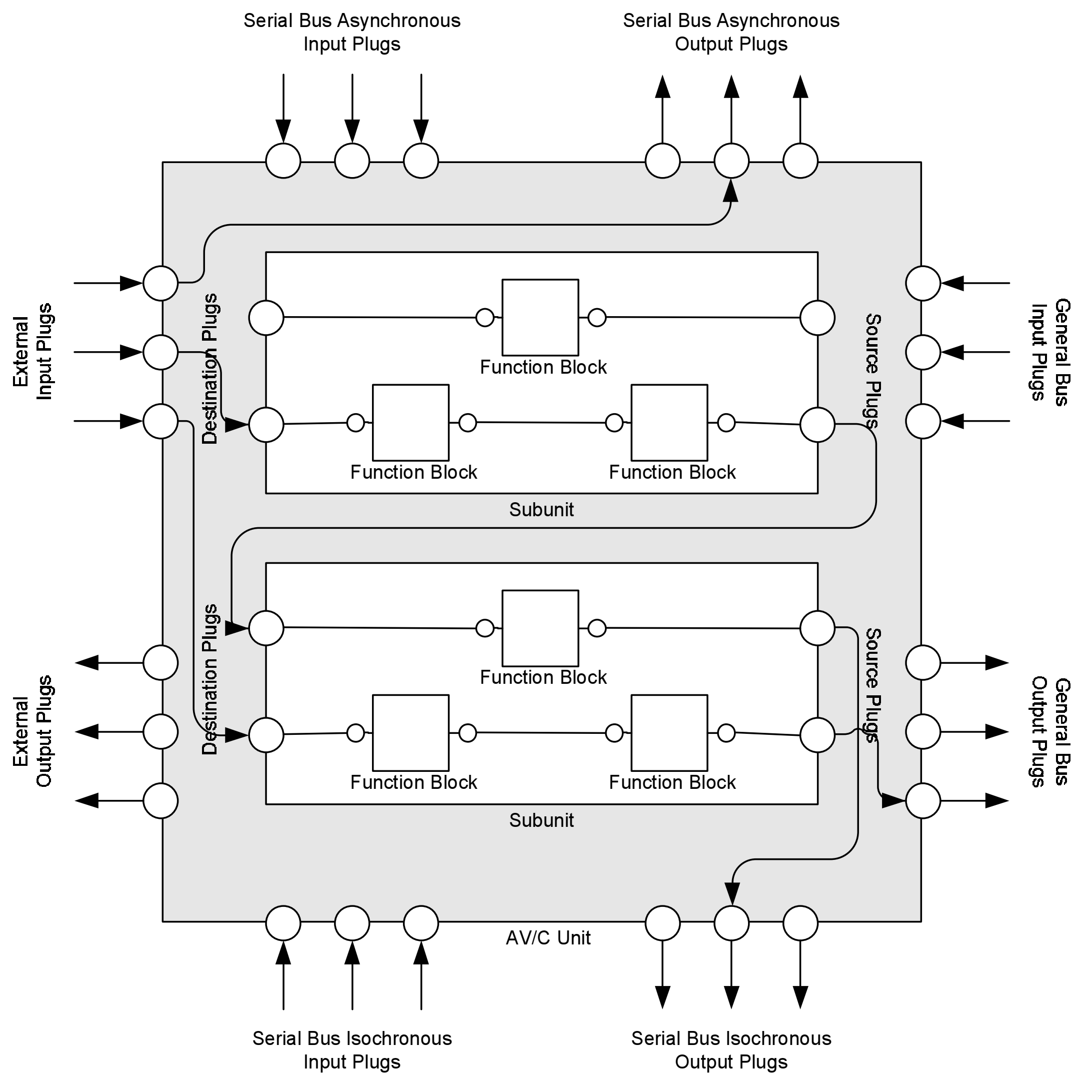

Figure 3.3: AV/C unit, subunit and function block model

Units

The device as a whole is considered to be an AV/C unit. Each unit has a number of plugs which logically represent the point at which an external device can be connected, based on concepts defined in IEC 61883-1. Plugs can be categorised as follows:

- *input or output:* input plugs are where the unit receives an audio or video stream, while the unit transmits streams from output plugs.
- *internal or external:* internal plugs are virtual representations of the end points of a stream carried on a network, and external plugs represent a physical connection point on the unit where a stream is received or transmitted in an analogue or digital format.
  - *isochronous, asynchronous or general bus:* internal plugs are further divided into isochronous plugs, which represent ends of isochronous data flows on an IEEE 1394 bus; asynchronous plugs, for flows of asynchronous transactions on an IEEE 1394 bus; and general bus plugs, for streams carried on buses besides IEEE 1394.

Plugs can be connected to each other within the unit. These connections can be permanent to represent physically wired connections, or non-permanent to logically represent a connection that can be established or broken. Connections may be made from an input plug to a subunit or output plug, or from a subunit to an output plug.

Subunits

AV/C units can optionally contain a number of subunits, each of which conceptually represents an aspect of the device's functions. While there are many subunit types for different aspects of different devices, all subunits receive and transmit streams from/to the rest of the unit through destination and source subunit plugs, respectively. Subunits are entirely contained within the unit, so all subunit plugs are internal and reception and transmission from and to other devices must be through a connection to a unit plug.

Function blocks

Subunits may contain a number of function blocks. Function block types are detailed in the specification for the subunit that they form part of. Function blocks similarly have plugs to connect to other function blocks and subunit plugs. In the case of the audio subunit, function blocks may also contain separate controls which affect the audio passing through the function block in different ways (e.g. volume, balance, etc.).

Command type	Possible response codes
control	not implemented, accepted, rejected, interim
status	not implemented, rejected, in transition, stable
notify	not implemented, rejected, changed, interim
specific inquiry	not implemented, implemented
general inquiry	not implemented, implemented

Table 3.1: Possible response codes for each AV/C command type

Control model

The concepts involved in AV/C control of devices are described below.

The participants in control are the controller, which issues commands, and the target, which receives the commands and returns responses. Thus an AV/C transaction consists of a command and the response to that command. The target of the command is a specific part of the AV/C unit model described above that the command is addressed to. It can be the unit as a whole, a subunit within it, a function block within a subunit, or even an individual control within a function block.

Each command falls into one of five broad command types: *control* directs a unit or subunit to perform an action; *status* queries the current status of the target; *notify* requests notification of a future status change; and *specific inquiry* and *general inquiry* test whether the target supports a given control command, with and without that commands set of parameters respectively. For each of those command types, there is a set of response codes with which the target can respond, given in table 3.1. If the target does not implement the command, it will return a response with the *not implemented* response code; an *accepted* response code indicates that the action was performed; *rejected* indicates that the target does implement the command but is not currently able to perform it; *implemented* confirms an inquiry command; *stable* returns target state when it is not changing while *in transition* is used when the state is currently changing; *changed* is used in responses to notify commands after the state has changed; and *interim* indicates that the target will complete its response at a later time.

Every command contains an opcode, which indicates the specific action to perform or status to query. Certain ranges of opcodes may only be addressed to units, other ranges to subunits only and others to either. Most opcodes can be used in more than one command type.

Thus the possible AV/C commands form a matrix of at least three dimensions, with target, command type, and opcode axes. Other dimensions exist in, for instance, the case of commands addressed to function blocks within the audio subunit, which have a control attribute that indicates an aspect of the control to query or alter.

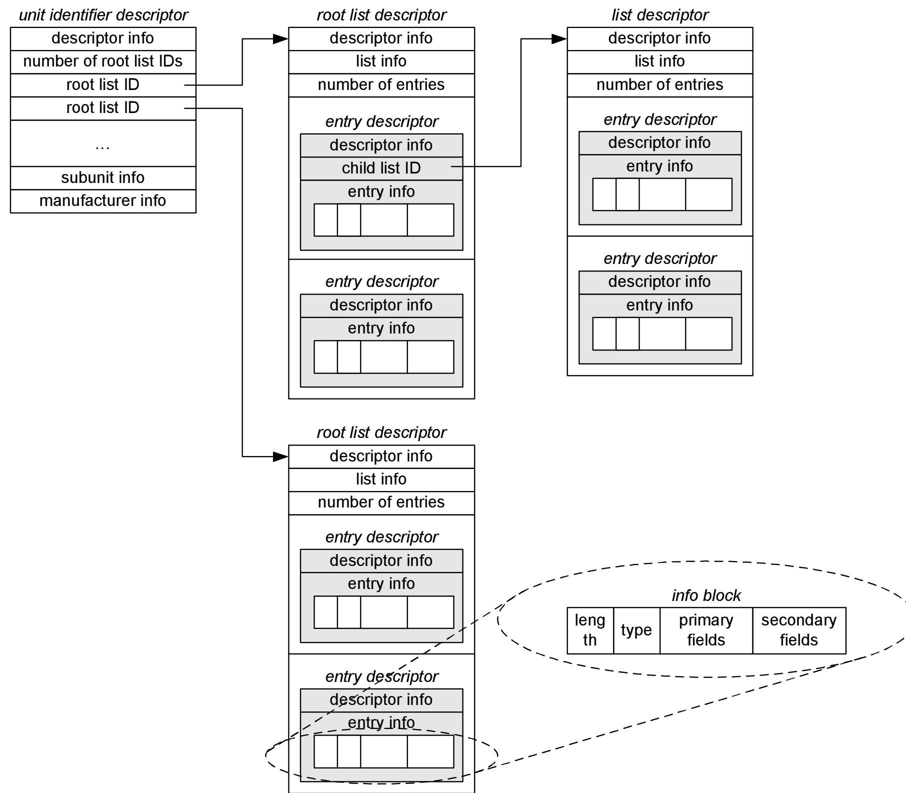

Figure 3.4: AV/C descriptor hierarchy, adapted from [28]

Descriptor mechanism

The AV/C Descriptor Mechanism Specification [28] specifies how information describing aspects of an AV/C unit can be represented and stored in the unit and retrieved by controllers. It aims to reduce the amount of knowledge required by the controller to discover and control a device, by describing the details of the device, thus allowing flexible controllers to automatically configure themselves to control a variety of AV/C devices based on the descriptor information each provides. Implementation of the descriptor mechanism, though, is optional.

Descriptors are structured data that represent the features of the device. The data structures form a hierarchy consisting of a unit identifier descriptor, which contains information regarding the unit and the structure of its descriptors; list descriptors, which contain a number of entry descriptors; and entry descriptors, which contain an item of stored data (an info block) and/or a reference to a list descriptor. The contents of the info blocks that describe each aspect of the AV/C unit are defined in the relevant AV/C unit or subunit specification. An example of how these elements can form the descriptor hierarchy is shown in figure 3.4.

The descriptor mechanism specification defines opcodes which can be used to search and navigate through the descriptor hierarchy, as well as read and write descriptors and info blocks.

3.3 Home Audio/Video Interoperability

The Home Audio/Video Interoperability (HAVi) architecture is designed to allow audio-visual devices in an IEEE 1394 home network to co-operate, in order to allow the development of applications for the home network. The open HAVi standard means applications can include devices from many manufacturers. The architecture makes use of the Java language's cross-platform portability to ensure platform neutrality and compatibility with future devices. A set of compliance levels allows the architecture to scale across a wide variety of devices.

Origin

The HAVi specification was developed by a group of eight major consumer electronics companies. The architecture is promoted by the HAVi Organisation, a non-profit organisation which now includes a number of companies besides the founders.

Products

Mitsubishi Digital Electronics, now a member of the HAVi Organisation, produces a range of HDTV products which are HAVi-compliant, using the "NetCommand" branding for their HAVi implementation. The range includes high-end rear projection TVs, such as the WS-65813 [29], which are able to receive MPEG-2 video streams via the IEEE 1394 bus and are HAVi Full AV Devices (explained on page 29). An HDTV digital video recorder, the HS-HD2000U [30], is also capable of playing and recording MPEG-2 streams via its IEEE 1394 interface, and is a HAVi Base AV device. When attached to the bus, the video recorder will appear as an icon shown on the TV's display, its interface can be presented on the TV and it can be controlled using the TV's infra-red remote control. [31]

Specifications

The HAVi Specification [32] includes all specifications for the HAVi Architecture, including the details of the software elements and the various functional control modules, as well as the APIs that they provide. Functional control modules are defined for tuners, video recorders, clocks, cameras, AV discs, amplifiers, displays, AV displays, modems and web proxies.

Device model

The conceptual device model that HAVi uses is composed of devices and functional components. Each physical unit is represented as a device, while different aspects of the device that can be controlled are

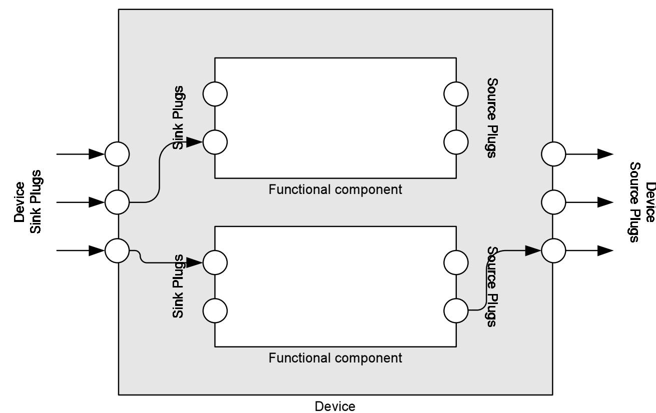

Figure 3.5: HAVi device and functional component model

represented as functional components. Both devices and functional components use plugs (based on IEC 61883-1 concepts) to represent the end points of connections, where sink plugs receive input and source plugs provide output connections. The model is illustrated in figure 3.5.

Software model

The HAVi architecture introduces a model of software objects, each of which provides a service, that interact with each other to provide HAVi's control applications. The objects are termed software elements and are identified by unique software element identifiers. Software elements register themselves with the Registry, itself a software element. They communicate with each other by passing messages via the Messaging System software element, which acts as a hub for the messages. The message passing system is able to deliver intra- and inter-device messages, thus software elements need not be aware of whether the elements with which they communicate are local or remote.

Software elements

A number of software element types are defined:

- The *1394 Communication Media Manager* provides access to the IEEE 1394 bus facilities, including asynchronous transactions and isochronous streams, and notification of bus topology changes. Other Communication Media Managers could provide support for other network types, though none have been developed yet.

- The *Messaging System* allocates software element identifiers and passes messages between software elements. Messages are carried between devices using an FCP Control Transaction Set, and delivered to software elements via a callback function.

- The *Registry* provides a distributed directory service for software elements to locate other software elements by ID or by type, whether on the local device or a remote device, and retrieve their capabilities and properties.

- The *Event Manager* delivers notifications of events. It receives event notifications from software elements and delivers them to other software elements which have subscribed for notifications of certain types. Delivery is by calling a function registered during subscription.

- The *Stream Manager* provides abstractions for handling media streaming between functional components. Connections can be made using IEC 61883 connections on the IEEE 1394 bus, internal connections within a device, or external cabling.

- *Device Control Modules* (DCMs) are the software interfaces through which devices are controlled. *Functional Component Modules* (FCMs) control the functional components within devices. The DCM classes are stored in DCM code units which can either be embedded into a controller or loaded from the controlled device itself. A Havlet can be contained within the DCM code unit. Like a Java applet running within a web browser, a Havlet is a Java program stored in a Java archive that can be executed on a controlling device and displayed on its user interface to allow the user to control the device.

- The *DCM Manager* handles loading and unloading DCM code units in devices which control DCMs.

- The *Resource Manager* manages resource and FCM allocation and sharing and provides a scheduling service.

- The *Data Driven Interaction (DDI) Controller* displays DDI user interface components on a display and handles user interaction with those components.

- Control applications are classed as *Application Module* software elements and also register with the Registry.

The software elements within a device are described by the Self Describing Device (SDD) data stored in the IEEE 1394 configuration ROM. The software elements are depicted in figure 3.6.

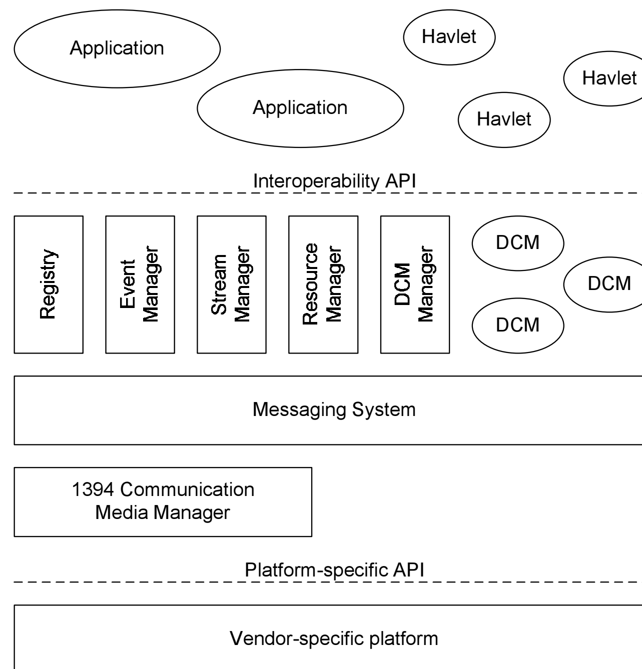

Figure 3.6: HAVi software elements, taken from [32]

HAVi-compliant devices are classed into four levels of compliance. Devices of different computing capabilities can comply at the most suitable level, allowing the HAVi architecture to scale across a range of devices. The classes are as follows:

- *Full AV Devices* (FAVs) are devices which have a user interface and can be used to control other HAVi devices. They include Application Modules and a Java runtime environment, in order to run Havlets, and every type of software element to support them. They may contain embedded DCM code units for controlling certain devices.

- *Intermediate AV Device*s (IAVs) are also capable of controlling other devices, though they lack the Java runtime environment. The Application Module, DDI Controller, and Resource, Stream and DCM Manager software elements are also optional. These lighten the computing demands HAVi places on the device, but limits IAV devices to controlling those types of devices for which they have an embedded DCM code unit.

- *Base AV Devices* (BAVs) are not capable of controlling other devices, and need not contain any software elements, though they must contain a DCM and SDD data so that an FAV or IAV can control them. Since HAVi's Messaging System is optional, communication from controller to BAV can be via HAVi messages or another means.

- *Legacy AV Device* (LAV) are devices which are not HAVi-compliant themselves, but are capable of being controlled via some means. A HAVi-compliant bridge can provide a DCM and translate control communication to and from HAVi messages on the LAV's behalf.

User interfaces

HAVi devices can be controlled by other devices executing DCM API functions, or by the user through a user interface. Two types of user interface (UI) display and control are supported by HAVi.

Level 1 UI is the simpler of the standards, since controllers act as dumb terminals for a user interface provided by the target device using the DDI protocol. The targets store a description of their user interface, composed of DDI elements which represent user interface controls, in the SDD data. Controllers load the DDI elements through the DCM, render them on their displays and allow user interaction with the controls. User manipulations of the controls are returned to the DCM, and the DCM may update the state of the controls when status-changing events occur using the DDI protocol.

The Level 2 UI standard uses Java applications running on an FAV to create the user interface. Java's Abstract Window Toolkit (AWT) user interface toolkit is used for the display of user interface controls. AWT allows the interface to adapt to the characteristics of the device it is displayed on, and provides a wide range of controls with which the interface can be developed.

3.4 Common aspects

While UPnP, AV/C and HAVi were developed by different groups, employ a range of technologies and have different strengths and weaknesses, all are able to control home entertainment devices, and thus all have certain common aspects. The approaches taken to these various issues are compared below.

Architectural details

- *Origins.* The protocols are all developed by industry alliances of companies. All groups recognise that achieving widespread acceptance of their respective standards in the market will require multi-vendor backing and a wide range of available and compatible products.

- *Openness of standards.* All protocols are standardised in specifications, but the ease with which those standards can be acquired varies. The UPnP specifications are the most open, recognising the role that the freely available standards of the Internet Engineering Task Force (IETF) played in the growth of the Internet. The UPnP architecture is comprised of open Internet standards freely available from bodies such as the IETF and World Wide Web Consortium (W3C), and extensions

to those standards. Those extensions have been published as draft standards. The UPnP Device Architecture itself is published on the UPnP web site. The HAVi Specification is available on the HAVi Organisation's web site, but for evaluation purposes only – implementors must license certain patents from the authors of the specification. AV/C documents are only available to members of the 1394 Trade Association.

- *Extensibility.* The protocols examined all have specifications for certain classes of device functionality, namely UPnP's DCPs, AV/C's subunits and HAVi's functional components. In future, specifications can be added for types of devices not yet imagined, allowing the standards to adapt and grow without risking obsolescence. In addition, the three protocols can be extended in vendor-specific ways to allow vendors to add complete support for all features of their products. UPnP allows non-standardised devices and services, AV/C commands can use the "vendor-dependent" opcode and a company identifier for any nature of control, and HAVi devices may implement non-standard DCMs.

- *Scalability.* Since embedded devices have a wide range of computing capabilities, the protocols can be scaled to match those capabilities. HAVi explicitly defines a set of compliance levels, though variation within those levels is possible. UPnP devices can omit control, eventing and/or presentation services by not listing URLs for those services in their device descriptions. At a minimum, an IP stack, SSDP and an HTTP server must be implemented. Designers of AV/C devices can choose whether or not to support each individual command, and controllers are able to check whether they are implemented using the General Inquiry and Specific Inquiry command types.

- *Modelling of devices and functionality.* Each protocol is based upon a conceptual model of a networked, controllable device, but those models are similar in some ways. All consider the device as a whole (a "device" or a "unit") and aspects of functionality contained within. Only UPnP allows devices nested within other devices as "embedded devices" within a "root device". AV/C and HAVi incorporate the IEC 61883-1 plug model, which could not have been used in UPnP since it is based on the IEEE 1394 bus.

- *Driverless control.* The protocols all have a means of allowing a device to be controlled by a controller without the need to manually install driver software on the controller. Driver installation on PCs is a source of frustration for consumers and not likely to be accepted in the home enter-

tainment market. In all cases, a means is provided to allow the controlled device to project a user interface on the controller's display.

- *Message transport.* Controllers and controlled devices must communicate for a variety of purposes, such as control and event notification. In the case of UPnP, most of the chapters use a different group of protocols for communication, as shown in figure 3.1. In AV/C, all communication uses the AV/C FCP Control Transaction Set and the basic fields in an AV/C frame. HAVi uses the Messaging Service software element to provide a general message passing system which can deliver messages by calling functions in the software element that they are destined for.

Implementation details

- *Service discovery.* All protocols include a means for automatic discovery of devices and services on the network. Chapter 1 of the UPnP specification uses announcements regularly broadcast across the network, while AV/C and HAVi make use of the IEEE 1394 bus' ability to enumerate nodes on the bus during the self-identification process and the configuration ROM to identify nodes which implement either standard. The UPnP method makes less efficient use of network bandwidth, but is portable to network media other than IEEE 1394.

- *Description.* Once controllable devices have been discovered, controllers learn the characteristics of the devices before controlling them. UPnP uses XML documents made available using HTTP, AV/C uses its descriptor mechanism and HAVi stores a description in the SDD data in the configuration ROM.

- *Control and status queries.* The three protocols provide methods for a controller to invoke a command on the device and query status of aspects of the device. UPnP uses SOAP over HTTP to deliver XML-formatted requests and the return values and queried values. Requests are delivered to a URL given for each UPnP service. AV/C uses an FCP Command Transaction Set for control and status commands, each addressed to a specific part of the unit model. HAVi control is through the general message passing system, which delivers messages via callback functions to a software element.

- *Status change notification.* Notification of events that occur on a device and change its status can be relayed to interested controllers. The protocols under investigation all use a publisher/subscriber model. UPnP uses GENA XML-formatted notifications carried over HTTP. The subscriber registers a callback URL and the duration of the subscription with the publisher, which delivers noti-

fications as they occur. AV/C provides a mechanism where a status notification can be triggered as soon as the state changes, though subscriptions only last for a single state change and must be renewed each time. HAVi software elements can register a callback function with the Event Manager software element to be called when state changes.

- *User interface presentation.* All three protocols provide means for controlled devices to display a graphical user interface on a controller's display, without pre-loaded software on the controller. UPnP uses HTML rendered in a web browser, which takes advantage of the ubiquity of the web browser, but this can limit the complexity of the interface. AV/C's Panel subunit allows flexible user interfaces using a set of standard interface controls. The HAVi Level 1 user interface is similar and based on the DDI elements, but the Level 2 user interface provides the most flexibility by uploading a complete control program to the controller.

Chapters 5 and 6 describe the implementation of UPnP and AV/C respectively. Only two protocols were needed for the purpose of developing a prototype multiprotocol device, and UPnP and AV/C were selected ahead of HAVi as they both seem to be more actively developed (judged by the number of new UPnP DCPs published and revisions to the AV/C standards) by their standards bodies than HAVi.

3.5 Summary

This chapter has described and compared three control protocols.

UPnP is an HTTP- and XML-based standard designed for use in a variety of network types by the UPnP Forum. The UPnP Device Architecture defines the protocol and model used in chapters covering Addressing, Discovery, Description, Control, Eventing and Presentation. Control and eventing are achieved by means of HTTP requests.

AV/C is an IEEE 1394-specific standard for control of consumer electronics from the 1394 Trade Association. A general specification describes AV/C devices and control, and is extended by function-specific subunit specifications. Control commands are addressed to a specific portion of the AV/C unit. They have a control type and an opcode to indicate the action, and their responses have a response type which corresponds to the control type. Descriptors can provide a hierarchy of information regarding the device.

HAVi is a standard for allowing devices connected to an IEEE 1394 home network devices to interoperate. The HAVi Specification includes all the details of the HAVi model and the family of software elements that make up a HAVi implementation. Different classes of AV device implement the HAVi

standard at different levels of completeness. Two methods are provided for remote display of a user interface, including running a complete Java application on the controller.

Although the above protocols have different backgrounds, they are all capable of controlling devices on a home entertainment network, and thus have common aspects. These include the bases upon which they are standardised, their extensibility, scalability, conceptual device models, the ability to control without drivers, as well as the manner in which they allow remote control of devices.

The following chapter covers an example of a home entertainment device and how it can be controlled.

Chapter 4

An Example Home Entertainment Device

This chapter introduces the Yamaha RX-V1000, a typical home entertainment device, which was used in the development of a prototype network-controlled home entertainment device. The features and remote control capabilities of the product are presented, its serial control protocol is described and its set of state variables determined. A control system to control it from a PC is examined.

4.1 The Yamaha RX-V1000

The RX-V1000, pictured in figure 4.1, was selected as a device for use in constructing a prototype home network-attached device. It is a high-quality audio-visual receiver (AVR) capable of receiving audio and video input from a range of devices (such as a CD player, a DVD player, or the built-in FM/AM tuner) in a variety of analogue and digital formats, decoding surround sound encodings (such as Dolby Pro Logic or DTS) to produce multi-channel sound, and passing the selected audio and video to output devices (such as speakers and a TV set). It is intended to form the centre of a home entertainment system.

The RX-V1000 can be used in two-room installations, where one program is presented on the main zone speakers, and another on the zone 2 speakers. To allow users to control it from zone 2, a remote control system (such as the Crestron SmarTouch [33]) can be connected to an RS-232 serial port on the RX-V1000. The serial port allows the remote control system to send commands of the same sort as those sent by the AVR's ordinary IR remote controller, as well as receive status notifications from the AVR. Thus the serial port makes an ideal interface through which a prototype network control system can communicate with the device. The device can be linked to other Yamaha devices using a pair of in and out remote control 3.5 mm jack connections, however, the details of the proprietary communication protocol used on those connections were unavailable, so development was focused on control using the RS-232 port.

Figure 4.1: The Yamaha RX-V1000 AV Receiver

4.2 RX-V1000 serial control protocol

The documentation of the RX-V1000's serial control protocol [34] describes how communication with the device is established, how a controller can control the AVR, and how the AVR reports changes in its state.

The controller sends the Ready command to the AVR to initiate communications. The AVR responds with the Configuration command, which includes a structure which contains the current values of all settings on the AVR, giving the controller a complete picture of the current state of the device. The Reset command allows the controller to return all those settings on the AVR, which can be controlled through the serial control protocol, to their factory defaults.

There are two types of control commands that the controller can send to the AVR. System commands have a command type and a parameter. There are system commands for the controller to set the main volume and zone 2 volume, enable or disable and specify the frequency of report commands, and read and set the text displayed on the RX-V1000's LCD display. Operation commands direct the AVR to perform the action indicated by a command code. The set of commands matches the set which the infrared remote control can send. Amongst many others, there are operation commands to select input from the CD player in the main zone, set the sound processor to produce sound that matches the acoustics of a jazz club, increase the main volume by one step, and start the tuner seeking for the next radio station.

The AVR sends Report commands to indicate that an aspect of the device's state has changed. System state reports concern the system as a whole. There are system reports which indicate that the device is shutting down, or is overheating. Playback state reports are sent when the internal playback state changes such as when the surround sound format changes, the frequency of a digital audio input changes, or when the tuner finds a station or goes out of tune. Operation reports inform the controller of a change in state caused by the user pressing a button on the front panel or IR remote, or a command received from the

serial port. For example, selecting a different input or DSP program or turning the main volume knob will be reported to a controller on the serial port in operation reports.

The serial protocol provides no means for a controller to query the state of an aspect of the AVR, except by sending a Ready command which the AVR will answer with a Configuration command including the values of all aspects of system state.

It is not necessarily easy to find the operation report command that corresponds to an operation command. This is as a result of the slow speed and full-duplex communication of the serial interface: the RX-V1000 might be transmitting a sequence of other report commands as the operation command is sent to it. The resultant report command would be queued in the AVR's buffer until the preceding commands had been transmitted. The controller can attempt to match the report to the operation command by checking the source field and report type in incoming report commands, but that could match the report from a previous command. This problem could be solved in the same way as the Domain Name System (DNS) protocol matches responses to queries, which are sent asynchronously using UDP datagrams. DNS resolvers tag each query that they send with a different identifier number, which the DNS server copies into its response to that query. [35]

4.3 A state variable model of the RX-V1000

An examination of the serial control protocol documentation and user manual [36] allowed a model of the state variables of the device to be distilled. By correlating system and operation commands with the corresponding report commands and elements of the configuration structure, a list of 36 controllable aspects was found. Each of those aspects was considered as a variable and given a name and description. A data type for each variable was determined based on the possible values that the variable could take. Data types assigned were either string, integer, boolean or enumerated sets. Those variables whose values could be found in the configuration structure or in a report command were considered readable, and those that could be set using a system or operation command considered writable.

The results of this analysis are found in appendix A. As an example, the main volume setting for the main zone is represented by the "volume" variable. The variable is readable since the current value is given in the configuration command, and updated by report commands. It can be changed with a system command, thus it is also writable. The volume setting is represented as an integer containing the values used in the configuration and report commands.

Some aspects of the RX-V1000 cannot be controlled or monitored from the serial port so they have not been included in the model. Examples include the treble and bass controls or defining preset stations

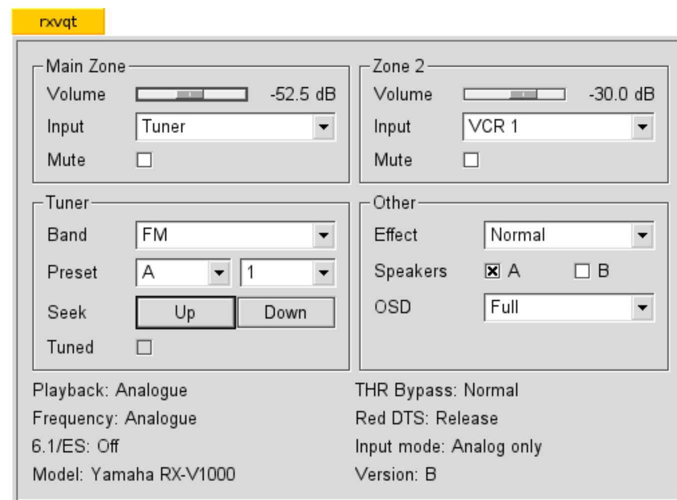

Figure 4.2: *rxvqt* user interface

on the tuner, which can only be set on the front panel of the AVR. Future devices which are designed to be controlled across a home entertainment network should allow control of as many aspects as possible remotely.

The state variable model allows one to consider the functionality of the AVR more clearly than can be seen from the list of commands in the RS-232 protocol specification. The list of variables represents the functionality in a simple form before one attempts to apply the conceptual model of a control protocol to the device.

4.4 Controlling the RX-V1000

An application which controls the AVR was developed for purposes of testing RS-232 control and designing a user interface to represent the AVR. *rxvqt* runs on a PC connected to the RX-V1000's RS-232 port. Its user interface (shown in figure 4.2) shows the current state of the AVR and allows the user to control it.

rxvqt is written in C++ and uses the Qt 3 toolkit. Qt provides cross-platform GUI and application support classes [37], so *rxvqt* has been used on FreeBSD and Linux systems and could be compiled for use on other platforms, including Windows and Mac OS. A thread receives report commands from the AVR through the serial port. It uses functions from *librxv* (see section 7.5) to parse the RX-V1000 RS-232 format, then updates the user interface controls to reflect the current state of the device. *rxvqt*'s communication with the AVR is normally through the serial port, but can also be through the program's standard input and output file descriptors (*stdin* and *stdout*) so that it can be used with the *splitrxv* and

simrxv virtual RX-V1000 simulators introduced in sections 7.5 and 7.6. When the user alters controls on the user interface, *rxvqt* sends the appropriate command to the AVR to effect that change on the device.

4.5 Summary

The Yamaha RX-V1000 AV receiver is an example of a device commonly found in home entertainment systems. It provides an RS-232 serial port which allows remote control of the AVR, and makes it well suited for the development of a prototype network controlled device. The serial control protocol includes control commands which allow controllers to change the state of the device, and report commands which indicate the current state. A state variable list has been drawn up to represent the controllable functions of the device in a simple manner. The *rxvqt* application allows a PC to control the AVR.

The following chapter examines UPnP in detail, based on experience gained during the development of a UPnP control server to control the RX-V1000.

Chapter 5

Implementing Universal Plug and Play

A working implementation of the UPnP Device Architecture version 1.0 specification [16] was created to allow a UPnP control point to control the Yamaha AVR. An overview of the system is presented, and the details of each of the six chapters of the UPnP specification and the implementation of each chapter are described in this chapter.

5.1 Initial feasibility study

Development of the UPnP protocol stack began on a Digital Harmony DHIVA embedded development system. The Digital Harmony Interface for Video and Audio (DHIVA) was designed as a transceiver to convert between digital media streams on an IEEE 1394 bus and older analogue and digital formats, in a package small enough to be installed in a host device [38]. The main circuit board contained an embedded computer with an ARM7TDMI CPU running at 25 MHz, 1 MB of RAM, 2 MB of Flash memory and had interfaces to connect to daughterboards, the CPU of the host device and a debugger. Modular daughterboards available for the DHIVA included a three-port IEEE 1394 physical layer interface board and an audio interface board with a variety of audio and MIDI inputs and outputs.

The DHIVA ran the ThreadX embedded operating system augmented by Digital Harmony's DHIVA build environment which provided support for the IEEE 1394 and audio hardware. Software images for the DHIVA were built on a PC using the Green Hills Software Optimizing C Compiler, which compiled C and C++ to ARM7 machine code, optimising the size of the executable by making use of the ARM CPU's reduced-size Thumb instruction set. These images were then written to the DHIVA's Flash memory and booted from there, or loaded directly into memory from the PC using an in-circuit emulator (ICE). Combined with the ICE, the Green Hills MULTI development environment allowed real-time debugging of software running on the DHIVA from the PC.

Using the DHIVA software environment and an RFC 2734 [39] based IP-over-1394 TCP/IP stack [40], a simple HTTP/1.0 [41] web server was written. However, performance of the web server was poor: it took the server a number of seconds to serve each page or graphic. This was partly attributable to the slow speed of the DHIVA's 25 MHz ARM7TDMI CPU, and partly to the software layers beneath the web server. Around the same time, Digital Harmony ceased operations, so development was moved from the DHIVA platform.

The development of the web server on the DHIVA provided insight into development on embedded systems. Based on the characteristics of embedded systems that Bentham describes [42, p vi], desired attributes of an embedded computer in a home entertainment system were identified:

1. *Fast performance.* Embedded systems use slower CPUs than other computers since they cost less and generate less heat and consume less power. At the same time, users expect rapid response. Thus software should be written to make efficient use of the CPU.

2. *Fast start up.* Other home entertainment devices turn on immediately, thus users will not be satisfied with new devices which take a long time to boot.

3. *Reliability.* Embedded systems can run continuously for substantial lengths of time, and should remain reliable. Entertainment programs should not be interrupted by software failure.

4. *Low memory usage.* Embedded systems have limited RAM to keep costs low, particularly on systems where the RAM is included on the same die as the CPU. Dynamic allocation of memory must be carefully considered, because once memory is exhausted there is no swapfile on secondary storage to resort to.

5. *Portability.* A wide variety of processors and operating systems are used in embedded systems. In order for an implementation to be of most use, it should be portable to different platforms. Thus the software should avoid use of features that are specific to a few platforms and follow commonly-used standards, such as the ANSI C standard.

5.2 Design of the implementation

Further development of the UPnP implementation was done on a PC running Red Hat Linux, bearing embedded development concerns in mind to allow for a possible future port to an embedded platform. One such platform considered was the eCos embedded operating system, which provides programmers with an API similar to that of Linux and UNIX systems, though reduced in size [43]. A port from

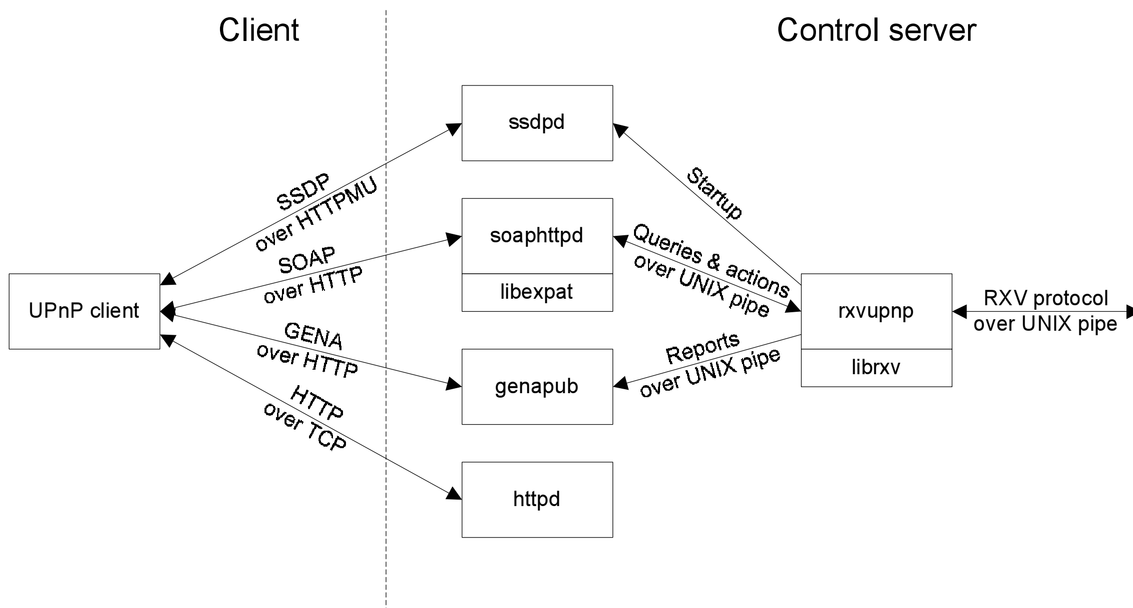

Figure 5.1: Components of the UPnP control server and its interactions with a client

the Linux platform to eCos would require some changes to the implementation (e.g., the POSIX thread library used by the *genapub* program described in section 5.8 is not available), but should be possible. On an embedded computer with a larger main memory, an operating system based on the Linux or NetBSD kernel could be used to provide the complete set of APIs used by the UPnP implementation, making porting very simple.

The complete UPnP stack was implemented in a number of separate programs which run concurrently, each providing a separate part of the UPnP protocol stack. The programs communicate amongst each other via UNIX pipes. As separate programs, the functions of the programs are clearly specified and the interfaces between them well-defined. This design allows transactions of different protocols in the UPnP stack (such as GENA and SSDP) to be served simultaneously by the device, and has allowed the components to be tested individually or replaced in a modular way. If the system were to be ported to a very light-weight operating system like ThreadX, the programs would need to be implemented as threads within a single application.

The components of the implementation are illustrated in figure 5.1, a more detailed representation of the UPnP control server and client shown in figure 1.1. Each block represents a separate program, with rectangles beneath indicating the libraries linked to it, and lines between blocks indicating a communications channel. The four programs shown interacting with the client correspond to the four columns of the UPnP protocol stack in figure 3.1.

5.3 The chapters of the UPnP Device Architecture

Each chapter of the UPnP Device Architecture and its implementation is examined below by considering a set of questions. While not all questions apply to each chapter, the headings are used as a framework.

Under the heading "Analysis of the specification", the details of the chapter, its objectives and intentions are considered in a general context and compared with similar protocols. "Application in an embedded system" sections discuss issues concerning the implementation of the chapter in an embedded computer based on experience gained during the feasibility study. "Application on an IEEE 1394 bus" sections consider issues specific to UPnP on an IEEE 1394 bus rather than any other network type. "Design of the implementation" examines the data and program structures of the program which implements the chapter. "Performance of the implementation" details the results of testing the implementation. "Possible improvements to the implementation" sections list ways in which the implementation could be enhanced for better compliance with the UPnP Device Architecture, improved performance or maintainability. "Possible improvements to the specification" section speculate on how future revisions of UPnP could improve on the current standard and address problems identified during the implementation.

5.4 UPnP chapter 0: Addressing

Analysis of the specification

All network protocols used within the UPnP protocol stack are based on IP protocols, so each UPnP-compliant device's first task is to acquire an IP address with which to communicate with other hosts on the network. In order to achieve ease-of-use, the UPnP Device Architecture requires that devices attempt to automatically acquire an IP address by DHCP (Dynamic Host Configuration Protocol) [44] or Auto-IP [45], though there is no reason why devices should not be configured manually on a managed network. Devices should first attempt to lease an IP address from a DHCP server, since DHCP is widely-deployed in managed and unmanaged networks to centralise the assignment of IP configuration to network devices.

In the absence of a DHCP server, the Auto-IP method is used to select an address from the link-local IP range. A device picks an address at random from 169.254.0.1 to 169.254.255.254 then uses the Address Resolution Protocol (ARP) [46] to query whether any other device on the network is already using that address. Should the address be in use, another is selected and tested again. Once an unused IP address is found, the device uses it and will answer future ARP queries to defend its claim to that address. Since addresses in the 169.254.0.0/16 range are intended to be confined to the local network

media, the Auto-IP standard prohibits IP routers from forwarding packets with a source address in that range to the Internet or other connected networks.

The behaviour specified by chapter 0 is the same as that of modern versions of Microsoft Windows without manual IP settings, so it is already accepted on many managed networks, while still being able to work correctly on an unmanaged network.

Application in an embedded system

Both DHCP and Auto-IP are relatively simple protocols which should not require much code to implement.

Users expect embedded devices to start working as soon as they are turned on, but the methods specified in the chapter can take some time to acquire an IP address. Section 4.1 of the DHCP specification suggests that clients retransmit unanswered DHCPDISCOVER messages at increasing intervals from approximately 4 to 64 seconds [44]. This sequence of retransmissions would take roughly 128 seconds to complete — an unacceptably long delay for an embedded system. The DHCP client used in an embedded UPnP implementation should therefore be configured to retransmit fewer times, or at more frequent intervals, to reduce the time before falling back to Auto-IP.

The Auto-IP test for whether the selected IP address is already in use also requires a far shorter delay. Section 2.2.1 of the Auto-IP draft standard describes a probing process that involves four delays of one to two seconds, implying an average probe length of six seconds. However, section 2.3 suggests that on reliable network media, which can indicate when the link is up, the delay could be reduced to range from 0 to 200 milliseconds, thus reducing the probe length to an average of 400 milliseconds [45]. As the IEEE 1394 bus does provide reliable transmission of packets, and nodes are aware of when they are connected to the bus, the shorter timings can be used in a UPnP implementation.

Application on an IEEE 1394 bus

As mentioned previously in section 5.1, TCP/IP can be carried over an IEEE 1394 bus using the means standardised in RFC 2734 [39].

The use of DHCP allows UPnP-compliant devices to co-operate with networks controlled by a residential Internet gateway which includes a DHCP server (e.g. Windows' Internet Connection Sharing service or a home broadband router).

Design of the implementation

In later versions of the Linux 2.4 kernel, the *eth1394* driver complies with RFC 2734. This implementation was used for development of the UPnP control server.

Since DHCP is available as a standard feature in many embedded operating systems, and since this part of the UPnP stack is the lowest-level and hence least portable layer, a new implementation was not written. The standard DHCP client included in Red Hat Linux, based on the Internet Software Consortium's freely redistributable DHCP Distribution, was used in the prototype system.

Performance of the implementation and possible improvements to the implementation

Since no custom implementation of chapter 0 was developed, no comments can be made regarding its performance or improving it.

Possible improvements to the specification

Within an IEEE 1394 bus, each device is allocated a unique physical ID in the self-identification process that follows a bus reset [8, ch 16]. This ID can be used to select an IP address in a link-local range without fear of collision. However, this method cannot be specified in UPnP since the standard is intended to be media-independent.

In future, the specification should include the use of IPv6 auto-configuration methods to acquire an IPv6 address. RFC 2462 describes methods for selecting a link-local IPv6 address and testing it for uniqueness on the network [47], much as Auto-IP does for IPv4.

5.5 UPnP chapter 1: Discovery

Analysis of the specification

Chapter 1 describes the means by which control points can ascertain the list of available devices and services on the network using the Simple Service Discovery Protocol (SSDP). SSDP has been specified in an IETF Internet draft [17] apparently written specifically for UPnP. It is based upon unicast and multicast HTTP-over-UDP (HTTPU and HTTPMU, respectively), also described in an Internet draft written for UPnP [18].

The fundamental operations in SSDP are presence announcement, search request, and shutdown announcement, shown in figure 5.2. Each announcement contains a tuple of notification type (NT), indicating whether it refers to a device or service; the unique service name (USN), which uniquely identifies

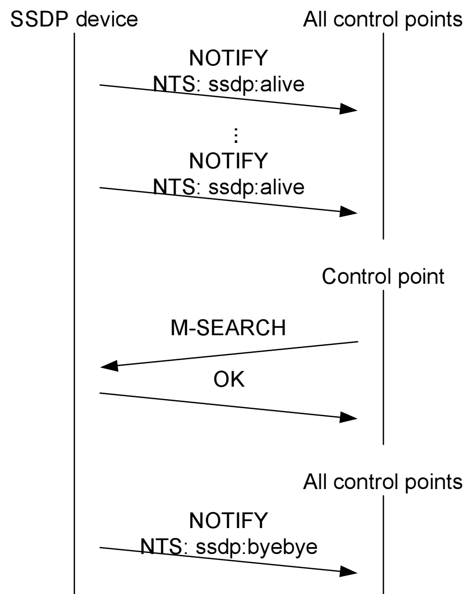

Figure 5.2: SSDP announcement and search processes

the specific device or service being announced; the description URL, and a time-to-live value. Control points can maintain a cache of announcements or search for each device they wish to communicate with. Search requests for a specific USN or category of USNs can be broadcast across the network, and those devices which announce a matching device or service will send an announcement in reply to the searching control point. An announcement is only valid for the time-to-live period or until a shutdown announcement cancels it, though it can be renewed by reissuing the announcement.

Comparable protocols are the NetBIOS name service (used in Windows Networking) where browse lists are maintained on an elected browse master or a pre-configured WINS server [48], also using presence announcements, or the IETF Zeroconf working group's DNS Service Discovery, where lists of service pointers are maintained in a multicast DNS domain [49].

The specification intends the discovery function to operate without configuration, and to be reliable and simple to implement. HTTPU and HTTPMU are intended to build on the popularity of the HTTP protocol and perhaps reuse existing code. In practice, however, it differs substantially from the HTTP/1.1 RFC [50] (for instance, there are no body sections at all) and an existing implementation would have to be of a very flexible design to be of any use.

Application in an embedded system

An embedded device that wishes to advertise itself and its services using SSDP must send advertisements at regular intervals and on request. The preparation of those advertisements is straightforward and requires little memory, since the SSDP draft limits the size of advertisements to 512 bytes or less (citing

a 512 byte limitation common in UDP implementations). Interpreting search requests requires variable-length string parsing, though the 512 byte limit bounds the amount of memory required. Multiple threads are not necessarily required.

Application on an IEEE 1394 bus

The bandwidth available on an IEEE 1394 bus and the small number of nodes in a home entertainment system make the traffic wasted by the SSDP protocol less significant than on lower bandwidth or larger scale networks.

Design of the implementation

The *ssdpd* program implements the SSDP protocol. It runs as a separate process and does not need to communicate with other components of the UPnP implementation, since all the data required is specified at compile time. As shown in figure 5.3, *ssdp_device* structs in a linked list describe the root and embedded devices and point to lists of *ssdp_service* structs, which describe each service provided by the device. The data structure could allow devices and services to be added or removed during execution, however those changes are less likely to occur in an embedded system, and the program has not been written to support that.

When started, *ssdpd* declares the presence of its devices and services by walking the *ssdp_device* structure, and then preparing and sending an announcement for each NT and USN pair. The announcement is sent as an HTTPMU NOTIFY request with NT and USN HTTP headers. Thereafter, *ssdpd* awaits the arrival of search requests on UDP port 1900, or the expiry of the its announcements. On receipt of an M-SEARCH request, *ssdpd* parses the headers to extract the search type (ST) value, searches the *ssdp_device* structure for services or devices of that type, and sends unicast HTTPU responses for any matches. Some time before the expiry of the last broadcast announcements, *ssdpd* will resend all announcements.

Performance of the implementation

The implementation was tested with the SSDP client service in Windows ME and XP. The test device appears in Windows' "Network Places" folder within seconds after the control server has started.

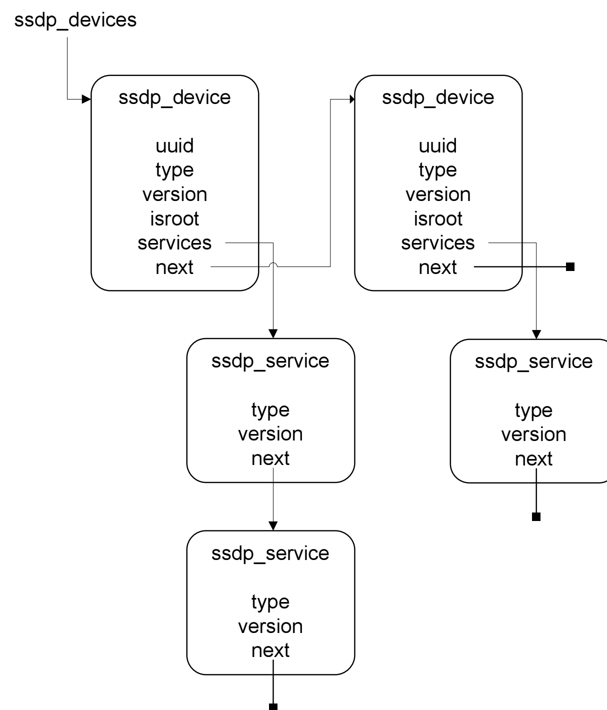

Figure 5.3: SSDP device and service data structure

Possible improvements to the implementation

The UPnP Device Architecture defines a maximum wait (MX) header for M-SEARCH requests. UPnP devices should pause for a random delay from zero seconds to the number of seconds specified in the MX header before sending their responses, in order to prevent a flood of responses by spreading them over a few seconds. This implementation of SSDP ignores the MX header and sends responses immediately. Implementing the delay could be done by spawning a thread for each response which would pause before sending the response, or by creating a queue of responses. The first would add complexity to the currently single-threaded *ssdpd* program, and the second would increase memory requirements. However, in the context of the IEEE 1394 home entertainment network, it is unlikely that there would be enough devices on the network to create a flood of responses that would saturate an IEEE 1394 bus – even the slowest speed, S100, offers bandwidth of nearly 100 Mbps.

Possible improvements to the specification

Perhaps the chief criticism of the SSDP protocol is its inefficient use of bandwidth. The announcements could be sent in a more compact format than HTTP headers. The SSDP draft specification also identifies situations where a storm of queries and responses could flood a network [17, section 6.3.1].

Were one to design a service discovery protocol exclusively for use on an IEEE 1394 bus, the protocol would not need to require periodic re-announcement of services, since all nodes on the bus are aware of

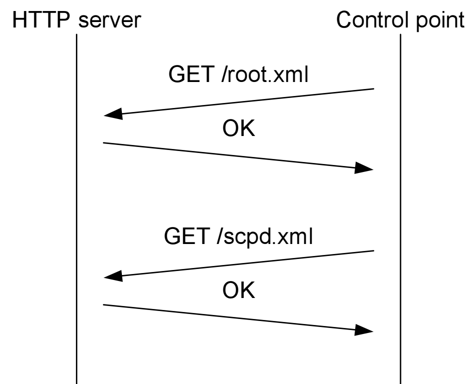

Figure 5.4: HTTP retrieval of root and service descriptions

other nodes connecting to or leaving the network from the subsequent bus reset. The services and devices available on each node would still need to be announced, though no bandwidth would be wasted repeating those advertisements.

5.6 UPnP chapter 2: Description

Analysis of the specification

The Description chapter provides the framework for control points to learn the nature of a device and the service it provides. The SSDP announcement includes an HTTP URL from which the root device description can be retrieved; that device description is an XML document containing URLs of descriptions for embedded devices and services, and control and eventing URLs for each service, in addition to a human-readable description of the device. The process followed is represented in figure 5.4.

Every service provided by the device or an embedded device is described by a Service Control Protocol Definition (SCPD) XML document. The SCPD lists the actions, effectively functions which can be called by a control point using the Control protocol covered in section 5.7; and state variables, a state variable representing some aspect of the current state of the device, provided by the service. For each action, the sets of parameters expected and returned are given, and each is bound to a state variable. State variables list their data type (from a range of types specified in the chapter) and their possible values are enumerated.

Device and service templates are used by UPnP Forum working committees to provide descriptions for standardised classes of devices. These templates contain certain values, actions and state variables that must be present in all devices which implement the standard. This allows for devices from different manufacturers to operate in similar ways, but still allows manufacturers to add additional functionality

to their devices. Non-standard devices and services can also be created and named outside the upnp-org namespace.

The Description phase represents the stage at which the UPnP device and service model is most clearly expressed. It provides for human control of devices, as well as automatically-discovered interaction between devices.

Application in an embedded system

Chapter 2 requires nothing more than a static-content HTTP server to serve the description documents, which can be implemented on the smallest of embedded computers [42, p 269].

Application on an IEEE 1394 bus

HTTP is carried via TCP connections between client and server, which are provided on IEEE 1394 networks by the IP-over-1394 encapsulation [39]. Even though the DHIVA HTTP server developed during the feasibility study proved slow, no fundamental problems regarding HTTP over IEEE 1394 were found.

Design of the implementation

Since none of the standardised UPnP device type templates available at the time of development suited the RX-V1000 AVR, a non-standard device type ("urn:cs-ru-ac-za:device:avr:1") and service type ("urn:cs-ru-ac-za:service:avrsvc:1") were used. This allowed the UPnP implementation to be matched to the RX-V1000 and its native control protocol – rather than trying to fit between two conceptual models of an AVR.

The RX-V1000 was modelled as a single root device, though the controls for the Main and Zone 2 zones and tuner could have been modelled as separate embedded devices. The root device description document is shown in appendix B.

Having identified all the variables of the AVR using the method described in section 4.3, the service provided by the root device's state variable list included entries and types for all readable variables. For those that were readable and writable, an action to set that variable's value was included in the action list. Write-only variables, such as Autotune Up/Down, were included as actions without arguments. The service description document is shown in appendix C.

Current:

```
<argument>
   <name>volume</name>
   <direction>in</direction>
   <relatedStateVariable>volume</relatedStateVariable>
</argument>
```

Proposed:

```
<argument>
   <name>volume</name>
   <direction>in</direction>
   <type>float</type>
</argument>
```

Figure 5.5: Current and proposed means of indicating the type of an argument

Performance of the implementation

The device and service descriptions for the AVR were tested using the UPnP clients in Windows ME and XP and the generic control point in the UPnP SDK, and all were able to parse the descriptions and access the control and presentation URLs.

Possible improvements to the implementation

The device and service description for the RX-V1000 was quite large – 24 kilobytes. Those files are statically compiled into the HTTP server so they are always loaded in memory. Since the XML document repeats a similar group of tags for each action description, the contents of the file are very repetitive. In order to save memory at the cost of increased CPU usage, the file can be compressed, using Huffman coding or a similar algorithm; or it can be generated by program code which would iterate through arrays of action and variable names to produce the document on demand.

Possible improvements to the specification

As noted by those who have implemented UPnP devices, the intended model that all actions in the service description should directly relate to state variables has been found to be flawed [51]. At present, the RelatedStateVariable element given for each Argument only serves to indicate the data type of the argument. This can be better done by stating the data type directly, in a manner such as that shown in figure 5.5.

5.7 UPnP chapter 3: Control

Analysis of the specification

The Control chapter defines how services within a UPnP device can be controlled and the values of state variables queried by a control point. Actions are remote procedure calls which are invoked with a set of arguments, and are executed by the service, which returns a set of result values. Queries allow a control point to poll once for the value of a state variable, as opposed to receiving immediate notification of variable value changes, as in the case of Eventing, examined in section 5.8. Queries can be considered a form of action with a list of variable names in the arguments list and their values returned in the response, though they do not effect any changes to the state of the service.

SOAP, an XML-based standard for exchanging structured information [19], is used to carry action and query invocations from the control point to the service, and return their responses. SOAP defines envelope, header and body elements used in the XML messages, while chapter 3 of the UPnP specification defines the action or query names and argument elements within the body. Though SOAP messages can be transported over different protocols, in the case of UPnP messages are transported using HTTP over TCP.

The HTTP binding chapter of the SOAP 1.1 protocol, current at the time the UPnP Device Architecture was published, required the use of a SOAPAction HTTP header, and contained a URI to indicate "the intent of the SOAP HTTP request" [52, section 6.1.1]. In order to comply with the standard, UPnP uses the action name to form the SOAPAction header. The HTTP binding allows the optional use of the HTTP Extension Framework, which formalises a means for HTTP extensions to add new headers in their own namespaces and specify optional or mandatory understanding of those headers, and allowing HTTP proxies to filter based on the extensions used [53]. UPnP requires that control points attempt requests using the framework should they fail without. This allows UPnP control to operate in environments where the use of the framework is or is not enforced.

The Control chapter makes use of the existing SOAP standard, with the SOAP HTTP binding, to such an extent that the only new aspects introduced are the contents of the bodies of the requests and responses. This allows reuse of HTTP servers and clients and SOAP-handling libraries (of which there are many) in the development of UPnP-compliant devices and control points.

Application in an embedded system

The implementation of the SOAP HTTP binding requires an HTTP server on the embedded system which is capable of generating dynamic responses based on the execution of an action, rather than merely

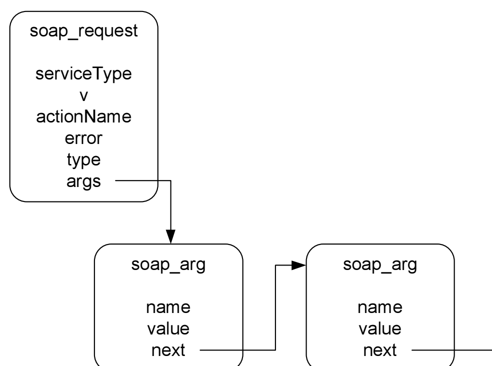

Figure 5.6: SOAP request data structure

the retrieval of files required by the previous chapter. The parsing of SOAP XML messages is more complicated and must be more flexible than the parsing of a binary data structure. These increase the amount of program code in the embedded system and the demands placed on the CPU.

Since the sizes of the SOAP messages received and sent vary, it is easiest to allocate memory dynamically, which makes peak memory use unpredictable unless message size limits are set.

Application on an IEEE 1394 bus

From the perspective of the network, the control protocol appears to be nothing more than HTTP requests and responses, and, as noted in section 5.6, HTTP can be carried over an IEEE 1394 bus without problems.

Design of the implementation

The SOAP server was first developed as a CGI (Common Gateway Interface) program which was called by the Apache HTTP Server to handle HTTP requests to the control URL. This allowed development to focus on the SOAP protocol by allowing Apache to handle the HTTP protocol.

A *soap_request* structure (figure 5.6) is used to track the state of each request, including the request type, an error code, and the list of in- and out-parameters. After each request's HTTP headers are checked, the SOAP envelope is parsed. The *soap_parsexml* function uses the *expat* XML parser toolkit [54] to parse the XML document. While many SOAP-handling and XML-parsing libraries are available, *expat* was selected as it is: available under a liberal open-source licence, which allows it to be included in commercial products; written in C, like the rest of the UPnP implementation, which simplifies compilation and linking; very fast, proving to be twice as fast as its nearest competitor in a benchmark test [55]; and small, making it suitable for use on an embedded computer. Either *expat* version 1.2 or 1.95 can be used since the parsing required is simple enough to not rely on any of the functionality

added in the later version. Version 1.2 is smaller, though version 1.95, the most recent, is written in ANSI-compliant C, making it more widely portable.

expat is a stream-based parser which reads a stream of XML and calls predefined callback functions as certain parts of the XML document are reached. In the *soap_parsexml* function, handlers for opening and closing tags and character data are installed, the body of the HTTP request is read into *expat*'s buffer, and the parse is begun. As the parser descends into the XML structure, the opening tag handler records the tag depth and checks that the appropriate tags appear at the correct points. It adds a *soap_arg* struct to a linked list on the *soap_request* structure for each argument once the arguments list has been reached. The character data handler is enabled at that point and it adds the values of the parameters to the *soap_arg* struct.

Once parsing is completed and the service type, action name and arguments have been extracted, the action must be executed. To do so, the CGI script would have to communicate with the device and instruct it to perform the action. However, since CGI scripts are started by the HTTP server and run to service only a single HTTP request, the CGI script would need to establish a connection to the device for each request.

In order to overcome this problem, a purpose-specific HTTP server, *soaphttpd*, was developed. It incorporates the SOAP functions, and listens on a different TCP port to that used by the static content HTTP server. *soaphttpd* is started by the *rxvupnp* program, which handles communication with the AVR (described in section 5.10), and is permanently connected to it via its *stdin* and *stdout*. *soaphttpd* passes the action or query to *rxvupnp* in a simple text format, which *rxvupnp* executes or answers on *soaphttpd*'s behalf and returns the response, from which the out-arguments are placed on the request's arguments list. The two designs identified for the SOAP service are illustrated in figure 5.7.

The *soap_fmtresponse* function takes the *soap_request* structure and prepares the response XML document based on the state of the request and its arguments list. The XML is returned to the client as the body of the HTTP response.

Performance of the implementation

The performance of the SOAP server was assessed by timing how quickly it responded to user input, where instant response is most desirable.

Testing was performed using two Linux PCs, one running the UPnP control server and the other acting as a control point, and a DapDesign FireSpy400 bus analyser on an IEEE 1394 bus. The control server was run on the slower of the PCs, which had a 233 MHz Pentium II CPU, as its performance was closer to that of an embedded system. The controller was run on the faster 866 MHz Pentium III

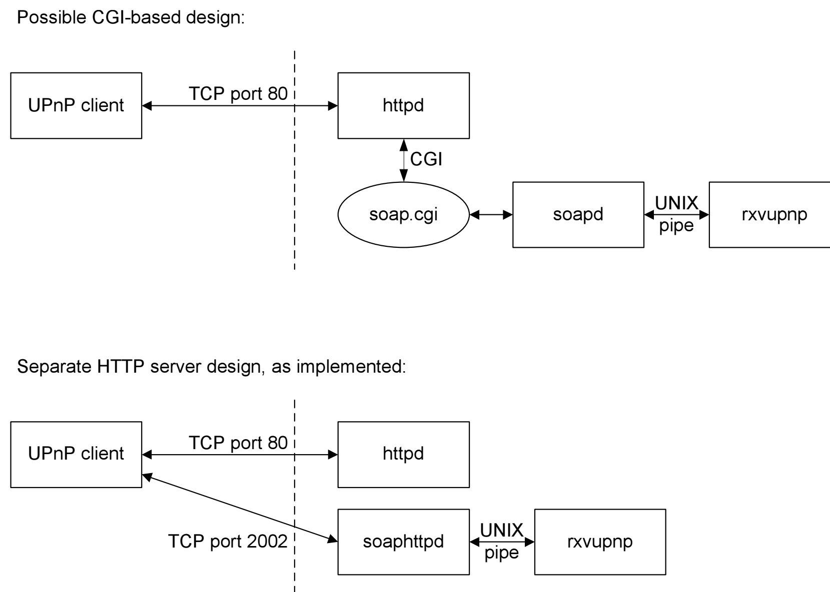

Figure 5.7: SOAP service designs

system. The controller was a Perl script which used the *HTTP::Request* module from the LWP (Library for WWW in Perl) to send three different SOAP requests to the control server, and high-resolution timing facilities from the *Time::HiRes* module to measure total time elapsed. The requests were a badly-formatted SOAP request to test error handling; a *queryStateVariable* action to retrieve the value of a single state variable; and a *getallvariables* action to retrieve the values of all state variables. Each request was sent ten times with a one second delay between each run, and each time the SOAP server would parse the request, query *rxvupnp* for the variable values, format the response and return it to the controller. Note that this included no serial communication with the AVR, as the aim was to determine how much time the control communication took. The FireSpy400 was used to measure the time elapsed between the time the first IEEE 1394 frame of the request reached the bus, and the time that the last frame of the response was transmitted on the bus. The controller script recorded the total round trip time by timing how long the *request()* function took to return, which is the time that a real controller might wait.

The results of the first runs were found to be inconsistent with the subsequent runs, so they were discarded, and the results in table 5.1 reflect the last nine runs of each test. Possible causes of the slow times of the first runs were the initial ARP queries, the results of which would be cached in the machines' ARP tables for the remaining runs; and the operating system's scheduler increasing the control server's priority from idle. The complete result set appears in section D.1.

The times for the first and second tests are similar, even though in the first case there was no need to query *rxvupnp*, and the error response is longer than the single variable response. The third test takes substantially longer to process, as its response includes the values of more than 40 variables. There is a

Test	Time on the bus (in ms)				Round trip time (in ms)			
	Min	Avg	Max	Std dev	Min	Avg	Max	Std dev
Invalid request	4.216	4.283	4.591	0.116	6.233	6.307	6.806	0.188
queryStateVariable	4.253	4.279	4.296	0.015	6.254	6.285	6.306	0.017
getallvariables	5.673	5.693	5.717	0.017	7.682	7.711	7.736	0.019

$n = 9$ for all tests

Table 5.1: Results of *soaphttpd* performance testing

consistent difference between the time measured by the bus analyser and that measured by the controller of almost approximately 2 milliseconds. This time would be taken by processing performed on the controller PC by the TCP/IP stack in the kernel and the Perl HTTP classes.

Possible improvements to the implementation

The SOAP programs could make better use of memory. The arguments list, argument values and response document are created using dynamically allocated memory, which could be exhausted by a sufficiently large request to a small embedded system. The functions could be redesigned to use static memory, or limits placed on the size of input the programs will accept.

The HTTP handling functions in *soaphttpd* don't handle mandatory HTTP headers as strictly as the HTTP Extension Framework dictates, and the XML parsing functions don't necessarily follow the SOAP Processing Model in the SOAP standard [19, section 2]. However, fixing either issue would increase code size and make the programs less flexible about the input they accept.

The *expat* library could be shrunk further. At present, even when compiled using the reduced code size option, *expat* adds at least 52 kilobytes to the executable (using the GCC 3.2 C compiler for Linux on the x86 architecture). XML-parsing functionality not used by *soap_parsexml* could be pared from *expat* to reduce the amount of memory used and possibly improve processing speed.

Possible improvements to the specification

Measures could be taken to reduce the command-response latency. Points which could be addressed are the network latency and processing latency. Approaches to reducing the first should attempt to avoid establishing a TCP connection per control action or query. That could be done using a connection-less protocol such as UDP, though the limits that would place on message size would be unacceptable; or the use of persistent HTTP connections, as HTTP version 1.1 allows TCP connections to be kept alive for multiple HTTP transactions [50, section 8.1]. Thus each control point would only need to open a single TCP connection to the device for its control purposes.

The Web Services Description Language (WSDL) [56] is used to describe web services which use SOAP over HTTP, and is used by development environments such as Microsoft Visual Studio .NET. Devices could publish WSDL descriptions of their control services in addition to, or instead of, the UPnP chapter 2 service descriptions to allow programmers easier access.

5.8 UPnP chapter 4: Eventing

Analysis of the specification

Eventing refers to the process by which a control point is informed of events that change the state of a state variable in a device it controls. For example, if a device supports eventing on a "volume" state variable, control points can be notified of the new volume level when the volume knob is turned.

Chapter 4 of the UPnP specification describes a publisher/subscriber model in which each service that has evented state variables is published by the device, and control points may subscribe to that service to receive events. Services that support eventing specify an event subscription URL and mark each state variable as evented or not in their service descriptions. Subscribers may *subscribe* to receive notification of a service's events, *renew* that subscription before it expires, or *cancel* it to stop receiving events. Publishers send *event messages* to subscribers. Subscription and event messages are communicated using HTTP augmented with General Event Notification Architecture (GENA) [20] headers, and state variable values are formatted according to the UPnP event XML schema in the event messages.

A control point can open a subscription to receive event notifications for a certain service by sending a SUBSCRIBE HTTP request to the eventing URL given in the service description, indicating how long the subscription should last (the timeout) and the callback URL on the control point to which event messages should be sent. If the subscription is accepted, the publisher will send a response with a subscription identifier (SID) and the agreed timeout. As soon as possible, the publisher will send an initial event message in a NOTIFY request to the callback URL for the subscription. The body of the initial event contains the current values for all evented state variables in the service. When state variables change afterwards, the publisher will send event messages with new state variable values. All event messages include a sequence number, incremented for each event message to allow the subscriber to detect missing or out of order messages. No means is provided for subscribers to request retransmission of missing messages, short of cancelling and reopening the subscription to force another initial event message. Sometime before the timeout length has passed, the subscriber must renew its subscription or the subscription will lapse. This is done by sending another SUBSCRIBE request, this time including the SID of the subscription to be renewed. Subscriptions can be explicitly cancelled by sending an

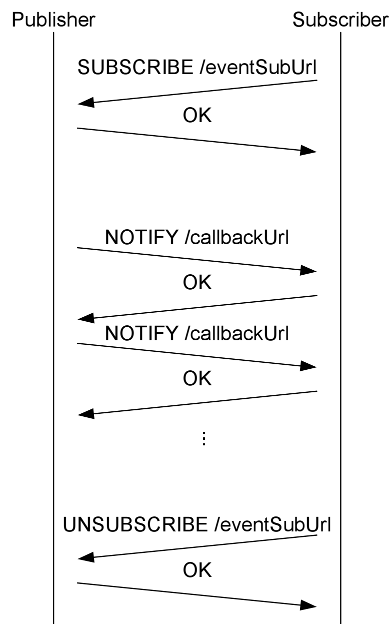

Figure 5.8: Communication between subscriber and publisher

UNSUBSCRIBE request, giving the SID to be unsubscribed. The process of subscription, notification and unsubscription is shown in figure 5.8.

The eventing specification extends the existing HTTP standard for use in a new application. While the value of the use of HTTP in SSDP was questioned in section 5.5, in this case, that value was confirmed during the development of a test subscriber program in Perl. The program used the *HTTP::Request* class from the LWP to open a subscription and the *HTTP::Daemon* class to provide a callback URL to receive event messages. Neither task required more than ten lines of code. Similarly, XML-handling libraries can be employed to parse the contents of the event messages. However, the use of HTTP was found to cause poor performance, discussed below.

Application in an embedded system

In most circumstances, the publisher will need to be able to accept a number of subscriptions at once, and thus the design of the publisher calls for multiple threads (further explored on the next page). This requires that the embedded processor and operating system support multithreading. Should it be acceptable that a device only support a single subscription, a single-threaded publisher could be designed, though this has not been investigated.

Generating the XML in the event messages is relatively simple – the simplest event notification only requires three sets of tags. There is no need for the publisher to parse XML, which would be more complicated.

```
avrsvc|volume|-52.5
avrsvc|zone2volume|-30.0
```

Figure 5.9: Examples of state variable change messages

Application on an IEEE 1394 bus

No issues specific to IEEE 1394 buses were identified.

Design of the implementation

The *genapub* program is an event publishing daemon. It receives notification of state variable changes from *rxvupnp*; handles subscription messages to maintain a list of subscriptions; and prepares and sends event messages to its subscribers.

State variable changes are passed from the *rxvupnp* program via *genapub*'s *stdin*. The messages are sent in a simple text format not specific to the RX-V1000, examples of which are given in figure 5.9. The variable values are received and stored as strings, already formatted for inclusion in the event message XML document, so that *genapub* need not be aware of the data types of the variables or how to format them.

Subscription messages from control points could be handled by a CGI script run by a general-purpose HTTP server. However, this presents a problem similar to that faced by the SOAP server in section 5.7: a publishing program must run continuously to receive events from the device, so the subscription CGI script would need an interface to the publishing daemon. In order to avoid that complexity, a simple HTTP server, of the same design as that used in *soaphttpd*, was incorporated into *genapub* and listens on its own TCP port.

The *genapub* main loop uses the standard UNIX *select* system call to wait for new events or new connections to the HTTP server. Possible means of sending the event messages to the subscribers are considered below.

Sending event messages to the subscribers

genapub's most important task is to distribute event notifications received from the device to a number of subscribers in the shortest time possible. However, distribution of event messages is done with an HTTP callback, so a TCP connection must be established for each subscriber for each event. This can take a substantial length of time: a three-way handshake must be performed, the HTTP request sent, the HTTP response received, and the connection closed. Further, subscribers might be connected at different bus speeds and they may respond slowly or not at all. The simplest approach would be to deliver each

message to each subscriber in sequence; however, this is unsatisfactory as it would allow slow subscribers to delay delivery to faster subscribers.

This problem requires a repeater which, given a source of messages, will deliver each message to each recipient – this is analogous to an Ethernet hub that repeats every frame received to each of its ports. A number of possible approaches were considered:

1. *One producer spawns n threads.* A producer thread could spawn a throw-away thread for each message for each subscriber. While this is very easy to implement, since it avoids thread synchronisation issues, it has numerous disadvantages. Many threads would be needed on the publisher, as well as the subscriber, which might receive a number of messages from the publisher simultaneously. The publisher might deliver messages out of sequence - though the client could counter that by buffering messages. It would be impossible to coalesce or skip messages should the network become backlogged since each message-delivering thread would not know how many other threads were attempting to deliver messages to that subscriber.

2. *One producer and queue; n consumer threads.* Based on the well-known producer-consumer problem [57, p 39], a producer thread would add messages to the head of a queue. Each message would be tagged with a counter initially set to the number of subscribers. Consumer threads, one per subscriber, would process each new message placed on the queue and decrement the counter once done. The message could be removed from the queue once its counter reaches zero. Were a consumer thread to terminate, it would need to run through its backlog of messages, and decrement their counters, otherwise those messages would never be removed from the queue. In this approach, consumer threads would be able to coalesce and skip messages, and messages can be delivered in sequence since only one thread communicates with each subscriber. The disadvantage of this approach is the effort it takes to ensure that each event is seen by each consumer thread.

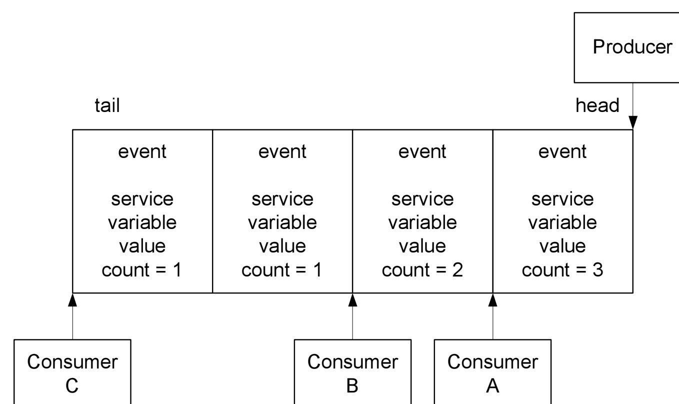

Figure 5.10: One producer and queue; *n* consumer threads

3. *One producer; n consumer threads and mailboxes.* The mailbox data structure introduced by Tanenbaum [57, p 50] was considered as a means of communicating events from the producer to the consumers. A mailbox allows multiple processes to send and receive messages to and from a buffer, and holds queues of processes waiting to send and receive messages to or from the mailbox. Each process has a semaphore used to wake it when a message is available for it, and a mutex protects shared data in the structure. The mailbox structure is shown in figure 5.11.

   The mailbox design is intended for distributing each message to any receiving process, whereas *genapub* requires that each message be distributed to all receivers. This could be handled by using one mailbox per subscriber, though there would only be one sender and receiver process for each mailbox, making the complexity of the send and receive queues in the mailbox unnecessary.

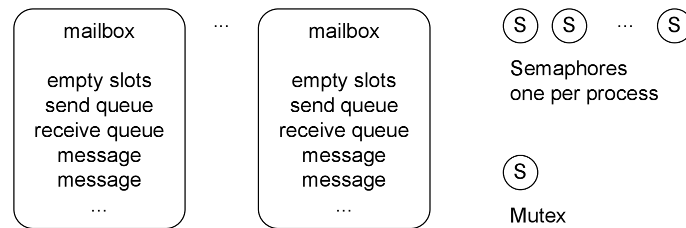

Figure 5.11: The mailbox data structure (adapted from [57, p 54])

4. *One producer; n consumer threads and queues.* Based on the previous approach, and similar to the second, this approach uses a simpler data structure than the mailbox, but achieves the same end. A single producer thread could add a copy of each new message to a queue per subscriber. A consumer thread per subscriber would remove and deliver messages from its queue. While this approach has the advantages of the previous option and is easier to implement, it could require *n*-times as much memory. It is important that the producer thread should be able to add messages to the queues without blocking, as that would allow slower subscribers to delay faster subscribers.

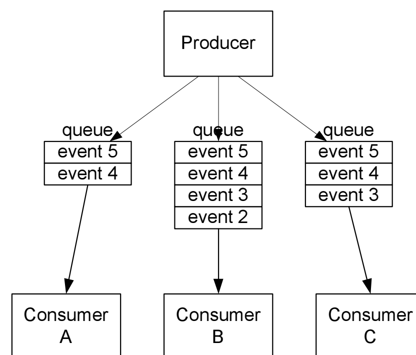

Figure 5.12: One producer; *n* consumer threads and queues

The fourth approach was chosen for *genapub*. Though the second and third approaches are both viable, and have lower memory requirements, the memory saved comes at the expense of a more complex algorithm.

Handling an event message backlog

Once a means for distributing event notifications to the threads which communicate with subscribers was selected, methods of selecting which events to send to those subscribers were considered. This becomes necessary since there are certain times when a state variable will change value many times in a short period. For example, it was observed that when the volume knob on the AVR was turned rapidly, an event occurred for each intermediate value that the volume changed to as the knob was turned. Within this burst of events, those describing intermediate volume levels are far less significant to the user than the final level that the volume settles at, however, the transmission of those events can delay the transmission of the final volume level to the control point.

Chapter 4 of the UPnP specification mentions that some state variables may change too frequently to be evented at all, and that for other state variables, the publisher could "filter, or moderate the number of event messages sent due to changes in a variable's value." [16, section 4] However, no methods of performing such filtering are suggested. A solution to the problem should provide as much up-to-date data to the subscriber as soon as possible in the correct sequence. In order to achieve that, the publisher can reduce the amount of data to be transmitted to the subscriber by either dropping certain messages, since it may be acceptable to ignore state changes that have already been undone; or by grouping a number of state variable changes into a single XML document. Thus the following possible means of selecting events to send were identified:

1. Send all events in the queue. The simplest option is to dispatch all events in the queue continuously within a single event message, thereby emptying the queue. While events would be grouped, every event would still be sent to the subscriber.

2. Send as many events from the tail of the queue until a variable appears a second time. This avoids repeating a variable within a single event message, which the specification is unclear about.

3. Use the first option, but optimise first by passing through the queue from the head, discarding all but the *n* most-recent states of each variable.

4. Count the number of items in the queue before sending, and should the queue be shorter than a certain limit, apply the second option; should it be longer than the limit, discard all changes to variables which have subsequently changed.

The third option is implemented in *genapub*. The *gena_queueevent* function ensures that, before adding a variable state change to a queue, no more than (by default) three changes of that variable are present on the queue. Thus the memory used by the queue is limited, but the producer thread has to perform the task, increasing the chances that slow subscribers can delay faster subscribers. Alternatively, the optimisation pass could also be performed in the consumer threads (in the *gena_eventloop* function) – the choice between the two represents a trade-off between reduced memory usage and a quicker producer thread.

The *gena_eventloop* function prepares an event message with as many state changes as are in the queue and sends it to the subscriber.

Data structure

The data structures used to describe services and record subscriptions and event queues are shown in figure 5.13. The list of subscriptions and the queue for each subscription were stored as linked lists, dynamically allocating memory from the heap as subscriptions are opened and events queued. The subscription list could be stored in a fixed-size array, though *genapub* places no limit on concurrent subscriptions at present. The queues could be stored in bounded buffers, but the producer thread would block when a queue became full.

While memory is allocated for each event queued, the maximum memory usage is predictable and, for embedded designs, can be contained by limiting the maximum numbers of concurrent subscriptions, evented variables and the size of variable names and values.

Synchronisation

Since the subscription and event lists are accessed by multiple threads, synchronisation of those threads must be examined to avoid non-determinism. The protection of the event queues is comparable to that required for the bounded buffer example [57, p 43] in that a mutex provides mutual exclusion for each queue structure, and a counting semaphore synchronises the producer and consumer threads. It differs in that only one semaphore is needed since the buffer is currently unbounded. A mutex protects the subscriber list to ensure that subscriptions can be added and removed safely, and, likewise, a single mutex protects the values in all the variable lists to ensure that changes are made atomically.

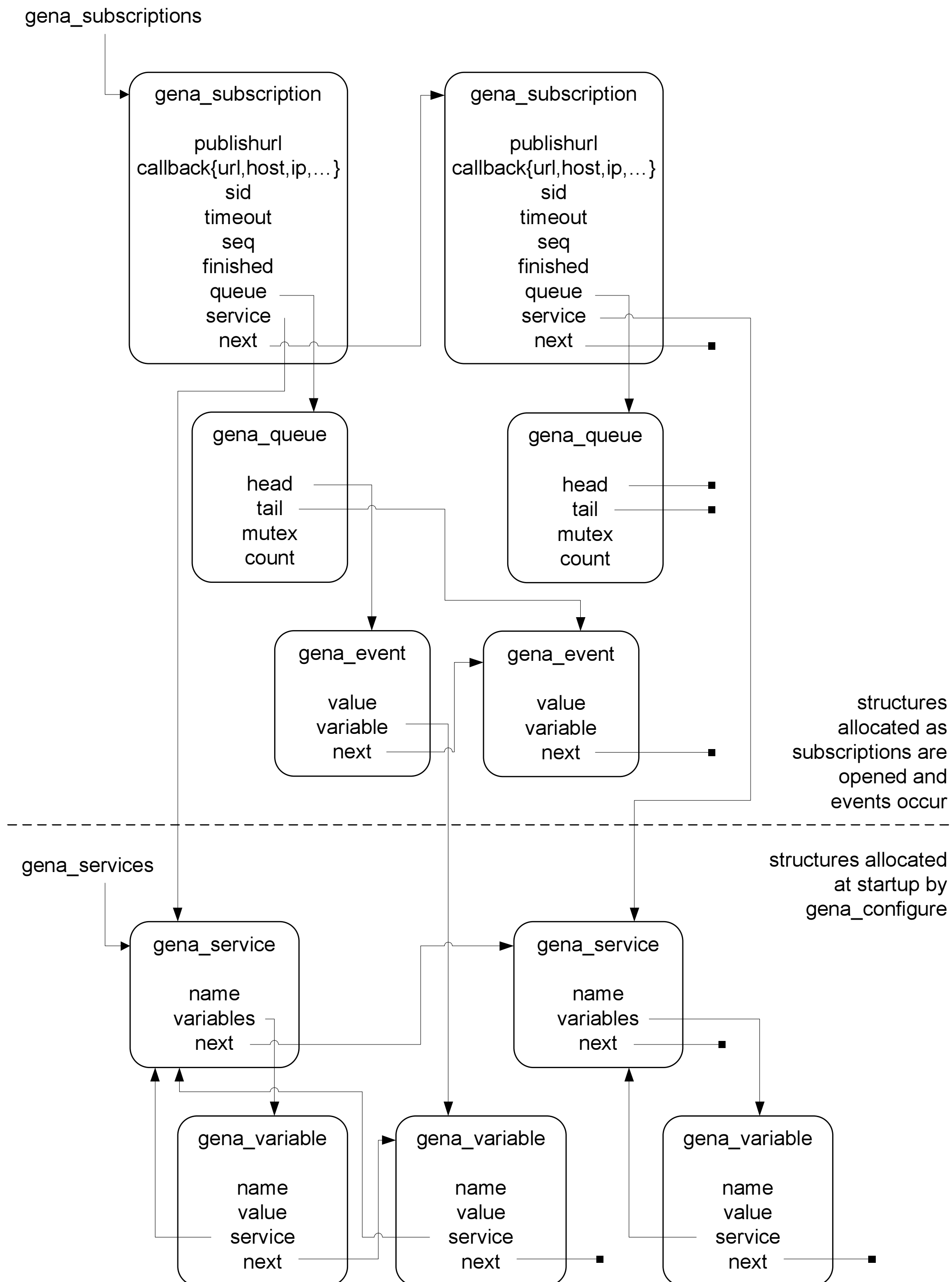

Figure 5.13: genapub data structure

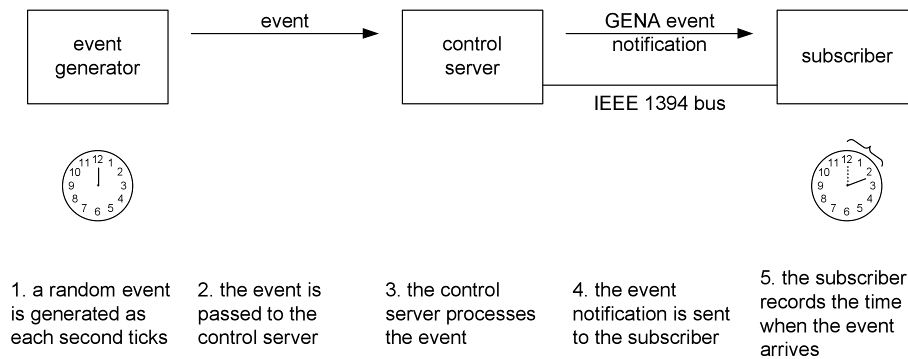

Figure 5.14: *genapub* performance testing experiment

Since the message-sending (consumer) threads spend most of their time waiting on a semaphore, they are unable to notice when their subscription expires. A reaper thread periodically checks for expired subscriptions to remove, doing so by setting the finished flag to true, then signalling the semaphore to wake the thread for the subscription so that it can terminate itself.

Performance of the implementation

In order to determine the extent of the delay caused by establishing a TCP connection to deliver an event message, an experiment to measure message delivery time was conducted.

The same PCs used in testing the SOAP server in section 5.7 were used for a GENA publisher and a subscriber. Prior to the test run, the real-time clocks on the publisher and subscriber were synchronised and their rates adjusted to match each other using the *clockspeed* package, which includes tools for synchronising clocks across a network [58]. This provided the closest synchronisation possible, and testing showed that the clocks differed by less than a millisecond after synchronisation.

On the control server, a Perl script using functions provided by the *Time::HiRes* module generated an event precisely as the real-time clock ticked each new second. The events were passed to *genapub*, which then prepared and sent the event message over a TCP connection to the subscriber. The subscriber ran a Perl script which received the event messages and recorded the time of arrival, allowing the calculation of the time between when the event would have occurred and the time the notification arrived, i.e. the delivery latency. The experiment sequence is shown in figure 5.14.

For comparison, the standard ICMP *ping* program was run between the same machines. ICMP is a connection-less IP protocol [59], which the *ping* program uses to send an echo request packet to another machine, which in its turn responds with an echo response packet.

The results of the test are summarised in table 5.2, and given in full in section D.2. The GENA average delivery latency is more than five times worse than the *ping* round trip time. This is partly due

Test	Delivery latency (in ms)			
	Min	Avg	Max	Std dev
Delivery of GENA event notifications	2.566	2.636	2.699	0.044

Test	Round trip time (in ms)			
	Min	Avg	Max	Std dev
ICMP *ping*	0.123	0.136	0.144	0.007

$n = 9$ for both tests

Table 5.2: Results of *genapub* performance testing

to the processing time required to prepare the event notification message, but also due to TCP overhead. Each ICMP ping sends only two packets across the network, while *tcpdump* recorded nine TCP packets in each GENA notification transaction. Thus, if the number of network packets involved in each transaction were reduced, the overall delivery latency would be improved.

Possible improvements to the implementation

At present, the *gena_services* structure is created by code in the *gena_configure* function, which must be altered when a service or state variable is changed. This could be avoided by either using a script to automatically generate the *gena_configure* function from the device description XML during compilation; or the *gena_configure* function could be rewritten to parse the device description at startup. While the latter would ease changes to the device description, it would require an XML-parsing library and parser routines, substantially increasing the size of the *genapub* program and startup time. For those reasons, the former would be more suitable for embedded systems.

Possible improvements to the specification

The most obvious problem with the eventing specification is the need to establish many short-lived TCP connections. This problem is the same, though perhaps more acute, as that faced in the Control chapter (see on page 56). A connection-less protocol could alleviate the problem. For instance, an eventing protocol which used multicast UDP packets to broadcast event messages would not require devices and control points to agree to subscriptions, but would force switched or routed networks to transmit every event message to all nodes on the network, wasting network bandwidth for those nodes which are not interested in receiving those event messages. A protocol which used unicast UDP in place of TCP, but still used subscriptions, could solve that issue.

In both cases, sending an event message would not require the overhead of the TCP three-way handshake, but there would be no guarantee that the event message had been received by the subscriber. While UDP-based streaming media protocols allow receivers to request retransmission of missing packets, that only solves the problem because the receivers buffer a substantial length of the stream before playing it to allow time for retransmissions to be received. In an eventing context, the stream of events is far less regular, and low latency is critical. Thus connection-less protocols would only be feasible on very reliable network media (such as the IEEE 1394 bus).

Another approach would be to increase the duration of the TCP connections, by using HTTP/1.1 persistent connections. This would be an ideal solution to the problem, however UPnP chapter 4 gives no indication whether HTTP/1.0 or HTTP/1.1 compliance is required. Thus *genapub* has been designed to support the lower level for wider compatibility.

5.9 UPnP chapter 5: Presentation

Analysis of the specification

The device description optionally includes a URL for a presentation page which the user can access with a web browser. Presentation pages may be used to describe the device, show its status, and control it. Chapter 5 is the shortest in the UPnP Device Architecture, and besides specifying that HTTP and HTML should be used, it leaves all other aspects up to the discretion of the implementor.

Application in an embedded system

A simple presentation page merely requires a static-content HTTP server, such as the one implemented on the DHIVA during the feasibility study. A page which shows the device state would require dynamic generation of web pages, and device control using HTML forms would require that the server be able to handle the form return values. Thus an HTTP server of an appropriate level of complexity can be developed to suit embedded computers of different capabilities.

Application on an IEEE 1394 bus

As noted in section 5.6, HTTP can be carried over an IEEE 1394 bus without problems.

Design of the implementation

The presentation page for the RX-V1000 aims to show device state and allow it to be controlled, while allowing the page to be easily accessible to many users and placing as few demands on the device as possible. In order to avoid developing a second means of control, the SOAP service was used for control operations from the presentation page. The sample device included in the Microsoft UPnP Development Kit makes use of a SOAP client in an ActiveX control to send control messages to the device, since Microsoft's Internet Explorer browser provides no standard facilities for using SOAP. The ActiveX control will only run on PCs running Microsoft Windows and Internet Explorer, excluding certain clients. However, the Mozilla web browser includes a SOAP API [60] accessible from JavaScript scripts in web pages, and is available for a variety of platforms.

Shown in figure 5.15, the presentation page displays a set of controls, which the user can use to control the AVR. For example, the volume controls for the Main and Zone 2 zones are shown as sliders. A JavaScript function embedded in the presentation page sets the values shown on the controls by invoking the getallvariables SOAP action, which returns the current values of all state variables, then setting each control to the value returned. On input from the user, such as moving a volume slider, a JavaScript event-handler function invokes the appropriate SOAP action to make the change on the AVR. The user interface has the same layout of controls as *rxvqt* (see section 4.4) and the AV/C controller, *avcqt* (see section 6.8), for ease of use.

Performance of the implementation

Control using the presentation page is noticeably less responsive than using a native control point application, mostly due to the slow JavaScript interpreter. However, this was not regarded as critical, since the presentation page is not intended to be the primary means of control: it is meant to be quick and easy to access, while a control point application can be installed for longer-term use.

Possible improvements to the implementation

The presentation page is only able to receive state variable values by polling the SOAP interface. Implementing an eventing subscriber would allow the page to be continuously updated. This could be done by incorporating a subscriber API in the browser or writing a browser plug-in that could be downloaded from the device.

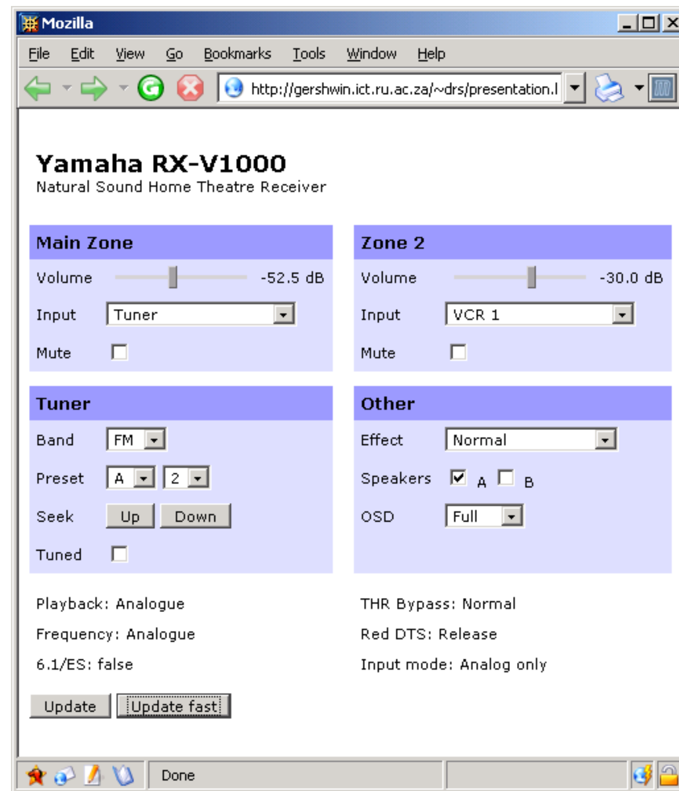

Figure 5.15: Presentation page for the RX-V1000

Possible improvements to the specification

While the presentation chapter could be greatly expanded to a complete user interface system, such as the AV/C Panel Subunit, there is justification for the current, non-specific chapter. Firstly, the current specification matches UPnP's goals, given in the Device Architecture's introduction:

> Universal Plug and Play is a distributed, open networking architecture that leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and public spaces. [16]

The Addressing, Discovery and Description chapters achieve proximity networking, and the Control and Eventing chapters describe standardised protocols to allow devices to interact with each other, but the user interface is not mentioned. Secondly, many existing devices already have their own control web pages, which, given the openness of the specification, can now be easily incorporated into a UPnP compliant device. Thirdly, this allows for competition between vendors and differentiation between their products.

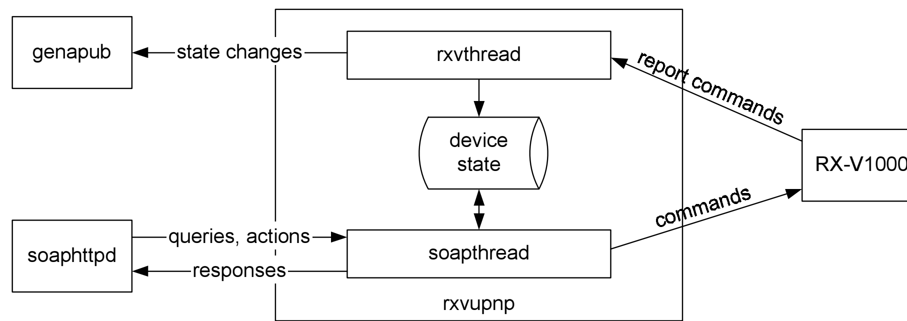

Figure 5.16: *rxvupnp* data flows

5.10 Coordinating the UPnP control server: the *rxvupnp* program

The last part of the UPnP implementation is the *rxvupnp* program which starts the *ssdpd*, *soaphttpd* and *genapub* programs, and proxies their communication with the AVR. *rxvupnp* is the only part of the implementation which has specific knowledge of the RX-V1000's RS-232 control protocol. It links in the *librxv* library (covered in greater detail in section 7.5) and uses its functions and constants to format action commands and parse report commands. As it understands the RX-V1000 control protocol entirely, it is able to attach to the AVR directly, communicating via the serial port device; or through *splitrxv* (covered in section 7.5), communicating via *stdin* and *stdout*. Since they are independent of the RX-V1000 control protocol, the other programs (i.e. *ssdpd*, *soaphttpd* and *genapub*) can be reconfigured and reused to create a UPnP implementation for an entirely different device. *rxvupnp* splits the communication between the AVR and the UPnP protocol programs into channels allowing the programs to be implemented as separate processes which can be developed and tested independently or replaced.

rxvupnp is implemented as two threads, shown in figure 5.16: one which waits for input from the AVR; the other for input from *soaphttpd*. Since the current *genapub* program only requires unidirectional communication, no thread is required to listen for input from it. Similarly, *ssdpd* only needs to be started and requires no communication with *rxvupnp*. The RX-V1000 thread receives report commands, updates *rxvupnp*'s copy of the state of the device, and passes an event notification to *genapub*. The SOAP thread receives queries and actions from *soaphttpd*, answering queries from its record of the device state, and passing actions towards the AVR. Queries must be answered from a copy of the device state since the RX-V1000 serial control protocol provides no means of querying the values of specific state variables. Synchronisation measures used are a mutex to prevent simultaneous reads and writes to the device state, and a binary semaphore which the SOAP thread waits upon when it sends a command. The RX-V1000 thread signals when the result of that command is received.

rxvupnp and *genapub* both keep a record of the device state. This duplication wastes memory and CPU time, but allows *genapub* to be tested independently of *rxvupnp*.

5.11 Summary

A UPnP control server that allows the RX-V1000 to become UPnP-compliant has been developed. While development was begun on the DHIVA embedded system, it was later moved to a Linux PC, though the control server has been designed so that it could be ported back to an embedded system. The control server is comprised of a number of connected programs, each of which implements a portion of the UPnP Device Architecture.

The Addressing chapter of the UPnP specification defines how devices acquire an IP address using DHCP or AutoIP in order to fit in managed networks and operate independently on unmanaged networks.

Discovery is based on the SSDP protocol which describes how devices and services can be advertised and searched for on the network, though in a manner which has been criticised as being wasteful of network bandwidth. SSDP is implemented in the *ssdpd* program.

The Description process provides information about the device and the services that it provides using XML documents served by an HTTP server. Descriptions were prepared for the RX-V1000.

The Control chapter specifies how UPnP devices can be controlled by other devices, and state variables queried using SOAP. A SOAP CGI program and stand-alone HTTP server, *soaphttpd*, were developed using the *expat* XML parser. Performance testing showed that *soaphttpd* could answer queries in times ranging from 4.2 ms to 5.7 ms.

Eventing allows controllers to subscribe for notifications of state-changing events on the device. UPnP uses GENA and HTTP for delivery of event notifications, as implemented in the *genapub* publisher. The development of *genapub* raised questions of how best event messages can be sent to subscribers, and how events should be selected for transmission when a backlog occurs. *genapub* was found to be capable of delivering a notification to a subscriber approximately 2.6 ms after becoming aware of the event.

User interface presentation is via a web page which can be displayed in a web browser. The Mozilla web browser's built-in SOAP functions allow full control of the RX-V1000 from the browser, without the need for additional software installation.

The *rxvupnp* program handles communication between the AVR and the other components of the control server.

The following chapter examines AV/C in depth and an implementation for the RX-V1000.

Chapter 6

Implementing AV/C

An AV/C control server containing an AV/C unit and an audio subunit was developed, and allows the Yamaha AVR to be controlled from an AV/C control point. This chapter examines the implementation of the unit and subunit, its communication with the AVR, the development of control point software, and the performance of the control server.

6.1 Design of the implementation

The *rxvavc* program is the AV/C control server for the RX-V1000. Like the *rxvupnp* UPnP control server, it is able to communicate with the AVR via RS-232 or via its *stdin* and *stdout*. Unlike UPnP, all communication with the client is via a single protocol, the Function Control Protocol (see section 6.2), so the AV/C control server is implemented as a single program. All AV/C commands are processed by a stack of protocol layers, each layer implemented as functions in *rxvavc*. Figure 6.1 provides a more detailed view of the AV/C control server and client based on figure 1.1.

The optional AV/C descriptor mechanism [28] was not implemented in *rxvavc* as it is beyond the focus on control of devices, and is particularly complex. For similar reasons, descriptor support was omitted from the Crest Audio FireBoB prototype. The implications are that control points are not able to discover the AV/C unit automatically, and must have specific knowledge of the unit and subunit to be able to control it.

6.2 Function Control Protocol

The AV/C General Specification [23] specifies that all AV/C commands and responses shall be carried using the Function Control Protocol (FCP) [24].

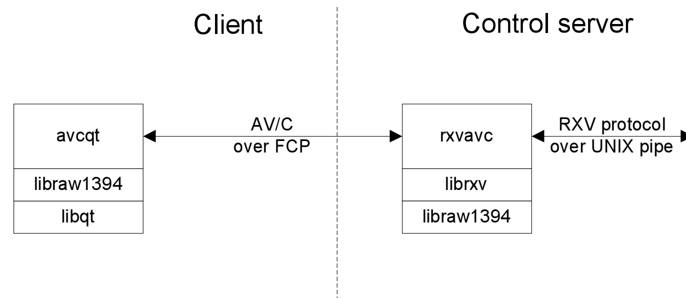

Figure 6.1: Components of the AV/C control server and client

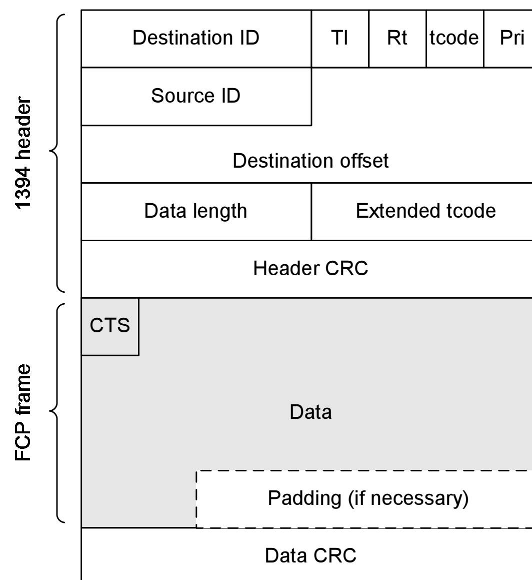

Figure 6.2: An FCP frame in an IEEE 1394 asynchronous write packet, adapted from [24]

FCP provides a simple means of encapsulating control commands and responses to them on an IEEE 1394 bus. An FCP frame is a unit of data transferred in an asynchronous packet of at most 512 bytes. It consists of a 4 bit Command/Transaction Set (CTS) identifier, followed by data in a format specific to that CTS. The CTS identifier allows FCP to be used for a variety of control protocols without interference between them. In the case of AV/C, all FCP frames are marked with a CTS value of 0. Command frames are transmitted by an asynchronous write to the target node, starting at register address FFFF F000 0B00, while responses are written starting at FFFF F000 0D00 on the controller node. The format of an FCP frame as it would transmitted on the bus is shown in figure 6.2.

Since the UPnP control server had been developed on a Linux system, the AV/C control server was written for the same platform. The IEEE 1394 drivers for Linux include kernel drivers for general IEEE 1394 support and specific PCI-to-1394 interfaces. A set of drivers provides support for IEC 61883-6 audio, IEC 61883-2 video, RFC 2734 TCP/IP and SBP2 SCSI carried over the IEEE 1394 bus. The

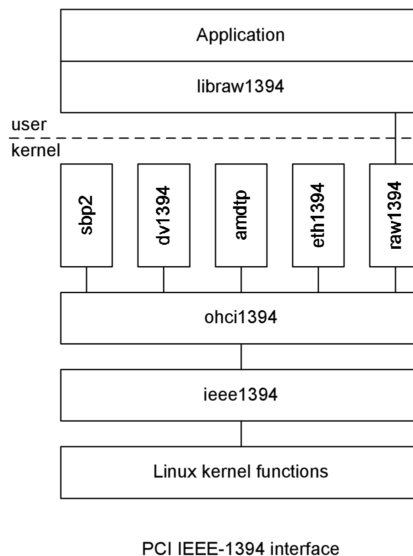

Figure 6.3: Linux IEEE 1394 and *raw1394* driver stack

raw1394 driver allows user applications low-level access to the facilities of the IEEE 1394 bus, including asynchronous transactions and isochronous streams. The *libraw1394* library provides a standard interface for applications to use the *raw1394* driver. The software stack is shown in figure 6.3. [61]

During initialisation, *rxvavc* opens a handle to an IEEE 1394 interface and registers its FCP handler function to be called by *libraw1394* whenever an incoming FCP frame is received. The FCP handler ignores response frames and any commands where the CTS is not zero, and passes AV/C commands to the AV/C handler function for processing. After processing, the response frame is returned to the FCP handler, which writes it to the response register on the node from which the command was received.

Frames are passed between protocol layers in *rxvavc* using two streams. The FCP handler writes the entire command frame to an upwards stream and reads the response from a downwards stream. At each layer above, the layer will read as many bytes as necessary from the upwards stream and interpret them, then write the corresponding response bytes to the downwards stream. The process is represented diagrammatically in figure 6.4. This approach was chosen after difficulties were experienced handling the variable length fields in AV/C commands using an approach which overlayed a struct over the packet data. The AV/C address fields, which can optionally be extended into subsequent bytes, were particularly troublesome. The advantages that it offers include: the first bytes that each layer reads from the upwards stream are the bytes that it is expected to process; the layers can copy fields directly from the command to the response, as is required for many fields by the AV/C specifications; and once each layer has completed its processing of the frame, it need only pass the stream handles to the next layer for further processing.

On the Linux system that *rxvavc* was developed on, the streams are created using the UNIX *pipe* system call, which returns a pair of file descriptors, one of which can be written to and the other read

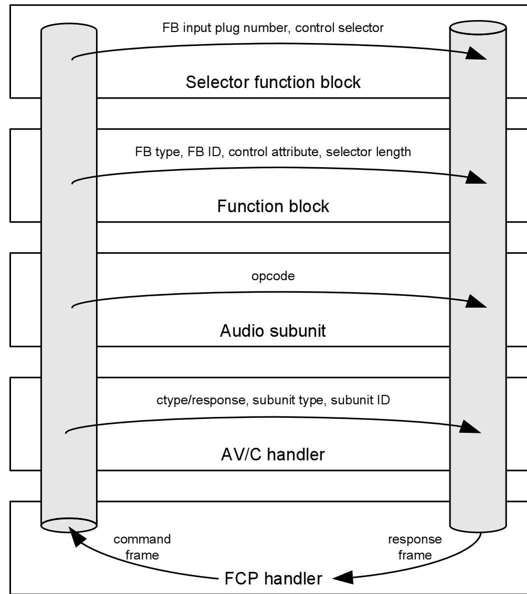

Figure 6.4: Stream-based communication between AV/C layers

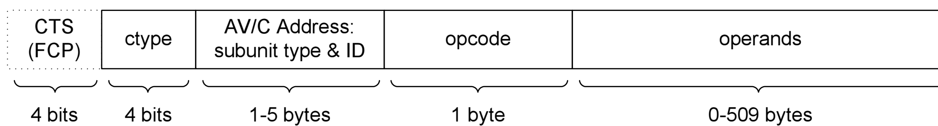

Figure 6.5: AV/C frame format, based on [23]

from, representing the ends of the pipe. The *rxvavc_fcphandler* function creates the pipes, and writes the incoming frame to the write end of the upwards stream, and calls other functions which each read a portion of the frame from the read end of the upwards stream and write to the write side of the downwards stream.

6.3 AV/C

All AV/C command frames use the basic format shown in figure 6.5. The AV/C address indicates the specific subunit (given by subunit type and ID, the instance of that type) or the AV/C unit itself that the command is addressed to. Subunit type and ID can be encoded in a single byte or each can be extended by one or two extra bytes, allowing up to 535 subunit types and 513 subunit IDs. The use of extension bytes preserves compatibility with previous versions of the specification, which were limited to 28 subunit types and 5 subunit IDs. However, parsing the address fields is complicated by checking to see whether the field is extended, and the parsing of fields later in the frame is also penalised in non-stream-based processing, as the fields are no longer at fixed offsets in the frame.

The *ctype* (command type) field indicates the type of command. The *opcode* field provides a more specific indication of the type of operation to perform or status to query, and the operands are parameters for that operation.

The *rxvavc_avchandler* function reads the command type, subunit type and subunit ID from an incoming frame. It checks these fields as required by rules 1 and 2 of the AV/C response rules [23, section 8.3]. Frames which include an unknown command type are not responded to, and commands addressed to a subunit not present in the unit will be responded to with the *not implemented* response code. It passes the frame to either the unit or the audio subunit handler function based on the AV/C address of the frame. Those functions will perform the action indicated by the command type, opcode and operands.

6.4 Unit

The *rxvavc_unit* function receives command frames addressed to the unit as a whole. It reads the frame's opcode and executes a function depending on the value of that opcode. The Unit Info, Plug Info, Subunit Info and Power opcodes are implemented. The mandatory Unit Info opcode is only valid in status commands, when it returns information about the unit or device as a whole: a subunit type to indicate the nature of the unit, and the manufacturer's company ID code. The Subunit Info opcode is likewise mandatory and only valid in status commands, and returns information about the subunits contained within an AV/C unit. It returns a group of four entries from the subunit table, which is the list of subunit type and ID pairs in the unit. The Plug Info status command returns the numbers of serial bus isochronous and asynchronous plugs and external input and output plugs that are present on the unit. Plug Info can also be addressed to a subunit. The Power opcode can be used in control, status and notify commands. In a status command, the response indicates whether the unit is powered up or not. In a control command, the controller can direct the unit to switch on or off.

rxvavc only implements status commands for these opcodes. Each function reads the operands defined for that opcode and performs the checks required for compliance with rules 3 to 6 of the AV/C response rules. The combination of ctype and opcode is checked for validity, the frame length is compared against the expected length, and any operands whose values must be from a limited set are checked. Should the command frame pass all tests, the function will return the values requested.

6.5 Audio subunit

The AV/C audio subunit [25] provides a means of controlling the audio functions of an AV/C-controlled device. An audio subunit is composed of a number of function blocks which have fixed connections to each other and the subunit's plugs. The function block types defined are:

- Selector function blocks selects one of a number of inputs to pass to a single output.
- Feature function blocks manipulate the audio flowing from its input plug to its output plug, controlling the muting, volume, balance, tone, equalisation, delay and loudness of the audio.
- Processing function blocks provide more sophisticated processing of audio. There are a number of subtypes of processing function blocks:
  - Mixer processing function blocks alter the levels of individual audio channels passing through them.
  - Generic processing function blocks can be used to provide any processing functions not possible with other processing function blocks, in a vendor-defined manner.
  - Up/down-mix processing function blocks change the number of channels in an audio source and can, for example, convert surround sound to stereo sound.
  - Dolby Pro Logic processing function blocks decode surround sound that has been encoded using Dolby Pro Logic.
  - Reverberation processing function blocks and chorus processing function blocks add reverberation and chorus effects, respectively, to audio to recreate the acoustics of certain types of venue.
- Codec function blocks decode digital audio which has been non-linearly encoded, such as MPEG audio.

Since the facilities the audio subunit provides match the functionality of the RX-V1000 reasonably closely, an audio subunit was designed and implemented. This is unlike the situation of the UPnP control server, where no suitable standard device type was available so a UPnP device and service were designed for the RX-V1000.

Design of the audio subunit for the RX-V1000

Since the RX-V1000 provides independent input selection, volume and muting controls for the main zone and zone 2, two streams of function blocks are used in the audio subunit. For each zone, a selector

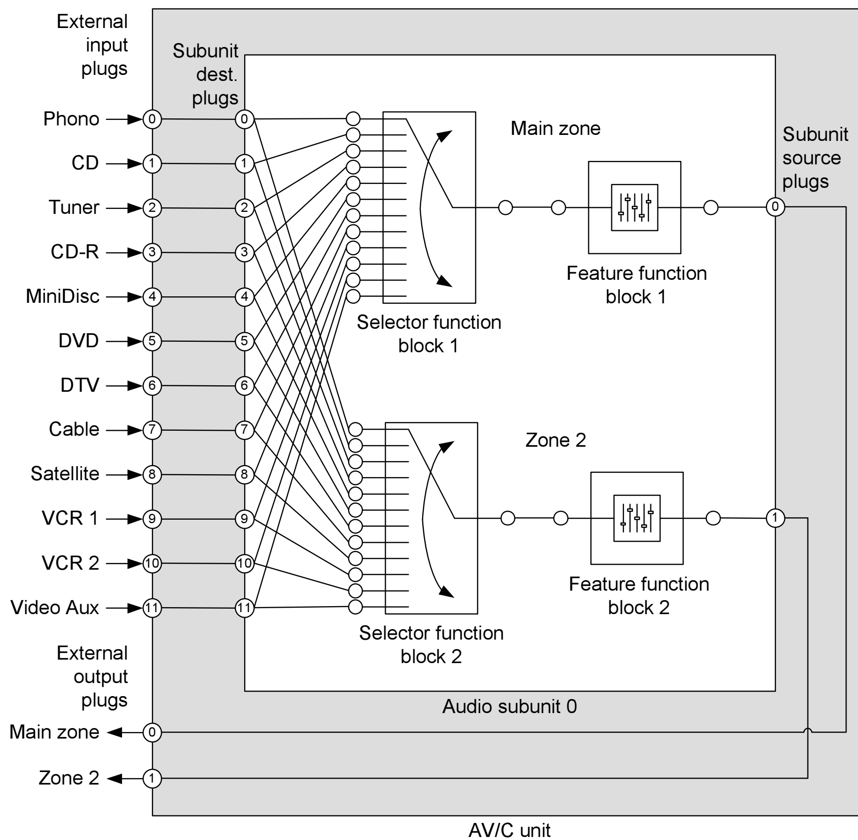

Figure 6.6: RX-V1000 AV/C unit

function block selects one of the RX-V1000's twelve inputs for output to that zone, and a feature function block provides volume and muting controls for that audio stream. The feature function block is placed after the selector function block as feature function blocks may only have a single input plug and a single output plug. Permanent connections logically connect the external input plugs on the unit to the destination subunit plugs on the audio subunit, and external output plugs to the source subunit plugs. The design of the AV/C unit and audio subunit for the RX-V1000 are shown in figure 6.6.

Alternative design options

The model described above is the one that has been implemented in *rxvavc* but aspects of the design could possibly have been done differently. Those choices are considered below.

It is possible to model the two zones as separate audio subunits, with each subunit containing one selector and one feature function block, but no clear advantage was seen in using that design for the RX-V1000.

Instead of permanent connections from the external plugs to the subunit plugs and selector function blocks within the subunit, a non-permanent connection could be made from one of the external plugs to a subunit destination plug for each of the zones. Selector function blocks were chosen because con-

trolling their selector control via Function Block commands was considered simpler than establishing connections with the Connect command.

The Speaker A and Speaker B controls on the AVR, used to select which sets of speakers the main zone output is played on, are difficult to model using the audio subunit model. They cannot be modelled using a selector function block as it can only select a single output from a number of inputs, while the speaker controls control multiple outputs. They could be modelled using non-permanent connections from a source subunit plug for the main zone to two external output plugs representing the A and B speaker sets. The presence or absence of a logical AV/C connection to an output plug would enable or disable the speaker set.

Other state variables that the RS-232 protocol provides access to (listed in appendix A) cannot be accommodated in the audio subunit model. Even though it is designed for flexibility by allowing function blocks of the different types to be assembled in any combination, there are certain features present in the device that no function block type provides. Examples include the volume banks, which switch between preset volume levels; and the on screen display control, which is a video-related control and thus naturally cannot modelled in an audio subunit. However, though they cannot be handled within the audio subunit model, they could be handled with a set of RX-V1000-specific commands using the vendor-dependent opcode.

A tuner subunit could be used to represent the RX-V1000's FM/AM tuner, based on the AV/C Tuner Model and Command Set specification [62]. This was not implemented as the audio subunit provided sufficient control functionality for the purposes of this study. Further, the RS-232 control protocol provides only limited control of the tuner's functions.

Implementing the audio subunit

The audio subunit function, *rxvavc_audio0*, receives frames addressed to the audio subunit from the AV/C handler function. The Plug Info and Function Block opcodes are handled by *rxvavc_audio0*. The Plug Info command is implemented in both the unit and the audio subunit, and when addressed to the audio subunit returns the numbers of destination and source plugs present on the subunit, as opposed to plugs on the unit.

The Function Block command is used for access to function blocks within the subunit. The operands section of the command contains common fields (the function block address and the control attribute) followed by fields specific to the type of function block addressed, as illustrated in figure 6.7. The function block address indicates the function block type and ID, in a similar way to how subunits are

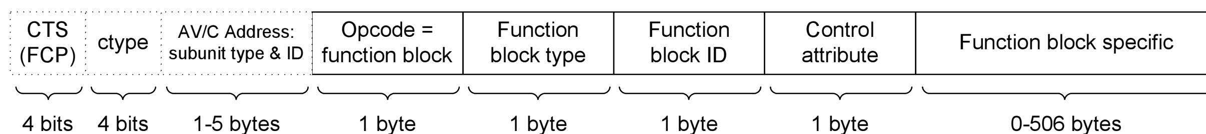

Figure 6.7: Function block command frame format, based on [25]

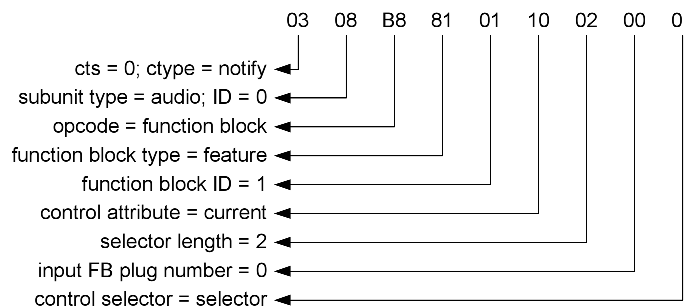

Figure 6.8: Example of an AV/C frame containing a selector function block command (byte values are in hexadecimal)

addressed within an AV/C unit. The control attribute indicates which aspect of the specific control within the function block is being controlled or queried.

The *functionblock* function reads the common function block fields, then passes the frame to the function which handles commands for the function block type the command was addressed to.

Selector function block

The *functionblock_selector* function handles commands addressed to either of the selector function blocks within the audio subunit. The command includes function block plug number and control selector fields. The control selector may only indicate the single selector control present in a selector function block. In control commands, the function block plug number field specifies which of the AVR's twelve audio/video sources should be presented in a zone, and in status responses indicates which source is currently selected. An example of a complete selector function block command is decoded in figure 6.8.

Feature function block

The *functionblock_feature* function handles commands for the feature function blocks. Feature function blocks can contain a variety of controls, which alter the audio passing through them, and the format of the operands for the function block command is specific to each control type. The feature function blocks for the RX-V1000 only implement the volume and mute controls. Since the mute control merely represents a boolean value, only the current control attribute is implemented, but the for the volume control the

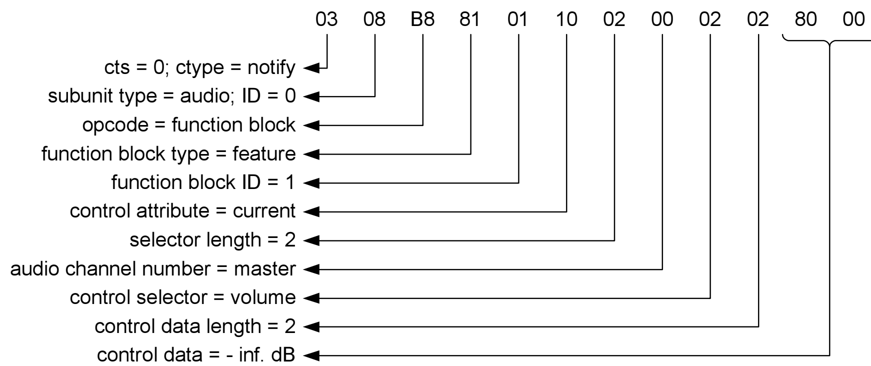

Figure 6.9: Example of an AV/C frame containing a feature function block command (byte values are in hexadecimal)

minimum, maximum and resolution control attributes are also supported. These allow a control point to set the correct ranges on the volume control that it presents to the user, and are useful in the RX-V1000 since the ranges differ between the two zones. An example of a complete feature function block command is decoded in figure 6.9.

6.6 Communicating with the device

Incoming communication from the AVR is handled by a separate thread, in the same way that the *rxvthread* thread of the *rxvupnp* program (see figure 5.16) receives AVR input for the UPnP control server. Once file handle for communication with the AVR is opened during *rxvavc* start up, the *rxvavc_rxvthread* function is run as a thread. It receives report commands, updates the record of the device's state, and checks for notifications waiting to be sent, as discussed below in section 6.7.

The audio subunit functions send commands to the AVR as they process AV/C control commands. As in the case of the SOAP thread in *rxvupnp*, a semaphore is used to notify the AV/C thread when the corresponding report command is received.

6.7 Notify commands

AV/C Notify commands are sent by a controller to an AV/C target, which immediately sends a response that indicates the current state of an aspect of the target, and, at some time in the future when the value changes, it sends a second response that indicates the new state. This allows control points to be notified of state variable changes without explicit polling, much as UPnP's eventing chapter (discussed in section 5.8) does, but, in the AV/C case, subscriptions expire after a single state change. Notify commands have

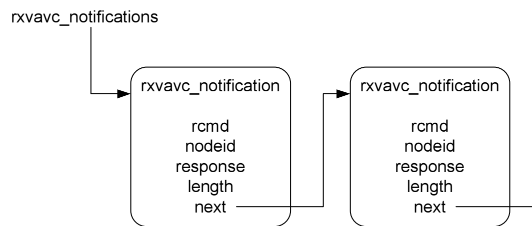

Figure 6.10: Notifications data structure

been implemented in the selector and feature function blocks and their implementation requires more complicated processing.

For all the commands handled by the function blocks, the first response to a notify command is identical to the response for a status command – except for the response code: *stable* in response to a status command, and *interim* in response to a notify command. The second response to a notify command, dispatched as soon as the control attribute in question has changed value, also has the same format, though it uses the *changed* response code and clearly contains a different value. Thus the functions that handle the function blocks implement notifies in the same way as status commands, but the FCP handler saves a copy of the response frame before transmitting it. The destination IEEE 1394 node address of the controller, and the RX-V1000 response code sent by the AVR when the value has changed, are recorded with the saved frame in a linked list, shown in figure 6.10.

Whenever *rxvavc_rxvthread* receives a report command from the AVR, it checks the response code against the pending notifications in the list. Should any match, the pre-formatted frame will be patched to include the current value that the report command gives and to set the AV/C response code to *changed*. The *rxvavc_fcpwrite* function writes the frame to the controller and the notification is removed from the list.

This implementation provides the most functionality and is relatively processor-efficient, but uses more memory than other methods would. The current implementation allows for any number of control points to request notifications of the state of any control in the subunit – they are all queued on the linked list – but the AV/C specification allows for targets that are only able to handle a single notify request, and suggests ways in which those targets could respond to multiple requests for notifications [23, section 9.4.1]. That would limit memory use to a single request, and would save the processor time taken to check through the list as each report command is received. Alternatively, a target could allow one notification per control, limiting memory usage to a statically-allocated array as large as the number of controls, but still allowing a control point to receive notification of all state changes without resorting to polling.

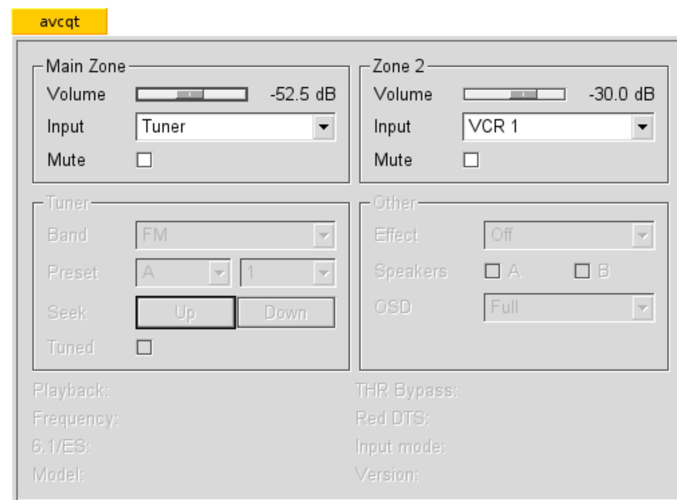

Figure 6.11: *avcqt* user interface

The use of pre-formatted responses does increase memory requirements, but very slightly: the example frames in figures 6.8 and 6.9 are nine and twelve bytes respectively, thus far smaller than the XML documents used for eventing in the UPnP control server (see section 5.8). Memory could be saved at the expense of increased CPU usage if responses were formatted immediately before being sent.

6.8 An AV/C control point

An AV/C control point, *avcqt*, was developed to test the AV/C control server. *avcqt* is based on *rxvqt*, described in section 4.4. It shares the same Qt-based user interface (pictured in figure 6.11) so that the protocol used for communicating with the device is invisible to the user. Some controls are disabled, as *rxvavc* cannot control all the functions of the RX-V1000 for the reasons discussed in section 6.5. *avcqt* uses *libraw1394* to send commands and receive responses, whereas *rxvqt* uses *librxv* functions (covered in detail in section 7.5) to communicate with the AVR.

avcqt keeps an array of pre-formatted AV/C frames. When the user adjusts a control on the user interface, the program patches the appropriate frame to set the *ctype* to Control and the value to the new value of the control, then sends the frame with an asynchronous write transaction. Frames received from the target, i.e. responses to control, status or notify commands, are compared with the pre-formatted frames to determine how to handle them. This method saves parsing each field of the frame, and is possible because the responses that AV/C targets send are almost identical to the commands.

During initialisation, *avcqt* sends a Notify command to every control of every function block. This serves two purposes: the immediate response from the target is used to set the initial values of *avcqt*'s user interface controls; and the target will notify the controller as soon as the state of any control changes.

Test	Time on the bus (in ms)				Round trip time (in ms)			
	Min	Avg	Max	Std dev	Min	Avg	Max	Std dev
Unit status command	0.494	0.505	0.519	0.008	0.525	0.539	0.552	0.009
Function block status command	0.530	0.543	0.569	0.012	0.562	0.577	0.602	0.013

Test	Delivery latency (in ms)			
	Min	Avg	Max	Std dev
Delivery of Changed response	-0.004	0.055	0.103	0.042

$n = 9$ for all tests

Table 6.1: Results of *rxvavc* performance testing

When those notifications are received, *avcqt* immediately resends the Notify command so that state change notifications continue.

The control point is designed specifically for the AV/C unit and audio subunit implemented in *rxvavc*. A more flexible control point could use the descriptors of the unit and subunits to configure itself for the target, though *rxvavc* would need to be extended to publish descriptors.

6.9 Performance of the AV/C control server

The speed with which *rxvavc* answers requests was tested using experiments similar to those used to test the UPnP control server, described on pages 54 and 65. The same Linux PCs and FireSpy400 bus analyser were used in the experiments.

The control server's ability to respond to status commands was tested. The controller used was a C program which sent pre-formatted frames to the control server and timed the time which elapsed until the response was received using the *gettimeofday* C library function. The first frame was a power status command addressed to the unit, and the second a function block status command addressed to a selector function block in the audio subunit. Each frame was sent ten times, though the result of the first run was ignored as in the UPnP tests. The times recorded are summarised in table 6.1 as "Round trip time", and the time for the same transaction as measured by the bus analyser as "Time on the 1394 bus". All the times that were recorded are listed in section D.3.

The two test frames have the same functionality as the *queryStateVariable* SOAP test, but the responses are received in as little as a tenth of the time. This is attributable to the length of time each control server spends parsing the requests and preparing the responses. In the case of UPnP, that involves parsing and preparing XML documents which are hundreds of bytes long, while comparable AV/C frames can be less than ten bytes. The function block status command takes longer to execute than

the unit status command since there are more layers of processing involved. The differences in the times as measured by the controller and the bus analyser average 0.033 ms, which is also substantially smaller than in the SOAP case. This is because the layers beneath the AV/C controller (i.e. the *raw1394* driver and *libraw1394*) perform less work than those beneath the UPnP controller, since the frame formats are simpler.

The timing of the delivery of notifications was attempted. Using the same method described in the performance testing in section 5.8, the PCs' real-time clocks were synchronised. On the control server, a C program generated an event notification as the clock ticked every new second. On the controller, a further program registered a notify command with *rxvavc*, waited for the next notification and recorded the time at which it arrived.

The results returned are summarised as "Delivery of Changed response" in table 6.1 and available in full in section D.4. However, they are obviously inaccurate – the minimum time shown in the results table is negative. This is because the times are a fraction of a millisecond when the two clocks could only be synchronised to within a millisecond's precision. However, it is clear that the notifications were delivered far quicker than those delivered via GENA over HTTP. This can be attributed to the fact that very little processing was required to send the AV/C notifications as the response were pre-formatted, and that only a single IEEE 1394 write transaction was needed to deliver the response to the controller.

6.10 Summary

An AV/C control server which allows the RX-V1000 to implement AV/C control has been developed, though without implementing the descriptor mechanism. The control server implements each layer of AV/C as a separate function. FCP was implemented on the Linux system using facilities provided by the *raw1394* driver and *libraw1394* library. The FCP handler handles commands marked with the AV/C CTS identifier by passing them to the AV/C handler function. Frames are passed between layers through an upwards and a downwards stream, implemented using UNIX pipes on Linux.

The AV/C handler function reads the address, *ctype* and *opcode* fields from the frame and, based on those values, passes them to the Unit or Audio subunit handler functions. The Unit function implements Unit Info, Plug Info, Subunit Info and Power opcodes.

The Audio subunit has been designed using two sets of selector and feature function blocks for the main zone and zone 2 to model the basic capabilities of the AVR. Alternative designs for the subunit have been considered. The Audio subunit handler function handles Plug Info and Function Block commands by passing them to specific handler functions.

Incoming status notifications from the RX-V1000 are handled by a separate thread, which also participates in the handling of Notify commands. Notify commands are received by the Audio subunit, which adds pre-formatted responses to a queue. They are dispatched by the RX-V1000 handler thread when it receives the relevant status change notification from the AVR.

An AV/C control point was developed based on *rxvqt* for testing purposes.

Performance testing experiments showed that the AV/C control server was able to respond far quicker than the UPnP control server. Status commands could be handled in approximately 0.5 ms. Changed responses to Notify commands were dispatched quicker than could be accurately timed.

The following chapter looks at the simultaneous use of multiple control protocols on a single device.

Chapter 7

Synchronisation

Having examined the implementations of the Universal Plug and Play and AV/C protocols independently, the issues involved in the simultaneous use of both protocols in a single home entertainment device are considered. In this chapter, the nature of the problem is examined, possible solutions are proposed and evaluated, an implementation of one of those solutions is discussed, and its operation tested.

7.1 The need for synchronisation

When designing a home entertainment device to be used in a home entertainment network, it is desirable that that device should be compatible with the greatest range of control points, so that it can be marketable to as many consumers as possible. Those control points are likely to use different control protocols for a number of reasons:

1. differing ages of the equipment, since older equipment may have been purchased prior to the standardisation of other protocols;
2. the different strengths and weaknesses of different protocols mean that a device may have been designed to use a particular protocol that it is best-suited to; and
3. home entertainment systems include components from a number of manufacturers, particularly as products from PC vendors and from consumer-electronics vendors meet in a converged home entertainment network, since those groups will be allied to different industry groups (e.g. the UPnP Forum is lead by Microsoft and other vendors in the PC market while the 1394 Trade Association is comprised of mostly consumer-electronics companies).

Thus, it is a goal for new devices to be able to support two or more control protocols, in order to interoperate with as many other devices on the network as possible.

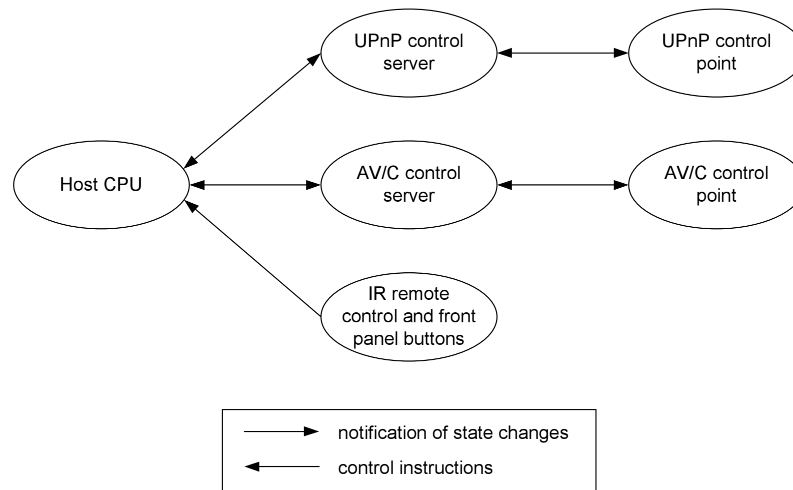

Figure 7.1: Components involved in the control of a device using two protocols

While a device could allow the user to select one protocol to use (in the case of the *rxvavc* and *rxvupnp* control servers, each is able to communicate with the AVR by itself, so the device would only need to provide a mechanism for the user to pick one program to run), the user would have to switch between them if the home entertainment system included control points using different protocols. This inconvenience would defeat the purpose of remote control, that is to make the user's interaction with the device easier. Therefore devices supporting multiple protocols should be able to support them at the same time.

It will not necessarily be easy for a home entertainment device to incorporate multiple control servers. Many devices may only have a single interface for control (such as the RX-V1000, which has a single RS-232 port) and that interface may use a simple communications protocol intended for use by a single controller. Further, it makes sense to design the host CPU with one control interface, to allow the CPU to concentrate on the primary functions of the device, and offload all network control functions to another CPU. Thus the set of control servers requires a means of time-multiplexed access to the communication channel with the host device.

7.2 A model of the problem

A scenario in which a home entertainment device supports the AV/C and UPnP protocols is modelled in figure 7.1. The components of the system are the AV/C and UPnP control points; the AV/C and UPnP control servers running on the network processor CPU; the host device itself; and other controllers, such as the infra-red remote control and the front panel buttons.

Records of the state of the device (that is, the values of the set of state variables) are stored in various parts of the system. At the very least, it must be stored in the device itself (on the far left of the diagram), and in the control points, displayed in the user interface (on the the far right of the diagram). Copies can be held by system components in between, such as the control servers. A number of sources of state changes can be identified: external causes (e.g. a tape runs out, a timer expires, or a radio station goes out of tune), the remote control and front panel, and any of the control protocols.

Within the system, information flows in two directions: control directives and state variable queries from the control points, through the services, to the device; and notifications of state changes from the device, through the services, to the control points. These correspond to leftwards and rightwards arrows in the diagram. Thus solutions must address two aspects: synchronising the state of the device across all components of the system, and managing control of the device.

7.3 Desired attributes of possible solutions

In order to assess possible solutions, desirable attributes should be identified.

Synchronisation of state should aim for consistency of state across system components, so that the state shown on each component is identical to that on every other component; accuracy, so that the state shown on each component is as close to the actual state of the device as possible; and low latency, so that state changes are reflected in every part of the system as quickly as possible. All of these serve to reduce the user confusion that would arise if different control points were to display different values at the same time – only one could be correct, but the user would not know which was correct and which was not. Considered in terms of the model described above, for each point at which state changes enter the system there must be a reliable path for state change notifications to propagate to each copy of the state, and those paths should operate as quickly as possible.

Records of the device state are mandatory at the points identified above, but between those points there should be as few copies of the device state as possible, primarily to limit the potential for discrepancies between those copies, but also to save the memory and CPU resources that would be required to maintain them.

Control instructions need to be managed so that, at a minimum, simultaneous commands from different control servers cannot overlap on a serial communications channel to the host CPU. At a higher level, it is desirable that the implications of commands should not conflict. For instance, it takes a number of seconds for a VCR to eject a tape. Were a play command to be sent while a tape was being ejected, it could not be completed.

Since the solutions must be implemented within an embedded system, the embedded development concerns found in section 5.1 must not be forgotten.

7.4 Possible approaches to synchronisation

The synchronisation problem can be solved in many ways. Some possible approaches are discussed below and evaluated against the above-mentioned criteria, using the Yamaha RX-V1000 AVR as an example device.

Polled access to the device

The simplest way to share the device would be for the control servers to take turns to access the device. Each control server would open the serial port, poll for the status of the AVR and and send commands as necessary, while holding a mutex to ensure that only one control server accesses the serial port at one time. While easy to implement, this approach has a number of disadvantages.

Every time a control server begins communication with the RX-V1000, it must send the *ready* command to synchronise communications. The transmission of the *ready* command and the RX-V1000's *configuration* response would take a significant length of time: the *ready* command is five bytes and *configuration* 44 bytes, so sending them at the 9600 bps 8-N-1 serial rate used by the RX-V1000 will take at least $\frac{5+44}{9600/(8+1)} seconds = 45ms$. Even when assuming that the response can be sent immediately, establishing communications adds substantial overhead.

The configuration response includes the current values of every state variable on the AVR. Listing each variable places unnecessary load on the host device's CPU; parsing the response likewise taxes the processor that the control servers run on.

A polling system introduces delays in updating the status of the AVR. When a change in status occurs, some time will elapse between the event and the control server learning about it on its next poll, in addition to the delay as the notification is transmitted across the network to the control point. As these delays lengthen, so the chances of inconsistent state being displayed on UPnP and AV/C control points increase. The option is even less attractive considering that the RX-V1000 is capable of sending immediate notification of status changes.

An implementation of this approach could provide reliable paths for synchronisation of state, though the delays would be unacceptably long. Control servers would not need to keep records of the device's state, since they could answer state variable queries by issuing the *ready* command and sending the value for the variable given in the *configuration* response, and handle control instructions by passing the

response received from the device on to the control point. Provided that the control servers have exclusive access to the device during communication with the device, there is no chance that serial communications could overlap.

Simulating virtual devices

A second way to share control would be to implement a sharing layer, which presents a simulation of the AVR's binary command channel to each of the control servers while communicating with the AVR itself. Status notifications from the AVR could be repeated to the control servers without the need for polling. If a command were issued by one server while a command from the other server was in progress, the sharing layer could buffer until the current command was completed. An advantage of this approach is that each control server is able to control the AVR on its own without modification.

A more complex variation would involve the complete simulation of the device. Commands received from control servers would be applied to a model of the state of the device, then passed to the actual device. While this could allow the sharing layer to detect the impact of conflicting commands, it requires that it understand the state of the device in such fine detail that it is able to predict how the device will react to the command when adjusting the internal model. Being able to predict the device's response would allow it to send the response to the control servers immediately, rather than waiting for the response from the device.

This approach can provide reliable paths for synchronisation, and because event notifications are made available to all control servers immediately, the notification delays inherent in the previous approach would not occur. For control servers to answer state variable queries, they would either need to issue the ready command each time as in the previous approach, or maintain a copy of the device state from an initial configuration response and update it as each event notification is received. As all commands to the device pass through the sharing layer, it is able to ensure that only one command is transmitted at a time and any others that arrive can be delayed.

High-level function interfaces

A third option is to implement a layer of abstraction. This could provide a higher-level interface to the AVR to the control servers. This layer would translate from the AVR's binary commands to higher-level language function calls to the control servers. Communication with the AVR would be as in the previous approach, and likewise could include a simulation of the state of the device.

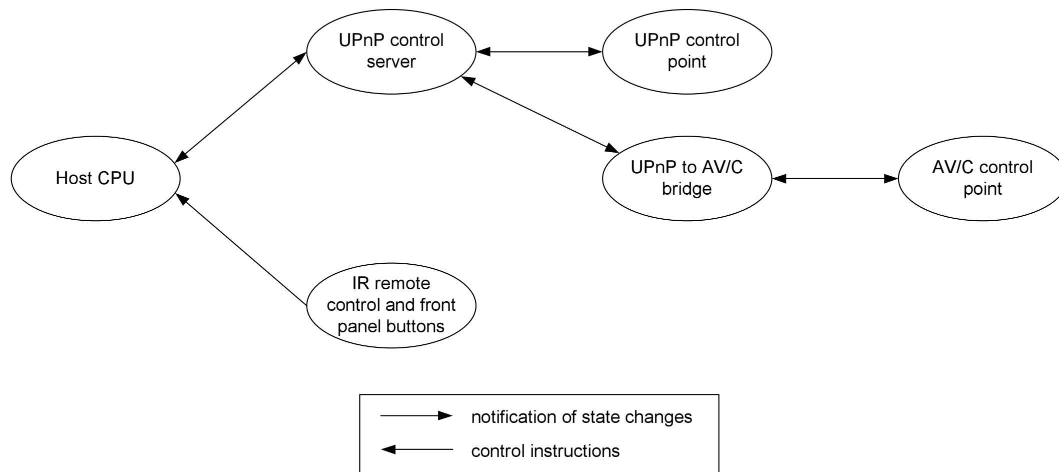

Figure 7.2: Using a protocol-conversion bridge for multiprotocol control

This approach's advantage is that control servers would not need to be as familiar with the AVR's protocol and thus could be more readily adapted to other devices. But it would be substantially more demanding on the CPU, and would involve the design of another functional model of the device.

Since this approach only differs from the last approach in the manner in which the abstraction layer communicates with the control servers, concerns of synchronisation, numbers of copies of the state and control arbitration are the same as those in that approach.

Bridging

A protocol-conversion bridge could be used to provide support for multiple protocols. The control server for AV/C, for instance, would be implemented as a client of the UPnP control server. It would receive commands from control points in the AV/C protocol, translate them to UPnP and pass them on towards the AVR. When the bridging control server receives an event notification subscription request, it would need to open a subscription using the other protocol itself. In that manner, it would act as a protocol transaction proxy. The bridge could be implemented within the AVR, or as a separate unit on the network providing protocol bridging for all other devices on the network. An example of a UPnP-to-AV/C bridge is shown in figure 7.2.

Bridging would not be as efficient as the approaches above, since commands and notifications would need to be translated twice: once from the device's native protocol to the first control protocol, and once from that protocol to the second. Mismatches between the control capabilities and device models of the control protocols could limit the functionality of the protocol implemented using the bridge, as all functions would have to be expressed in the least capable protocol at one stage.

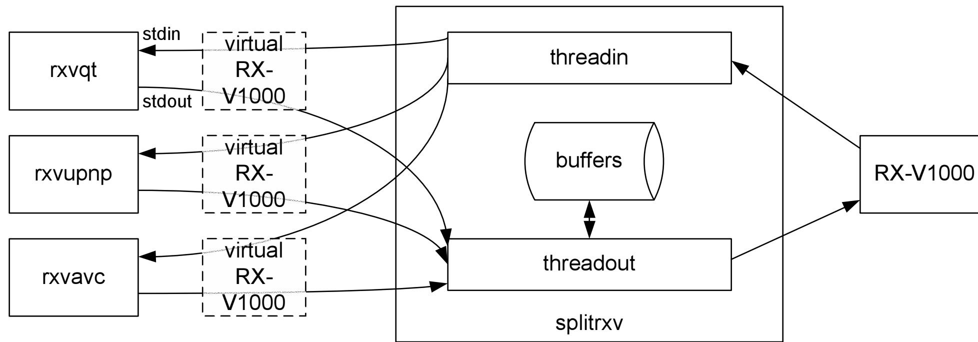

Figure 7.3: *splitrxv* data flows

This approach lengthens the paths that commands and events take, but could be implemented reliably. A protocol bridge need not increase the number of copies of state, as it could query the other control server to answer queries. Only one control server is able to communicate with the device, so it would be responsible for ensuring that commands do not overlap or conflict.

7.5 Implementing a virtual device simulator

Since the polled access and function interface approaches were regarded as too inefficient, and the bridging approach was considered more appropriate for use in a separate adapter device, only the virtual device simulation approach was implemented and tested. The implementation is described below, and test results are discussed in section 7.6.

The *splitrxv* program

splitrxv is a simple layer which provides virtual RX-V1000s to a number of control servers, all based on a single real RX-V1000. It operates by opening the serial port to communicate with the real AVR, then starts an arbitrary number of programs (usually a combination of the *rxvavc* and *rxvupnp* control servers and the *rxvqt* controller) as child processes, opening a pair of pipes to each child's *stdin* and *stdout* for communication with that child. This creates the illusion for those programs that the AVR is connected on *stdin* and *stdout*. Figure 7.3 shows the virtual devices and how *splitrxv* creates them.

Within *splitrxv*, two threads pass data in each direction. *threadin* performs a blocking read on the serial port, and immediately writes the incoming bytes to all of the pipes to the control servers' *stdin*s. Because the data written to UNIX pipes is buffered by the kernel, *splitrxv* does not use a more complex means of delivering the data to the children. On systems where UNIX pipes are not available, data could be delivered in the same way that *genapub* delivers event messages to its threads, as discussed in

section5.8. Since each program is able to buffer partial reports from the RX-V1000, *threadin* need not modify the data streams in any way.

threadout controls the flow of data in the opposite direction, from the control servers to the device. It uses the *select* system call to await the arrival of data from any of the pipes attached to the child programs' *stdout*s. Data received is added to a buffer for each program and afterwards, all complete commands that are in the buffer are passed to the serial port. Thus as soon as a control server sends a complete command to *splitrxv*, it is passed on, but should a program send an incomplete command, it will be buffered until all of it is received. Recognising complete commands requires that *threadout* be able to understand the serial protocol. In the case of the RX-V1000 serial protocol, every command is followed by an ASCII ETX (end of text) byte, which makes the task simpler. Depending on the times at which data is received from the children, it is possible that commands from different programs will be interleaved. In the case of the RX-V1000, this is acceptable since all commands are atomic. For devices where commands depend on a preceding command, *threadout* would need to examine the contents of the commands to determine whether it was safe to allow other commands to be sent yet.

splitrxv represents a minimal implementation of virtual devices. No attempt is made to understand the state of the device, or the implications of the commands sent to the actual device. Each virtual device will only reply to a command from a control program once *threadin* has received the response from the actual device.

The *librxv* library

Since the task of understanding and speaking the RX-V1000 communication protocol must be performed by each control protocol, reuse of the protocol-specific functions can simplify the implementation of the control servers, and possibly make them less device-specific. The *librxv* library provides functions to support communication with an RX-V1000. It could be extended to support other similar Yamaha AV receivers, such as the RX-V3000 or RX-V3200, which use a similar protocol but have a different set of capabilities, but unfortunately these devices were not available for testing.

The *rxv_fmt* functions (listed in table 7.2) take parameters given as constants defined in *librxv* and prepare binary strings to be sent to the AVR representing initialisation, remote control, reset and system commands. The constants for each state variable are listed in appendix A. The *rxv_scan* functions extract the parameters from a binary string received from the AVR. The client program can determine which of the types of commands it has received by comparing the first byte against the *RXV_START* constants, then calling the appropriate function to decode it. While this approach requires that the client program understand some of the protocol, it simplifies the *librxv* function interface. The functions operate on RX-

Function name	Arguments	Return	Description
Parsing functions: AVR-to-controller messages			
rxv_scan_configuration	char *s, struct rxv_configuration *config	int	Extracts configuration data from a configuration message.
rxv_scan_report	char *s, unsigned int *typ, unsigned int *grd, unsigned int *rcmd, unsigned int *rdat	int	Extracts typ, grd, rcmd and rdat from a report command.
rxv_scan_text	char *s, unsigned int *rcmd, char *ddat	int	Extracts rcmd and ddat from a text report command.
Parsing functions: Controller-to-AVR messages			
rxv_scan_ready	char *s, unsigned int *tout	int	Extracts tout from a ready command.
rxv_scan_remotecmd	char *s, unsigned int *cmd	int	Extracts cmd from a remote command.
rxv_scan_reset	char *s	int	Checks for a valid reset command.
rxv_scan_systemcmd	char *s, unsigned int *cmd, unsigned int *data	int	Extracts cmd and data from a system command.
Formatting functions: Controller-to-AVR messages			
rxv_fmt_ready	int timeout	char *	Formats a ready command to begin communication with the RXV.
rxv_fmt_remotecmd	int cmd	char *	Formats an operation command. Takes an RXV_OP_* command.
rxv_fmt_reset		char *	Formats a reset command to set all settings to factory defaults.
rxv_fmt_systemcmd	int cmd, int data	char *	Formats a system command. Takes an RXV_CMD_* command and an RXV_ARG_* argument.
Formatting functions: AVR-to-controller functions			
rxv_fmt_configuration	struct rxv_configuration *config	char *	Formats a configuration report.
rxv_fmt_report	int typ, int grd, int rcmd, int rdat	char *	Formats a report command.
Utility functions			
rxv_modelname	char *code	const char *	Translates a model code from the configuration struct to a model name.
rxv_print_configuration	struct rxv_configuration *config	void	Dump a configuration struct.
rxv_reset_configuration	struct rxv_configuration *config	void	Set a configuration struct to defaults.
rxv_volume_ftoi	float v, int zone	int	Convert a volume in dB to its value in integer format for the given zone.
rxv_volume_itof	int v, int zone	float	Convert a volume in integer format to its value in dB for the given zone.

Table 7.2: *librxv* functions

V1000 format commands stored in strings, rather than performing device I/O themselves, to allow the library to be independent of the transport used to communicate with the AVR. *librxv* includes descriptive text strings for various state values in constant arrays so that client programs use the same names for states.

7.6 Testing the system

The combination of the AV/C and UPnP control servers and *splitrxv* was tested to verify that control instructions and change notifications flow across the system. The tests aimed to provide automatic and systematic validation of control and notification delivery throughout the system, rather than measure performance, as each control server's performance was measured in isolation in the previous chapters. Based on the model of section 7.2 and requirements of section 7.3, the tests verified in a practical manner that changes entering the system from any source notified all event subscribers and updated all copies of the device state.

A Perl program, *verify.pl*, oversaw the tests using a simulated RX-V1000 and a set of AV/C and UPnP clients on the device- and client-sides of the control servers respectively. Both the control server and testing software were run on a single PC to give *verify.pl* access to both sides of the control server system.

The components and interactions of the test system are diagrammed in figure 7.4. All of the programs shown in the Control Servers section were under test, the programs in the rest of the diagram performed the tests.

The RX-V1000 simulator

A software simulator of the RX-V1000, named *simrxv*, was developed. The simulator maintains a model of the AVR, and accepts commands in the RX-V1000 format and responds in the manner that the AVR itself would. Thus it is capable of communicating with the *splitrxv* sharing layer, or directly with the *rxvavc* or *rxvupnp* control servers, or the *rxvqt* controller. In the test scenario, *simrxv* provides a precisely controlled and monitored source of event notifications to the control servers and destination for commands from them.

The *simrxv* program provides a simple command line interface through which the values of state variables can be queried and changed. The command line interface is equivalent to the front panel of the real AVR which displays its state on an LCD display and where its buttons change the AVR's state.

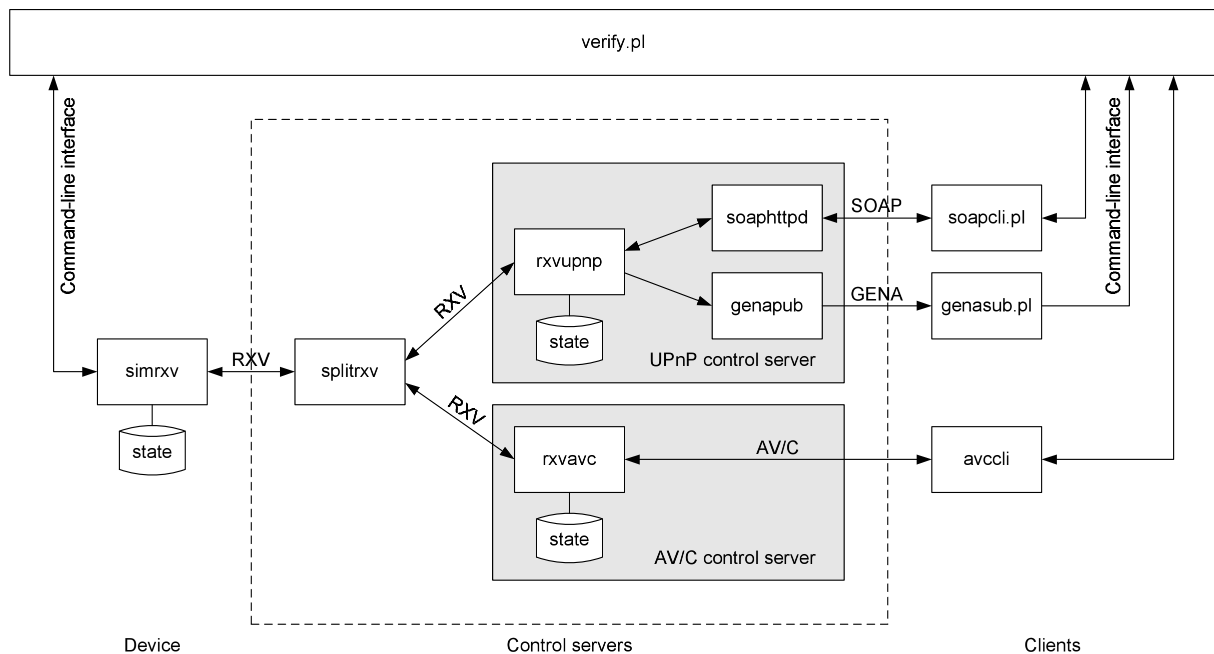

Figure 7.4: Control server testing system

simrxv starts a program and communicates with it using the RX-V1000 serial format through its *stdin* and *stdout*. Thus the RS-232 port on the RX-V1000 corresponds to a pair of UNIX pipes in *simrxv*.

For each control command received through either interface, *simrxv* updates its state model, sends the appropriate RX-V1000 report command, and prints the name and new value of the variable on the command line interface. Though the RX-V1000 format does not provide a means to query state variable values, queries received on the command line are answered based on the values in the state record.

A real RX-V1000 could be used for testing, but *verify.pl* would not be able to control it itself, so a human test operator would need to press front panel buttons at the appropriate times and confirm that the AVR's state had changed. *simrxv* allows fully automatic testing of the control server software, reducing the time taken to run the tests and eliminating the possibility of human error.

AV/C and UPnP clients

avccli is a simple AV/C client which *verify.pl* uses to send commands and state variable queries to the AV/C control server and receive event notifications from it. *soapcli.pl* is a SOAP client capable of issuing commands to the UPnP SOAP server and querying state variable values from it on behalf of *verify.pl*. *genasub.pl* opens a subscription with the UPnP GENA server and reports the event notifications that it receives to *verify.pl*.

All three clients use the same command line interface as *simrxv*. For example, a volume change can be effected by entering “volume=-30.0” at the command line of any of *simrxv*, *avccli* or *soapcli.pl*. When they receive notification of that event, *simrxv*, *avccli* and *genasub.pl* will print the same string.

Test method

verify.pl performs the tests by sending a command via the *simrxv*, *soapcli.pl* or *avccli* command line. The SOAP and AV/C protocols return the new value of the state variable after executing the command, so when sending a command via SOAP or AV/C, *verify.pl* checks that the same value is returned. It then checks that event notifications are received from all of GENA, AV/C and *simrxv* by reading from the *genasub.pl*, *avccli* and *simrxv* command line interfaces. Finally, it checks that all copies of the device state have the new value by sending a state variable query to the SOAP and AV/C servers and *simrxv*. Since all four interfaces that *verify.pl* has to the control system use the common format described above, it merely compares the string it receives with the command string it originally sent in all the checks it performs.

verify.pl sends a random command through all of the entry points into the system and checks that the change propagates through all paths through the system, and performs a number of iterations of the test.

Results

Initial testing revealed some bugs in the control servers, but once those were corrected, *verify.pl* found that all commands propagated across the system successfully. An example of output from a test run is given in figure 7.5. The run began by sending an instruction to set the volume to -53 dB using the SOAP protocol, and checked that the correct response was returned by *soapcli.pl*. Then the three event notification subscribers (i.e. *genasub.pl*, *simrxv* and *avccli*) where checked to see whether they have received the notification. The three copies of the device state were checked for consistency using *avccli*, *simrxv* and *soapcli.pl*. Subsequently, the volume level was set to -75.5 dB and -56.5 dB by instructing *simrxv* and *avccli*, respectively, to initiate the change then performing the same set of checks, though in different orders. For every check done, the volume level was compared with the expected volume level, the “OK” indicates that they were as expected.

7.7 Summary

This chapter has considered the simultaneous use of multiple control protocols to control a device. There are practical reasons why it would be an advantage for manufacturers to support multiple control proto-

```
test run 0
sent "volume=-53.0" to soapcli
recv "volume=-53.0" from soapcli; OK
recv "volume=-53.0" from genasub; OK
recv "volume=-53.0" from simrxv; OK
recv "volume=-53.0" from avccli; OK
recv "volume=-53.0" from avccli; OK
recv "volume=-53.0" from simrxv; OK
recv "volume=-53.0" from soapcli; OK
sent "volume=-75.5" to simrxv
recv "volume=-75.5" from genasub; OK
recv "volume=-75.5" from avccli; OK
recv "volume=-75.5" from simrxv; OK
recv "volume=-75.5" from simrxv; OK
recv "volume=-75.5" from avccli; OK
recv "volume=-75.5" from soapcli; OK
sent "volume=-56.5" to avccli
recv "volume=-56.5" from avccli; OK
recv "volume=-56.5" from genasub; OK
recv "volume=-56.5" from simrxv; OK
recv "volume=-56.5" from avccli; OK
recv "volume=-56.5" from soapcli; OK
recv "volume=-56.5" from avccli; OK
recv "volume=-56.5" from simrxv; OK
```

Figure 7.5: Test results from *verify.pl*

cols in a single device, and they should be supported simultaneously. However, this is not necessarily easy and calls for time-multiplexed access to the host CPU in the device.

An examination of a model of the scenario revealed that solutions to the problem must synchronise the state of the device across the system and manage the control of the device. Synchronisation can be achieved by ensuring that fast, reliable paths exist for state change notifications to reach all copies of device state whenever a state change occurs from any source. Control commands must not be allowed to overlap, and should be kept from conflicting.

Four possible methods of synchronisation were discussed. A polling system would be simple to implement, but might not be able to keep all parts of the control system synchronised. Virtual copies of the device could be simulated, one for each of the control servers to interact with. A high-level function interface could provide a level of abstraction from the device's native control protocol. Protocol-conversion bridging could allow a control server to be implemented as a client of another control server.

The *splitrxv* program implements the virtual device simulation approach for the RX-V1000. *librxv* provides a common library of functions for formatting and parsing communications to and from the AVR.

A testing system has verified that commands and event notifications flow through the control servers reliably. *verify.pl* uses an AVR simulator and AV/C and UPnP clients to test every path from the start of an event to event notification delivery, and checks each copy of the device state.

Chapter 8

Conclusion

This work has considered the future of home entertainment systems integrated into a home network, and how a home entertainment device on such a network can be controlled using multiple control protocols.

It is expected that the deployment of home networks will increase; more homes will have networks, and those networks will link more devices around the house, by uniting currently isolated clusters of devices into a single cross-platform network. These networks can be used for data-centric, multimedia-centric, home management and value-added service applications. The IEEE 1394 bus provides an ideal network technology for deployment in homes, as it is easy to use, can carry a wide range of applications, is able to cover the home using different media including wireless networks, provides abundant bandwidth, and excels in the delivery of digital media streams.

As home entertainment devices incorporate network connections, home entertainment systems will merge with the home network to form home entertainment networks. Home entertainment networks will be comprised of home entertainment devices around the house which are able to share digital media across the network in a range of multimedia-centric applications. In order to enable effective use of the capabilities of those home entertainment devices, they must be able to be controlled from other devices on the network.

Networked control involves a controller and a controlled device, communicating using a common control protocol to send instructions from controller to controlled devices, and device status information from controlled device to controller. A number of such control protocols have been specified by different industry groups. Amongst them are:

- UPnP specifies a network-independent, open standards-based device architecture to allow easy connection, configuration and control of networked devices. UPnP makes use of Internet protocols such as TCP/IP, HTTP and XML. The six chapters of the UPnP Device Architecture cover

discovery of devices on the network, description of devices and the services they provide, control of devices, status-change event notification, and user interface presentation.

- AV/C is designed for control of consumer electronics equipment using the IEEE 1394 bus. AV/C makes use of the IEEE 1394-specific FCP protocol for all communication between the controller and the control server. Each device is represented as an AV/C unit, and a number of subunits within it represent areas of the device's functionality. A range of subunit types, including media sources, media processors and a user interface, have been standardised by the 1394 Trade Association.
- HAVi is an architecture based on a collection of software elements. Elements such as the IEEE 1394 communication abstraction, Messaging Service, Registry and Event Manager, provide services to a set of Device Control Modules and the Functional Component Modules within them that represent devices and their functionality. User interfaces can be provided using the DDI protocol, or by a Java application running on the controller.

Since different controllers use different control protocols, manufacturers of controlled devices can increase the number of controllers with which their products will be compatible by designing their devices to support multiple control protocols. This expands the number of consumers to which they can market their product.

To test the feasibility of implementing the concept of multiprotocol control, a prototype network-connected home entertainment device which supports multiple control protocols was developed. The IEEE 1394 bus was chosen as the network to which the prototype would be connected, as it is suitable for home networking and a foundation of the AV/C control protocol. An existing home entertainment product, the Yamaha RX-V1000 AV receiver, was selected as the basis of the prototype device. The RX-V1000 was chosen because it can be controlled using a simple RS-232 serial control protocol. The control protocol was understood and the capabilities of the device were identified, leading to the development of *rxvqt*, which allows a PC to control the AVR via the serial port.

In order to connect the AVR to the IEEE 1394 bus, an embedded computer must interface with the AVR and the bus, and run the control servers. This was done using, at first, a Digital Harmony DHIVA and, later, a PC running Linux. It is expected that a future commercial version of the system would include an small embedded computer system within the casing of the AVR.

The UPnP device architecture was implemented for the RX-V1000. As there was no standardised device type that matched the AVR, a non-standard UPnP device and service were designed for it. The protocols were implemented as separate programs, each communicating with the client independently and communicating with the RX-V1000 via a controlling program. The *ssdpd* program uses SSDP to

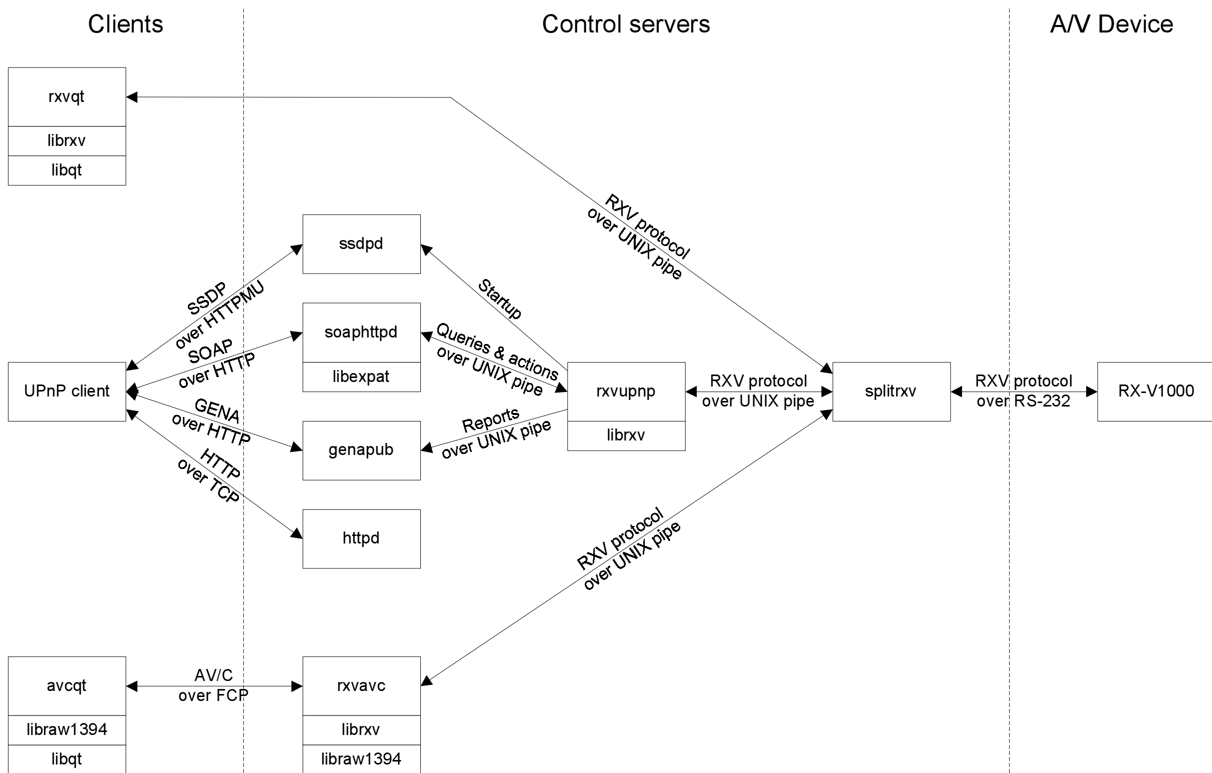

Figure 8.1: Software components of the control servers

advertise the device and the service it provides to the rest of the network. The device descriptions are served using an HTTP server. The *soaphttpd* program receives control directives and status queries in SOAP messages received via HTTP connections. *genapub* is a GENA event notification publisher which notifies subscribed controllers of status changes on the AVR. The presentation page for the device makes use of the Mozilla web browser's SOAP classes to allow the device to be controlled without the need for drivers or a specific control application.

An AV/C control server was also developed for the RX-V1000. Support for the FCP protocol was implemented using the *libraw1394* library. The functionality of the RX-V1000 was modelled using an audio subunit consisting of two selector function blocks and two feature function blocks. The various layers of the protocol were implemented as functions in the *rxvavc* program. Communication between the layers was via an upwards pipe from which functions read bytes of the command frame, and a downwards pipe to which they wrote bytes of the response frame. Notify commands were handled by pre-formatting a response then awaiting the report from the AVR, which triggers that response. Performance testing showed that the AV/C control server could respond to state variable value queries in as little as a tenth of the time that the UPnP control server took, while event notifications were delivered too quickly to be measured accurately.

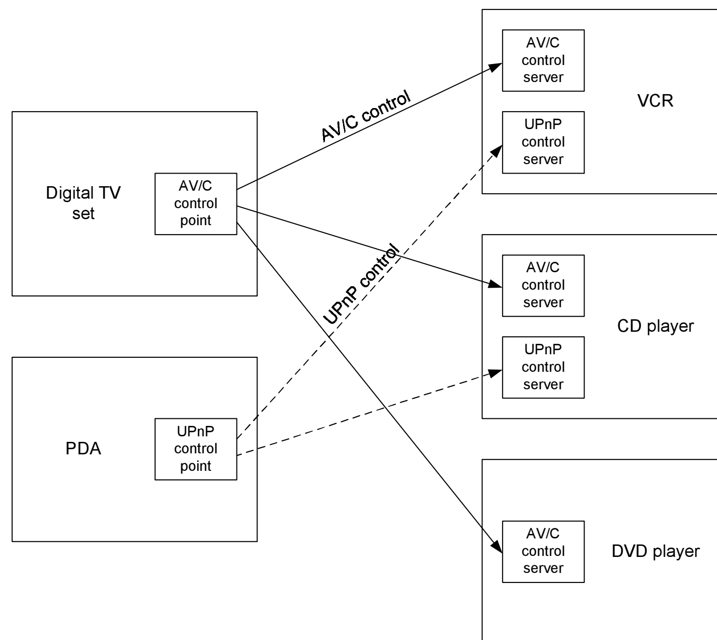

Figure 8.2: A home entertainment scenario made possible by multiprotocol control

Having developed independent control servers for UPnP and AV/C, the issues involved in using them simultaneously were examined. Criteria for evaluation of possible solutions were identified. Approaches to synchronisation involving polled access to the device, the simulation of virtual devices, the use of high-level function interfaces, and the use of a protocol-conversion bridge were considered. The *splitrxv* program implements the virtual device simulation method, and *librxv* handles the RX-V1000 serial control protocol for the control servers. The interactions of all the control server programs are shown in figure 8.1, a more detailed version of figure 1.1 which incorporates the UPnP and AV/C sections shown in figures 5.1 and 6.1.

Once all components of the multiprotocol control server system were developed, they were tested to determine whether the system operated correctly. *verify.pl* performed the testing by sending control instructions through each means of control, then checking that notifications were delivered over all paths through the system and querying all copies of device state to see that they had the correct values. The successful results of the tests showed that all components of the system operated correctly.

Thus the feasibility of the use of the IEEE 1394 bus in home networks, networked control of home entertainment devices, and the simultaneous use of multiple control protocols on a single device has been demonstrated.

A home entertainment system which includes devices capable of multiprotocol control will make the hypothetical control scenario shown in figure 8.2 possible. The digital TV set and PDA are used as controllers: they display the user interfaces of the VCR, CD and DVD players on their screens and allow

the user to issue instructions to the controlled devices. The TV is capable of controlling AV/C devices while the PDA implements a UPnP control point. The VCR and CD player can both be controlled using UPnP and AV/C, thus they can be controlled by the TV and the PDA, allowing the user to use whichever is most convenient for him or her. Since the DVD player only supports AV/C, it cannot be controlled from the PDA.

Appendix A

State Variable Model of the RX-V1000

The table below lists and describes the variables which were identified in the RX-V1000 based on information in the RS-232 control specification [34] and the owner's manual [36]. The Type column indicates the data type used to represent the value; Access, whether the variable can be read and/or written; Cfg, whether or not the value is reported in the configuration command; Values, the *librxv* constants that represent the values the variable can take; Write, the commands that can be used to change the variable; and Report, the remote command type sent by the AVR when the variable changes.

Name	Type	Access	Cfg	Values	Write	Report
typ	string	R	yes	rxv_modelname()		
	Model identification					
ver	string	R	yes			
	Firmware revision ID					
system	enum.	R	yes	rxv_desc_system[]		
	Indicates whether the system is busy or idle					
power	boolean	R/W	yes		RXV_OP_POWER_*	RXV_RCMD_OP_POWER
	Power status of the unit					
input	enum.	R/W	yes	RXV_RDAT_OP_INPUT_*, rxv_desc_input[]	RXV_OP_INPUT_*, rxv_op_input[]	RXV_RCMD_OP_INPUT
	Currently selected input for playback and/or display in the main zone					
sixchinput	boolean	R/W	yes		RXV_OP_SIXCHINPUT_*	RXV_RCMD_OP_INPUT & RXV_RCMD_OP_INPUT_SIXCHINPUT
	When enabled, input is taken from the 6 channel external decoder input jacks rather than that selected by *input*					
inputmode	enum.	R/W	yes	RXV_RDAT_OP_INPUTMODE_*, rxv_desc_inputmode[]	RXV_OP_INPUTMODE_*, rxv_op_inputmode[]	RXV_RCMD_OP_INPUTMODE
	Selects an audio input if the selected source provides multiple digital and/or analogue audio streams					
mute	boolean	R/W	yes		RXV_OP_MUTE_*	RXV_RCMD_OP_MUTE
	Mutes audio output in the main zone					
zone2input	enum.	R/W	yes	RXV_RDAT_OP_INPUT_*, rxv_desc_input[]	RXV_OP_ZONE2INPUT_*, rxv_op_zone2input[]	RXV_RCMD_OP_ZONE2INPUT
	Currently selected input for playback and/or display in zone 2					

Name	Type	Access	Cfg	Values	Write	Report
zone2mute	boolean	R/W	yes		RXV_OP_ZONE2MUTE_*	RXV_RCMD_OP_ZONE2MUTE
	Mutes audio output in zone 2					
volume	integer	R/W	yes	0-199 (0=-inf, 1=-99 dB, 2=-98.5 dB, 199=0 dB)	RXV_CMD_VOLUME	RXV_RCMD_OP_VOLUME
	Master volume control for the main zone					
zone2volume	integer	R/W	yes	0-80 (0=-inf, 1=-79 dB, 80=0 dB)	RXV_CMD_ZONE2VOLUME	RXV_RCMD_OP_ZONE2VOLUME
	Master volume control for zone 2					
program	enum	R/W	yes	RXV_RDAT_OP_PROGRAM_*, rxv_desc_program[], rxv_rdat_program[]	RXV_OP_PROGRAM_*, rxv_op_program[]	RXV_RCMD_OP_PROGRAM
	Selects the sound field program applied to the audio by the digital signal processor					
effect	boolean	R/W	yes		RXV_OP_PROGRAM_*	RXV_RCMD_OP_PROGRAM
	Enables or disables the digital signal processor's sound field processing					
eskey	boolean	R/W	yes		RXV_OP_ESKEY_*	RXV_RCMD_OP_ESKEY
	Enables or disables the internal Dolby Digital Matrix 6.1 and DTS ES decoder					
osd	enum.	R/W	yes	RXV_RDAT_OP_OSD_*, rxv_desc_osd[]	RXV_OP_OSD_*, rxv_op_osd[]	RXV_RCMD_OP_OSD
	Controls the amount of information shown in the on-screen display					
sleep	enum.	R/W	yes	RXV_RDAT_OP_SLEEP_*, rxv_desc_sleep[]	RXV_OP_SLEEP_*, rxv_op_sleep[]	RXV_RCMD_OP_SLEEP
	Sets the sleep timer to shut the unit down after a certain length of time					
tunerpage	enum.	R/W	yes	RXV_RDAT_OP_TUNERPAGE_*, rxv_desc_tunerpage[]	RXV_OP_TUNERPAGE_*, rxv_op_tunerpage[]	RXV_RCMD_OP_TUNERPAGE
	Selects a page of preset stations					
tunerpreset	integer	R/W	yes	1-8	RXV_OP_TUNERPRESET_*, rxv_op_tunerpreset[]	RXV_RCMD_OP_TUNERPRESET
	Selects a preset station within the selected page					
homebank	enum.	R/W	yes	RXV_RDAT_OP_HOMEBANK_*, rxv_desc_bank[]	RXV_OP_HOMEBANK_*, rxv_op_homebank[]	RXV_RCMD_OP_HOMEBANK
	Selects a bank of preset control settings					
volumebank	enum.	R/W	yes	RXV_RDAT_OP_VOLUMEBANK_*, rxv_desc_bank[]	RXV_OP_VOLBANK_*, rxv_op_volumebank	RXV_RCMD_OP_VOLUMEBANK
	Selects a bank of preset volume levels					
speakera	boolean	R/W	yes		RXV_OP_SPEAKERA_*	RXV_RCMD_OP_SPEAKERA
	Enables or disables speaker set A in the main zone					
speakerb	boolean	R/W	yes		RXV_OP_SPEAKERB_	RXV_RCMD_OP_SPEAKERB
	Enables or disables speaker set B in the main zone					
playback	enum.	R	yes	RXV_RDAT_PLAY_PLAYBACK_*, rxv_desc_playback[]		RXV_RCMD_PLAY_PLAYBACK
	Indicates the surround sound format of the audio being played					
frequency	enum	R	yes	RXV_RDAT_PLAY_FREQUENCY_*, rxv_desc_frequency[]		RXV_RCMD_PLAY_FREQUENCY
	Indicates the sampling frequency of digital audio input					
esstatus	boolean	R	yes			RXV_RCMD_PLAY_ESSTATUS
	Indicates whether the internal Dolby Digital Matrix 6.1 and DTS ES decoder is active or not					
thrbypass	enum.	R	yes	rxv_desc_thrbypass[]		RXV_RCMD_PLAY_THRBYPASS
	Indicates whether audio is being passed through or bypassing the DSP					
reddts	enum.	R	yes	rxv_desc_reddts[]		RXV_RCMD_PLAY_REDDTS
	Indicates whether the DTS decoder is waiting after decoding a stream or released					
headphone	boolean	R	yes			RXV_RCMD_OP_HEADPHONE

Name	Type	Access	Cfg	Values	Write	Report
	Indicates whether headphones are plugged in					
tunerband	enum.	R/W	yes	RXV_RDAT_OP_TUNERBAND_*, rxv_desc_tunerband[]	RXV_OP_TUNERBAND_*, rxv_op_tunerband[]	RXV_RCMD_OP_TUNERBAND
	Sets the tuner to the AM or FM band					
tunertuned	boolean	R	yes			RXV_RCMD_PLAY_TUNER TUNED
	Indicates whether the tuner is tuned to a station or not					
autotune	enum.	W	no	up, down	RXV_OP_AUTOTUNE_*	
	Starts the tuner seeking for the next station					
volumestep	enum.	W	no	up, down	RXV_OP_VOLUMESTEP_*	
	Increments or decrements the main zone master volume					
zone2volume step	enum.	W	no	up, down	RXV_OP_ZONE2VOLUMESTEP_*	
	Increments or decrements the zone 2 master volume					
noguard	enum.	R	no	RXV_RDAT_SYS_NOGUARD_*		RXV_RCMD_SYS_NOGUARD
	Indicates whether the system is ready, or busy or powered off					
warning	enum.	R	no	RXV_RDAT_SYS_WARNING_*		RXV_RCMD_SYS_WARNING
	Indicates system warning conditions					
zone2power	boolean	W	no		RXV_OP_ZONE2POWER_*	
	Controls the power for zone 2					

Appendix B

UPnP Root Device Description


```xml
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>http://gershwin.ict.ru.ac.za/~drs/</URLBase>
  <device>
    <deviceType>urn:cs-ru-ac-za:device:avr:1</deviceType>
    <friendlyName>Yamaha RX-V1000 AV Receiver</friendlyName>
    <manufacturer>Yamaha Corporation</manufacturer>
    <manufacturerURL>http://www.yamaha.com/</manufacturerURL>
    <modelDescription>Natural Sound Home Theatre Receiver</modelDescription>
    <modelName>RX-V1000</modelName>
    <modelURL>
      http://www.yamaha.com/cgi-win/webcgi.exe/DsplyModel/?gAVR00010RX-V1000
    </modelURL>
    <serialNumber>Y109210WY</serialNumber>
    <UDN>uuid:123456</UDN>
    <iconList>
      <icon>
        <mimetype>image/png</mimetype>
        <width>32</width>
        <height>32</height>
        <depth>8</depth>
        <url>icon.png</url>
      </icon>
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:cs-ru-ac-za:service:avrsvc:1</serviceType>
        <serviceId>urn:cs-ru-ac-za:serviceId:avrsvc</serviceId>
        <SCPDURL>scpd.xml</SCPDURL>
        <controlURL>http://gershwin.ict.ru.ac.za:2002/soap</controlURL>
        <eventSubURL>http://gershwin.ict.ru.ac.za:2001/gena/avrsvc</eventSubURL>
      </service>
    </serviceList>
    <presentationURL>presentation.html</presentationURL>
  </device>
</root>
```

Appendix C

UPnP Service Description


```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>getallvariables</name>
      <argumentList>
        <argument>
          <name>typ</name>
          <direction>out</direction>
        </argument>
        <argument>
          <name>ver</name>
          <direction>out</direction>
        </argument>
        <argument>
          <name>system</name>
          <direction>out</direction>
        </argument>
        <argument>
          <name>volume</name>
          <direction>out</direction>
          <relatedStateVariable>volume</relatedStateVariable>
        </argument>
        <argument>
          <name>input</name>
          <direction>out</direction>
          <relatedStateVariable>input</relatedStateVariable>
        </argument>
        <argument>
          <name>mute</name>
          <direction>out</direction>
          <relatedStateVariable>mute</relatedStateVariable>
        </argument>
        <argument>
          <name>zone2volume</name>
          <direction>out</direction>
          <relatedStateVariable>zone2volume</relatedStateVariable>
        </argument>
        <argument>
          <name>zone2input</name>
          <direction>out</direction>
          <relatedStateVariable>zone2input</relatedStateVariable>
        </argument>
        <argument>
          <name>zone2mute</name>
          <direction>out</direction>
          <relatedStateVariable>zone2mute</relatedStateVariable>
        </argument>
        <argument>
          <name>tunerband</name>
          <direction>out</direction>
          <relatedStateVariable>tunerband</relatedStateVariable>
        </argument>
        <argument>
          <name>tunerpage</name>
          <direction>out</direction>
          <relatedStateVariable>tunerpage</relatedStateVariable>
        </argument>
        <argument>
          <name>tunerpreset</name>
          <direction>out</direction>
          <relatedStateVariable>tunerpreset</relatedStateVariable>
        </argument>
        <argument>
          <name>tunertuned</name>
          <direction>out</direction>
          <relatedStateVariable>tunertuned</relatedStateVariable>
        </argument>
        <argument>
          <name>effect</name>
          <direction>out</direction>
          <relatedStateVariable>effect</relatedStateVariable>
        </argument>
        <argument>
          <name>program</name>
          <direction>out</direction>
          <relatedStateVariable>program</relatedStateVariable>
        </argument>
        <argument>
          <name>speakera</name>
          <direction>out</direction>
          <relatedStateVariable>speakera</relatedStateVariable>
        </argument>
        <argument>
          <name>speakerb</name>
          <direction>out</direction>
          <relatedStateVariable>speakerb</relatedStateVariable>
        </argument>
        <argument>
          <name>osd</name>
          <direction>out</direction>
          <relatedStateVariable>osd</relatedStateVariable>
        </argument>
        <argument>
          <name>power</name>
          <direction>out</direction>
          <relatedStateVariable>power</relatedStateVariable>
        </argument>
        <argument>
          <name>sixchinput</name>
          <direction>out</direction>
          <relatedStateVariable>sixchinput</relatedStateVariable>
        </argument>
        <argument>
          <name>sleep</name>
          <direction>out</direction>
          <relatedStateVariable>sleep</relatedStateVariable>
        </argument>
        <argument>
          <name>eskey</name>
          <direction>out</direction>
          <relatedStateVariable>eskey</relatedStateVariable>
        </argument>
        <argument>
          <name>headphone</name>
          <direction>out</direction>
          <relatedStateVariable>headphone</relatedStateVariable>
        </argument>
        <argument>
          <name>homebank</name>
          <direction>out</direction>
          <relatedStateVariable>homebank</relatedStateVariable>
        </argument>
        <argument>
          <name>volumebank</name>
          <direction>out</direction>
          <relatedStateVariable>volumebank</relatedStateVariable>
        </argument>
        <argument>
          <name>playback</name>
          <direction>out</direction>
          <relatedStateVariable>playback</relatedStateVariable>
        </argument>
        <argument>
          <name>thrbypass</name>
          <direction>out</direction>
          <relatedStateVariable>thrbypass</relatedStateVariable>
        </argument>
        <argument>
          <name>frequency</name>
          <direction>out</direction>
          <relatedStateVariable>frequency</relatedStateVariable>
        </argument>
        <argument>
          <name>reddts</name>
          <direction>out</direction>
          <relatedStateVariable>reddts</relatedStateVariable>
        </argument>
        <argument>
          <name>esstatus</name>
          <direction>out</direction>
          <relatedStateVariable>esstatus</relatedStateVariable>
```

```
      </argument>
      <argument>
        <name>inputmode</name>
        <direction>out</direction>
        <relatedStateVariable>inputmode</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setpower</name>
    <argumentList>
      <argument>
        <name>power</name>
        <direction>in</direction>
        <relatedStateVariable>power</relatedStateVariable>
      </argument>
      <argument>
        <name>newpower</name>
        <direction>out</direction>
        <relatedStateVariable>power</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setinput</name>
    <argumentList>
      <argument>
        <name>input</name>
        <direction>in</direction>
        <relatedStateVariable>input</relatedStateVariable>
      </argument>
      <argument>
        <name>newinput</name>
        <direction>out</direction>
        <relatedStateVariable>input</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setsixchinput</name>
    <argumentList>
      <argument>
        <name>sixchinput</name>
        <direction>in</direction>
        <relatedStateVariable>sixchinput</relatedStateVariable>
      </argument>
      <argument>
        <name>newsixchinput</name>
        <direction>out</direction>
        <relatedStateVariable>sixchinput</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setinputmode</name>
    <argumentList>
      <argument>
        <name>inputmode</name>
        <direction>in</direction>
        <relatedStateVariable>inputmode</relatedStateVariable>
      </argument>
      <argument>
        <name>newinputmode</name>
        <direction>out</direction>
        <relatedStateVariable>inputmode</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setmute</name>
    <argumentList>
      <argument>
        <name>mute</name>
        <direction>in</direction>
        <relatedStateVariable>mute</relatedStateVariable>
      </argument>
      <argument>
        <name>newmute</name>
        <direction>out</direction>
        <relatedStateVariable>mute</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setzone2input</name>
    <argumentList>
      <argument>
        <name>zone2input</name>
        <direction>in</direction>
        <relatedStateVariable>zone2input</relatedStateVariable>
      </argument>
      <argument>
        <name>newzone2input</name>
        <direction>out</direction>
        <relatedStateVariable>zone2input</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setzone2mute</name>
    <argumentList>
      <argument>
        <name>zone2mute</name>
        <direction>in</direction>
        <relatedStateVariable>zone2mute</relatedStateVariable>
      </argument>
      <argument>
        <name>newzone2mute</name>
        <direction>out</direction>
        <relatedStateVariable>zone2mute</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setvolume</name>
    <argumentList>
      <argument>
        <name>volume</name>
        <direction>in</direction>
        <relatedStateVariable>volume</relatedStateVariable>
      </argument>
      <argument>
        <name>newvolume</name>
        <direction>out</direction>
        <relatedStateVariable>volume</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setzone2volume</name>
    <argumentList>
      <argument>
        <name>zone2volume</name>
        <direction>in</direction>
        <relatedStateVariable>zone2volume</relatedStateVariable>
      </argument>
      <argument>
        <name>newzone2volume</name>
        <direction>out</direction>
        <relatedStateVariable>zone2volume</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setprogram</name>
    <argumentList>
      <argument>
        <name>program</name>
        <direction>in</direction>
        <relatedStateVariable>program</relatedStateVariable>
      </argument>
      <argument>
        <name>newprogram</name>
        <direction>out</direction>
        <relatedStateVariable>program</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>seteffect</name>
    <argumentList>
      <argument>
        <name>effect</name>
        <direction>in</direction>
        <relatedStateVariable>effect</relatedStateVariable>
      </argument>
      <argument>
        <name>neweffect</name>
        <direction>out</direction>
        <relatedStateVariable>effect</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>seteskey</name>
    <argumentList>
      <argument>
        <name>eskey</name>
        <direction>in</direction>
        <relatedStateVariable>eskey</relatedStateVariable>
      </argument>
      <argument>
        <name>neweskey</name>
        <direction>out</direction>
        <relatedStateVariable>eskey</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setosd</name>
    <argumentList>
      <argument>
        <name>osd</name>
        <direction>in</direction>
        <relatedStateVariable>osd</relatedStateVariable>
      </argument>
      <argument>
        <name>newosd</name>
        <direction>out</direction>
        <relatedStateVariable>osd</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setsleep</name>
    <argumentList>
      <argument>
        <name>sleep</name>
        <direction>in</direction>
        <relatedStateVariable>sleep</relatedStateVariable>
      </argument>
      <argument>
```

```
        <name>newsleep</name>
        <direction>out</direction>
        <relatedStateVariable>sleep</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>settunerpage</name>
    <argumentList>
      <argument>
        <name>tunerpage</name>
        <direction>in</direction>
        <relatedStateVariable>tunerpage</relatedStateVariable>
      </argument>
      <argument>
        <name>newtunerpage</name>
        <direction>out</direction>
        <relatedStateVariable>tunerpage</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>settunerpreset</name>
    <argumentList>
      <argument>
        <name>tunerpreset</name>
        <direction>in</direction>
        <relatedStateVariable>tunerpreset</relatedStateVariable>
      </argument>
      <argument>
        <name>newtunerpreset</name>
        <direction>out</direction>
        <relatedStateVariable>tunerpreset</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>sethomebank</name>
    <argumentList>
      <argument>
        <name>homebank</name>
        <direction>in</direction>
        <relatedStateVariable>homebank</relatedStateVariable>
      </argument>
      <argument>
        <name>newhomebank</name>
        <direction>out</direction>
        <relatedStateVariable>homebank</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setvolumebank</name>
    <argumentList>
      <argument>
        <name>volumebank</name>
        <direction>in</direction>
        <relatedStateVariable>volumebank</relatedStateVariable>
      </argument>
      <argument>
        <name>newvolumebank</name>
        <direction>out</direction>
        <relatedStateVariable>volumebank</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setspeakera</name>
    <argumentList>
      <argument>
        <name>speakera</name>
        <direction>in</direction>
        <relatedStateVariable>speakera</relatedStateVariable>
      </argument>
      <argument>
        <name>newspeakera</name>
        <direction>out</direction>
        <relatedStateVariable>speakera</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>setspeakerb</name>
    <argumentList>
      <argument>
        <name>speakerb</name>
        <direction>in</direction>
        <relatedStateVariable>speakerb</relatedStateVariable>
      </argument>
      <argument>
        <name>newspeakerb</name>
        <direction>out</direction>
        <relatedStateVariable>speakerb</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>set</name>
    <argumentList>
      <argument>
        <name></name>
        <direction>in</direction>
        <relatedStateVariable></relatedStateVariable>
      </argument>
      <argument>
        <name>new</name>
        <direction>out</direction>
        <relatedStateVariable></relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>settunerband</name>
    <argumentList>
      <argument>
        <name>tunerband</name>
        <direction>in</direction>
        <relatedStateVariable>tunerband</relatedStateVariable>
      </argument>
      <argument>
        <name>newtunerband</name>
        <direction>out</direction>
        <relatedStateVariable>tunerband</relatedStateVariable>
      </argument>
    </argumentList>
  </action>

  <action>
    <name>doautotuneup</name>
  </action>
  <action>
    <name>doautotunedown</name>
  </action>

  <action>
    <name>dovolumestepup</name>
  </action>
  <action>
    <name>dovolumestepdown</name>
  </action>

  <action>
    <name>dozone2volumestepup</name>
  </action>
  <action>
    <name>dozone2volumestepdown</name>
  </action>

  <action>
    <name>dozone2poweron</name>
  </action>
  <action>
    <name>dozone2poweroff</name>
  </action>

</actionList>
<serviceStateTable>
  <stateVariable sendEvents="yes">
    <name>volume</name>
    <dataType>float</dataType>
    <defaultValue>-99.5</defaultValue>
    <allowedValueRange>
      <minimum>-99.5</minimum>
      <maximum>0</maximum>
      <step>0.5</step>
    </allowedValueRange>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>input</name>
    <dataType>string</dataType>
    <defaultValue>Phono</defaultValue>
    <allowedValueList>
      <allowedValue>Phono</allowedValue>
      <allowedValue>CD</allowedValue>
      <allowedValue>Tuner</allowedValue>
      <allowedValue>CD-R</allowedValue>
      <allowedValue>MiniDisc/Tape</allowedValue>
      <allowedValue>DVD</allowedValue>
      <allowedValue>Digital TV/LaserDisc</allowedValue>
      <allowedValue>Cable</allowedValue>
      <allowedValue>Satellite</allowedValue>
      <allowedValue>VCR 1</allowedValue>
      <allowedValue>VCR 2/DVR</allowedValue>
      <allowedValue>Video Aux</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>mute</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>

  <stateVariable sendEvents="yes">
    <name>zone2volume</name>
    <dataType>int</dataType>
    <defaultValue>-80</defaultValue>
    <allowedValueRange>
      <minimum>-80</minimum>
      <maximum>0</maximum>
      <step>1</step>
    </allowedValueRange>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>zone2input</name>
    <dataType>string</dataType>
    <defaultValue>Phono</defaultValue>
    <allowedValueList>
      <allowedValue>Phono</allowedValue>
      <allowedValue>CD</allowedValue>
      <allowedValue>Tuner</allowedValue>
      <allowedValue>CD-R</allowedValue>
      <allowedValue>MiniDisc/Tape</allowedValue>
      <allowedValue>DVD</allowedValue>
      <allowedValue>Digital TV/LaserDisc</allowedValue>
      <allowedValue>Cable</allowedValue>
      <allowedValue>Satellite</allowedValue>
      <allowedValue>VCR 1</allowedValue>
      <allowedValue>VCR 2/DVR</allowedValue>
```

```
      <allowedValue>Video Aux</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>zone2mute</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>

  <stateVariable sendEvents="yes">
    <name>tunerband</name>
    <dataType>string</dataType>
    <defaultValue>FM</defaultValue>
    <allowedValueList>
      <allowedValue>FM</allowedValue>
      <allowedValue>AM</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>tunerpage</name>
    <dataType>char</dataType>
    <defaultValue>A</defaultValue>
    <allowedValueList>
      <allowedValue>A</allowedValue>
      <allowedValue>B</allowedValue>
      <allowedValue>C</allowedValue>
      <allowedValue>D</allowedValue>
      <allowedValue>E</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>tunerpreset</name>
    <dataType>int</dataType>
    <defaultValue>1</defaultValue>
    <allowedValueRange>
      <minimum>1</minimum>
      <maximum>8</maximum>
      <step>1</step>
    </allowedValueRange>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>tunertuned</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>

  <stateVariable sendEvents="yes">
    <name>effect</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>program</name>
    <dataType>string</dataType>
    <defaultValue>Off</defaultValue>
    <allowedValueList>
      <allowedValue>Off</allowedValue>
      <allowedValue>Hall A</allowedValue>
      <allowedValue>Hall B</allowedValue>
      <allowedValue>Hall C</allowedValue>
      <allowedValue>Live Concert</allowedValue>
      <allowedValue>Freiburg</allowedValue>
      <allowedValue>Royaumont</allowedValue>
      <allowedValue>Village Gate</allowedValue>
      <allowedValue>The Bottom Line</allowedValue>
      <allowedValue>The Roxy Theatre</allowedValue>
      <allowedValue>Arena</allowedValue>
      <allowedValue>Anaheim</allowedValue>
      <allowedValue>Bowl</allowedValue>
      <allowedValue>Disco</allowedValue>
      <allowedValue>Game</allowedValue>
      <allowedValue>5/6/8ch Stereo</allowedValue>
      <allowedValue>Pop/Rock</allowedValue>
      <allowedValue>Classical/Opera</allowedValue>
      <allowedValue>Mono Movie</allowedValue>
      <allowedValue>Variety Sports</allowedValue>
      <allowedValue>Spectacle</allowedValue>
      <allowedValue>Sci-fi</allowedValue>
      <allowedValue>Adventure</allowedValue>
      <allowedValue>General</allowedValue>
      <allowedValue>Normal</allowedValue>
      <allowedValue>Enhanced</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>speakera</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>speakerb</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>osd</name>
    <dataType>string</dataType>
    <defaultValue>Off</defaultValue>
    <allowedValueList>
      <allowedValue>Full</allowedValue>
      <allowedValue>Short</allowedValue>
      <allowedValue>Off</allowedValue>
    </allowedValueList>
  </stateVariable>

  <stateVariable sendEvents="yes">
    <name>power</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>sixchinput</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>sleep</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>eskey</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>headphone</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>homebank</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
    <allowedValueList>
      <allowedValue>Main</allowedValue>
      <allowedValue>A</allowedValue>
      <allowedValue>B</allowedValue>
      <allowedValue>C</allowedValue>
    </allowedValueList>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>volumebank</name>
    <dataType>boolean</dataType>
    <defaultValue>false</defaultValue>
    <allowedValueList>
      <allowedValue>Main</allowedValue>
      <allowedValue>A</allowedValue>
      <allowedValue>B</allowedValue>
      <allowedValue>C</allowedValue>
    </allowedValueList>
  </stateVariable>

  <stateVariable sendEvents="yes">
    <name>playback</name>
    <dataType>string</dataType>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>thrbypass</name>
    <dataType>string</dataType>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>frequency</name>
    <dataType>string</dataType>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>reddts</name>
    <dataType>string</dataType>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>esstatus</name>
    <dataType>string</dataType>
  </stateVariable>
  <stateVariable sendEvents="yes">
    <name>inputmode</name>
    <dataType>string</dataType>
  </stateVariable>
 </serviceStateTable>
</scpd>
```

Appendix D

Performance Test Data

The minima, averages, maxima and standard deviations exclude the first runs. Times are in milliseconds.

D.1 UPnP SOAP tests

Invalid request

Run	First frame	Last frame	Time on the bus	Round trip time	Difference
0	0	12.398366	12.398366	38.892000	26.493634
1	1023.925090	1028.515828	4.590738	6.806000	2.215262
2	2033.110331	2037.376689	4.266358	6.272000	2.005642
3	3043.130127	3047.364665	4.234538	6.233000	1.998462
4	4053.126485	4057.342896	4.216411	6.234000	2.017589
5	5063.129293	5067.368815	4.239522	6.234000	1.994478
6	6073.143860	6077.403381	4.259521	6.255000	1.995479
7	7083.138021	7087.389221	4.251200	6.246000	1.994800
8	8093.162252	8097.408936	4.246684	6.245000	1.998316
9	9103.146912	9107.385335	4.238423	6.234000	1.995577
Minimum			4.216411	6.233000	1.994478
Average			4.282599	6.306556	2.023956
Maximum			4.590738	6.806000	2.215262
Standard deviation			0.116469	0.187735	0.072125

queryStateVariable

Run	First frame	Last frame	Time on the bus	Round trip time	Difference
0	10113.586405	10117.931152	4.344747	6.366000	2.021253
1	11123.165365	11127.461365	4.296000	6.303000	2.007000
2	12133.169047	12137.451355	4.282308	6.288000	2.005692
3	13143.181946	13147.448140	4.266194	6.267000	2.000806
4	14153.175680	14157.428752	4.253072	6.254000	2.000928
5	15163.185954	15167.455343	4.269389	6.277000	2.007611
6	16173.194275	16177.486877	4.292602	6.296000	2.003398
7	17183.199585	17187.473307	4.273722	6.280000	2.006278
8	18193.218770	18197.513631	4.294861	6.306000	2.011139
9	19203.211487	19207.498027	4.286540	6.295000	2.008460
Minimum			4.253072	6.254000	2.000806
Average			4.279410	6.285111	2.005701
Maximum			4.296000	6.306000	2.011139
Standard deviation			0.014783	0.017208	0.003444

getallvariables

Run	First frame	Last frame	Time on the bus	Round trip time	Difference
0	20213.661092	20219.473226	5.812134	7.854000	2.041866
1	21223.227844	21228.922546	5.694702	7.718000	2.023298
2	22233.237710	22238.926412	5.688702	7.706000	2.017298
3	23243.248149	23248.951152	5.703003	7.727000	2.023997
4	24253.247294	24258.922384	5.675090	7.688000	2.012910
5	25263.248678	25268.926921	5.678243	7.696000	2.017757
6	26273.253947	26278.970439	5.716492	7.729000	2.012508
7	27283.259094	27288.931864	5.672770	7.682000	2.009230
8	28293.283427	28298.979004	5.695577	7.719000	2.023423
9	29303.261902	29308.978617	5.716715	7.736000	2.019285
Minimum			5.672770	7.682000	2.009230
Average			5.693477	7.711222	2.017745
Maximum			5.716715	7.736000	2.023997
Standard deviation			0.016538	0.019162	0.005339

D.2 UPnP GENA tests

Delivery of GENA event notifications

Run	Latency
0	2.647000
1	2.699000
2	2.641000
3	2.610000
4	2.685000
5	2.631000
6	2.566000
7	2.666000
8	2.643000
9	2.584000
Minimum	2.566000
Average	2.636111
Maximum	2.699000
Standard deviation	0.044228

ICMP *ping* test

Run	Round trip time
1	0.139
2	0.130
3	0.140
4	0.137
5	0.128
6	0.144
7	0.142
8	0.123
9	0.142
Minimum	0.123
Average	0.136
Maximum	0.144
Standard deviation	0.007339

D.3 AV/C status tests

Unit status command

Run	First frame	Last frame	Time on the bus	Round trip time	Difference
0	0	0.606730	0.606730	0.659000	0.052270
1	1009.728414	1010.235575	0.507161	0.544000	0.036839
2	2019.730815	2020.227905	0.497090	0.529000	0.031910
3	3029.737935	3030.240967	0.503032	0.538000	0.034968
4	4039.740255	4040.234558	0.494303	0.525000	0.030697
5	5049.747559	5050.266622	0.519063	0.552000	0.032937
6	6059.753520	6060.267843	0.514323	0.547000	0.032677
7	7069.765564	7070.267700	0.502136	0.541000	0.038864
8	8079.762695	8080.274943	0.512248	0.542000	0.029752
9	9089.767517	9090.271423	0.503906	0.533000	0.029094
Minimum			0.494303	0.525000	0.029094
Average			0.505918	0.539000	0.033082
Maximum			0.519063	0.552000	0.038864
Standard deviation			0.008098	0.008689	0.003266

Function block status command

Run	First frame	Last frame	Time on the bus	Round trip time	Difference
0	10099.776672	10100.311910	0.535238	0.569000	0.033762
1	11109.785726	11110.323629	0.537903	0.573000	0.035097
2	12119.786051	12120.323771	0.537720	0.568000	0.030280
3	13129.794067	13130.362671	0.568604	0.602000	0.033396
4	14139.799276	14140.330098	0.530822	0.562000	0.031178
5	15149.816345	15150.354655	0.538310	0.582000	0.043690
6	16159.810649	16160.364909	0.554260	0.586000	0.031740
7	17169.814819	17170.355428	0.540609	0.572000	0.031391
8	18179.821391	18180.351379	0.529988	0.562000	0.032012
9	19189.827922	19190.376851	0.548929	0.582000	0.033071
Minimum			0.529988	0.562000	0.030280
Average			0.543016	0.576556	0.033539
Maximum			0.568604	0.602000	0.043690
Standard deviation			0.012332	0.012875	0.004063

D.4 AV/C notify tests

Delivery of Changed response

Run	Latency
0	0.017000
1	0.102000
2	0.065000
3	0.023000
4	0.100000
5	0.051000
6	-0.004000
7	0.103000
8	0.057000
9	-0.002000
Minimum	-0.004000
Average	0.055000
Maximum	0.103000
Standard deviation	0.042444

Appendix E

Contents of the CD-ROM

The accompanying CD-ROM contains the source code for the software written for this project, and this thesis in LyX and PDF formats. The directory structure is listed below, with cross-references to the sections of the thesis that describe the programs.

			Section
/cvs/	CVS repository.		
/src/	A checked-out copy of the contents of the CVS repository. The software can be built on a FreeBSD or Linux system using the makefiles in this directory.		
	avcqt/	AV/C controller for the RX-V1000 AV/C control server. Requires Qt 3 and *libraw1394*, thus can only be built on Linux.	6.8
	description/	UPnP XML description files and HTML presentation page.	5.6
	dumps/	Captured examples of the RX-V1000 RS-232 format and SSDP transactions.	
	exp/	Experimental code.	
	genapub/	UPnP GENA eventing server.	5.8
	librxv/	Library for formatting and parsing RX-V1000 commands.	7.5
	rxvavc/	AV/C control server for the RX-V1000. Requires *libraw1394*.	6.1
	rxvqt/	Graphical application for controlling an RX-V1000 via RS-232. Requires Qt 3.	4.4
	rxvupnp/	UPnP control server coordination program for the RX-V1000.	5.10
	simrxv/	Software simulator of the RX-V1000.	7.6

			Section
	soap/	UPnP SOAP service. Includes CGI and stand-alone HTTP server versions. Requires *expat*.	5.7
	splitrxv/	RX-V1000 sharing layer.	7.5
	ssdpd/	UPnP SSDP service announcement daemon.	5.5
	thesis/	The source for this document in LyX format.	
	verify/	Control server testing system. The AV/C client requires *libraw1394*.	7.6
/thesis.pdf	This document in PDF format.		

List of References

[1] "Digital Home White Paper," Digital Home Working Group, white paper, June 2003. [Online]. Available: http://www.dhwg.org/resources/DHWG_WhitePaper.pdf

[2] B. Rose and K. Scherf, "Connecting Legacy Devices on the Home Network: The Challenges of Connectivity," Parks Associates, white paper, May 2003.

[3] G. O'Driscoll, *The Essential Guide to Home Networking Technologies*. Prentice Hall, 2000.

[4] K. Scherf, "The Emergence and Growth of Entertainment-Centric Home Networks," Parks Associates, white paper, July 2002.

[5] *Standard for a High Performance Serial Bus*, IEEE Std. 1394-1995, May 1995.

[6] *Standard for a High Performance Serial Bus - Amendment 1*, IEEE Std. 1394a-2000, May 2000.

[7] *Standard for a Higher Performance Serial Bus - Amendment 2*, IEEE Std. 1394b-2002, December 2002.

[8] D. Anderson, *FireWire System Architecture*, 2nd ed., ser. PC System Architecture Series. Addison-Wesley, December 1998.

[9] M. J. Teener, "What's new about 1394b?" Zayante, Inc., November 2000. [Online]. Available: http://www.1394ta.org/Technology/About/ppt1.PDF

[10] "1394 Trade Association Completes New 1394 Wireless Specification," 1394 Trade Association, December 2003, press release. [Online]. Available: http://www.1394ta.org/Press/2003Press/december/12.08.a.htm

[11] *Digital Transmission Content Protection Specification*, Information Version, Rev. 1.3, November 2003. [Online]. Available: http://www.dtcp.com/data/info_20031124_dtcp_Vol1_1p3.pdf

[12] T. Fout, "Understanding Universal Plug and Play," Microsoft Corporation, white paper, June 2000. [Online]. Available: http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc

[13] "About the UIC," UPnP Implementers Corporation. [Online]. Available: http://www.upnp-ic.org/uic/about/

[14] M. Schmitz, U. Warrier, and P. Iyer, *WANIPConnection:1 Service Template*, UPnP Forum Std., Rev. 1.01, November 2001. [Online]. Available: http://www.upnp.org/download/UPnP_IGD_DCP_v1.zip

[15] "BridgeCo launches UPnP-compliant Wireless Audio Adapter," BridgeCo AG, September 2003, press release. [Online]. Available: http://www.bridgeco.net/news/BCO_Launches_Wireless_Audio_Adapter.shtml

[16] *Universal Plug and Play Device Architecture*, Microsoft Corporation Std., Rev. 1.0, June 2000. [Online]. Available: http://www.upnp.org/download/UPnPDA10_20000613.htm

[17] Y. Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright, *Simple Service Discovery Protocol/1.0 Operating without an Arbiter*, IETF Expired Internet Draft, October 1999. [Online]. Available: http://www.upnp.org/download/draft-goland-http-udp-04.txt

[18] Y. Y. Goland and J. C. Schlimmer, *Multicast and Unicast UDP HTTP Messages*, IETF Expired Internet Draft, October 2000. [Online]. Available: http://www.upnp.org/download/draft-goland-http-udp-04.txt

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Recommendation, June 2003. [Online]. Available: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[20] J. Cohen, S. Aggarwal, and Y. Y. Goland, *General Event Notification Architecture Base: Client to Arbiter*, IETF Expired Internet Draft, September 2000. [Online]. Available: http://www.upnp.org/download/draft-cohen-gena-client-01.txt

[21] R. Foss, B. Moses, and R. Laubscher, "A Legacy Adapter Component of a 1394-Based Professional Studio Architecture," in *111th Convention of the Audio Engineering Society*, September 2001.

[22] "BridgeCo Networked Loudspeaker Product Information," BridgeCo AG, December 2003. [Online]. Available: http://www.bridgeco.net/products/loudspeaker/adp_dm1_14_pma_nls.pdf

[23] *AV/C Digital Interface Command Set General Specification*, 1394 Trade Association Std., Rev. 4.1, September 2001.

[24] *Digital Interface for Consumer Audio/Video Equipment - Part 1: General*, IEC Std. 61 883-1, 1998.

[25] *AV/C Audio Subunit Specification*, 1394 Trade Association Std., Rev. 1.0, October 2000.

[26] *AV/C Panel Subunit Specification*, 1394 Trade Association Std., Rev. 1.21, 2002.

[27] *AV/C Disc Subunit General Specification*, 1394 Trade Association Std., Rev. 1.0, January 1999.

[28] *AV/C Descriptor Mechanism Specification*, 1394 Trade Association Std., Rev. 1.1, January 2002.

[29] "WS-65813," Mitsubishi Digital Electronics America, Inc., 2004. [Online]. Available: http://www.mitsubishi-tv.com/2003_2004_product/projection_tvs/ws_65813.html

[30] "HS-HD2000U," Mitsubishi Digital Electronics America, Inc., 2004. [Online]. Available: http://www.mitsubishi-tv.com/2003_2004_product/set_top_products/hs_hd2000u.html

[31] "Mitsubishi's IEEE 1394 and HAVi Technology," Mitsubishi Digital Electronics America, Inc., 2001. [Online]. Available: http://www.mitsubishi-tv.com/ieee.html

[32] *The HAVi Specification*, HAVi, Inc. Std., Rev. 1.1, May 2001.

[33] "SmarTouch STS - ST-COM," Crestron Electronics, Inc. [Online]. Available: http://www.crestron.com/products/show_products.asp?cat=4&id=176&type=residential

[34] Y. Yoshizawa and H. Kawai, *RX-Vx00 RS-232C Interface*, Yamaha Electronics Corporation, November 2000.

[35] P. V. Mockapetris, *Domain Names: Implementation and Specification*, IETF RFC 1035, November 1987.

[36] *Yamaha RX-V1000 Natural Sound AV Receiver Owner's Manual*, Yamaha Electronics Corporation, November 2000.

[37] "Qt 3.1 Whitepaper," Trolltech AS, white paper, October 2002. [Online]. Available: ftp://ftp.trolltech.com/qt/pdf/3.1/qt-whitepaper-a4-web.pdf

[38] "DHIVA Transceivers (IEEE 1394) Technology Fact Sheet," Digital Harmony Technologies, Inc., January 2001.

[39] P. Johansson, *IPv4 over IEEE 1394*, IETF RFC 2734, December 1999.

[40] B. H. Klinkradt, "An Investigation into the Application of the IEEE 1394 High Performance Serial Bus to Sound Installation Control," M.Sc. (Computer Science) thesis, Rhodes University, 2003.

[41] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, *Hypertext Transfer Protocol – HTTP/1.0*, IETF RFC 1945, May 1996.

[42] J. Bentham, *TCP/IP Lean: Web Servers for Embedded Systems*. CMP Books, 2000.

[43] "About eCos." [Online]. Available: http://ecos.sourceware.org/about.html

[44] R. Droms, *Dynamic Host Configuration Protocol*, IETF RFC 2131, March 1997.

[45] S. Cheshire, B. Aboba, and E. Guttman, *Dynamic Configuration of Link-Local IPv4 Addresses*, IETF Internet Draft, September 2003. [Online]. Available: http://www1.ietf.org/internet-drafts/draft-ietf-zeroconf-ipv4-linklocal-10.txt

[46] D. C. Plummer, *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*, IETF RFC 826, November 1982.

[47] S. Thomson and T. Narten, *IPv6 Stateless Address Autoconfiguration*, IETF RFC 2462, December 1998.

[48] *Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods*, IETF RFC 1001, March 1987.

[49] S. Cheshire and M. Krochmal, *DNS-Based Service Discovery*, IETF Internet Draft, June 2003. [Online]. Available: http://www1.ietf.org/internet-drafts/draft-cheshire-dnsext-dns-sd-01.txt

[50] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, IETF RFC 2616, June 1999.

[51] D. R. Worley, "Re: Question about relatedStateVariables and actions in general," March 2002, e-mail posted to the UPnP World mailing list.

[52] D. Box, D. Enhebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, *Simple Object Access Protocol (SOAP)*, W3C Note, May 2000. [Online]. Available: http://www.w3.org/TR/2000/NOTE-SOAP-20000508

[53] H. F. Nielsen, P. J. Leach, and S. Lawrence, *An HTTP Extension Framework*, IETF RFC 2774, February 2000.

[54] J. Clark, "expat - XML Parser Toolkit." [Online]. Available: http://www.jclark.com/xml/expat.html

[55] C. Cooper, "Benchmarking XML Parsers," *xml.com*, May 1999. [Online]. Available: http://www.xml.com/pub/a/Benchmark/article.html

[56] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[57] A. S. Tanenbaum, *Modern Operating Systems*. Prentice-Hall, 1992.

[58] D. J. Bernstein, “clockspeed,” February 2000. [Online]. Available: http://cr.yp.to/clockspeed.html

[59] J. Postel, *Internet Control Message Protocol*, IETF RFC 792, September 1981.

[60] R. Whitmer, “SOAP Scripts in Mozilla,” March 2003. [Online]. Available: http://lxr.mozilla.org/seamonkey/source/extensions/webservices/docs/Soap_Scripts_in_Mozilla.html

[61] “IEEE 1394 for Linux,” February 2004. [Online]. Available: http://www.linux1394.org/

[62] *AV/C Tuner Model and Command Set*, 1394 Trade Association Std., Rev. 1.0, April 1998.