The Remote Configuration of Devices within Home Entertainment Networks

THESIS

Submitted in fulfilment of the requirements for the Degree of

Master of Science (Applied Computer Science)

by

Colin Dembovsky

January

2002

Supervisor: Prof. R. Foss

Abstract

This thesis examines home entertainment network remote configuration solutions. It does so by inspecting four home entertainment networking solution specifications – HAVi, Jini, AV/C and UPnP. Two of these (AV/C and UPnP) are implemented partially for a system allowing a TV to configure an Audio/Video Receiver (AV/R) remotely on the network (a process known as remote configuration).

The two implementations are then more closely investigated and several implementation differences in the approach between the remote configuration method of device configuration and other methods of device configuration are discerned. These different approaches are then categorised into one of two theoretical models of communication for configuring devices on home entertainment networks – the Rendering model and the Programmed model. By classifying a particular method of device configuration into one of the two models, manufacturers can quickly determine the inherent strengths and weaknesses of that method.

Keywords: remote configuration, home entertainment networking, HAVi, Jini, AV/C, UPnP

Acknowledgements

I would like to thank all first and foremost my supervisor, Richard Foss for all his support, ideas and patience – thanks for supervising so well! Thanks are also due to Digital Harmony Technologies in Seattle for funding my MSc, for flying me to Seattle for three weeks and for providing me with a DHIVA. Thanks especially to Walt Jones for his ideas and input which essentially decided the direction of this thesis and to the other DHT guys who helped me at various stages. Brad Klinkradt and Dave Sieborger for their assistance with the DHIVA and the DHT stack. Peter Clayton and the Rhodes University Computer Science department for their continued support over the years.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 A New Generation of Home Entertainment Networking

Think of the devices that most people live with – from televisions to DVD-players and hi-fi's. Most of these devices contain embedded microprocessors and technology people know very little about. However, users prefer their devices this way – they want devices that simply plug in and work. How DVD-players get their streams to TVs, how resources on the network are managed and other such problems are of little consequence to the people who use these devices. Hence more and more intelligence must be placed into these devices to make the devices themselves handle all the low-level hassles of networking.

The importance of properly designing the next generation of computer-enhanced devices and appliances is becoming more and more noticeable. Manufacturers are being required to place enough intelligence into these devices to make them operate "invisibly" – users plug them in and they simply work. But the problem is deeper – users can now purchase a whole range of different devices from a multitude of manufacturers. What if they purchase several devices from different vendors? The devices are going to have to connect to each other and "understand" each other even if they are produced by different manufacturers. This means that manufacturers are going to need to produce devices that implement home entertainment networking solutions that are "open" – that any other manufacturer can implement – in order to remain competitive in a user-driven market.

However, there exist many such home entertainment networking solutions and just which solution is implemented is a difficult question for a manufacturer. How does a manufacturer go about deciding which home entertainment networking solution is best suited to its suite of devices?

Since the challenges of this ideal interconnected digital world – where every device can "speak" to every other device regardless of manufacturer, make or model – are varied and many, this thesis focuses on one aspect of home entertainment networking – *the configuration of home entertainment devices*. Configuration involves changing or selecting features or setting up the device in order to utilise certain functionality of the device. This can be achieved in an number

of ways. *Remote configuration* involves connecting two devices on a home entertainment network and then allowing the user to configure one of the devices (remotely) via the other device. For example, if a user connects a hi-fi and a TV together, remote configuration would allow the user to configure the hi-fi by simply interacting with its user interface displayed on the TV. The user never has to physically touch the hi-fi itself. Other methods of configuring devices include utilising device and service specifications and control messages inherent in home entertainment network standards.

1.2 Project Objectives

This project seeks to investigate some of the home entertainment networking solutions that are currently being widely used, and ultimately provide the manufacturer of home entertainment devices with a comprehensive aid for enabling their devices to be configured on home entertainment networks, especially by the method of remote configuration. Two current home networking entertainment solutions are partly implemented to demonstrate remote configuration and these systems are fully functional and able to be used commercially.

There are many home entertainment networking solutions available, and this thesis begins by examining four of the most popular ones in Chapter 2 – HAVi, Jini, AV/C and UPnP. This inspection gives rise to several *theoretical* differences between these networking standards.

The theoretical differences include connection medium, the location and use of services on the network and what type of user interface the home entertainment networking solution provides. While comparing these features provides a fair understanding of the differences between the solutions, many questions are still left unanswered. For instance, how difficult is it to place the correct command (semantics) into the syntax that the solution requires? Are the graphical element types defined in the solution sufficient for any arbitrary user interface? Questions like these can only be answered by implementing the solutions and comparing the implementations.

However, since implementation of an entire home networking solution was not possible given certain time constraints, it was decided that a small sub-section of each home networking solution would be implemented. This section focuses on the configuration of devices on the network, and especially remote configuration of devices, and each home entertainment networking solution's approach to remote configuration is discussed in detail in Chapter 3.

Chapters 4 and 5 detail the implementation of two of the home entertainment networking solutions remote configuration systems – the AV/C Panel Subunit and UPnP presentation mechanism respectively. The same system is implemented by both home entertainment networking solutions – a TV is connected to an Audio/Video Receiver (AV/R) in such a manner that the AV/R can be remotely configured by a user from the TV. Another subsidiary goal emerges in the AV/C implementation – a set of tools for manufacturers to use for further remote configuration systems with other devices is provided. HAVi and Jini were not implemented at all since there were no HAVi or Jini components in place already. AV/C was selected because of the availability of an AV/C stack and UPnP because of the availability of Microsoft's UPnP Development Kit.

Chapter 6 discusses the differences experienced between configuring devices remotely and using other methods of control. The comparison and contrast leads up to two models of communication for home device configuration – Rendering and Programmed. While these models vary significantly, the main difference appears in the amount of knowledge of devices on the network is programmed into a control point. Programmed model approaches to device configuration characteristically involve controllers that have intimate knowledge of the capabilities and services of the devices on the network, while Rendering model approaches involve control points that possess little to no foreknowledge of the devices on the network. These models (or approaches) aid manufacturers in analysing methods of device configuration available in different home entertainment networking standards and allow manufacturers to see immediately their inherent advantages and disadvantages.

Finally, Chapter 7 draws some conclusions about remote configuration of devices on home entertainment networks from the experience and findings of the previous chapters.

Chapter 2 - Home Networks

2.1 Introduction

Home networking is a young, dynamic and emerging industry that stimulates research. This chapter will examine three main topics: the economics of home networks, the elements of home networks, and popular home entertainment networking solutions. By examining these three topics, a foundation is laid for further detailed discussion of home networks and remote configuration on these networks.

2.2 The Economics of Home Networks

The market for home networks is growing. According to Parks Associates, a research firm based in Dallas, the year 2000 $150 million market is expected to grow to $3.4 billion by 2004 [19]. People today are becoming increasingly interested in home networks.

For many years technologists, as well as media and market strategists, have been hailing in the digital revolution in home networking. They have painted bold and creative pictures about how computing could change the way people live. Hence people are no longer content to be confined to one room or one TV set. Many users wish all their devices to communicate with each other, as well as to be able to control lights, air conditioners and security systems from anywhere in the house.

Most of the devices in a modern home have some sort of computing power in them – there are embedded microprocessors in washing machines and alarm systems. Most of these computers are invisible to the user – they are *ubiquitous*. Ubiquitous computing was a phrase coined by Dr. Mark Weiser of PARC (Palo Alto Research Centre). The vision of the ubiquitous computing research group at PARC was to move computing "back into its place, to reposition it in the environmental background" [1]. Ubiquitous computing is an important concept relevant to the way users interact with devices in the home.

Computers and intelligent devices (devices that have embedded microprocessors built into them) are also *pervasive* – they seem to inhabit every part of peoples' lives. Today's users do not want to have to install drivers for their fridges and microwave ovens. Users expect their

networks to be intelligent enough to configure and communicate without human interaction while at the same time be robust enough to work correctly all the time.

This means there exists a wide scope for creating a new generation of devices that make use of computing to enhance their functionality as well as their usability. Ian O'Sullivan names these devices *Electrodomestic Network Devices (ENDs)* [2].

2.2.1 The IEDMM

The Information Economy Derivative Market Model (IEDMM) [3] is a model that O'Sullivan defines as a guide for key players in the Home Networks Industry and provides a framework for identifying sources of value in this industry. He defines four tiers that identify major points of value.

- **Tier 1: Infostructure** – this tier is the substructure of the physical system enabling the delivery of content and services to users. The key players in this tier are Telco's, cable operators and Internet Service Providers.
- **Tier 2: ENDs** – this tier is responsible for the rendering of the content and services to the user. The key players on this tier are obviously device manufacturers – but also standards bodies.
- **Tier 3: Digital Media Commodities** – this tier encompasses the raw content – anything that is rendered to digital format; from songs and voice messages to movies, text and graphics.
- **Tier 4: Services** – this tier is primarily concerned with co-ordination and consolidation of the content of Tier 3, as well as the maintenance of the customer-vendor relationship.

Tiers 3 and 4 are derivative markets, since they rely on Tiers 1 and 2 to provide the infrastructure on which content and services operate. They are poised to generate larger profit margins in the value market.

This thesis presents work important to Tier 2. What intelligence devices posses and how they communicate with each other is determined in Tier 2, and is critical to Tiers 3 and 4.

2.3 Home Network Elements

At a high level, the task of home networks is to connect anything and everything in a home together. Obviously the most visible components of a home network are the ENDs. There must be a *connection medium* that connects the devices. Furthermore, there needs to be software

that will allow meaningful interaction between devices (a *home networking solution*). In order to make the network able to interact with the outside world, some sort of *residential gateway* needs to exist.

Figure 1 shows the typical components of a home network.

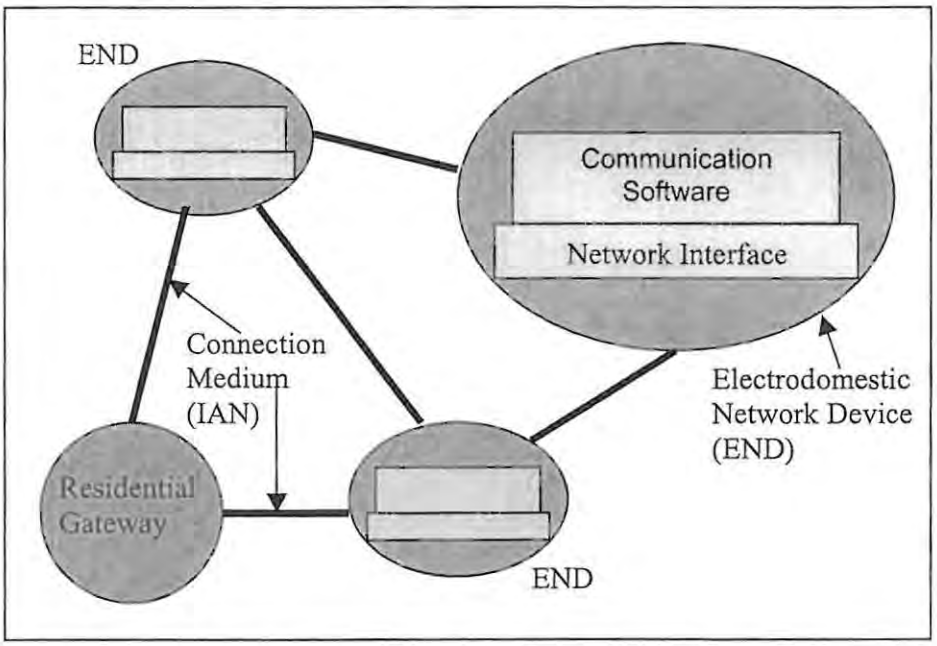

Figure 1 - Elements of a typical home network.

2.3.1 Electrodomestic Network Devices (ENDs)

O'Driscoll [10] divides ENDs into three categories.

- **Appliances** – these are devices that are mechanical in nature (for example a fridge)
- **Electronics** – these are devices that are both physical and logical in nature (for example a digital TV)
- **Computers** – these devices are logical in nature (for example a PC)

Generally users lack the technical know-how to be able to install or set up the networks themselves – they will therefore primarily interact with the ENDs. It is therefore imperative to fit these devices with sufficient intelligence to be able to configure themselves onto the home network.

For the purpose of this thesis, only devices that are used for entertainment are considered. These devices are mainly audio/visual in nature. These devices form a subset of the entire home network called the *home entertainment network* (see § 2.3.4). Home lighting, temperature and security networks are examples of other subsets that may be part of a home network.

2.3.2 The IAN

The connection medium for a home entertainment network is called the Information Appliance Network (IAN) by O'Sullivan [2]. It is a high-speed home data network that allows ENDs to connect to one another. Most current networking solutions do not cater for all the requirements of a typical home network. Typical solutions are anything from existing copper power and telephone cabling, to new digital cabling, to wireless solutions. The connection media of future home networks will most likely be a mix of these solutions.

Successful IANs have several important features (note that some of the features are inherent in the devices connected on the IAN rather than the IAN itself):

- **Leveraging existing technologies** – users are not inclined to rewire their entire houses for new technology. Use must be made of existing wiring as far as possible. Another alternative gaining popularity is wireless solutions.
- **High Bandwidth** – this applies especially to A/V networks that stream media.
- **Intelligence** – the network needs to be 'plug-and-play'. Users cannot afford to get professionals to set up their network. The devices need to be intelligent enough to configure themselves.
- **Robustness** – the network must be able to handle traffic, hot-plugging of devices and errors efficiently and invisibly.
- **Interoperability** – the devices on the network must be able to communicate meaningfully with one another.
- **They must be future-proof** – users will probably not upgrade if professional intervention is required. Devices must be able to upgrade via internet downloads or other user-friendly means.

2.3.3 The Residential Gateway

The residential gateway is a device that links the home network with the rest of the world, effectively providing a bi-directional interface between every device in the home and the rest of the world. They are important devices for home networking as without them, the devices in the

home are completely isolated from the rest of the world. The simplest gateway device is a modem, but gateway devices with far more capabilities are being developed and marketed.

Residential gateways increase the value of broadband services by allowing broadband data (such as video) to be distributed to a multitude of devices around the home instead of to just one PC (as in the case of a modem).

2.3.4 Home Entertainment Networks

There exists a subtle difference between home networks and home entertainment networks. For the purposes of clarity, this thesis defines *home networking* to mean the entire plethora of networked appliances in a household. Furthermore, it defines a subset of home networking called *home entertainment networking.* Home entertainment networking defines the connection and networking of audio/visual (A/V) devices such as TVs and DVD players. Generally speaking, A/V devices usually have a high degree of complexity inherent in them, and so many of the principles that work for such devices can be scaled down to work for other non-A/V devices.

2.4 Home Entertainment Networking Solutions

Central to the success of future home entertainment networks is the ability for devices to communicate with each other – this means that they must be able to discover each other, use each others' services and share network resources *regardless of their make and model.* This has far reaching implications for manufacturers.

Manufacturers of ENDs are faced with the fact that the vast majority of home networks will be *multi-vendor* networks. Various manufacturers make the devices on a typical home entertainment network. Hence, in order for the devices to be able to communicate with each other, they must 'speak' some common language. This shows the importance of *open* communication standards (or home entertainment networking solutions). Open home entertainment networking solutions are solutions that are public and are usually developed by teams of manufacturers and researchers rather than by one company.

There are many home entertainment networking solutions that deliver some answers to the challenges of home entertainment networking. This section concentrates on some of the more widely adopted solutions. Manufacturers are faced with the decision of which solution(s) to embed in their devices. This is a complex challenge since there are many solutions.

These solutions differ in their approach to home entertainment networking and thus also in their approach to a specific part of home entertainment networking known as remote configuration (discussed in more detail in Chapter 3). The solutions can be analysed in terms of common features and unique features. The common features are physical medium, abstraction of functions or services, locating other devices and services, using services, detecting device state changes and user interfaces.

2.4.1 IEEE1394

One of the main emerging digital connection standards is IEEE1394-1995 [9]. It is used by three of the four home entertainment networking solutions examined in this section.

IEEE1394-1995 (or just IEEE1394) is a low cost, high bandwidth, serial bus standard that allows the multiplexing of several streams of audio and video simultaneously. It does not rely on a PC being present and allows devices to be hot-plugged and unplugged.

Presently, IEEE1394 supports bi-directional transport of up to 400 Mbps (megabits per second) and will soon support up to 1.6 Gbps (gigabits per second). IEEE1394 has two communication modes – *isochronous* and *asynchronous*. Isochronous communication guarantees delivery of data packets at fixed intervals even if some packets must be dropped. An example of isochronous communication would be streaming video at 30 frames per second. Asynchronous communication on the other hand, guarantees that all packets will arrive at their destination, but makes no guarantee of when they will arrive. Asynchronous communications are typically used to send control messages on the network.

Enhancements to the IEEE1394-1995 standard have been made and are specified in the IEEE1394b-2000 standard. These enhancements include bus speeds of up to 1.6 Gbps, more advances bus arbitration methods and several clarifications to ambiguities in the IEEE1394-1995 standard.

2.4.2 HAVi

2.4.2.1 Introduction

HAVi [8] stands for Home Audio/Video Interoperability and was started by Grundig, Hitachi, Phillips and other companies. More companies have joined in their efforts since HAVi was started. It is an open, platform independent home entertainment networking solution.

HAVi specifies a number of APIs that allow manufacturers to develop HAVi compliant devices. HAVi provides a common interface that allows interoperability between devices from different manufacturers – in other words the devices can detect and use the functionality of other devices on the network. It focuses mainly on the delivery and processing of digital A/V content between devices.

2.4.2.2 Physical Medium

HAVi is designed for, but not limited to, the IEEE1394 serial bus as its transport medium.

The HAVI architecture employs a number of *managers* or software elements that manage different facets of the network. Each software element is a self-contained entity consisting of data and the functions to manipulate that data and has a unique name and identifier (the Software Element identifier or SEID). Software elements expose their functionality through well-defined interfaces. The software elements use SEIDs and a messaging system to communicate. The implementation of the messaging system differs depending on the manufacturer, but the messaging format is common to all devices on a HAVi network. There exist several device classifications in the HAVi specification. Full Audio/Video Devices (FAVs) support the entire HAVi protocol (see § 2.4.2.3 for more about HAVi device classification). Figure 2 shows the software elements (managers) in a typical FAV.

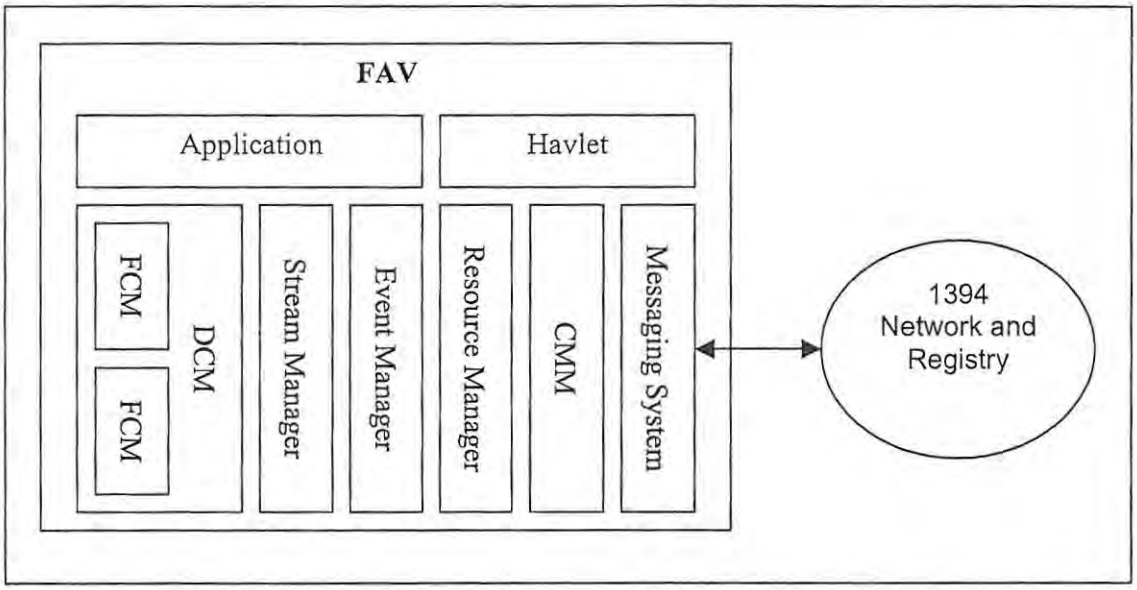

Figure 2 - Software elements in a typical FAV

One of these managers is the *communication media manager* (CMM). The CMM is basically a transcoding proxy – it allows any physical communication medium to talk to the HAVi network. The CMM provides two services – transport of data between devices and abstraction of network activities from the HAVi home entertainment networking solution.

The CMM allows HAVi to be dynamically aware of network changes that occur. It can detect and act upon topology changes that occur as a result of devices being hot-plugged and unplugged.

The interface from the network to the device is called a *Device Control Module* (DCM). The DCM also acts as a container for *Functional Control Modules* (FCMs). FCMs encapsulate services within the device (see § 2.4.2.3 for more details).

2.4.2.3 Abstraction of Devices and Services

HAVi ENDs can be separated into four categories:

- **Full A/V devices (FAV)** – supports 1394 and the full HAVi specification. Usually has a rich set of resources.
- **Intermediate A/V devices (IAV)** – supports 1394 and only a limited subset of HAVi
- **Base A/V devices (BAV)** – supports 1394, but does not run the HAVi solution directly. This device contains a Java-based control module that is uploaded to a FAV in order to allow the FAV to control this BAV.

- **Legacy A/V devices (LAV)** – these are devices that don't support either 1394 or HAVi. HAVi realises that the transition from existing legacy devices to new ENDs is going to be gradual and supports the legacy devices by allowing them to be part of the network through a FAV or IAV gateway.

HAVi views the system in terms of *controllers* and *controlled devices*. A controller is an application that serves as a host to controlled devices – it contains the hardware resources necessary to run software home entertainment network applications.

Each DCM in a device has zero or more *Functional Control Managers* associated with it. The FCMs represent functional components within a device, and each device may have any number of FCMs. HAVi applications can only communicate with functional components via their FCMs. Each FCM and DCM has a unique identifier called a HAVi unique identifier (HUID).

HAVi have defined a number of FCMs such as tuner, VCR, clock, camera, A/V disc, amplifier, display and A/V display. HAVi also define APIs to help manufacturers develop custom HAVi applications, DCMs and FCMs.

Besides providing APIs for the device, a DCM may also contain a device-specific application called a "havlet". Havlets are developed by the device manufacturers to provide a means of utilising any specialised function on the device.

2.4.2.4 Locating Other Devices and Services

All managers, DCMs and FCMs register themselves with a distributed, system-wide database called a *registry*.

The registry is a directory (present on all FAVs and IAVs) of objects available on the HAVi network. It provides APIs for registering and searching for objects. Objects register with the registry in order to be contacted. Their SEIDs (in the case of managers) or HUIDs (in the case of DCMs and FCMs) and attributes are stored in the registry. The registry also provides a query mechanism that allows objects to search for other objects according to a set of criteria.

2.4.2.5 Using Services

A device wishing to use services will query the registry to find a DCM or FCM that can perform the service. Thereafter, DCMs and FCMs are obtained via a DCM code unit present on each device. This code unit may be written in Java byte-codes or in native code and is related to the

particular device. The DCM management system is responsible for installing and uninstalling DCM code units. The DCM code unit is then run on the requesting device.

2.4.2.6 Detecting Device State Changes

Events are delivered through a service called the event delivery service (present on all FAVs and IAVs). The local event manager distributes events posted by objects to all interested objects. Objects register for notification about certain events with their local event manager.

Each event manager contains a list of events and the objects that wish to know of such events. When an event is posted via a well-defined interface to the event manager, the event manager checks its table and posts event notifications to all objects that have registered for notification of this event. Objects who have not registered for event notifications do not receive any. If events are posted globally, the event manager also notifies all other event manages of the event.

Events have buffers that can optionally be used to convey information about the event.

2.4.2.7 User Interfaces

The goal of a HAVi UI is to provide the user with a comfortable operating environment. Besides being able to control HAVi devices traditionally (as in front panels or remote controls) HAVi allows manufacturers to specify GUIs that can be rendered on various displays ranging from text-only to high-level graphics displays. The GUIs can also be rendered on remote devices, potentially from different manufacturers.

In order to support this, HAVi defines two mechanisms:

- **Level 1 UI** – this is the encoding of user interface elements which can be loaded and displayed and which generate messages in response to user actions.
- **Level 2 UI** – this level consists of APIs based on Java.

2.4.2.8 Other Features

Resources on a HAVi network are managed by two software elements – the stream manager and the resource manager. The stream manager manages the connections and bandwidth of the network. The resource manager deals with the allocation and de-allocation of hardware resources. The resource manager is used to perform "scheduled actions" (such as recording a programme at a specified time). The resource managers on a network will work together to prevent deadlock.

2.4.2.9 HAVi and Java

HAVi specify the Java programming language for the development of DCMs and home entertainment networking applications. Java has several benefits for HAVi

- It is object oriented
- It has strong typing
- It has support for standard computing functions such as I/O, networking and graphics
- It is platform independent
- It includes security through code verification
- It is widely adopted and supported

Programmers compile their code into Java *bytecodes*. These are similar to machine code except that they are processor independent. Bytecodes are verified before they are executed by the Java virtual machine present on a HAVi device.

All FAVs have a Java virtual machine capable of running any Java programme. This allows devices to be future-proof – new applications can simply be downloaded without having to recompile.

2.4.3 Jini

2.4.3.1 Introduction

Sun Microsystems have been developing Jini [20], their home entertainment networking software solution, for many years. It is the logical extension of Java. Jini relies on ENDs being able to interchange objects (data and code). Jini dispenses with the need for device drivers because each device supplies its own interfaces ensuring compatibility and reliability.

Some of the core features of Jini systems are:

- **Robustness** – Jini allows and encourages the development of highly reliable systems
- **Effortless "plug and play"** – Jini makes ENDs intelligent enough to discover one another, connect and communicate automatically
- **It is future-proof** – upgrades for Jini software on devices is easy and can be done by the users themselves
- **Support** – over 40 of the key home entertainment networking companies are behind Jini – besides all the support available for the underlying Java environment

2.4.3.2 Physical Medium

Jini is not bound to any specific medium. It can be used in limited bandwidth environments (for example cellular networks) or in broadband environments, such as home entertainment systems. Figure 3 shows the Jini stack. The operating system interfaces with the transport layer (the interface to the physical medium). Java (which is hardware and operating system independent) is the layer above the operating system. The next layer up is Jini, and the network services on the devices sit on the Jini layer.

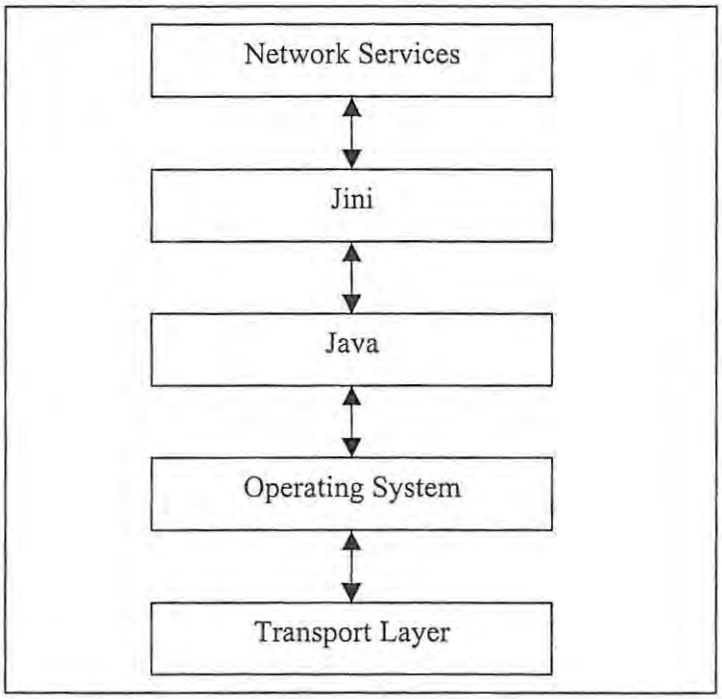

Figure 3 - The Jini stack

To ensure integrity of data on a Jini network, each series of communications between devices is grouped into an indivisible *transaction*. The transactions themselves may comprise many back-and-forth communications and acknowledgements.

Transactions are useful because, in the case that one of the devices is removed or crashes, the other does not continually try to contact it. A transaction will either succeed or fail. If a transaction succeeds, the device stops trying that transaction. If a transaction fails, it is attempted again at a later time.

2.4.3.3 Abstraction of Devices and Services

Jini is based on the simple model that devices connect together in *communities*. The formation of these communities requires no device drivers, no new cabling, no operating systems and no human intervention. The devices on a Jini network are abstracted simply to *devices* that offer *services*.

2.4.3.4 Locating Other Devices and Services

Before devices on a Jini network can interact with one another they each must join at least one Jini community. The device does so by finding a *lookup service* (basically a name server) that keeps track of the resources of that community. Finding the lookup service is called *discovery*.

Jini supports two forms of discovery – *serendipitous* and *direct*. Serendipitous discovery means the lookup service and services in the community find each other without any advance knowledge of each others' whereabouts. Direct discovery is like "hard-wiring" a service to make use of a particular lookup service (which may not be local).

Jini uses three discovery protocols for different scenarios:

- **Multicast Request Protocol** – used when a service first becomes available to find nearby lookup services
- **Multicast Announcement Protocol** – used by lookup services to announce their presence
- **Unicast Discovery Protocol** – used by a service that knows the address of a lookup service when it starts up

A device that is performing the discovery on the Jini network ultimately ends up being passed one or more references to any lookup services present. Once a device application has discovered a lookup service, it can query the lookup service to locate another device that offers any required service. The application is given a reference to the service it requires.

The lookup service itself is like a very advanced name server, allowing the applications on the network to search for particular types of objects. The lookup service returns object references to applications querying it. Devices use the lookup service to *publish* services and to *find* services. Each service in the Jini network has a proxy object – this is the object that is downloaded to other devices and enables those other devices to interact with the service. Each service publishes a reference to its proxy object with all the lookup services it can see. This process is called publishing a service.

Devices find the lookup service itself during discovery. Searches performed by the lookup service can be done by proxy object, by a specific unique service identifier or by the attributes of a particular service.

2.4.3.5 Using Services

Once a reference to a service has been obtained from the lookup service, the device requesting the service downloads a Java object that is used to interact with the other device. This means that no device drivers are necessary – the service is available via its own native interface (which is running on the device requesting the service) and so compatibility is guaranteed.

When devices register on the community, they publish their services in the registry. Other devices can then make use of the lookup service to find these services and then use them. But what happens if a device that has registered a service is suddenly removed from the network without being able to de-register? The network will still think that the device is connected and this may cause the network to hang or even crash.

To work around this problem, Jini employs *leasing*. Thus instead of granting access to a resource indefinitely, services can only register for certain amounts of time. The grantor of the lease (the lookup service) may deny a lease or prematurely terminate a lease. Services can also prematurely terminate a lease. Leases all have an expiry date and if a service wishes to remain registered, it must attempt to renew its lease. So if a device is suddenly removed from the network, its lease will eventually expire and the service will no longer be registered. The Jini network will continue as if the device had properly de-registered. Leasing enables a Jini network to be very robust by providing a way of freeing unused or unneeded resources.

2.4.3.6 Detecting Device State Changes

Jini services need to know about external state changes (events) that occur on the network from time to time. For example, a DVD player may need to be notified that the TV is off and it can stop streaming.

There are many other examples of remote events on a Jini network. As an illustration, O'Driscoll [10] uses the example of a digital camera that is connected to the network. If the user wished to print photos by sending them from the camera to a printer on the network, the camera would simply search a lookup service for a printer and be able to print. But what if the camera was connected to the network first? Then when a printer is plugged in, it would generate a remote

event to tell the camera that a printer is available. The camera could have a print button that is greyed out when a printer is not present, and when one comes on-line, the button would become clear and the user could print.

2.4.3.7 User Interfaces

Jini UIs are based on the functionality provided by the service. The UIs are objects called User Adapters and are discussed in detail in section 3.4.2 .

2.4.3.8 Other Features

Jini offers many benefits to users of home entertainment networking technologies and to developers and manufacturers. Here are some of them:

- **Device agnosticism** – Since devices come bundled with their own interface and no device drivers are required, Jini can support just about any device – from a full Multimedia PC to a light switch. Jini does not require devices to even have Java-compatible code embedded in them – as long as another device on the network can act as a proxy for it, it can successfully participate in the network.
- **Simplicity** – No device drivers, no human intervention required for network setup, hot-plugging and unplugging – these features make the Jini network easy for users to utilise.
- **Reliability** – Because of Jini's advanced lookup server, networks form "spontaneously". Services close together automatically form communities without user interaction. Also, the Jini network is largely self-healing – leasing and a natural redundancy inherent in the infrastructure reduce the effect of key A/V devices failing.
- **Small footprint** – The code required to implement Jini on a device is extremely small – it can be placed on devices with hardly any resources – such as lighting devices.

2.4.3.9 Jini and HAVi

Since both Jini and HAVi both use Java technology, they can naturally work together quite effectively using a HAVi-to-Jini bridge. Early in 2000, Phillips, Sony and Sun announced plans to work towards connecting the HAVi architecture with Jini.

2.4.4 UPnP

2.4.4.1 Introduction

Microsoft sponsors Universal Plug and Play (UPnP) [11], an open, cross-industry home entertainment networking solution. The UPnP architecture defines a set of common interfaces

that allows users to plug any device into their home entertainment network without having to install drivers or change configuration settings. Importantly, UPnP leverages existing technologies and industry standards.

Some of the core features of UPnP are:

- **Open Standards** - by leveraging existing, widely used standards, UPnP can work on a wide range of devices and eliminates the need for large-scale testing to ensure proper interconnectivity.
- **Scalability** – UPnP scales from small networks to larger networks.
- **Plug and Play** – UPnP provides automatic methods of discovery and connectivity that involve no user interaction.
- **Small Footprint** – UPnP devices require minimal computing resources.
- **Multivendor and Mixed Media Environment** – UPnP supports mixed media and multivendor environments by using its simple interfaces and the fact that it is network medium independent.
- **Integration of Legacy and Non-IP Devices** – Although UPnP uses IP, it allows bridging devices (devices that bridge between IP and some other transport mechanism) to be used on the UPnP network.

2.4.4.2 Physical Medium

UPnP is independent of the transport medium. Figure 4 shows the high-level architecture of UPnP. The physical medium utilises IP (Internet Protocol) [27]. This widespread protocol is chosen as the transport protocol of UPnP since it forms the backbone of web software. UPnP also utilises the HyperText Transfer Protocol (HTTP) [24], the Transfer Control Protocol [28] and the User Datagram Protocol (UDP). HTTP is the *de facto* standard for transferring data across the Internet between browsers and servers and all messages sent in UPnP are sent using HTTP (other than messages sent while addressing – see next section). UDP is a connectionless protocol allows a device to send information without having to establish a connection with the target device. UPnP will use HTTP over TCP or over UDP, but ultimately all messages are transferred using IP.

It must be noted that IP itself is independent of transport mechanism, and IP packets may be transported on an IEEE1394 network using the "IP over 1394" protocol [44].

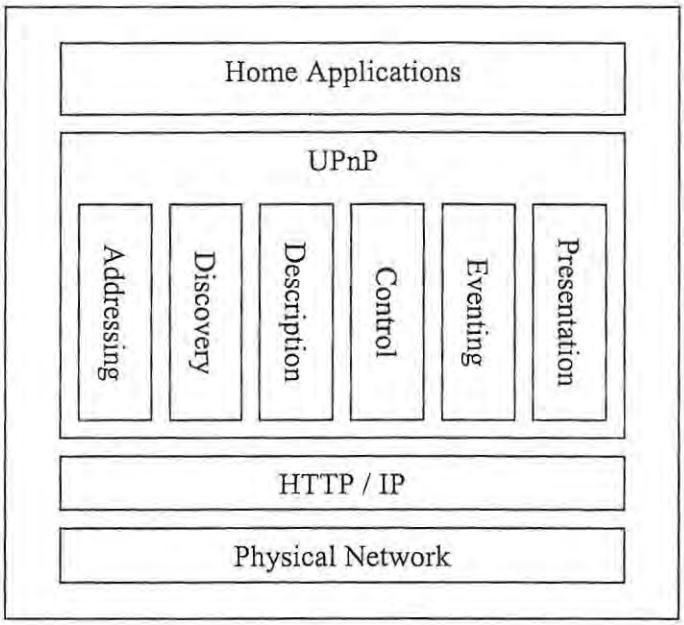

Figure 4 - The UPnP high-level architecture

2.4.4.3 Abstraction of Devices and Services

UPnP uses two terms to distinguish devices on a network. A *control point* or *user control point* contains software elements that allow it to communicate with a number of devices offering services on the network. *Devices* on the UPnP network offer *services* to the network (see next paragraph). Devices also contain software elements that allow them to communicate with user control points. User control points always initiate the communication session. Devices may also contain control points as well as provide services.

The software elements that a device contains are described using extensible Markup Language (XML). Each device has a description XML file that contains vendor-specific information about the device, such as model name and number. Devices may contain other *logical devices* as well as services. Services are functional units that the device can offer to the network. The description also contains Uniform Resource Locators (URLs) for control, eventing and presentation. Each service is described by a list of *actions* (or commands) and their associated *parameters* (or arguments) as well as a list of *variables* that represent the run-time state of the service. These variables are described in terms of type, range and event characteristics.

Using the best available method (DNS [29], AutoIP[30] or Multicast DNS [31]), UPnP devices obtain IP addresses when they first join the network. The choice between these methods depends on whether or not a DNS server or Dynamic Host Configuration Protocol (DHCP) [32]

server is connected to the network and what IP addresses have been hard-wired. This process is called *addressing* and is followed by *discovery*.

Domain Name Server (DNS) is an Internet service that translates human-readable domain names into IP addresses. The maps from domain names to IP addresses are stored on DNS servers and DNS is the protocol that is used to manage these maps. When no DNS server is present on the network, Multicast DNS is used to allow DNS requests to be multicast. Devices can then listen for requests containing their own name and respond accordingly.

Normally, computers joining an IP network obtain their IP address from a server using Dynamic Host Configuration Protocol (DHCP). The DHCP server ensures that no two computers on the same network have the same IP address. However, small home entertainment networks may not have a DHCP server, and when one is not present, UPnP makes use of the AutoIP protocol to obtain IP addresses. Devices use AutoIP to automatically choose an IP address without conflicts with existing IP addresses.

2.4.4.4 Locating Other Devices and Services

During discovery, devices utilise the Simple Service Discovery Protocol (SSDP) [23] to discover other services present on the network. SSDP allows control points to search for services and for services to advertise their presence. SSDP makes use of Multicast IP [45] which allows one message to be sent to many (or even all) nodes on the network.

When a control point initiates an SSDP search, the response it receives from a device contains a URL corresponding to the device's device description document. The control point retrieves this document using the URL provided by the SSDP search. This document proves the control point with enough information about the target in order to establish a TCP/IP connection. Devices offering services announce their presence to control points on the network. When control points are added, they search for services already present on the network. Each device advertises basic information about itself like type, identifier and a reference URL to a more detailed description.

2.4.4.5 Using Services

Following discovery, control points are aware of the networked devices, but know very little about them (assuming they do not possess any device or service specifications – see § 2.4.4.8). The control point uses the reference URL it received during discovery to retrieve a device's description document. This process is called *description*.

Once a control point has the address of the control URL of a device offering services (from its description) it sends appropriate *control commands* (actions) to the device. Control messages are described in XML using the Simple Object Access Protocol (SOAP) [25]. SOAP is used by UPnP to invoke commands remotely over the network. The effects (if any) of the action are described in the changes to the variables associated with the device. Actions on services behave like remote function calls and return any action-specific values. This process is called *control*.

2.4.4.6 Detecting Device State Changes

The description document of a device offering services includes a list of actions and run-time variables. These devices publish updates when these variables change. Control points subscribe to receive these updates (or event notifications). The event notifications are published by sending event messages. Event messages are formatted in XML using the General Event Notification Architecture (GENA) [26]. UPnP uses GENA to notify devices on the network of events that occur. When a control point first subscribes to receive event notification from a service, it receives a special event message that lists all the events and variables for that service, allowing the control point to initialise its state model of the service. Subsequently, all event subscribers are sent event messages for all events that occur within the service.

2.4.4.7 User Interfaces

A control point can present the user with a Web-like GUI on a browser if it is present in a device's description. This document is referred to as the *presentation*. Presentation pages are written in HyperText Markup Language (HTML) [33]. HTML is a platform independent language used to format pages on the World Wide Web.

The presentation page and control process can be linked. When this is the case, the presentation page responds to user actions by generating control messages. These control messages usually result in the device's state variables changing, and these changes are reflected as the presentation automatically updates the variables that it is displaying to the user. This is known as using the UPnP *presentation mechanism*.

The presentation page may also respond in other ways. For example, the page could respond to the user's actions by linking to other URLs within the device itself that cause the device to perform some action.

2.4.4.8 UPnP's Control Mechanism

UPnP's presentation mechanism is a complimentary (but not obligatory) mechanism that runs alongside UPnP's *control mechanism.* When a control point already possesses device or service specifications about certain devices, it can control it without having to rely on the presentation mechanism.

Consider the example of a TV (that knows about a specific VCR already) gets connected to that VCR. The UPnP process proceeds as normal from addressing to discovery. At this point the TV realises that a VCR that it possesses device and service specifications for is present on the network. The TV could then display an interface to the user – this interface has not been retrieved from the VCR (as would be the case if the presentation mechanism was utilised). The user could then interact with the interface and request that a certain programme be recorded. The TV would then issue the correct control command in order to perform the action. It would know the correct control command from the device and service specifications for the VCR that it already possesses.

The device and service specifications that the control mechanism would typically use are specifications that particular UPnP working groups have created – they are "standard" specifications for common devices and services.

2.4.5 AV/C

2.4.5.1 Introduction

The 1394 Trade Association (TA) is a voluntary, non-profit association that seeks to promote and grow the market for IEEE1394 compliant devices. They have a number of specifications, or standards, that the working groups within the TA have agreed upon. These standards document methods of performing IEEE1394-based operations (such as establishing and managing connections between devices).

2.4.5.2 Physical Medium

AV/C uses IEEE1394 as its physical medium. To abstract networking activities from the AV/C home entertainment networking solution itself, AV/C makes use of the IEC 61883-1 standard, Function Control Protocol (FCP) [13]. This allows AV/C to issue FCP "send" and "receive" commands without hassling about bus arbitration and clocking on and off of packets – these "low-level" operations are performed by FCP. The address space on the bus is specified in the

IEEE1212 standard [21]. This standard specifies 64-bit addressing and views all the memory of all the devices on the network as a shared memory. The most significant 16 bits specify the node address (comprised of 10 bits of bus id and 6 bits of physical id) and the remaining 48 bits are used to address 256 terabytes of data within each node. This massive block of memory is divided into registers that reside at certain addresses (such as the configuration ROM which resides in the second 1k of memory on each device).

The IEEE1394 architecture is divided into three layers – the physical layer, the link layer and the transaction layer. The physical layer encodes and decodes between symbols from the link layer and electrical impulses on the cable at the physical layer. The link layer provides addressing, error checking and framing (placing a communication from higher layers into messages that will be transported on the physical layer). The transaction layer implements a complete request-response protocol to perform IEEE1212 transactions. Transactions may be reads, writes or locks of the registers in the shared address space.

FCP is used to abstract from the transactions of the transaction layer by encapsulating device commands and responses within IEEE1394 asynchronous block write transactions. A device wishing to send a command addresses the FCP packet to the FCP Command register (address 0xFFFF F000 0B00 in the destination device's node address space). When the (low-level) asynchronous block write has been performed, the transaction layer of the destination device is notified and the FCP command register is read to obtain the packet. The destination device returns its response(s) to the FCP Response register (address 0xFFFF F000 0D00 in the source device's node address space) to complete the transaction. The header of each FCP frame includes the destination address of the device. The transaction layer of the network ensures that the FCP frame is read by the destination device.

AV/C commands and responses are in turn encapsulated in FCP frames. Each AV/C header includes a *ctype*, a *subunit type*, an *id field* and several (optional) *opcodes*. The ctype is one of five command types (control, status, and specific enquiry, notify and general enquiry) or a number of response types including accepted, rejected and not implemented. The subunit type and id (see § 2.4.5.3) are used to form an AV/C address identifying either the destination or source device (as is the case for commands and responses respectively). There exist several opcodes for each subunit type and ctype. The opcode defines the operation to be performed or the status being returned and may pertain to units, subunits or both.

2.4.5.3 Abstraction of Devices and Services

The AV/C home entertainment networking solution abstracts devices in terms of *units* and *subunits*. Each device has one unit that may contain several subunits. Each unit or subunit has a type (such as video monitor, tape recorder/player or tuner) and an instance id that uniquely identifies it. Every subunit defined by the TA has a specification that documents the services and commands to access those services that the subunit is supposed to provide.

Subunits are classified according to the following criteria [12]:

- availability of a transport mechanism
- signal input
- signal output
- signal processing
- perhaps the subunit does not have any signal input or output and is a utility of some kind
- for devices that have video and audio capabilities, separate subunits are used to represent these capabilities.

AV/C uses the terminology of *controller* and *target* devices. Controllers send commands to targets in order to use services provided on the target device.

Use is made of the AV/C *descriptor mechanism* to describe the units and subunit on a device. This mechanism supports the creation of various data structures (static and dynamic), which contain useful information about the AV/C units, subunits and their associated components. The structures are specified in such a manner that content navigation and selection of all media is as general as possible. This means that controllers can discover and access the services or media of a target – they need only know a little bit about the subunits that are present on the device – enough to interpret the information in the descriptors – content navigation is then standardised.

A descriptor is an address space on a target that contains attributes or other descriptive information. For example, the *subunit identifier descriptor* contains information unique to that type of subunit. All subunits of the same type will have the same subunit identifier descriptor. Most information in the descriptor structures is static, but may be change depending on the type of subunit and the technology it implements.

Other important descriptor structures are *object lists* and the *objects* they contain. Objects are defined generically and are defined as needed. An object represents each service in a subunit.

Objects may also be used to represent media. For example, objects could represent tracks on a compact disc. Objects lists are generic containers that contain objects or other object lists.

Objects and object lists can be used to model relationships where one entity is composed of several sub-entities. For example, an audio compact disc can be composed of the tracks on the disc. Object lists are arranged hierarchically and can be continued to any arbitrary level of complexity.

Each subunit identifier descriptor will have a reference to the root of the subunits object list hierarchy. A subunit identifier descriptor may refer to several root lists. Importantly, all the information contained in descriptor structures is stored in the device in a device-specific manner – that is, the manner of storage is not specified by AV/C. AV/C only specifies how this information is to be presented.

2.4.5.4 Locating Other Devices and Services

The IEEE1394 bus configures itself. When devices are added or removed from the network, device ids are allocated to each device. These ids are unique to each device, but may change when a *bus reset* occurs (bus resets occur when devices are plugged or unplugged, but can also be generated by software). The root node of the network maintains a list of all the device ids on the network so that any node can address any other node.

Controller devices use AV/C commands with various opcodes to obtain information about target devices, such as unit and subunit information. Using descriptor search commands, the controller can traverse the descriptor structure of the target and obtain object references to any object it is interested in using.

2.4.5.5 Using Services

Once the type of subunit has been identified by the controller using the descriptor mechanism, the controller issues commands to use the services provided by a specific subunit. For example, if an audio subunit is present on a target, the controller can issue any AV/C Audio Subunit Specification [22] control command.

2.4.5.6 Detecting Device State Changes

Device states are modelled by variables in the various subunits that it contains, and detection of state changes can occur in two ways: either the subunit spontaneously sends a message to the

controller to indicate the change, or the controller uses the status, general enquiry or specific enquiry commands to detect changes in the subunit.

2.4.5.7 User Interfaces

The TA defines a subunit that implements user interfaces for units (each unit may only have one of these) called the Panel Subunit. This specification is discussed in detail in section 3.4.3

2.5 Comparison of Communication Solutions

2.5.1 Physical Medium

All home entertainment networking solutions need to be able to support high bandwidth applications and streams. This means that high bandwidth mediums must be utilized at the physical layer. HAVi and AV/C comply immediately since they are solutions designed to operate on IEEE1394 networks. UPnP and Jini, however, do not require a high bandwidth network at the physical layer. Thus home entertainment networking solutions implementing UPnP (which is IP based) must ensure that the physical layer can support high bandwidths. Jini uses Java bytecodes, and hence can run on very limited networks. However, when used by home entertainment network solutions, it must ensure a physical medium that has high bandwidth.

2.5.2 Abstraction of Devices and Services

The abstraction of devices and services is an important feature of home entertainment networking solutions since the level of abstraction affects the amount of device-specific knowledge that is required to discover and use devices and services on a home entertainment network. For example, since AV/C abstracts its services at a fairly low level using defined subunits, a fair deal must be known in advance about the capabilities of the subunit in order to utilize its services. UPnP, by using XML to describe its devices and services, draws on a higher level of abstraction and hence very little information is required by a controller device to be able to utilize its services. Similarly, HAVi and Jini, both using objects to abstract their devices and services at a high level, allow a level of generality that AV/C does not provide.

2.5.3 Locating Other Devices and Services

There are three methods used by the four home entertainment networking solutions to effect device and service discovery on a home entertainment network: *inherent discovery*, *announcements* and *registries*.

Inherent discovery is used by AV/C; devices become aware of each other at the physical layer and from this knowledge can obtain further information about the devices and their services. This inherent knowledge comes from the automatic network configuration performed by the IEEE1394 network when devices are plugged and unplugged.

UPnP has two methods of discovery - announcements and searches. Each device broadcasts a message to announce its presence on the network. This announcement simply contains an URL to the description document of the device. Other devices can then use this URL to obtain further information about the services offered by the announcing device. Devices may also search the network for devices that offer certain services. Both these methods utilize SSDP packets and IP multicast addresses.

HAVi and Jini both make use of registries to implement discovery. In HAVi, all DCMs and FCMs register with the system registry when they join the network. In Jini, devices join a community as soon as they are added to the network. The community has a lookup service, which is conceptually the same as a registry. Devices on either network can then search the system registry (the registry for HAVi or the lookup service for Jini) for services.

Inherent discovery, announcements and registries obviously have different advantages and disadvantages. Inherent discovery and announcements are both fast and efficient methods of discovery, but divulge very little information about the discovered devices. They also make searching for specific services cumbersome (since all devices on the network must be queried until a required service is discovered). Registries are more efficient when it comes to searching for devices and services (since all the devices and services are registered in a central location), but require overhead to manage the registries and ensure that registry information is valid.

2.5.4 Using Services

Once devices and services have been discovered, two methods of using services are employed by the solutions: *running* and *commanding*.

HAVi and Jini both download the service object and run it locally. All functions that are actually performed on the target device are run like remote function calls. This method is called running a service. Running services has the advantage that no device drivers are required in order to successfully use services, since the service code itself is exchanged. Only a Java Virtual Machine is required for the services to be run.

UPnP and AV/C employ commanding in order to use services. Commanding happens when commands are sent from the controller to the target, and requires drivers that "translate" between the UPnP or AV/C commands and the native commands that perform the service.

2.5.5 Detecting Device State Changes

Each of the four home entertainment networking solutions uses a different mechanism for detecting device state changes (events).

HAVi uses its *event manager* for detecting events. This approach means that services requiring event notifications must register to receive them, but has the advantage that only registered services receive these event notifications.

Jini employs *remote events*, where services notify one another of events that have occurred. This is useful to the lookup service, because when new services join the community, the lookup service can send an event to notify the rest of the network that new services have been added. This means that all devices will be able to see all the events on the network, which could potentially flood the network.

Eventing occurs when a controller device registers to receive event notifications for certain events from the target itself, as is the case in UPnP. This approach is limited, since all events that occur in a service must be received – there exists no filtering mechanism to allow only certain events generate notifications. However, this method is the simplest of the four to implement since it is inherent in the service descriptions.

AV/C controllers may poll targets for configuration changes using status, specific enquiry and general enquiry commands or receive NOTIFY messages from targets. Polling usually reduces network traffic (since polling can be less frequent than event notifications), it has the disadvantage that it is difficult for the controller to gauge when exactly an event occurs.

2.5.6 User interfaces

The UIs employed by the various solutions differ widely. Chapter 3 discusses the UIs in detail and comparison is drawn between them in section 3.5

2.6 Summary

This chapter examines three important topics pertaining to home entertainment networking: the economics of home entertainment networking, elements of home entertainment networks and popular home entertainment networking solutions.

Research shows that there is a demand for new and improved technology in the home entertainment networking industry. It can be shown that better devices and improved connectivity allow for more (and more complex) services to be provided. It is also shown how different tier markets in the industry influence one another.

Core components of a home entertainment network include Electrodomestic Network Devices (ENDs), connection mediums or Information Appliance Networks (IANS) such as IEEE1394 and Ethernet, communication protocols and residential gateways.

This chapter also discusses the features of four primary home entertainment networking solutions – HAVi, Jini, AV/C and UPnP. Table 1 shows a comparison of the common features of these solutions.

	HAVi	Jini	UPnP	AV/C
Physical Medium	IEEE1394	Any (Java byte-code based)	Any (IP based)	IEEE1394
Device/Service Abstraction	DCMs and FCMs (Objects)	Java Classes	Device/Service XML Descriptions	Units and Subunits
Locating Other Devices/Services	Using Registry	Using Community Lookup Service	Using Addressing, Discovery and Description Processes	Using Descriptor Mechanism
Using Services	Running a DCM or FCM	Leasing Objects and Running Java Bytecodes	Using Control Process	Using AV/C Control Commands
Detecting Device State Changes	Using Event Manager	Remote Events	Using Eventing Process	Using AV/C Status Commands
User Interfaces	Level 1 and 2 Uis	User Adapters	Presentation HTML File	Panel Subunit

Table 1 - Comparison of the common features of the four communication solutions

Chapter 3 - Remote Configuration on Home Entertainment networks

3.1 Introduction

Home networking involves many diverse areas – from connecting devices and communicating between them to carrying data streams and providing bandwidth. Since there are so many areas, this thesis inspects one area of home entertainment networking and attempts to discuss home entertainment networking solutions in general from the experience and insights gained by examining this area – *device configuration* and especially *remote configuration*.

The term remote configuration has many meanings throughout the field of computer science. This chapter defines the meaning and use of remote configuration on home entertainment networks, and presents some possible solutions manufacturers could use to implement remote configuration.

3.2 Defining Remote Configuration

As far as it concerns home entertainment networks, remote configuration can simply be defined as *the ability of a home entertainment networking solution to allow one device to change the configuration state of another device.*

A distinction is drawn between the *configuration state* of a device and a *functional state* of a device. The functional state describes what the device is currently doing – for example, a video cassette recorder (VCR) may be playing a tape – the functional state would be "playing". If the VCR were rewinding, the functional state would be "rewinding". The configuration state describes how the features of a device are currently configured. For example, the equalizer settings of one configuration state of a Hi-fi might be set to "Rock", while another configuration state (with different equalizer settings) might be "Jazz".

Configuration states are typically changed less frequently than the functional states of the device. Users will press the "stop" and "play" buttons on a hi-fi (part of its functional state) far more frequently than change its equalization settings (part of its configuration state).

While functional state changes could be achieved remotely, this thesis focuses on configuration changes.

Remote configuration can be done in various ways, but most commonly it is accomplished by a device (the target) uploading an interface to its configuration onto another device (the controller). The controller then presents this interface to the user, who then interacts with the interface. All commands that are issued by the user are then transmitted from the controller to the target as they happen and the target device changes its configuration state. In order to remain consistent and ensure that the UI is always up to date, state changes occurring on the target device must be communicated to the controller as they happen.

Hence, user interfaces (or UIs) become the core problem when considering remote configuration. How UIs are specified, stored, transmitted, and displayed will affect the ability of a device to offer remote configuration.

Importantly, remote configuration also allows properties of the target device to be altered that cannot be altered by other control mechanisms.

3.3 User Interfaces

As industry moves from *static* devices to *dynamic* devices, the functionality and complexity of the devices increases. For example, fridges present static "interfaces" – they always look the same and people know what to expect from them. Now consider a fridge Electrodomestic Network Device (END) (see § 2.3.1). Perhaps it can determine what vegetables have run out, and perhaps it is linked to an on-line grocery store – at the touch of a button, the homeowner can order replacement vegetables on-line. The fridge END is more functional, but also more complex.

This means that in order to increase the user's ability to use the END, effective user interfaces (UIs) are required. An advanced END will go unused if its user does not know how to interact with it meaningfully.

Eustice et al. define a device called the Universal Information Appliance (UIA*) [4]. Every time this device comes into (wireless) contact with any other device (the target), it serves the user the

* © Copyright 1999 by International Business Machines Corporation.

UI of the target and becomes the gateway between the user and the target. There are some important features inherent in the UIA that are worth highlighting.

3.3.1 The UIA

Most devices we interact with on a day-to-day basis have static interfaces – coffee makers and toasters always look more or less the same. Eustice et al. postulate that the electronic world need not be constrained by physical limitations. They imagine users could interact with a device that gives them custom interfaces to ATMs, televisions, alarm systems and any other system requiring human interfacing.

They state that the components required to realise the UIA are:

- **A platform independent interface to the digital domain** – Eustice et al. develop a wearable computer that the user carries around
- **A wireless infrastructure** – the device, being extremely portable, is kept connected permanently or intermittently via a wireless link
- **Communication middleware** – this is a uniform method of accessing information and services across heterogeneous networks (networks that are comprised of different technologies) automatically and without user input

Eustice et al. point out that while physical systems have fixed interfaces (doors have doorknobs, for example), electronic interfaces can have any appearance. The problem Eustice et al. encounter is how to decide on the best interface to a logical function (such as redistributing resources on a network of computers). They note that most UIs are based on the *common denominator* of human experience and not on the *multiplicity* of human experience.

This has led to people being bombarded by a plethora of untailored, incompatible and non-uniform interfaces. Eustice et al. claim that the UIA transcends the general interface problem by "extending the concept of remote interaction beyond direct control of televisions and video cassette recorders (VCRs) to dynamic interaction with all electronic entities and digital information sources in one's environment."[4]. Remote interaction encompasses all the remote communication that occurs on a network, including remote configuration.

The architecture for the UIA revolves around events. As middleware, they use IBM's TSpaces [5]. This allows them to implement eventing in a platform-independent manner.

The middleware acts as messenger, database and file system. It stores a list of rules in a database that register actions to events. The rules have this format:

Rule = Client.EventClass (parameters) → Client.ActionClass (parameters)

In other words, "an event in entity A triggers an action in entity B". They further define "event" very loosely – any unit of action that can be parameterised becomes an event. This allows any client anywhere to register any event class and any event action at *run time*.

Eustice et al. define a language called MoDal (Mobile Document Application Language)[6] to represent UIA applications and interfaces. Importantly, they use XML (eXtended Markup Language [16]) to describe MoDAL, since XML is a relatively high level language and is also platform independent. UIA applications and interfaces described in MoDAL are compact enough to be dynamically retrieved over the network.

To illustrate the usefulness of the UIA in the field of remote configuration, consider the following example: a user is watching a program on the television in the living room when guests arrive. The user, wishing to record the rest of the program, retrieves the UI of the VCR in the bedroom onto the UIA and quickly selects the program (changing the configuration state) and presses record. The user then selects the hi-fi in the living room (retrieving its UI onto the UIA) and changes the sound configuration to "JAZZ", changes the source to "CD Shuffler", and presses play.

Without interacting directly with either of the target devices (the VCR and the hi-fi), the user has configured them quickly and easily. Using the UIA architecture, the UIA retrieves the interfaces of the target devices by requesting their MoDAL UI descriptions. It then displays the UI of the current target on its display. The UIA relays the user's commands back to the current target, which carries out the commands and changes its configuration state. Figure 5 shows how the UIA's interface looks different for each device while the scenario is being fulfilled.

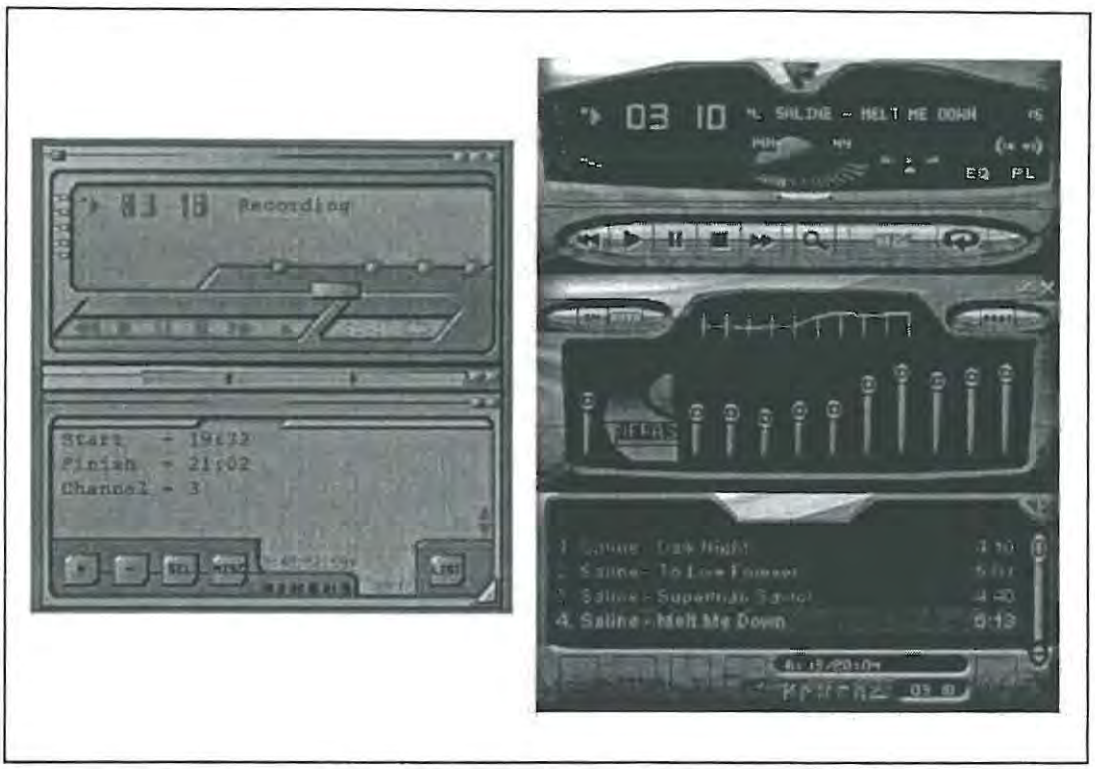

Figure 5 - The UIA interfaces for a VCR (left) and a hi-fi (right)

3.3.2 Desirable UIA Features for Home entertainment networks

Most device interfaces within home entertainment networks are static, and as such differ significantly from the dynamic interfaces of the UIA. However, the UIA highlights some features that it possesses which are required for successful operation and which are useful to remote configuration UIs:

- **A high bandwidth connection to the network** – in the case of the UIA, this connection is wireless – most home entertainment networks will use some sort of cabling. The essential requirement is a fairly high bandwidth with negligible latency.
- **Platform independent middleware** – different manufacturers manufacture the devices that are communicating and if they are to understand each other they require a cross-platform and platform independent message service (or middleware). The middleware must also be platform independent since it also serves as a database and file system on the network.
- **Run time Interfaces** – users do not want to wait for their devices to recompile for new interfaces – new interfaces and applications must be able to be added to the system at run time.

- **Compact document description** – not only does a well defined and compact language make implementing new applications and interfaces easier and faster, but it reduces network traffic and latency.
- **Uniformity** – one of the aims of the UIA is to present the user with interfaces that are uniform. Manufacturers can enhance the experience of the user and the usability of their products if they use uniform interfaces.

3.4 Solutions to Remote Configuration

Having looked at some of the requirements of UIs on home entertainment networks, remote configuration solutions must be found that conform to these requirements. This section views some of the solutions contained in some of the popular home entertainment networking solutions discussed in Chapter 2.

3.4.1 HAVi's Solution to Remote Configuration

HAVi caters for remote configuration by using a model called Data Driven Interaction (or DDI) [8]. DDI defines a *DDI Controller* (the software element or device performing the controlling) and a *DDI Target* (the software element or device being controlled). The controller uses a description of the UI obtained from the target called *DDI Data*. The DDI Data consist of a set of *DDI Elements*. It must be noted that DDI can be used to configure as well as control devices remotely.

3.4.1.1 DDI Elements

DDI Elements form the building blocks for the UI. They range in nature from simple text labels to buttons, ranges and text entry boxes.

All the DDI elements on the target are arranged into a hierarchy. The hierarchy serves three purposes:

- it allows "organised" navigation through the elements
- it indicates which elements logically belong together (and thus indicates that they should be rendered together)
- it lets the target know which element changes the controller should be notified of

The first element in the hierarchy is said to be the root element. If the controller is given a reference to the root element, it can use the root element to navigate to any arbitrary element in the hierarchy.

DDI elements are divided into two types – *organisational* and *non-organisational*. Organisational DDI Elements are DDI elements that are used to determine the hierarchical organisation of the other elements. There are two organisational elements: *panels* and *groups*.

Panels and groups are conceptually containers and both have (mandatory) lists of the element ids of the elements they contain. Panel elements may not be contained in other panels or in groups. However, panels are may be referenced in other panels by using the panel link elements which allows the user to navigate between panels. Groups may be contained in panels and other groups.

The DDI Input Device Model is also modelled abstractly to allow for flexibility in the implementation. Once of the ways this abstract modelling is achieved is to define whether or not the user may modify elements or whether or not elements generate "UserAction" messages. If elements have any user-modifiable attributes, they are termed *user-modifiable* (for example a slider); otherwise they are *non-user-modifiable* (such as buttons and icons). DDI elements that can be used by the controller to send "UserAction" messages are termed *interactive*, while those that do not are termed *non-interactive* (and are typically used to convey status or static information to the user). Panels and groups are non-user-modifiable and non-interactive.

All elements (except panels and groups) are non-organisational elements. These elements occupy "leaf" positions in the element hierarchy. Each element has attributes that suggest to the controller

- how the element is to be rendered (displayed to the user)
- whether or not the element is interactive (and how)
- whether or not the element is user-modifiable (and how)

The DDI non-organisational elements defined thus far are: text, panel link, button, choice, entry, animation, show range and set range.

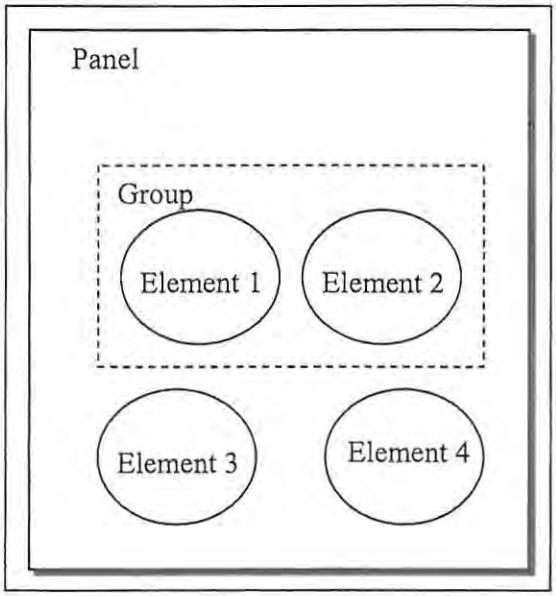

Figure 6 - An example of how DDI elements are arranged in panels and groups

Figure 6 shows an example of how non-organisational elements are contained within panels and groups (organisational elements). Figure 7 shows how DDI elements are arranged in a DDI hierarchy and how the hierarchy can be navigated from the root element.

3.4.1.2 Navigation Through the DDI Hierarchy

Sometimes the controller either chooses not to render entire panels or is incapable of rendering entire panels. When this happens, some of the elements in the panel are displayed (seen elements) and the rest of the elements in that panel are not displayed (unseen elements). The controller must provide some (implementation specific) method of bringing unseen elements into view. It might do this by using scroll-bars, for example. However, the implementation specific navigation may not generate "UserAction" messages, though it may cause the controller to obtain further elements from the target. This process is called *controller-driven navigation*.

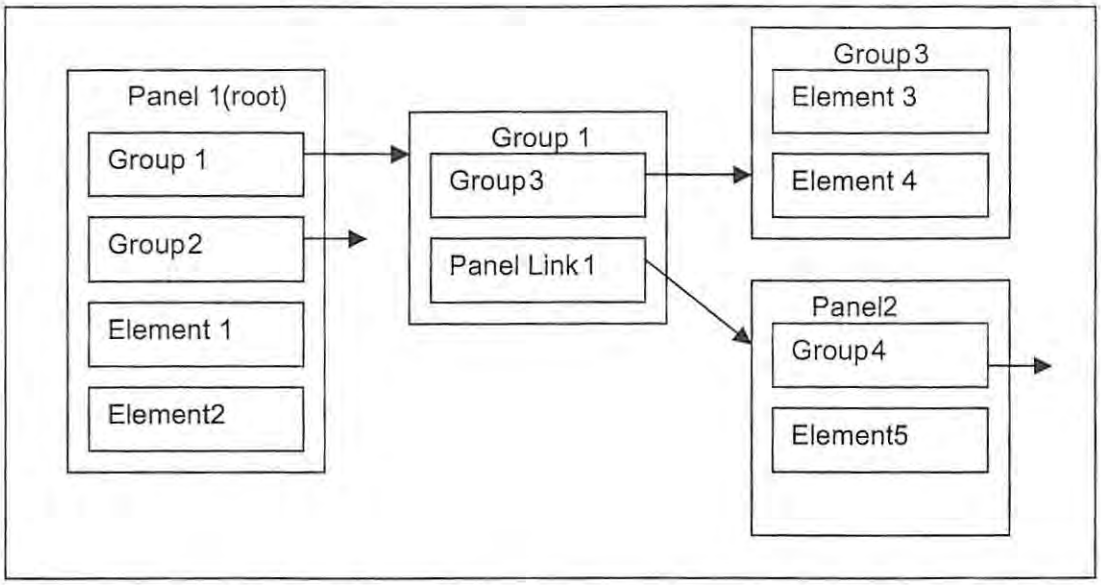

Figure 7 - The DDI hierarchy is navigable from the root element (Panel 1 in this case)

A second method of navigating through the DDI hierarchy is defined in the DDI model. It exploits a DDI element called a panel link element. This non-organisational element allows the user to switch from the current panel to other panels by selecting it. When this type of navigation is used, it is called *user-driven navigation*.

The DDI model allows a target to specify element hierarchies of arbitrary depth below the root panel. It also places no topological restriction on targets for relations between panels (for example cycles within the hierarchy are permissible).

3.4.1.3 Notification Scope for Target DDI Changes

At any time, a target may send "NotifyDDIChange" messages to a controller to notify the controller of any state changes that may have occurred to any elements – it may also send these messages in response to "UserAction" commands from the controller. However, this could lead to large amounts of traffic on the network, and the DDI employs *notification scope* to reduce this traffic.

The controller will specify to the target a description of a portion of the DDI hierarchy that is currently of interest – the *current* notification scope. The target will then only notify the controller of changes that occur to elements in this portion of the hierarchy. Any other elements may change, but the target will not notify the controller of these changes. The controller may change its notification scope at any time.

3.4.1.4 Data Driven Interaction

Both the controller and target are HAVi software elements residing on Full Audio/Video (FAV) or Intermediate Audio/Video (IAV) devices (see § 2.4.2.3). They may reside on the same or different devices (obviously for remote configuration they would be on different devices) and are implemented in Java bytecodes or in native code. They interact by sending HAVi messages to each other.

The controller interacts with a user by using the input and output devices (typically) of the device on which the controller resides. The I/O is implemented in an implementation independent manner. The target controls the device in which it resides in an implementation independent manner as well. The DDI model specifies interactions in a generic manner – thus a controller does not need to be implemented with a particular target in mind. DDI Data represent all target dependencies.

Figure 8 shows the typical DDI message sequence scheme. Remote configuration begins by the controller sending a "Subscribe" message to the target. The target notes the id of the software element that sent the message. When any events occur on the target that will be of interest to the controller (such as configuration state changes), the target will use the id of the subscribing controller to send it "NotifyDDIChanges" messages. It then returns the id of the root DDI element. The controller uses the "GetDDIElement(id)" operation to obtain any element it requires by navigating through the DDI hierarchy from the root element. The controller renders the DDI elements as they are retrieved from the target and sends any user commands that are specified in the DDI Data to the target using "UserAction" operations. Once the controller has unsubscribed, no more "NotifyDDIChange" messages are sent to it from the target.

The target responds to "UserAction" messages sent from the controller to the target. The response includes any information about the success or failure of the command or state changes that may have occurred in the target as a direct result of the action.

The "GetDDIElement" operation takes an element id as an argument and returns the actual element. Other operations exist to return lists of element ids or lists of elements. "Large" elements (such as bitmaps) are obtained by the controller by making use of the "GetDDIContent" operation.

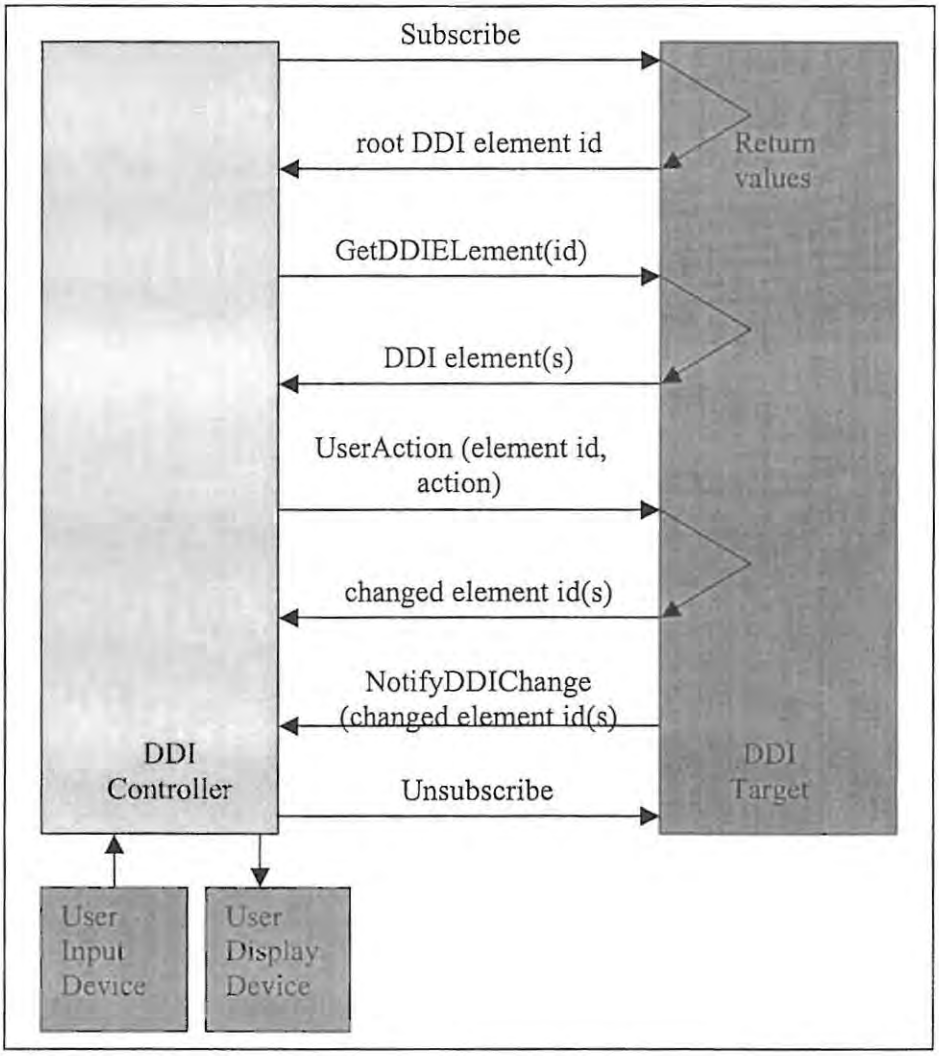

Figure 8 - The Typical DDI Message Sequence Scheme

It is possible for one target to be controlled by more than one controller and for more than one target to be controlled by one controller. Typically, the controller acts as a UI-controller and the target is typically a HAVi Device Control Module (DCM), although any application may act as either controller or target.

3.4.1.5 The DDI Output Device Model

The DDI defines an output model in an abstract way so as to allow application independence. The DDI data that are presented to the controller are "suggestions" of how the UI should look and feel. The device is responsible to render the elements as best it can.

Each DDI element has a set of attributes (like size, position, colour, etc.). Some are mandatory (these will always have values) and some are optional (they may or may not have values). Every DDI element furthermore has at least one mandatory label attribute (a string value).

There are three broad styles for a DDI controller to use when it presents a UI:

- **Full capability** – all of the elements are displayed as required by their attributes
- **Intermediate** – some of the elements are displayed
- **Basic** – only very few elements are displayed at any time; however, at least text string label attributes are rendered

Many elements have attributes that describe their size and location on the display device. The display device is assumed to be a rectangular array of discrete pixels. The elements are arranged hierarchically within organisational elements (panels and groups).

3.4.1.6 The DDI Input Device Model

Controllers receive input from the user in an implementation-specific manner. Controllers must, however, allow the user to somehow

- Change the value of a user-modifiable attribute of a user-modifiable element (for example changing the value of a slider). This causes the controller to send a "UserAction" message to the target with arguments depending on the nature of the element.
- Select any interactive element (for example press a button). This also causes the controller to send a "UserAction" message to the target with arguments specifying the particular selection
- Change the current panel to another panel. This causes a "UserAction" message to be sent from controller to target and, typically, the controller to obtain the new DDI elements contained in the new panel.
- Change the display to render elements that are not yet visible (for example scrolling down a panel). This does not typically cause "UserAction" messages to be sent and may cause visible elements to be "un-rendered" in order to make space on the display for new elements.

3.4.2 Jini's Solution to Remote Configuration

One of Jini's unratified standards is the ServiceUI architecture [7]. This standard is a first attempt at standardising how user interfaces are attached to Jini services. Remote configuration on Jini networks then falls under this standard, since a device might have a Jini service for configuring it and then a ServiceUI attached to this service to allow a user to run the service.

3.4.2.1 Separating UI and Functionality

Traditionally, UIs are designed to be built into applications. This results in a tight marriage of UI and functionality. Jini's ServiceUI architecture attempts to change this paradigm by separating UI from functionality.

The way this is done is by encapsulating UI and functionality into objects. The point of coupling between the functionality and UI objects is the Service Object Interface. The Service Object Interface describes "what" the service does. Furthermore, the only way to access the functionality of the service is through the Service Object Interface.

3.4.2.2 User Adapters

Service Object Interfaces capture the entire functionality of the service that they provide an interface for. In order to access the functions of the service, only a reference to the service object interface is needed. The service object does not include any UI code whatsoever.

In order to supply a UI to the user, a separate UI object must be created. This UI object "adapts" the service object interface into a form that the user can interact with. Figure 9 shows this diagrammatically. The UI object is the only object the user directly interacts with (the user interacts with the UI object by issuing commands using the UI). The Service object abstracts the services that reside on the device.

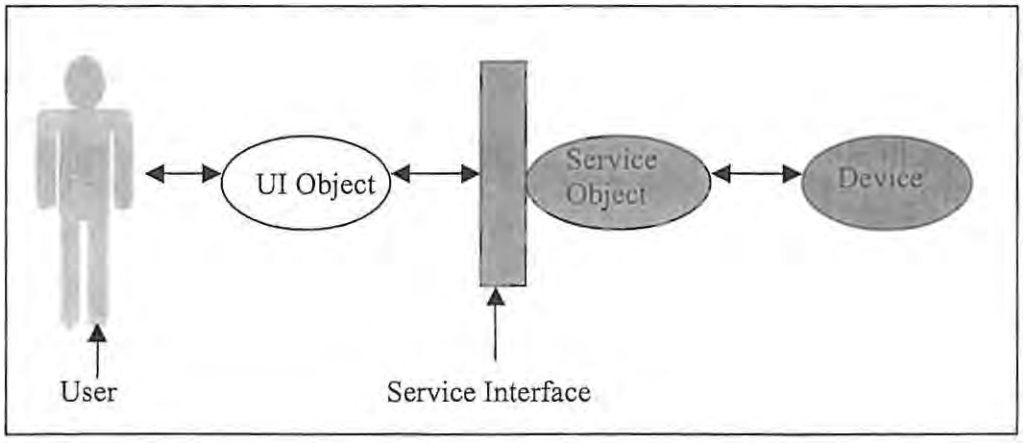

Figure 9 - A user interacting with a service via an UI Object

The Jini ServiceUI standard notes that the UI object could display to the user a graphical UI component, such as any AWT[1] UI component, but may not necessarily be graphical – it could be a speech interface or any other kind of human interface.

The advantage of separating UI and functionality now becomes clear – many UI objects can be associated with the same service. This allows UIs to be tailored to the preferences or capabilities of the user.

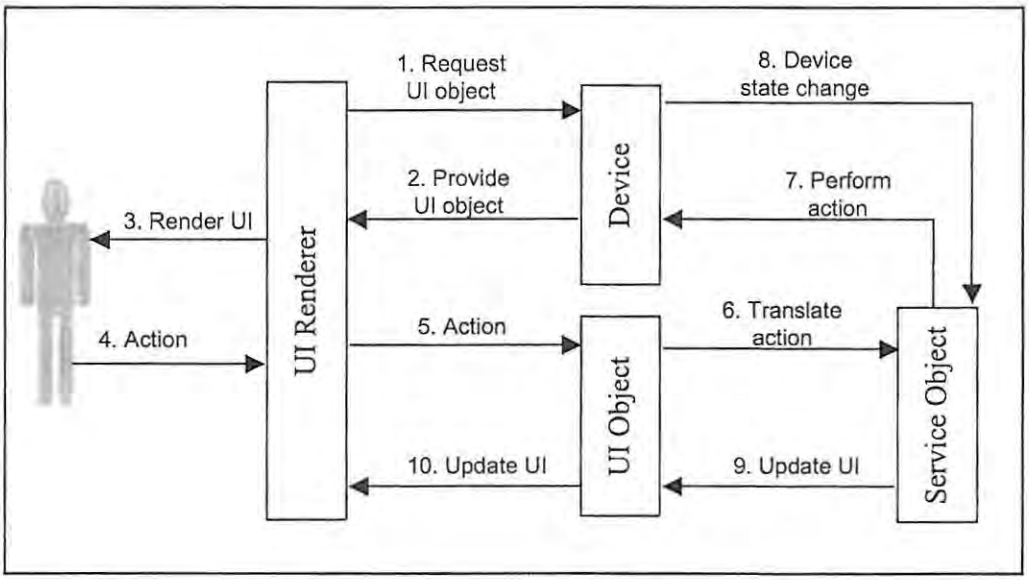

Figure 10 - Interaction between the User, UI Renderer, UI object, Service Object and Device

Figure 10 shows the interactions between the user, the UI renderer, the UI object, the Service object and the device. The UI renderer is the device that the user directly interacts with, but since the UI is not necessarily graphical in nature, the term "display" would be misleading. The UI renderer renders the UI to the user – whether it is a graphical display or a speech interface. The UI renderer requests the UI object from the device. The device then returns the UI object, and the renderer uses the methods and attributes of the UI object (detailed in the following section) to render the UI. The user then interacts with the UI, issuing commands. The UI renderer relays these commands to the UI object, which "translates the action" by mapping the action to an interface command that the Service object understands. The Service object in turn relays the action to the device and the action is performed. Any state changes or events

[1] Java's Abstract Windowing Toolkit

occurring in the device are relayed to the service object which tells the UI object to update the UI. The UI object then updates the UI.

3.4.2.3 A Closer Look at Jini's Service UIs

Each UI object has three elements:

- the **UI** itself
- a UI **factory** that produces the UI
- a UI **descriptor** that describes the UI

The factory produces and returns UI objects. The descriptor is a container for the factory and objects that describe the UI produced by the factory. Each UI descriptor has four public fields:

- **factory** – a reference to the UI factory
- **attributes** – a list of attribute objects describing the UI produced by the factory
- **toolkit** – a string that gives the name of the package needed to create the UI
- **role** – a string giving the name of a Java interface type

Using the attributes, toolkit and role fields, client programs can decide on the best suited UI – once decided upon, the factory is invoked to create the UI object.

Of particular interest is the role field. So far, the Jini service object architecture defines only three roles – MainUI, AdminUI and AboutUI. These roles have different relationships with the functionality that the UI is interfacing to – the service that is being offered by the device to the network. The MainUI is for interacting with this service; the AdminUI is for administrating the service and the AboutUI gives information regarding the service.

Although anyone can potentially create a role, programs that use the new roles will need some prior knowledge of the roles if they are to use them. So the architecture suggests that only the Jini consortium should make new roles, allowing programs to have advance knowledge of standard roles.

When creating a UI object for a service, the descriptor of the UI object is included in the attributes list of the service, thus linking the UI object to the service.

3.4.3 The AV/C Solution to Remote Configuration

One of the subunits defined by the 1394 TA for the AV/C home entertainment networking solution is the Panel Subunit [12]. It is this subunit that provides user interfaces to allow control and configuration of devices from a remote location on a home entertainment network using AV/C.

The panel subunit bases its UIs on panels. A panel is a container of elements or groups of elements and usually has a caption. Each of the elements within a panel are graphical "widgets" that allow the user to control the target device. Examples of elements are buttons and text entry boxes.

3.4.3.1 The Panel Subunit

The AV/C Panel Subunit provides data structures that describe the GUI elements as well as a user-manipulation command set for accessing these controls.

Users will use the controller device to interact remotely with a target device. The controller obtains the description of the target's GUI (the panel data) from the target. It then renders the panels and elements contained within the panel according to its capabilities. The user then manipulates the widgets and the corresponding actions are relayed to the target device. The target device can also send updates of its current state to update the GUI.

3.4.3.2 The AV/C Panel Subunit Model

The AV/C Panel Subunit model describes a way of accessing physical or logical controls of a device from a remote location. The AV/C Panel Subunit specification refers to devices as *control devices* (devices that display GUIs) and *target devices* (devices that are remotely controlled). The AV/C Panel Subunit model uses asynchronous connections [15] (see § 4.6.2 for details of asynchronous connections) to pass GUI information from targets to controllers. Each device is allowed only one panel subunit even if it has more than one logical or functional unit.

The panel subunit does not directly deal with media streams – it is responsible for translating user actions performed remotely to local internal actions. Sometimes these internal actions may involve media streams.

The AV/C Panel Subunit model defines two modes of operation – *direct mode* and *indirect mode*. In the direct mode, the controller obtains a GUI description from the target and renders it. The controller then conveys user actions to the target. In the indirect mode, the panel subunit on the target receives user commands but does not care about the GUI that is being used – it acts like a usual remote control. The GUI may be transmitted as a static bitmap image, and the controller can recognise user actions by detecting which part of the bitmap the user is interacting with.

The AV/C specification makes use of *plugs*. These plugs are logical, and represent points at which streams of data can flow, either out of the device or into the device. The plugs have associated attributes, such as id and information as to what sort of stream it supports. Plugs can be *source* or *destination* plugs. Destination plugs are used for streams coming into the device, while source plugs are used to channel out flowing streams. Connecting a source plug to a destination plug is known as establishing a connection between two plugs, and this connection allows the stream to flow between the plugs. Each panel subunit has no destination plugs but can support one or more source plugs (depending on implementation). The plugs are of the format specified by the IEC in [13] and a controller uses them to establish an asynchronous connection between itself and a target device.

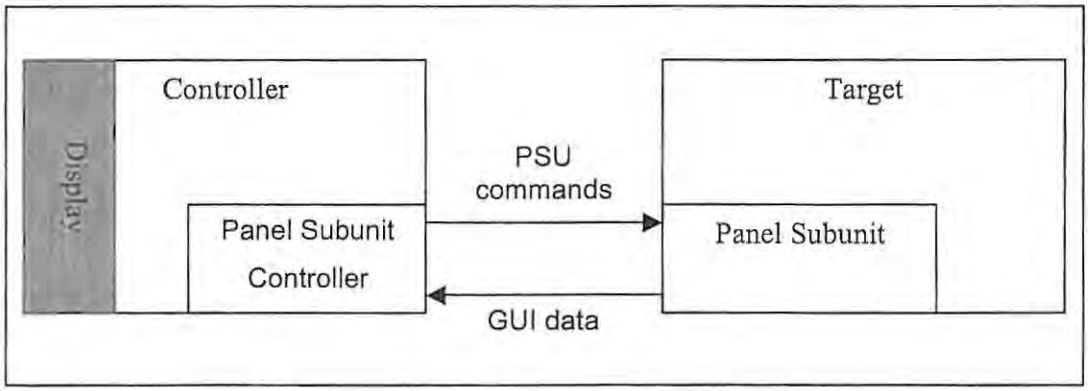

Figure 11 - The Panel Subunit model

Figure 11 shows the AV/C Panel Subunit model. Sessions occur as follows:

- **The controller opens a session** – the controller requests ownership of a source plug from the target device's panel subunit.
- **The controller establishes an asynchronous connection** – the controller connects one of its input plugs to the source plug of the target's panel subunit. The target can now push GUI data onto this connection and the controller can pop this data off when it arrives.

- **The controller requests GUI information** – the controller asks the target to push its GUI onto the asynchronous connection.
- **The target sends the GUI** – the target sends GUI information – this could be an entire panel (see section 3.4.3.4) or an individual GUI element, depending on the arguments passed by the controller in the previous step.
- **The user interacts with the GUI** – the controller renders the GUI and the user interacts with it. The controller sends any user actions performed to the target. The controller does not explicitly know what happens as a result of these actions.
- **The target sends any updates** – any state changes that occur in the device that affect the GUI are transmitted to the controller spontaneously by the target. This is how the controller obtains any response to actions performed by the user.
- **The controller closes the session** – once the user has closed the GUI, the controller breaks the asynchronous connection and closes the session with the target.

3.4.3.3 Input and Output Device Models

In order to allow controllers to implement GUI displays and read user inputs in device-dependant ways, the AV/C Panel Subunit model defines input and output models in a general and abstract way.

The AV/C Panel Subunit model defines three classes of panel subunit controller (the Output Device Model):

- **Full capability controllers** – these controllers have large, high resolution displays and can render all GUI elements exactly as they are (size, position, colour, etc.).
- **Intermediate controllers** – these controllers cannot render all the GUI elements exactly. Certain resolutions and/or attributes may be ignored (for example sound clips).
- **Basic controllers** – these are controllers that can only manage text strings and no graphics.

Since the input mechanisms of controllers can also vary greatly, the AV/C Panel Subunit defines its Input Device Model to allow general input mechanisms for its GUIs. GUI elements can either be

- **Interactive** – they invoke the controller to send user action commands to the target (for example a button), or
- **Non-interactive** – they do not invoke any actions (for example a label).

Furthermore, GUI elements may be

- **User-modifiable** – (such as sliders) or

- **Non-user modifiable** – (such as icons).

3.4.3.4 GUI Layout and Presentation

The AV/C Panel Subunit model defines the data structures necessary to communicate a GUI to the user. The target "suggests" how the controller should render the GUI – but since the display capabilities of controllers may differ greatly, the AV/C Panel Subunit model remains flexible on this point. The model can guarantee that something will be displayed, but not how it will look. For instance, a computer monitor could display all the graphics of a GUI, while a cell-phone LCD display may only display text strings.

The layout rules for the GUI elements are based on relative geometric co-ordinates (x and y co-ordinates). GUI elements are arranged hierarchically and are positioned relative to their parents. The elements are divided into *panels* (top level containers with no parents) and *groups* (of elements). Groups are children of the panel they are in.

All GUI elements are of a certain *type* (button, panel, etc.) and have particular *attributes* (size, colour, etc.). Each element has *mandatory* and *optional* attributes associated with it as well as at least one mandatory label attribute (a text string).

GUI elements are divided into *organisational* and *non-organisational* elements within a hierarchy.

The GUI hierarchy serves three main purposes:

- **Navigation** – the hierarchy allows the controller to navigate through the GUI elements in an organised way
- **Display suggestion** – the hierarchy groups certain elements together logically so that they can be displayed together physically
- **Clarity** – it allows the target and controller to easily reference particular elements

There are two organisational GUI elements – the *panel* and the *group*. The panel is used to represent, as a whole, a set of functions on the target that the user can control. Panels may not be contained within other panels (although special buttons called *panel links* can be used to navigate between panels) and the controller should render the elements within a panel together. The panel element itself is made of its attributes (for example size) and lists of the elements and groups that are contained within it.

Groups are used to bundle a smaller set of related functions that appear on the target. Groups may be contained within other groups. The group has its own attributes (for example size and position) and a list of the elements contained within it. Groups are used primarily when the controller is incapable of displaying the entire panel at once – it will then display groups and allow the user to navigate between groups as opposed to between panels.

An example of how panels and groups are related is shown in Figure 12 - a VCR panel. The panel could contain groups of functions for playback and for setting the timer to record. The user would see one panel with two groups labelled Playback and Record.

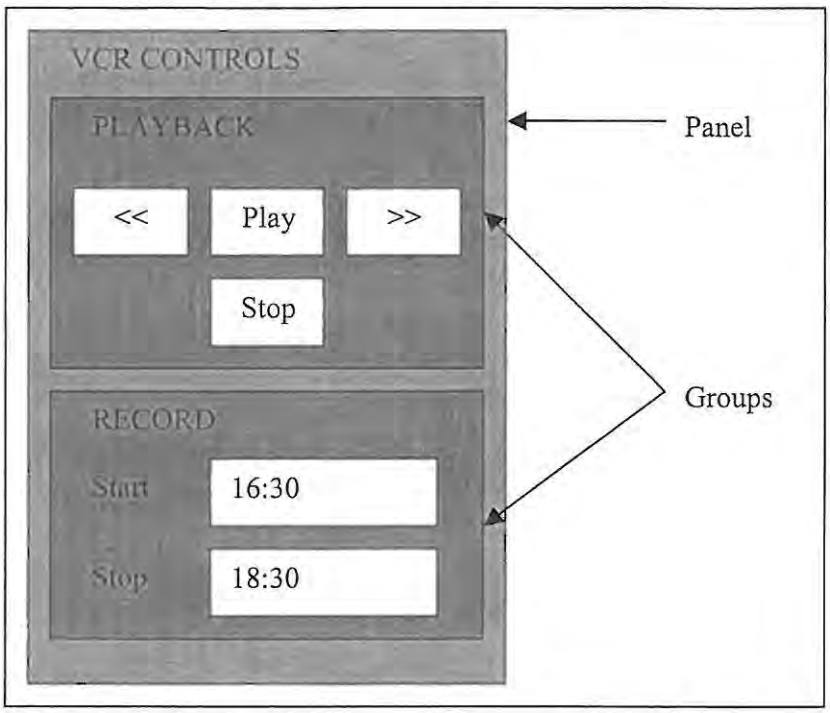

Figure 12 - A VCR Panel containing two groups of elements

Non-organisational GUI elements occupy leaf nodes in the GUI hierarchy. All non-organisational elements have mandatory and optional attributes that specify:

- **Rendering** – how the element should be rendered (including its size, position, etc.)
- **Changes** – if this element is user-modifiable, the new state of the element must be specified
- **User actions** – if this element is interactive, the description of the user action that is relayed to the target must be specified

Non-organisational elements in the AV/C Panel Subunit model are text (strings), button, icon, animation, choice, entry, show range, set range and panel link.

3.4.3.5 GUI Navigation

Navigation refers to how a user moves between collections of GUI elements (specifically non-organisational GUI elements), selects on-screen controls for manipulation and the way the controller shows the user that an element has been selected.

The controller handles navigation in a controller-dependant manner. The hierarchy of GUI elements on the target suggests navigation to the controller, but the controller may tailor navigation according to its capabilities.

The AV/C Panel Subunit specifies two modes of navigation:

- **Controller-driven** – this mode is used when a controller is unable to render a GUI element (either a panel or group and its entire contents or individual GUI elements). The controller must still use some suitable method to represent the un-rendered GUI elements to the user. The controller may add GUI elements that are not specified by the target in order to accomplish this (for example a controller with a small display could use scroll bars to allow the user to scroll to unseen elements).
- **User-driven** – used when a controller can render complete panels and elements. Panels may contain panel link elements. When a user selects a panel link element, it abandons rendering the current panel and retrieves the new panel. This allows the user to navigate between panels easily and is called user-driven navigation.

3.4.4 The UPnP Presentation Mechanism Solution to Remote Configuration

The UPnP architecture is designed around ad hoc networks that allow devices to easily connect to one another and use each others' services. Implementing remote configuration using UPnP is relatively simple if UPnP's presentation mechanism is used. This simply involves the manufacturer supplying the correct documents and some native functions in order to achieve a UPnP remote configuration system (see Chapter 5 for more details).

3.4.4.1 Setting Up a Device to Offer Services on a UPnP Network

In order for a device to offer services on a UPnP network, it must have the components shown in Figure 13. The web server implements HyperText Transfer Protocol (HTTP). The XML parser parses all the XML that is used to communicate between the server and client. There also needs to be SOAP, SSDP and GENA parsers (see § 2.4.4 for more details). The device-dependent function block is an interface to the actual functionality of the device, and all the

UPnP commands that are listed in its control document must be mapped therein. Finally, it needs to have three UPnP documents – XML device and service descriptions and an HTML presentation page.

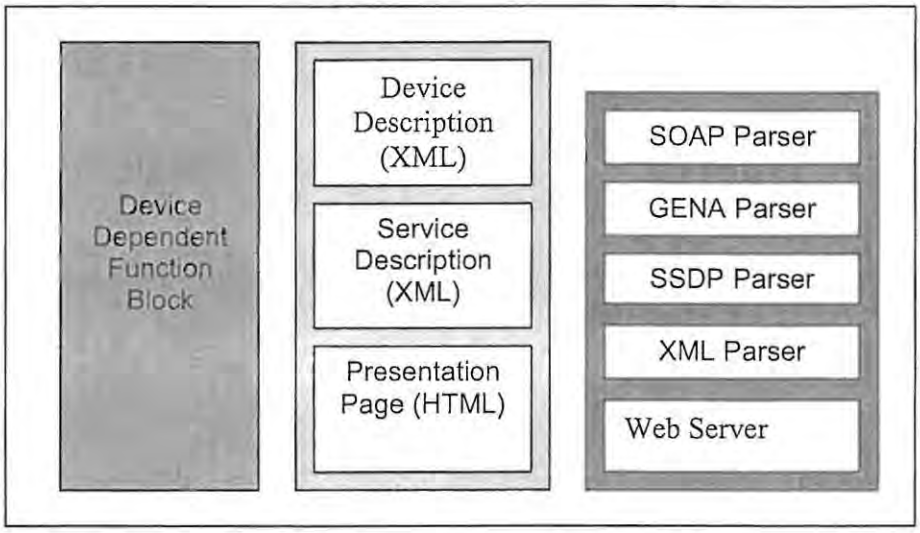

Figure 13 - Components of a Device Offering Services on a UPnP Network

Remote configuration using UPnP's presentation mechanism is then fairly simple. The process is as follows (see § 2.4.4 for more details):

- The device offering services broadcasts its presence on the network
- The control point then requests the device description XML document from the device offering services
- From the device description, the control point obtains the URL of the service description document and requests this document from the device
- The control point then subscribes to any events it wishes to (events are services) using SSDP
- The control point then requests the device's HTML presentation page from the device and displays it to the user in a standard browser
- Any user commands are then transmitted from the control point to the device offering services (using SOAP). The commands are parsed and passed to the device dependent function block which carries out the command on the device
- Any events that occur on the device are transmitted from the device to the control point (using GENA). The presentation page updates as necessary

An example of how the UPnP presentation mechanism could be used to implement remote configuration is to consider a UPnP Audio/Video Receiver (AV/R). Once the control point has

discovered the device, it requests a device description document. The device description document has URLs to the service description document and the HTTP presentation document. The control point obtains the service description document and the presentation page, which is shown in Figure 14, and the control point displays the presentation page in a browser on its display. The device has "exposed" some of its state variables (Input, Sound Mode, etc.) and shows their current values. The control point obtains the current values by sending SOAP messages to the device to query the variables. This occurs when the control point first displays the presentation page. The buttons in the right-most column allow the user to change these variables. Every time a button is pressed, a corresponding SOAP message is sent to the device. The device updates the variable being changed and sends the new current value back to the control point. The control point also subscribes to receive event notifications from the device (which uses GENA to communicate the events). If the variables change state on the device (a user may change the volume on the actual device for example), the device sends GENA messages to the control point to notify it of the variable and its new current value. The control point then updates its display to reflect the changes.

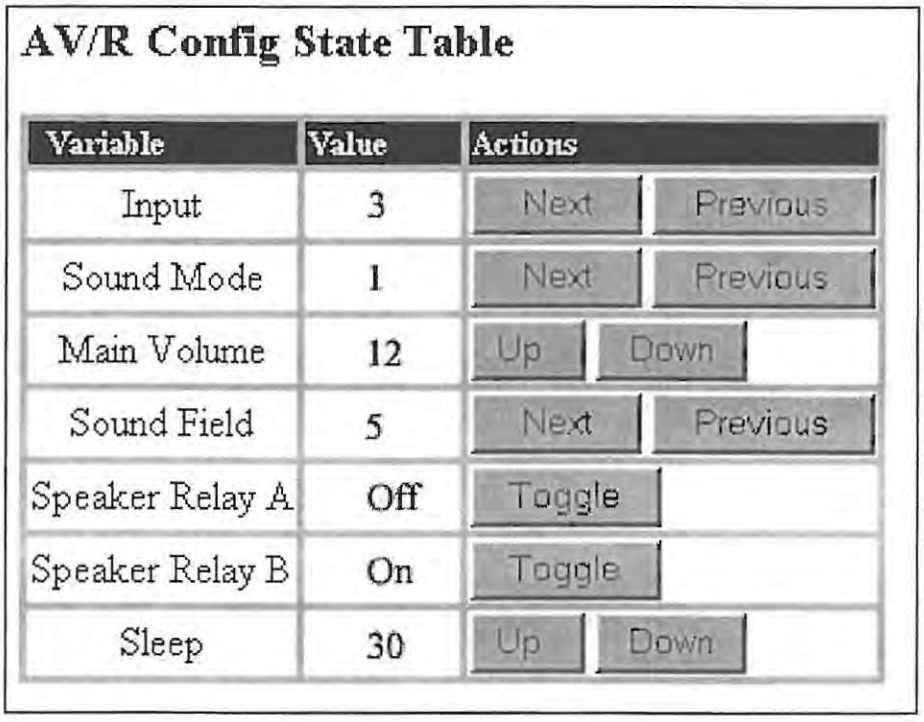

Figure 14 - The UPnP HTML presentation page for an AV/R

3.5 Comparing Remote Configuration Solutions

The AV/C Panel Subunit and HAVi's DDI model are strikingly similar. They define almost identical input and output models and define exactly the same GUI elements. Even the sequence of transactions between controller and target are the same. Because they are so similar, they are considered as one model (PS/DDI) for the purposes of further discussion.

The differences between PS/DDI, Jini's ServiceUI and UPnP are varied, but five main differences are worth noting (Table 2 summarises these differences):

- **UI Type** – this refers to the very nature of the UI – PS/DDI and UPnP are text/graphical based, while Jini's ServiceUI allows abstraction away from text and graphics to allow for any conceivable UI.
- **Controller Type** – this refers to the capabilities required by the controller for rendering the UI (bearing in mind that Jini caters for more than just visual display). Most devices (even simple LCD displays) can be used for PS/DDI and Jini UIs, while UPnP relies on an HTML browser.
- **Inter-device Relationship** – this refers to the relationship between the devices – for UPnP, a typical client-server relationship is established, while the other models exploit peer-to-peer relationships.
- **Functionality/UI Coupling** – this refers to the amount of abstraction away from the functionality of the device the UI achieves, or in other words, how dependent on the UI is the functionality of the device? For UPnP and PS/DDI, there is tight coupling between the UI and the functionality of the device, while Jini separates these two.
- **Target Specification** – this refers to the way in which the UI and functionality of the device is specified in the home entertainment networking solution. For PS/DDI, the UI is compiled into the firmware of the device (so-called *internal* target specification). For Jini, the UI is an object that is external to the communication layer (so-called *external object*). The UI and functionality of the device is specified using XML and HTML documents in UPnP and is thus said to have *external document* target specification.

3.6 Summary

This chapter has defined remote configuration as "the ability of a home entertainment networking solution to allow one device to change the configuration state of another device". It is shown that an important consideration when providing remote configuration on devices is User Interfaces. Furthermore, by looking at the Universal Information Appliance (the UIA), it has

shown some desirable User Interface features such as run-time interfaces, a compact document description language and uniformity.

Several alternatives exist for implementing remote configuration, and this chapter highlighted the solutions inherent in the four primary home entertainment networking solutions. HAVi makes use of Data Driven Interaction (DDI). Jini implements remote configuration using ServiceUI objects. AV/C makes use of the Panel Subunit. Remote configuration is a subset of the capabilities of UPnP.

The striking similarities between AV/C Panel Subunit and HAVi's DDI is noted, and the main differences between the four methods of implemented remote configuration are highlighted and summarised in Table 2.

	PS/DDI	Jini ServiceUI	UPnP
UI Type	Text/Graphical	Any	Text/Graphical
Controller Type	Any	Any	HTML Browser
Inter-device Relationship	Peer-to-peer	Peer-to-peer	Client-server
Functionality/UI Coupling	Coupled	Separate	Coupled
Target Specification	Internal	External Object	External Document

Table 2 - The main differences between Remote Configuration Solutions

Chapter 4 - The AV/C Panel Subunit Solution to Remote Configuration

4.1 Introduction

In the previous chapter, four methods of implementing remote configuration were presented – HAVi's Data Driven Interaction (DDI), Jini's ServiceUI object, AV/C's Panel Subunit and UPnP. The following two chapters discuss how two of these methods are used to implement a simple remote configuration system.

In order to better understand the inherent strengths and weaknesses of some of the potential solutions, implementations must be attempted and analyzed. For this purpose, a simple remote configuration system is implemented which allows remote configuration of an Audio/Video receiver (AV/R) via a TV, both on an IEEE 1394 network.

The AV/R is a Yamaha RXV 1000. The AV/R itself cannot communicate with the IEEE1394 network directly. The AV/R has to be *hosted* by another device capable of interfacing with the IEEE 1394 network on its behalf. The AV/R, however, does have an RS-232 serial port to allow for serial interfacing. The commands that can be sent to this serial port closely map to the infrared commands that its own remote control sends to it. The AV/R receives commands on this serial port from the device that is hosting it. The hosting device is capable of interfacing with the IEEE 1394 network and also has a serial port that enables it to communicate with the AV/R. Once the hosting device has been sufficiently tested, it will be embedded into the AV/R itself, enabling the AV/R to interface with the IEEE1394 network directly.

The first implementation (described in this chapter) makes use of the AV/C Panel Subunit specification to implement the TV-AV/R remote configuration system.

4.2 The Panel Subunit TV-AV/R

The purpose of the TV-AV/R system is to allow the user to be able to configure the AV/R from the TV without directly interacting with the AV/R itself. Hence some sort of interface to the AV/R must be provided so that the user has some way of inputting commands to the AV/R. The TV-AV/R system does this by embedding a Graphical User Interface (GUI) on the AV/R (the GUI

may be said to reside in the target, but it physically resides on the AV/R's host – more details of how the GUI is stored in the target are given later). The controller (the TV) must then retrieve this GUI and display it to the user. The user must then be able to interact with the GUI by being able to change the configuration options displayed by the TV. All the user's actions must then be relayed to the AV/R over the IEEE1394 network and the actions must be performed on the AV/R. Any events occurring on the AV/R that change its state on the GUI (such as another user changing the AV/R input physically) must then be relayed back to the TV and the GUI must be updated.

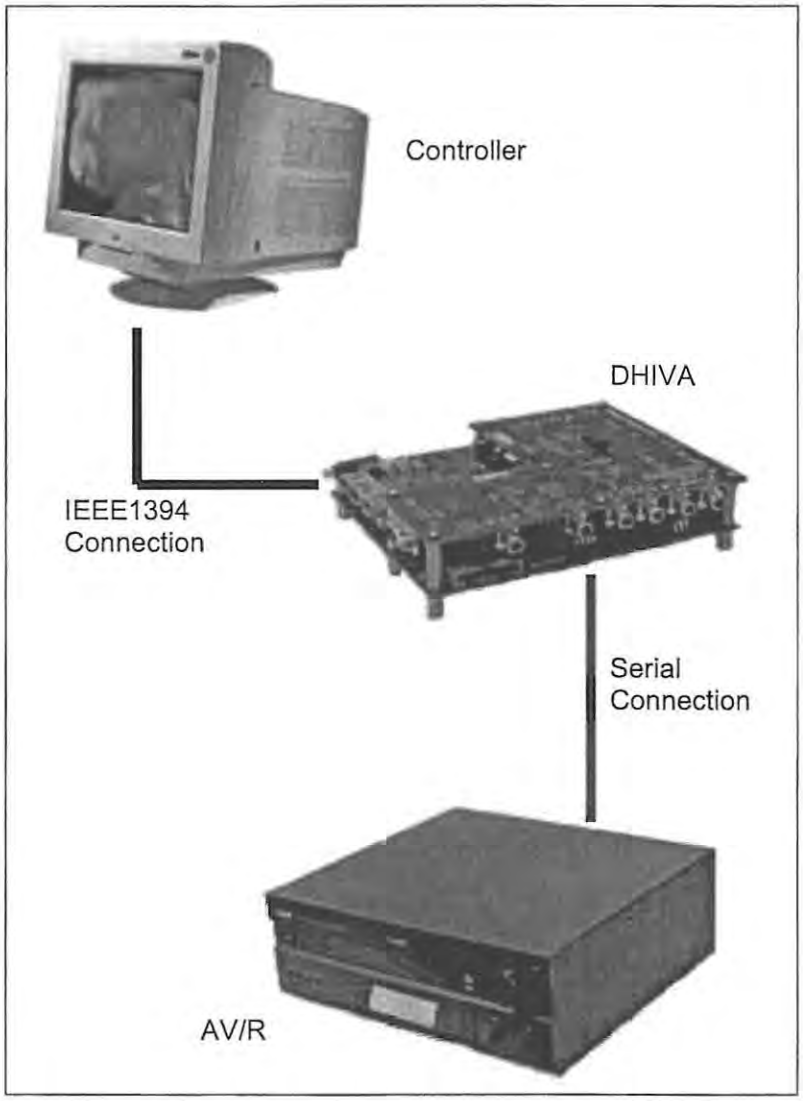

Figure 15 - Components of the AV/C Panel Subunit TV-AV/R Remote Configuration System

Some Digital TVs (DTVs) from manufacturers like Sony are capable of interfacing to IEEE 1394 networks. Since IEEE1394 is being adopted widely by manufacturers that produce TVs, it is reasonable to expect many more TVs that will have IEEE1394 capability built into them to emerge soon. For the purposes of this thesis, a "super" TV with IEEE1394 serial communication capabilities is simulated. Since the controller (the "super" TV) requires input and output capabilities to read user input and display the GUI to the user respectively, it was decided that a PC using a Windows application would simulate the TV controller for the AV/C Panel Subunit. This is because the PC (simulating the "super" TV) can readily receive user input, provide user output and is directly connected to the IEEE1394 network.

Physically, the AV/R is connected via a serial cable to a DHIVA (Digital Harmony Interface for Video and Audio [42]). This is an embedded computer that has a 1394 protocol stack. The DHIVA acts as host to the AV/R and is connected to the IEEE 1394 network. The controller device (the PC) is also connected to the IEEE 1394 network. The components of the system are shown in Figure 15. The hosting device (the DHIVA) stores the graphical elements and their spatial positions and relationships as well as the serial commands necessary to carry out the user's actions.

When designing the AV/C Panel Subunit implementation for the TV-AV/R system, two main goals emerged. Firstly, a working implementation had to be obtained. Secondly, a set of tools had to be developed to aid manufacturers in developing further remote configuration applications similar to this one (i.e. remote configuration systems for devices other than the TV and AV/R). Bearing these goals in mind, the work done in order to realize the remote configuration TV-AV/R system was five-fold. Each step was essential for the implementation, and a tool was created for each step. Future systems need only use the tools created in order to quickly and easily create a fully functioning remote configuration system. The five steps are:

- A Graphical User Interface (GUI) must be created (§ 4.4)
- The GUI must be represented in some hardware-independent manner (§ 4.3)
- The GUI must be stored natively on the target device (§ 4.5)
- The target device must be able to supply the GUI to the controller and respond to user actions received from the controller (§ 4.6)
- The controller must be able to request and display the GUI and send user actions to the target (§ 4.7)

The tools used are detailed in each section as listed above. They are:

- The GuiBuilder

- The GUI XML Document Type Definition (DTD)
- The GUI XML Parser
- The Panel Subunit
- The ControllerApp

4.3 The GUI XML Grammar

Specifying GUIs can be a very complex problem. Manufacturers are faced with a trade-off between *abstraction* of the GUI and *specificity* of the GUI. The abstraction of the GUI is the manner in which the GUI is represented – the "language" used to describe the GUI. The specificity of the GUI refers to how the GUI is implemented – how the "language" is used to display an actual GUI.

An important factor that comes into consideration at this point is the *equivalence* of GUIs. Two GUIs may appear different in terms of the colours, sizes and shapes that they display to the user. However, these factors form part of the "look-and-feel" of a GUI and are not considered when talking about equivalence of GUIs, since it is reasonable to expect the look-and-feel of GUIs to differ on different implementations and on different platforms. What is considered is the *functionality* of the GUI. For example, if on one GUI a user can only select one of several alternatives, and on another GUI the user can select several of these alternatives, the GUIs are not equivalent.

The more abstract the GUI representation is, the less hardware independent the GUI becomes. There is also a higher probability that different implementations reading the GUI "language" will display different GUIs - a situation that is undesirable since the manufacturer wishes consistency of the GUI across different platforms and implementations. The less abstract the GUI representation, the more consistency is gained at the cost of a complex and bloated GUI "language", and less hardware independence. The ideal is to have a representation that is abstract enough to be hardware-independent, while at the same time being specific enough that equivalent GUIs will be displayed even when the "language" is read by different implementations or platforms.

The AV/C Panel Subunit specification goes some of the way towards this ideal by specifying what GUI elements are available and what properties they have. However, the specification only spells out how this information is communicated and not how it is stored on the device.

A method of specifying how the GUI is laid out and how it is stored in the target was created by making use of an extended Markup Language (XML) [16] grammar. XML allows rigorous representation of data without hardware dependence. This suits the representational problem well, since the layout of the GUI must be well organized and hardware independent.

4.3.1 Selecting the GUI Elements Required

The AV/C Panel Subunit model is designed for remote control (as well as remote configuration) of any device. For the purpose of the TV-AV/R system, however, not all the GUI elements available are necessary. Hence only five elements are selected, and with these elements, virtually all configuration operations can be achieved:

- Panels (these are organizational and serve as containers for the other elements)
- Labels (these are static text fields)
- Links (these are buttons that link panels, i.e. clicking a link brings up the panel this link points to)
- Sliders (these allow a user to select a value within a well-defined range). This element is called a "Set Range" element in the AV/C Panel Subunit specification
- Scroller (these allow a user to scroll through a set of several alternatives, seeing only one at a time). This element is called a "Choice" element in the AV/C Panel Subunit specification

The AV/C Panel Subunit specification defines a number of GUI elements, each having it's own properties. Not all of these properties are required to sufficiently specify the elements required for the TV-AV/R system. Table 3 shows the elements and the attributes selected for them.

	Mandatory Attributes	Optional attributes
Panel	width, height, caption, aspect ratio,	Position
Label	width, height, caption	Position
Link	width, height, caption, image, interactive[2], panel-to-link-to	Position
Slider	width, height, caption, interactive[3], range type, default value	Position, min caption, max caption
Scroller	width, height, caption, interactive[3], choice type, chosen elements	Position

Table 3 - The properties associated with GUI elements

4.3.2 The XML Document Type Definition

The next step is to map these elements and their properties to XML notation. Formal methods exist for checking the "correctness" of XML files. One of these methods is Document Type Definitions, or DTDs. The DTD verifies that all the data in the XML files have correct tags, that variables are of the correct type and that all required data are present. The DTD file is listed in Appendix A and shows the XML GUI grammar created.

In order to create XML tags for the GUI language, a hierarchical structure is used, since this fits well with the nature of XML. Tags in XML can easily be nested, and this allows hierarchies to be created simply by nesting tags appropriately. At the highest level of the hierarchy is the PanelSubunit. The PanelSubunit consists of one or more Panels (i.e. the Panel tags are nested within the PanelSubunit tags). In the DTD, this relationship is shown as follows:

```
<!ELEMENT PanelSubunit (Panel+)>
```

The !ELEMENT precedes the name of the tag – in this case, PanelSubunit. Then (Panel+) is used to show that one or more Panel tags are contained within the start and end tags for the PanelSubunit (<PanelSubunit> and </PanelSubunit> respectively).

Next, a list of the attributes of the PanelSubunit tag are specified:

```
<!ATTLIST PanelSubunit
      name   CDATA #REQUIRED>
```

In this case, there is only one attribute, name. Name is specified as mandatory (#REQUIRED) and is character data (CDATA). In the XML, the PanelSubunit tag must look as follows:

```
<PanelSubunit name = "MyPSU">
```

At least one Panel must follow this line of XML before the closing tag, </PanelSubunit> can be used.

More complex relationships between tags are possible. For example, a Panel consists of one or more elements - Sliders, Scrollers, Links or Labels. This is specified as follows:

```
<!ELEMENT Panel (Scroller | Slider | Link | Label)+>
```

Another relationship exists in the Slider tag. The slider consists of exactly three tags - one SliderMin, one SliderMax and one SliderStep. This is specified thus:

```
<!ELEMENT Slider (SliderMin, SliderMax, SliderStep)>
```

Tags may also be empty (i.e. be able to contain no other tags). For example, the Link element cannot contain any other tags and is thus empty:

```
<!ELEMENT Link EMPTY>
```

The XML tags are relatively easy to decide on. At the top level is the PanelSubunit that consists of one or more Panels. The Panels each contain at least one, but any combination of the other elements. The PanelSubunit needs only one attribute – its name. The panel needs two attributes – its name and its caption. The name is used "internally" to enumerate the panels, while the caption is displayed to the user.

The Slider, Scroller, Link and Label all have a caption and a position attribute. It was decided that each panel should be divided into eight horizontal "slots" that elements could go into, one under the other. The position then simply defines which "slot" the element goes into, and ranges from one to eight.

The label needs no other attributes; while the Link needs an attribute (called "linkto") that holds the name of the panel this Link refers (or links) to.

The attributes for the Scroller and Slider are more complex. The Scroller is a container for several "choices" that are displayed to the user. Each choice has a caption, which is displayed to the user, and a corresponding host action (i.e. the action to be performed in order to select this "choice" on the actual AV/R). Since there may be many choices, it was decided to make each choice (with its attributes of caption and host action) a separate tag. These tags are then nested within the Scroller tag – one tag for each choice.

The Slider, over and above its caption and position, needs a maximum value, a minimum value, and a step value as attributes. It was decided that the display value need not necessarily be the value the AV/R uses. This allows the manufacturer flexibility in what is displayed to the user – some users may not mind working in decibels, having their volume range from –100dB to 0dB, while other users may prefer their volume ranging from 0 to 100. Hence, for each value (min, max and step) a separate tag is made to hold the display value as well as the host value and host action (both of which are used by the AV/R to carry out the action).

4.3.3 An Example – The ImagiRadio

An example is useful to illustrate the usefulness of using XML to describe a GUI. Imagine that a manufacturer installs remote configuration capabilities onto a device (called an *ImagiRadio*) and wishes the GUI to have two panels that allow for configuring the type of input and some audio features. The manufacturer sketches a picture of what the GUI should look like as shown in Figure 16.

The manufacturer describes the root panel (or first panel in the hierarchy as a panel with two panel link buttons that link to a further two panels called "Inputs" and "Audio" respectively. The Inputs panel has a scroller called "Signal Select" and a panel link back to the root panel. The Audio panel has a scroller called "Sound Mode", a slider called "Digital NR" (for Digital Noise Reduction) and a panel link back to the main panel. The slider ranges from 10 to 20. The scrollers all have various options, of which the user is allowed to select only one – the currently visible one.

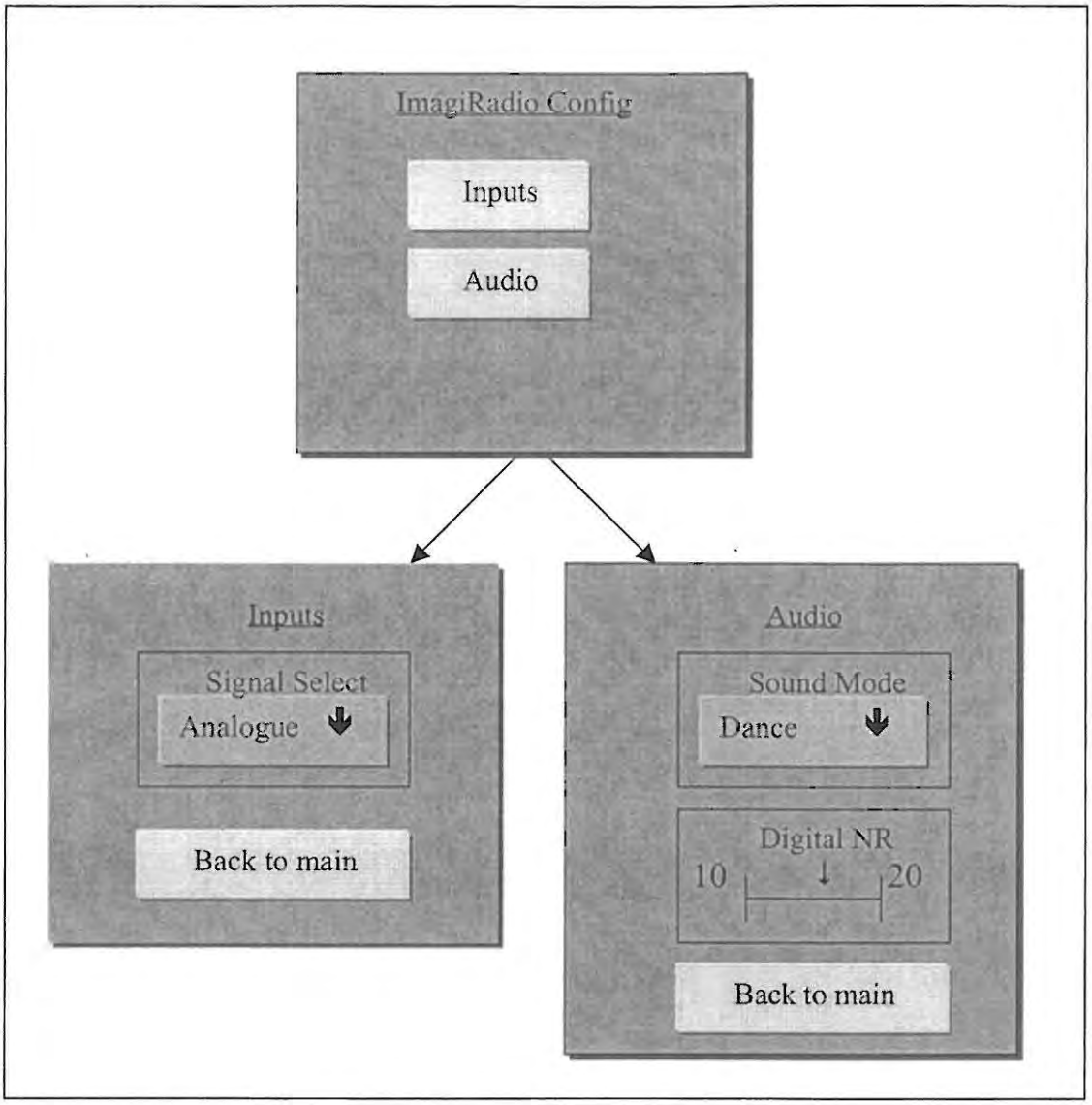

Figure 16 - The GUI layout as specified by the ImagiRadio manufacturer

Since the manufacturer's description of the GUI only lays out which elements go where and what attributes each element has, the GUI can be represented fairly abstractly (that is, without specifying look-and-feel).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE PanelSubunit SYSTEM "describeguiformal.dtd" >
3
4  <PanelSubunit name="ImagiRadio.xml" >
5   <Panel name="panel0" caption="Radio Setup" >
6      <Link caption="Inputs" position="2" linkto="panel1" />
7      <Link caption="Audio" position="3" linkto="panel2" />
8   </Panel>
9   <Panel name="panel1" caption="Inputs" >
10     <Scroller caption="Signal Select" position="2" >
11        <ScrollVal caption="Analogue" hostaction="SS:01" />
12        <ScrollVal caption="Digital" hostaction="SS:02" 13 />
13        <ScrollVal caption="DTS" hostaction="SS:03" />
14     </Scroller>
15     <Link caption="Back to Main" position="3" linkto="panel0" />
16  </Panel>
17  <Panel name="panel2" caption="Audio" >
18    <Scroller caption="Sound Mode" position="2" >
19        <ScrollVal caption="Hall 1" hostaction="SM:01" />
20        <ScrollVal caption="Hall 2" hostaction="SM:02" />
21        <ScrollVal caption="Jazz" hostaction="SM:03" />
22        <ScrollVal caption="Dance" hostaction="SM:04" />
23        <ScrollVal caption="Theatre" hostaction="SM:05" />
24    </Scroller>
25    <Slider caption="Digital NR" position="3" >
26       <SliderMin value="10" hostaction="DNR:" hostvalue="10" />
27       <SliderMax value="20" hostaction="DNR:" hostvalue="20" />
28       <SliderStep value="1" hostvalue="1" />
29    </Slider>
30    <Link caption="Back to Main" position="4" linkto="panel0" />
31  </Panel>
32 </PanelSubunit>
```

Listing 1 - An example of a GUI described in XML

Listing 1 shows the XML required to represent the GUI for the ImagiRadio. Line 1 of the document specifies that it is an XML version 1 document using UTF-8 encoding. Line 2 is a reference to the DTD of this document to allow any parser to verify the correctness of the XML. Lines 5 – 8, 9 – 16 and 17 – 31 are the three panels with the various elements they contain nested within the <Panel> and </Panel> tags. The Inputs scroller on panel one is seen in lines

10 – 14. Lines 11, 12 and 13 show how the various options available to the user are specified and how they are mapped to *host actions* – these are the serial codes transmitted to the AV/R when its corresponding option has been selected. The Slider (lines 25 – 29) must contain a SliderMin, a SliderMax and a Sliderstep element – these are used to specify both what appears on screen to the user and what action is taken when the slider value is changed. The link elements (lines 6, 7, 15 and 30) are all *empty* tags since they do not need to contain any elements themselves.

Notice that no information is given regarding what the elements look like – this achieves hardware and system independence. However, the structure of the GUI (which elements go in which panels and in which order they appear) is specified. This allows a GUI that is flexible in its look-and-feel without losing the layout that the manufacturer originally intended.

This XML grammar as defined by its DTD is itself the tool for this step. Future systems do not have to specify the DTD again; they simply use the DTD to specify the layout of the new GUI by using the tags and replacing the values appropriately. The XML grammar is comparable with the MoDAL system that Eustice et al. use to describe their UIA interfaces (see § 3.3.1). However, since MoDAL is used for controlling devices as well as configuring them, it is a more detailed and extensive grammar. The GUI XML grammar created for remote configuration is simpler and smaller. However, the aim of using the XML grammar is the same: to specify an interface to a service without having to specify the functionality of the service – i.e. separating interface from functionality.

4.4 The GuiBuilder

Since not all manufacturers are familiar with XML, and specifying GUIs in XML can be a rather arduous procedure, there exists the need to automate the process of specifying the GUI. To achieve this, the GuiBuilder application is created. The GuiBuilder is the tool for this step.

The GuiBuilder application is a WSIWYG environment that allows the manufacturer to lay out and specify a GUI using a graphical interface. Once the GUI has been laid out, the GuiBuilder generates the XML file corresponding to the layout of the GUI. This gives the manufacturer several benefits:

- Speed – since the layout is done graphically rather than by hand in XML
- Ease – since the manufacturer need not learn XML

- Flexibility – GUI layouts can be saved and edited, allowing the manufacturer to easily improve existing GUIs
- Efficiency – since the XML is generated automatically, the manufacturer is guaranteed of an XML file that is correct (in the grammatical sense)

The GuiBuilder is written in Visual Basic in a Windows environment since this environment lends itself to rapid development of graphical applications.

4.4.1 GuiBuilder Design

The design of the system is *object oriented.* Object orientation is a widely adopted approach for the construction of particular systems. Classes defined for one system can be re-used in other systems. Similarly, improvements to existing systems can be made by modifying certain classes without having to change the rest of the system. An important reason for using object orientation for remote configuration systems is the fact that the object orientation lends itself to separating core components of the system – in this case, the functionality can be separated from the user interface since classes will exist for both components.

The designs of object oriented systems are usually modeled in a particular modeling language, and for this thesis, UML (the Unified Modeling Language) [17] is used. UML is used to show the attributes and functions that each class possesses as well as how the classes in a system interact with each other.

Peter Coad et. al. [37] views any object oriented system as having four types of component. The problem domain component contains objects that relate directly to the problem being modeled. The human interaction component contains objects that provide an interface between the problem domain component and humans. The system interaction component provides interfaces from the problem domain to other systems, while the data management component provides interfaces between the problem domain and databases or file management systems.

These are used to categorize the objects of the remote configuration system – specifically into a problem domain component and a human interaction component, the functionality of the system is separated from the user interfaces of the system. This is important since the user interface for the AV/R may appear on several different controllers (TVs) and the separation aids in implementing a hardware independent system.

The design of the GuiBuilder (and the ControllerApp – see § 4.7) conforms to the following sequence:

- Use case diagram (to show the broad functionality of the system)
- Scenarios (to set out clearly what events occur in the system)
- Sequence diagrams (to show interactions between classes)
- Object Model (to show the classes with their attributes and member functions)

Figure 17 shows the GuiBuilder Use Case diagram. The only actor in the system is the manufacturer (who is an actor since the manufacturer is external to the system), and only two use cases exist – DefineGUI and ExportXML. These use cases describe the broad functionality of the system. Each use case may contain several paths of operation.

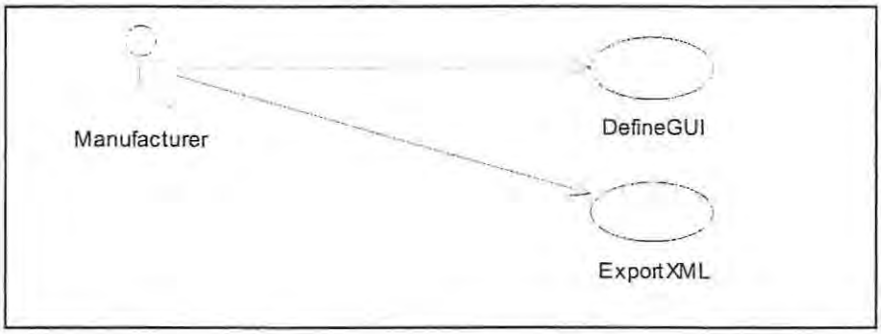

Figure 17 - The Use Case diagram for the GuiBuilder

Figure 18 shows a screen shot of what the GuiBuilder looks like as it opens. The GUI elements fill slots on the panels, and the entire "collection" of panels is exported to XML as the *PanelSubunit*. Figure 19 shows a panel with the name "Audio Setup" that contains four elements. The first element slot contains a label with the caption "(Advanced)". The following three element slots contain a Scroller with the caption "Surround", a Slider with the caption "Bass Gain" and a link button with the caption "Back to Main". Figure 20 shows what appears in the properties space when the manufacturer clicks on a Slider element. The Name box is the name that will appear to the user when the GUI is displayed on the TV. The min, max and step values (on the left, labeled "Display") are the slider values that will appear to the user. The remaining values (on the right, labeled "Host") and the "Host Action String" are used by the DHIVA to send serial commands to the AV/R to perform the user's actions.

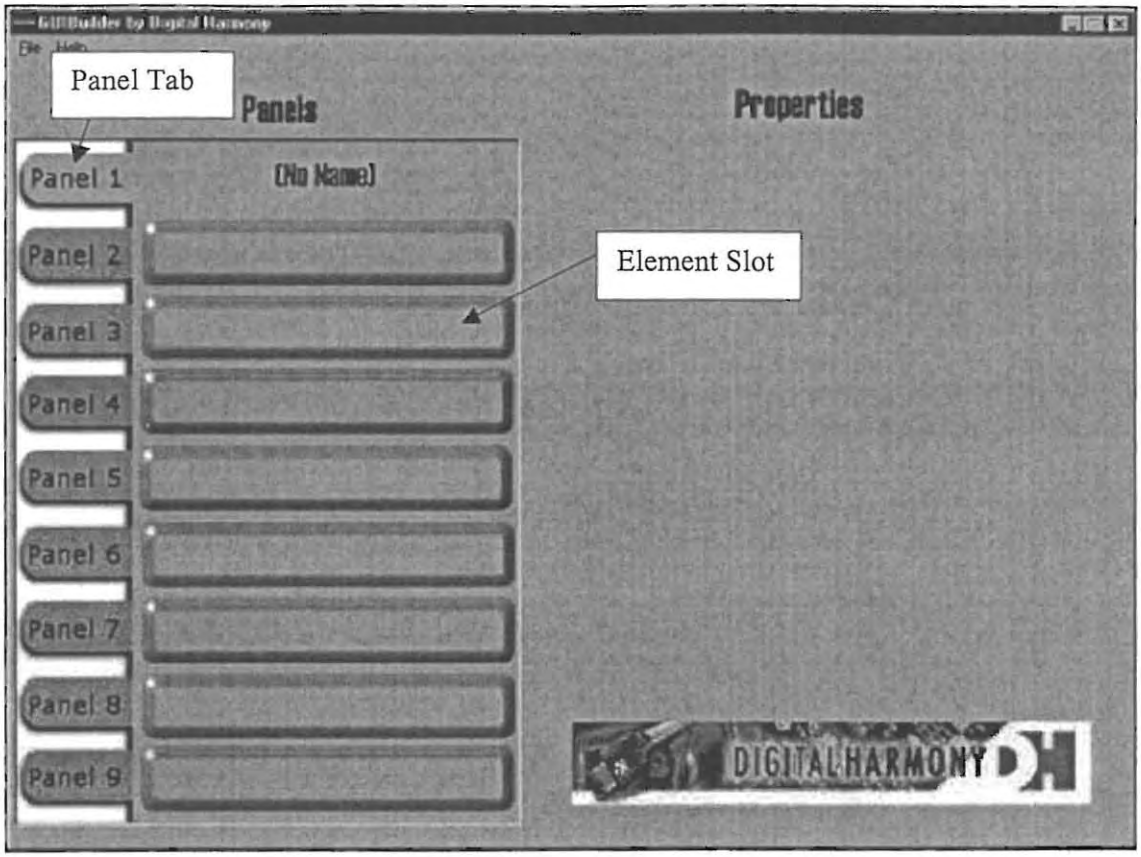

Figure 18 - A screen shot of the GuiBuilder application

Figure 19 - An example of a panel containing four elements

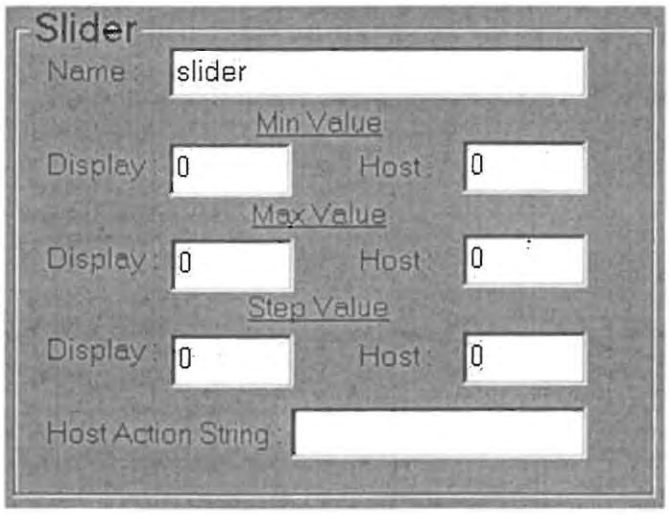

Figure 20 - The properties that are shown for Slider elements

4.4.2 GuiBuilder Scenarios

Several scenarios exist for the GuiBuilder system.

- Scenario 1: Add New Element. The manufacturer clicks an empty element slot and creates a GUI element (a label, a link, a slider or a scroller) for that slot.
- Scenario 2: Display Next Panel. The manufacturer presses a "panel tab". The corresponding panel with all its current elements is displayed to the manufacturer.
- Scenario 3: Enter Properties. The manufacturer clicks on an element. The properties for that element are displayed. The manufacturer modifies the properties of the element and the element is updated.
- Scenario 4: Export to XML. The manufacturer selects "Export to XML" button. After obtaining the filename for the XML file from the manufacturer, the program outputs the XML corresponding to the GUI the manufacturer has just specified.

4.4.3 GuiBuilder Sequence Diagrams

The sequence diagrams correspond to the scenarios and look as follows:

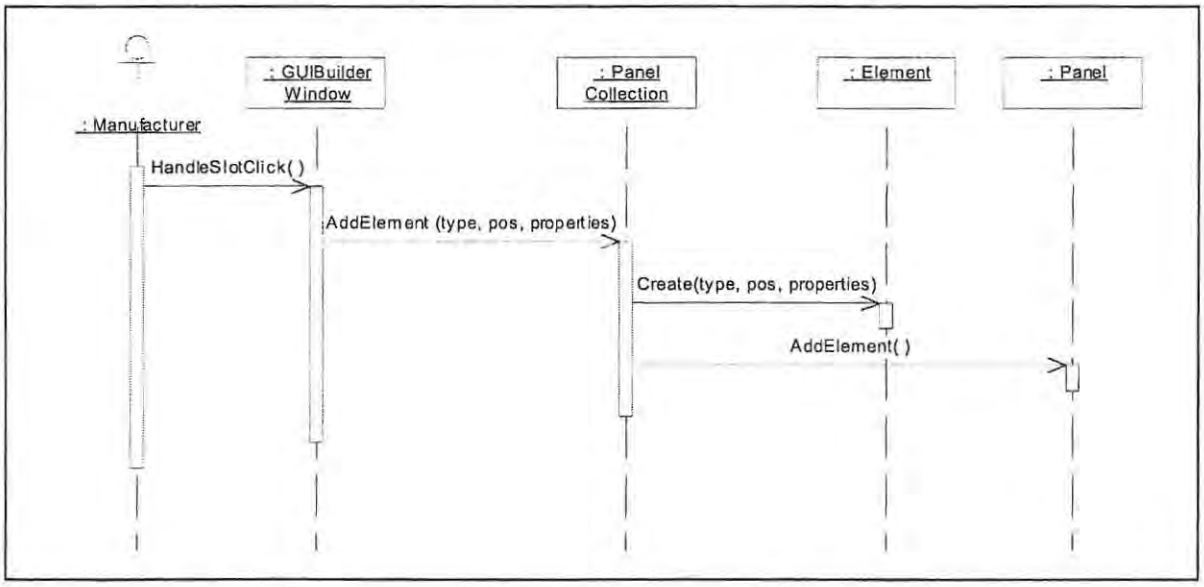

Figure 21 – GuiBuilder Sequence diagram for Scenario 1 - The manufacturer clicks on an empty element slot

All user interactions are done via the GuiBuilderWindow object. This class is the only class that is in the Human Interaction Component.

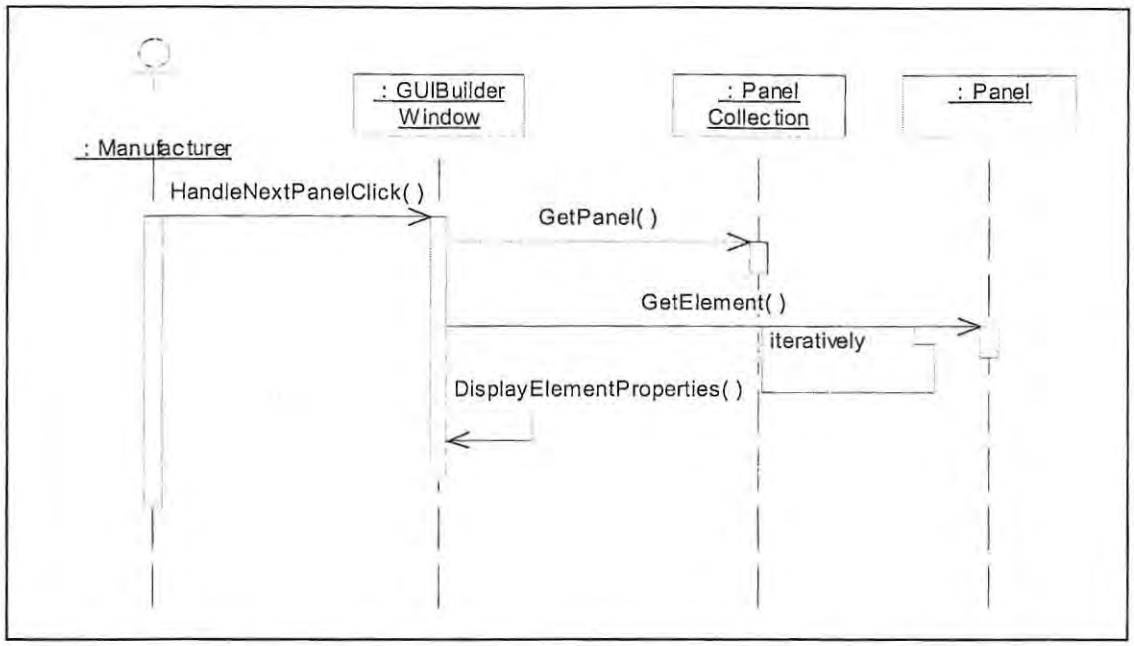

Figure 22 – GuiBuilder Sequence diagram for Scenario 2 - The manufacturer clicks on a panel tab

In the above sequence diagram, the GetElement function is called iteratively for all the elements contained within the new panel. Each time an element is retrieved, its properties are displayed.

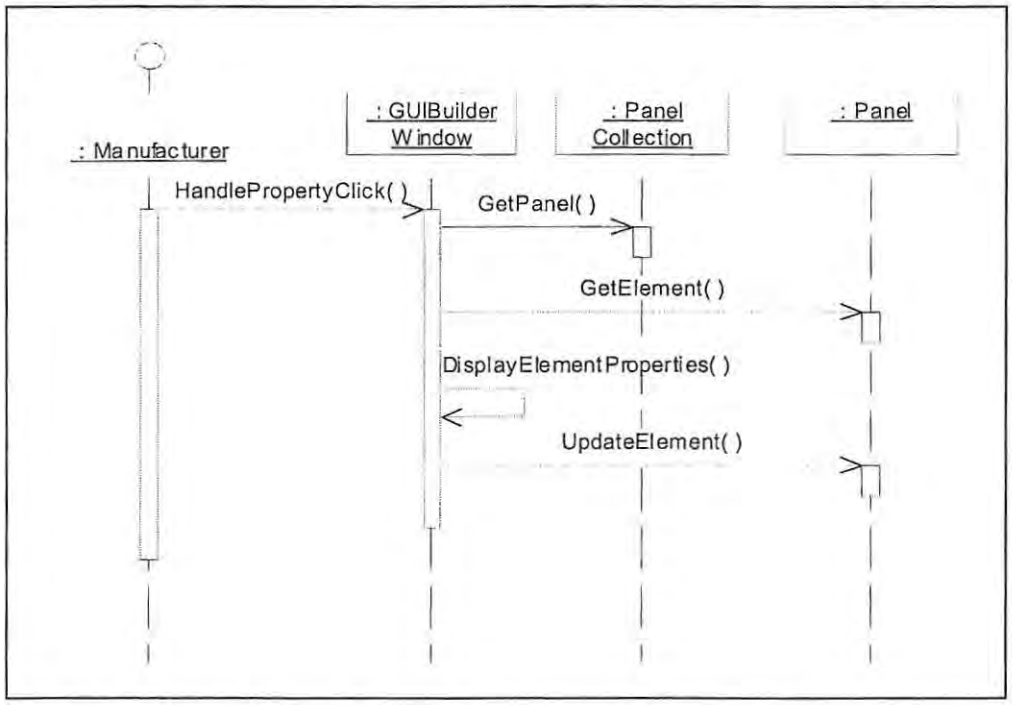

Figure 23 – GuiBuilder Sequence diagram for Scenario 3 - the manufacturer changes the properties of an element

In the sequence diagram above, the properties of the element are first displayed to the manufacturer. The manufacturer is then free to change any properties, and once the changes are complete, the element is updated to reflect the changes.

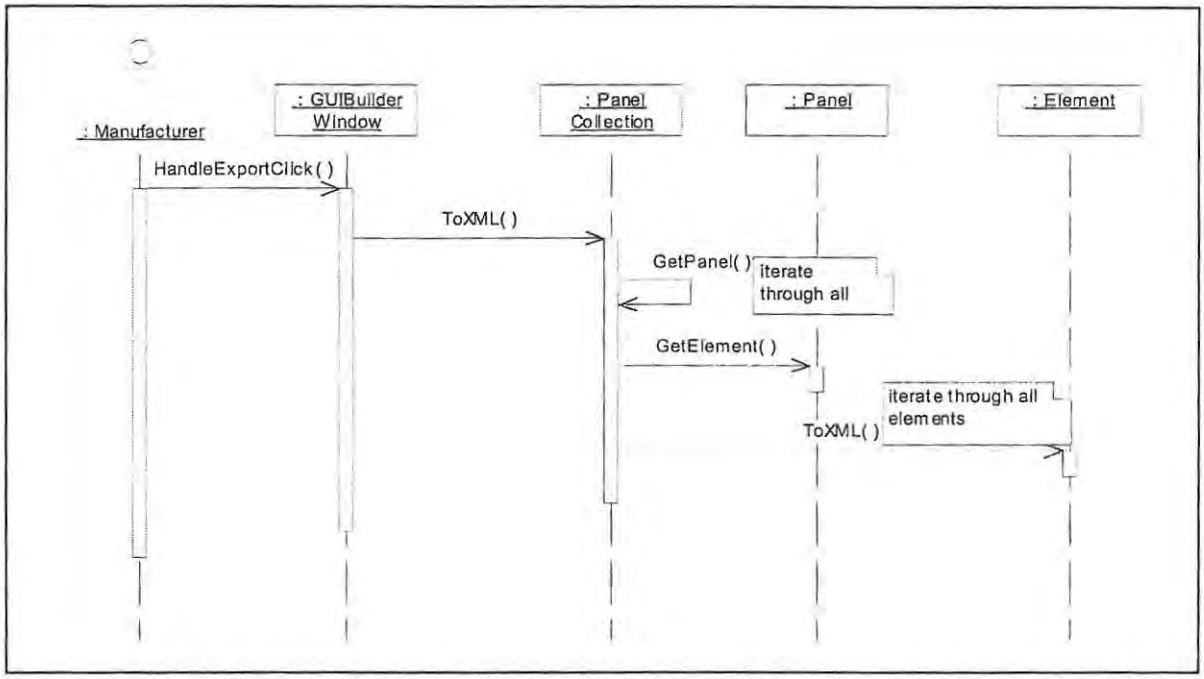

Figure 24 – GuiBuilder Sequence diagram for Scenario 4 - the manufacturer exports the GUI to XML

In the above sequence diagram, all the panels are iterated through, and each panel iterates through all of its elements, outputting XML for each one.

The completed object model for the GuiBuilder appears as follows:

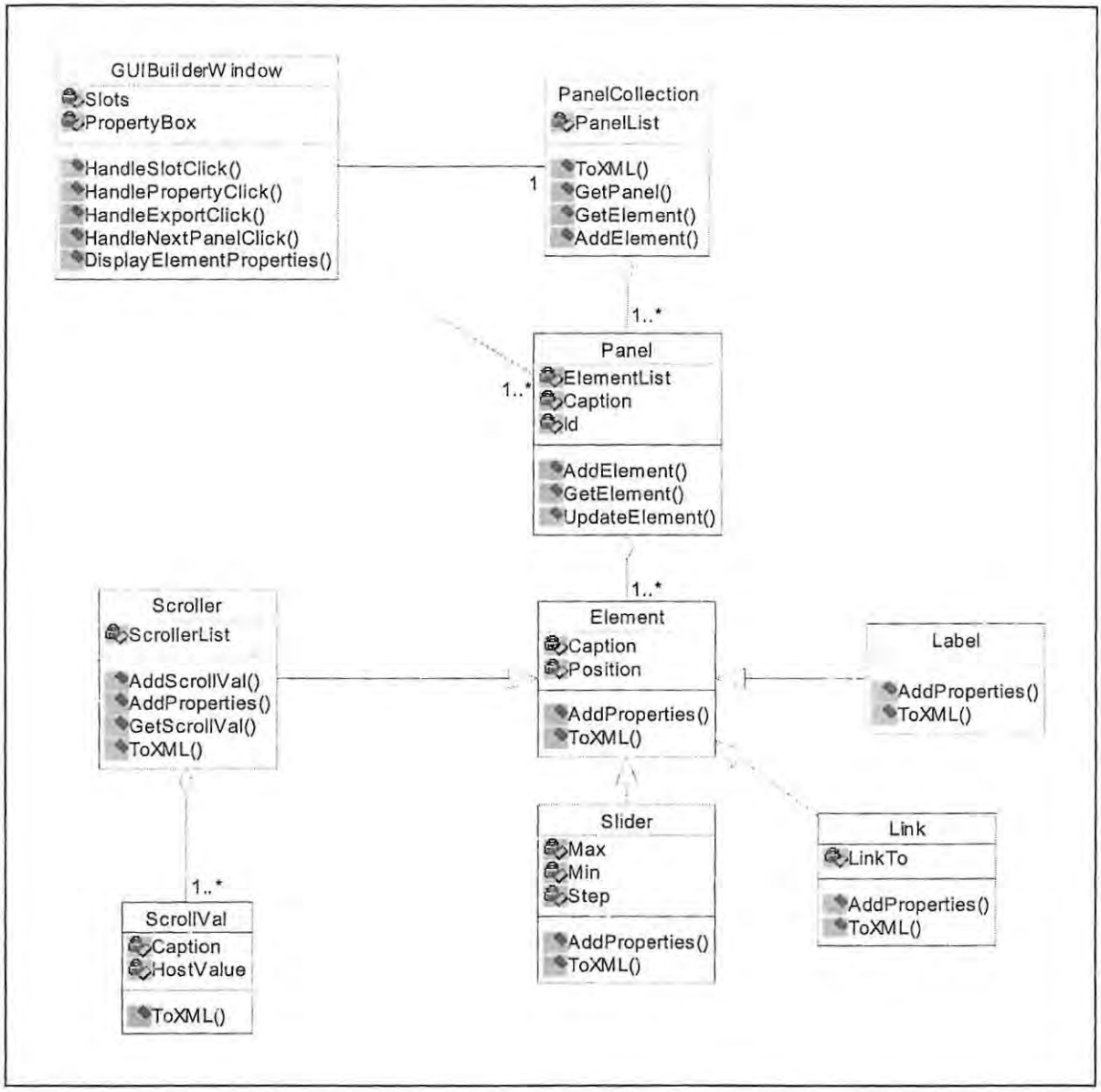

Figure 25 - The GuiBuilder object model

In the object model, lines represent associations; arrows denote inheritance and a line with a diamond head shows an aggregation relationship. So in the above diagram, the Scroller contains one to many ScollVals; the Scroller, Slider, Link and Label classes are specialisations of the Element class (they inherit from the Element class) and the GuiBuilderWindow class is associated with the Panel and PanelCollection classes (the association allows communication between the classes and allows the classes to make use of public attributes and member functions within associated classes).

4.5 The XML-GUI Parser

Now that the manufacturer has specified the layout of the GUI in XML, the XML must be stored in the target device where it will later be retrieved by the controller. In order to increase the speed and efficiency of the target device, a parser was built that parses the XML and converts it to native structures that can be stored on the target device. This prevents the target having to implement an XML parser. It also makes the GUI data immediately available to the target, instead of having to be parsed first.

The parsing could be done "by hand" – but this would be a very time-consuming activity and it is far better to automate this process too. However, parsers are not easy to build, and building a parser from scratch can be a complex problem. Hence a parser generator, Coco/R [18], was used to build a parser.

Coco/R requires a *grammar* (a formal description of the form of a language) to be created. Coco/R uses this grammar to generate a parser that can then be used to parse a file and confirm *syntactic* correctness. Later, *attributes* are added to the grammar (making it an *attributed grammar*) and these attributes are actions that perform *semantic* functions. Coco/R is usually used in compiler creation, and in a sense the XML-GUI parser "compiles" the GUI XML. However, instead of producing object code, it creates C structures that correspond to the GUI layout described by the XML. The attributed grammar is shown in Appendix C. C structures are chosen as the native storage medium since the target stack is written in C (see § 4.6).

4.5.1 Grammars

4.5.1.1 Checking Semantics

A grammar is a set of rules that define how certain words and symbols in a language (in this case, the language is XML) may be put together legally. The grammar then allows a parser to be able to parse a string (or sequence of symbols) to determine if it is an allowable string. Malformed sequences generate errors.

These grammar rules are specified in Extended Backus-Naur-Form (or EBNF) [38]. The EBNF rules have the form:

leftside = rightside

The leftside expressions are "conceptual" (they are called *non-terminals*) – they describe abstractly what their corresponding rightside expressions define. For example, consider the following rule:

```
Panel = "<Panel>" Element { Element } "</Panel>".
```

The rule defines what a panel looks like (panel is here an "abstract" term). Each panel begins with the string "<Panel>" and ends with the string "</Panel>" (these strings appear in the actual XML file and are called *terminals*). Between these two strings, an Element must exist, followed by an arbitrary number of other elements (this is shown by the braces). Element is again an "abstract" term, which has its own rule stating what it looks like in the form of strings that appear in the XML file to be parsed.

The *symbols* for the GUI-XML include all the tags that have been defined in the Document Type Definition (DTD). Variable strings are enclosed in quotation marks. The grammar checks for syntactic correctness by reading the XML file that is being parsed one character at a time. The grammar is structured in such a way that reading only one character at a time, the parser is able to determine whether or not that character is legal. This feature is called LL(1) formally. LL(1) takes its name from the method of parsing employed – the input XML file is read from **L**eft to right and substitutes from the **L**eft-most "abstract" term in the rightside of the rule, looking ahead only one character at a time to determine if the string being parsed is legal.

4.5.1.2 Performing Syntactic Actions

Coco/R, using the grammar created, constructs the parser. Coco/R defines a function for each production. This function is designed so that it exactly mirrors the production – and the function checks that the correct terminals appear in a string – when they do not, the function reports that it has a malformed string. The first (highest) function corresponds to the Start sentence (goal) of the grammar, and every time a production is required to perform a substitution, a function lower down (corresponding to the production being used) is called. These functions are usually recursive in nature, and so the parsing is said to be by *recursive descent*.

For example, consider the following production:

```
Label = "<Label" Caption Position "/>".
```

The function that would "service" this production looks like this (in pseudo-code):

```
function Label
{
```

```
        sym = ReadSymbol(); // reads the next string
        if (sym != "<Label") ReportError();
        call Caption(); // parse the next expected string
        call Position();// parse the next expected string
        sym = ReadSymbol(); // reads the next string
        if (sym != "/>") ReportError();
}
```

The attributes, once added to this production, perform specific actions and the production looks as follows when it is attributed:

```
Label<TLabel* &L> = (. char c[20], p[20]; .)
  "<Label" Caption<c> Position<p>
  (. int pos = atoi(p);
     L = CreateLabel (c, pos);  .) "/>".
```

The semantic actions that must be performed are enclosed by "(." and ".)" symbols. The angle brackets are used to denote actual or formal parameters. The formal parameter is a type declaration and is used to specify the arguments that must be passed to a function call. The actual arguments are the variables that are passed. So in the above example, L is a formal parameter (passed by reference to the "Label" function, while c, p and pos are all local variables declared inside the Label function.

The function produced by Coco/R for the attributed grammar production Label, appears like this:

```
function Label(Tlabel* &L)  // L passed by reference
{
        char c[20], p[20];
        sym = ReadSymbol(); // reads the next string
        if (sym != "<Label") ReportError();
        call Caption(c); // parse the next expected string
        call Position(p); // parse the next expected string
        int pos = atoi(p);
        L = CreateLabel (c, pos);
        sym = ReadSymbol(); // reads the next string
        if (sym != "/>") ReportError();
}
```

Notice how the position of the attributes (contained within "(." and ".)" symbols) in the grammar specify where they appear in the corresponding function. In this manner, any semantic actions that must be performed are simply placed as they would appear in real code in the grammar.

4.5.2 The XML-GUI Grammar

The productions of the XML-GUI grammar map simply to the XML tags defined. Since the formatting of the tags is strictly defined by the DTD, the parser is able to determine which production is being parsed by reading ahead only one character. This also allows the entire XML file to be compiled (or parsed) by reading it from top to bottom once – formally, the grammar is said to be LL(1) (see § 4.5.1.1). This inherent feature of the grammar allows the parsing to be a very fast and efficient procedure.

The grammar for the parser is closely related to the DTD since both can check the XML for syntactic correctness. The grammar, however, once attributed, produces code by Coco/R that not only checks syntactic correctness, but actually converts the XML to native C structures. It does this by creating a "blank" structure for each element and then filling in the properties directly from the XML. The following code shows what the C structure for a label looks like:

```
typedef struct TLabel {
      DH_PANEL_ELEMENT_ID id;
      char caption [STRLEN];
      int pos;
} TLabel;
```

The Label consists of a DH_PANEL_ELEMENT_ID (which is a unique 4-byte number that identifies this label), a caption of chars and an integer value representing its position in the panel (this integer ranges from 1 to 8). The following function shows how memory is allocated for the label and how its caption and position are "filled in":

```
TLabel* CreateLabel (char* c, int p) {
      TLabel* L;
      DH_FWSTATUS result = mpmMemBlockAllocate (sizeof (TLabel), &L);
      L->id.type = 0x17;  // panel subunit spec 1.0
      L->id.id = idNum++;
      strcpy (L->caption, c);
      L->pos = p;
      return L;
}
```

The function is called CreateLabel and returns a pointer to this label structure. It requires two input parameters – a character string (the label's caption) and an integer (the label's position). The function mpmMemBlockAllocate is used to allocate memory for the label. The id of the label is generated by using two bytes for its type (which is Label and is given as 0x17 in the AV/C

Panel Subunit specification) and two bytes for its unique id. The unique id is incremented each time an element is created, thus ensuring that each element has a unique id. Finally, the caption is copied into its place-holder and the position is assigned. The function then returns a pointer to this newly created label.

For an example as to how all this works, consider the following production:

```
Label = "<Label" Caption Position "/>".
```

This production states that the Label tag must begin with the terminal "<Label" and end with the terminal "/>". Between these terminals, exactly one caption and one position element must appear. Any deviations from this are syntactically incorrect.

The following code is the same production, but now it is attributed (see the previous section for more explanations on attributed productions):

```
Label<TLabel* &L> = (. char c[20], p[20]; .)
  "<Label" Caption<c> Position<p>
  (. int pos = atoi(p);
     L = CreateLabel (c, pos);  .) "/>".
```

Now consider a line from the XML file being parsed that looks like this:

```
<Label caption="Notice" position="2" />
```

The code first parses the "<Label" terminal. If this terminal is not found or is incorrect, an error message is generated. Next, there should be two strings following the "<Label" symbol – a Caption and a Position. When the parser recognizes that it must parse a non-terminal (Caption and Position are here both non-terminals), it calls the Caption function, passing to it the variable c. If the function is completed successfully, c will contain the string "Notice". Similarly, the Position function is called to parse the string following Caption and if it returns successfully, the position will contain the string "2". If either function fails, it is because the caption or position was specified incorrectly, and the parser generates an error message. Then the position p is converted from a string to an integer called pos and the Label L is created (memory is set aside for it and its caption and position are assigned) by calling the CreateLabel function shown above and passing it the caption and position. Finally, the terminal symbol "/>" is parsed and if this is successful, the XML line is successfully parsed and all the corresponding semantic actions have been performed.

4.6 The DHIVA

Once the XML structure of the GUI is defined and parsed into C structures on the target, the target must be able to receive AV/C commands from the controller and be able to respond to them appropriately.

The target itself is a Digital Harmony Interface for Video and Audio (DHIVA) and is an IEEE1394 network interface with an embedded ARM7 RISC processor and an IEEE1394 protocol stack. The protocol stack was developed by Digital Harmony Technologies, and implements audio, MPEG/DV video and control mechanisms. At the time of the inception of this thesis, the IEEE1394 stack had already been written in C, and so C is used to implement the AV/C Panel subunit. Figure 26 shows what the DHIVA looks like, as well as annotating several of its components.

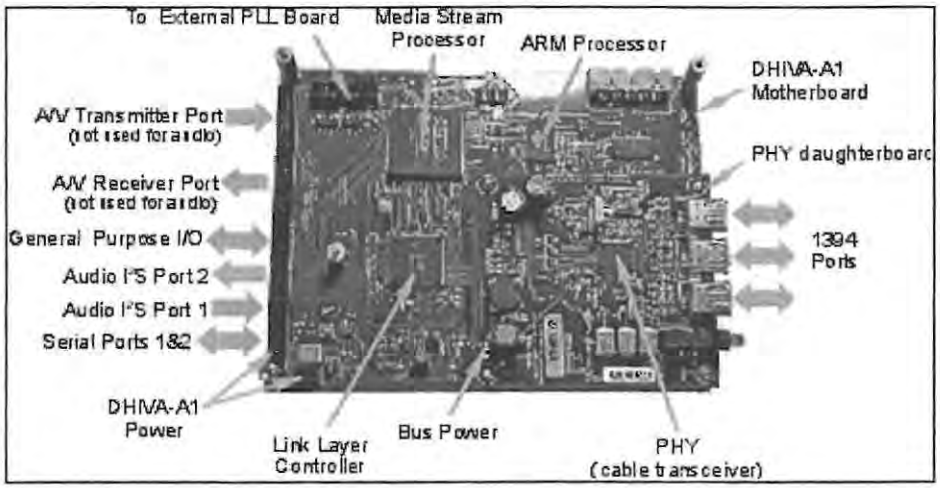

Figure 26 - An annotated picture of a DHIVA

Figure 27 shows the DHIVA's Digital Harmony Protocol stack [43]. The module shaded in gray is the Panel Subunit and the diagram shows where this module resides in relation to the other modules on the DHIVA. The AV/C handler (the block directly below the Panel Subunit block) receives AV/C packets from the FCP block, which in turn receives these packets from the link layer. The FCP packet strips the FCP headers and passes the AV/C packet to the AV/C handler. The AV/C handler then removes more headers and passes the packet to the relevant subunit. All panel subunit packets are passed to the panel subunit in this manner.

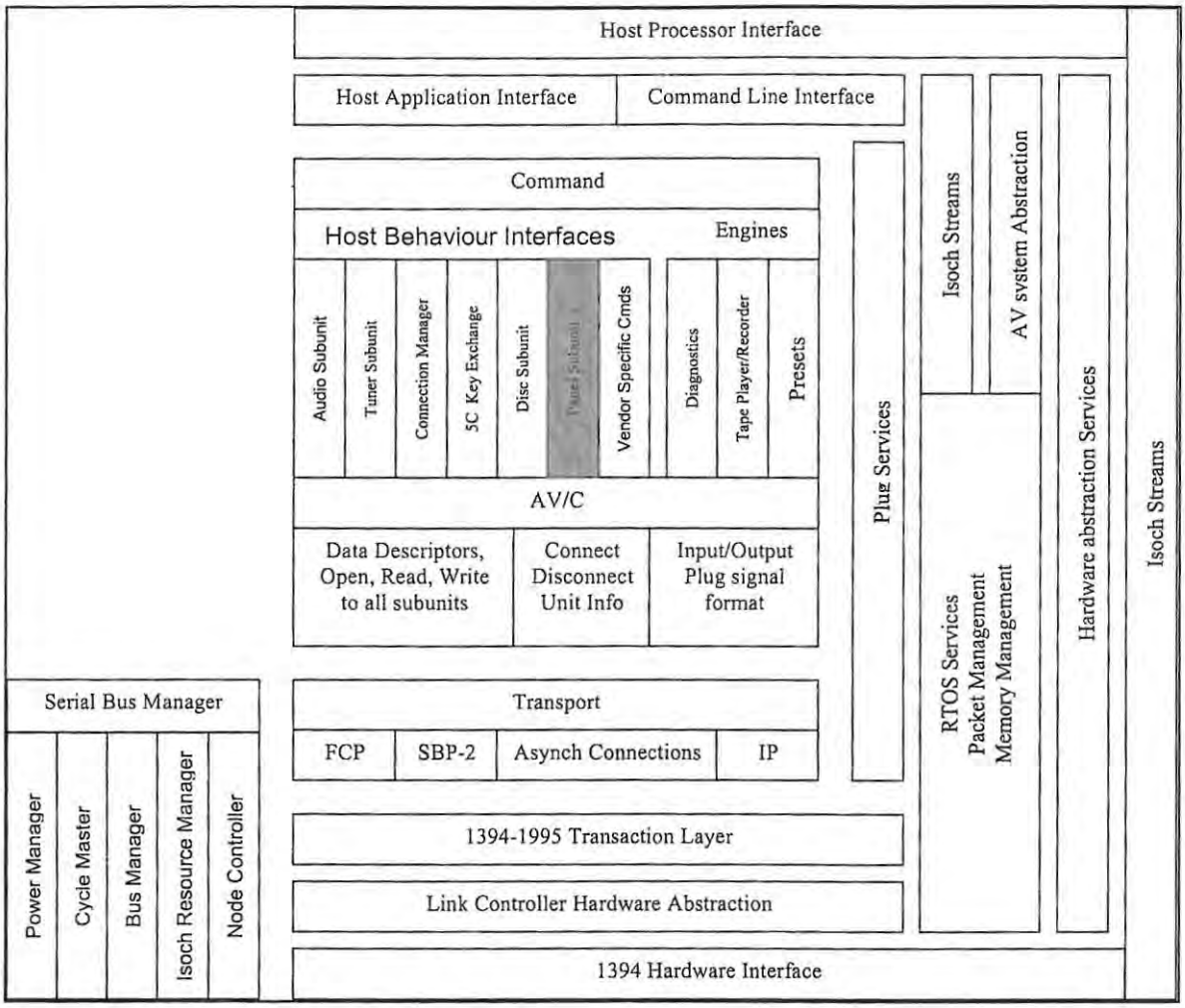

Figure 27 - The DHIVA's Digital Harmony Protocol Stack

4.6.1 DHIVA Ward/Mellor Diagrams

Since the DHIVA system is not object oriented, some other method has to be used to show the design of the AV/C Panel Subunit implementation. Ward/Mellor [34] provide a set of tools for the analysis of structured, real-time systems such as the DHIVAs. These tools allow diagrams to be drawn that show the flow of data (arrows) into and out of *transforms* (circles) that perform certain functions. *Data stores* are also denoted (by parallel lines). Figure 28, Figure 29 and Figure 30 show the Ward/Mellor diagrams for the DHIVA AV/C Panel Subunit system.

Figure 28 shows the main data flows into and out of the DHIVA AV/C Panel Subunit system. Only three types of command are sent to the DHIVA – open/close panel subunit messages, panel data requests and user configuration commands. These panel subunit messages originate at the controller and are transported on the IEEE1394 network in FCP frames (see §

2.4.5.2 for more details). The physical layer of the DHIVA recognizes that the messages are addressed to its node (see § 4.7.1 for more details) and decodes the message. The packet is passed to the FCP handler (a callback function) for further decoding. This function is called every time an FCP packet arrives at the DHIVA. The FCP handler then decodes the headers of the message and recognizes that an AV/C packet is encapsulated in the FCP packet. The FCP header is stripped and the payload (the AV/C packet) is passed to the AV/C handler (another callback function). This function then decodes the AV/C header and hands the payload to the relevant function (specified by the opcode of the message) for further processing.

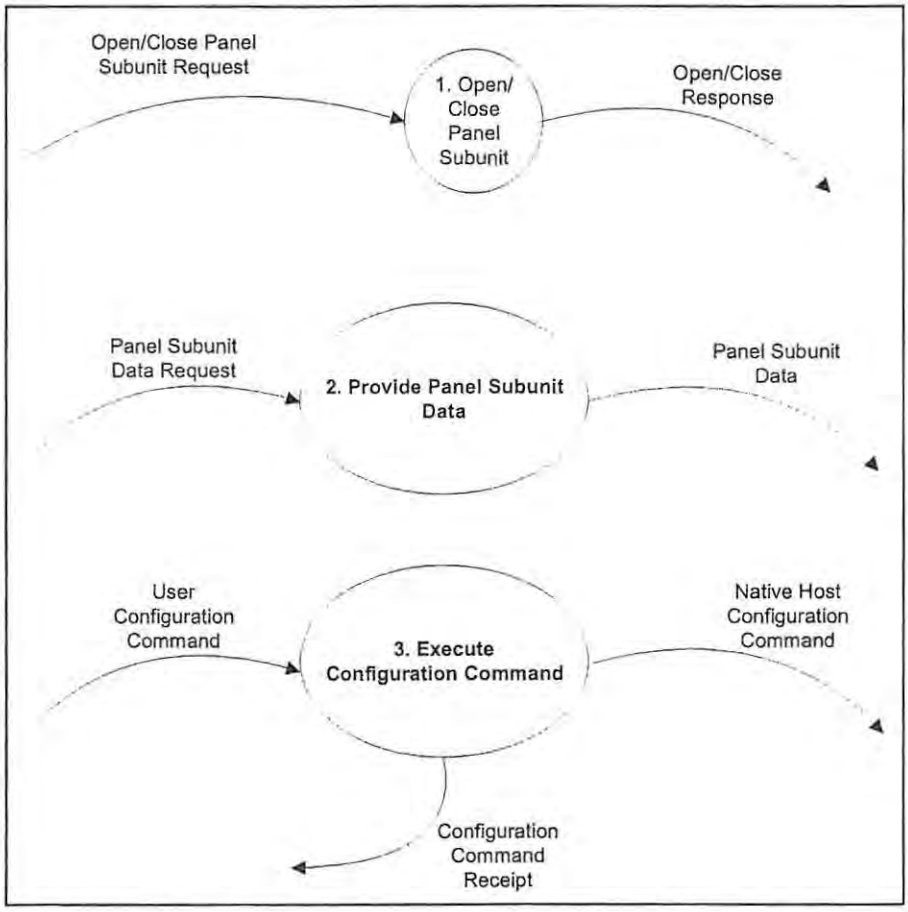

Figure 28 - The high level Ward/Mellor diagram of the DHIVA AV/C Panel Subunit system

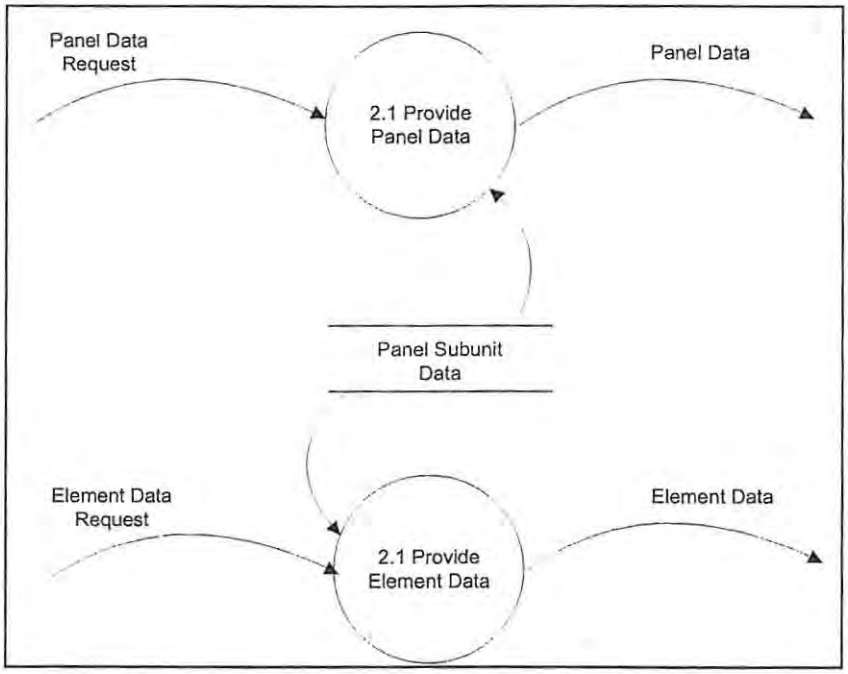

Figure 29 - Detail of Transform 2 of the Ward/Mellor diagram of the DHIVA AV/C Panel Subunit system

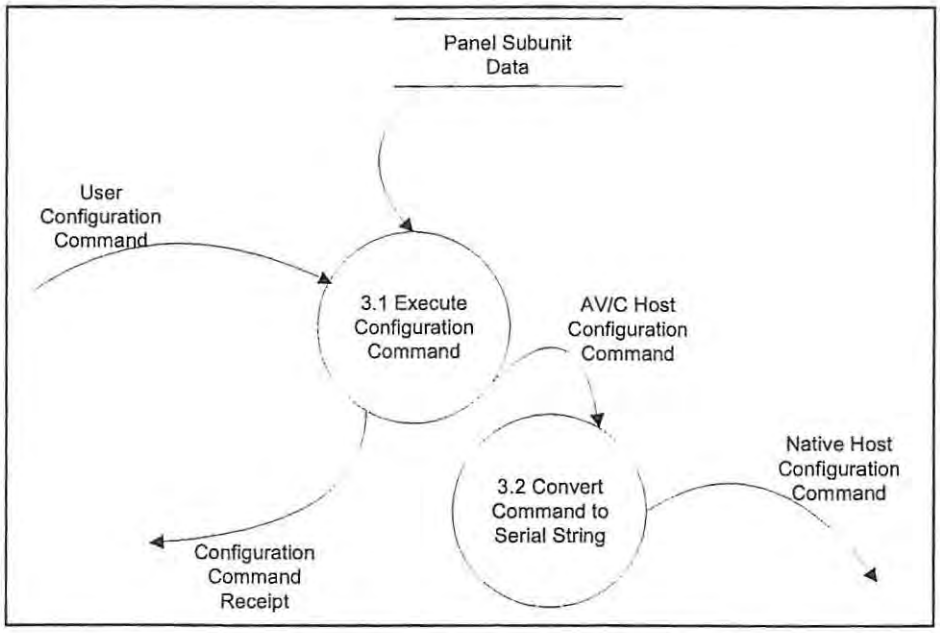

Figure 30 - Detail of Transform 3 of the Ward/Mellor diagram of the DHIVA AV/C Panel Subunit system

4.6.2 Open/Close AV/C Panel Subunit Messages

If the message is an "open panel subunit message", ownership of one of the DHIVA's source *plugs* [13] (a software object that allows connection and transfer of streams of data – see § 3.4.3.2) is given to the controller. Once the controller obtains ownership of the plug, it establishes a connection, or data pipe, between the controller and the target. All panel subunit data that is transferred from the target to the controller is done on this data pipe. This data pipe is an *asynchronous connection* [35]. Asynchronous connections enable transfer of large amounts of data over a serial bus. An AV/C packet is returned to the controller indicating the id of the source plug the controller now owns. If no plug is available, the DHIVA returns an AV/C packet with a REJECTED status and the controller will have to try to open the panel subunit again at a later stage. When the controller sends a "close AV/C Panel Subunit message", the source plug is freed. The panel subunit may have several source plugs, and controllers may only own one of these plugs at a time.

Figure 31 shows how the panel subunit is contained within the unit on the device. One unit that is present on each device represents that device, and the unit may contain several subunits that divide its functionality into logical "blocks". In the case of the AV/R, the unit and the panel subunit both physically reside in the DHIVA. When the controller begins working with the target, it issues an AV/C command to establish a connection between the unit's asynchronous plug and one of the panel subunit's free source plugs. Thereafter, it issues another command that connects its asynchronous plug to the units asynchronous plug, and a channel is opened for the transport of GUI data.

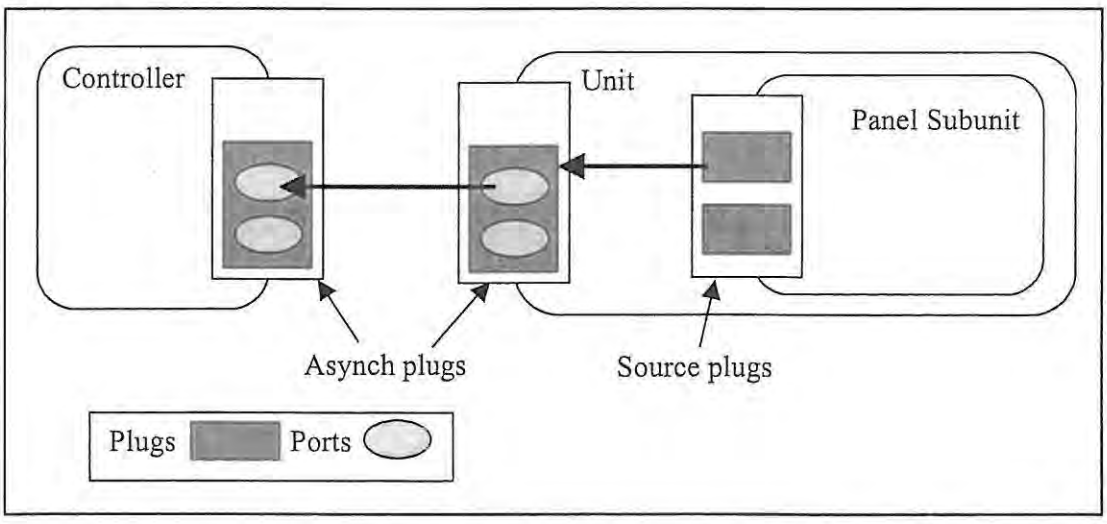

Figure 31 - The plugs of the panel subunit and its relation to its unit and the Controller

However, two implementation shortfalls prevent asynchronous connections being utilized. Firstly, the DHIVA does not yet support asynchronous connections. Secondly, the controller requires a device driver that enables it to transmit and receive FCP packets. This driver, however, only allows acknowledgement FCP packets to be read (see § 4.7.1 for more details). In other words, the only time the driver will allow an application to receive FCP packets is directly after sending an FCP packet. These two factors combined to force the use of an alternative method of transferring panel subunit data from the DHIVA to the controller.

The solution was to utilize the fact that the controller may only read acknowledgement FCP packets. The DHIVA sends two acknowledgement packets – the first contains the panel subunit data while the second contains the formal acknowledgement that the controller is really waiting for. The device driver on the controller is simply requested to read the "incoming FCP acknowledgement buffer" twice instead of once.

4.6.3 Panel Data Request Messages

While panels are AV/C Panel Subunit elements, the way they are treated is different from other elements. A request by the controller (the TV) for an element will simply cause the element to be returned by the target (the AV/R). A request by the controller for a panel causes the panel and the id's of all the elements it contains to be returned. Hence the transforms for the panel data request and the element data request are different and this is shown diagrammatically in Figure 29.

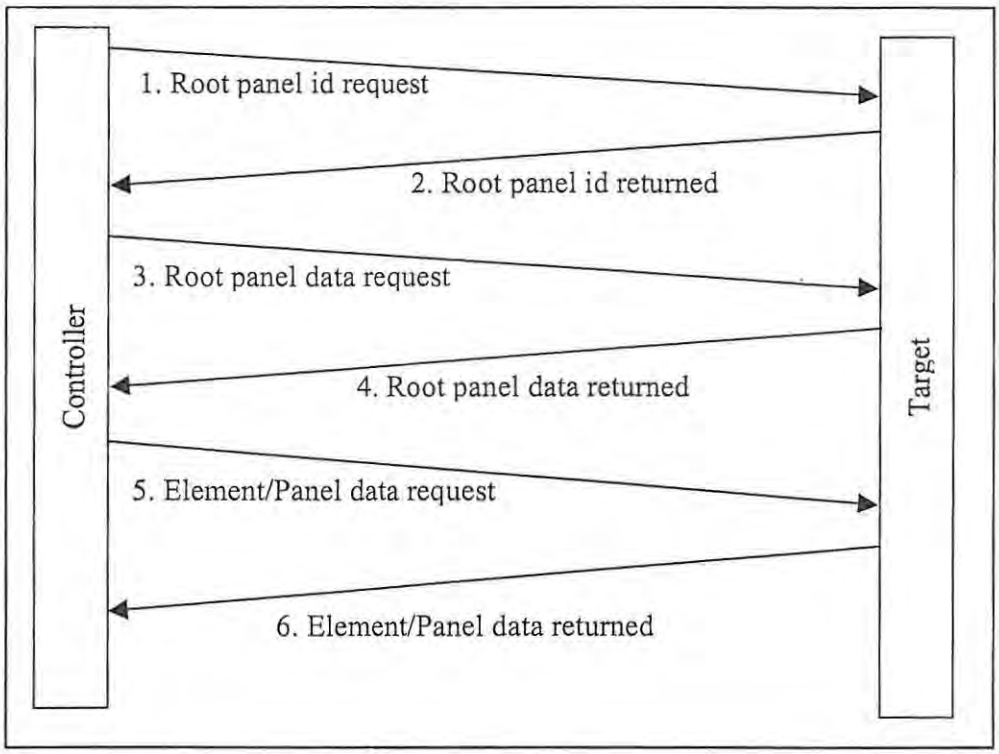

Figure 32 - Interactions between Controller and Target to allow the Controller to obtain GUI data

Once the controller opens the panel subunit, it will begin to request the target's GUI data. Figure 32 shows the interactions that occur between the controller and target when the controller at this stage. The first request is for the id of the root panel (the first or main panel of the GUI) shown in Step 1. The controller achieves this by requesting an element of the type "Panel" with the id "root". The id of the root panel is then found in the panel subunit structures stored on the DHIVA and returned to the controller in Step 2. The controller will then request the root panel data (Step 3). The DHIVA searches the panel subunit structures and builds a packet containing the root panel and the id's of all the elements contained within the panel (Step 4). The controller then iterates through each element id contained in the root panel, requesting the element itself from the DHIVA (Steps 5 and 6). Steps 5 and 6 are repeated for all the elements in the root panel.

The root panel will contain links to other panels. When a link's data is sent from the target to the controller, the id of the panel that the link is referring to is included in that data. The controller then requests the panel data using this panel id (Step 5). Once the panel data is returned (Step 6), the controller can iterate through all the id's of the elements contained in the new panel and

by repeating Steps 5 and 6 can retrieve all the element data needed. If other panel links occur in the new panel, the same process is repeated. In this manner the controller retrieves all the GUI data starting from the root panel.

The controller sends "Push GUI Data" packets in order to retrieve information. The panel data is in turn returned on the asynchronous connection. However, since the DHIVA is not capable of asynchronous connections, the GUI data is returned in the correct format using an FCP command. If this implementation is moved to a system capable of asynchronous connections, the GUI data packets would not need to change – they simply need to be sent along the asynchronous connection instead of by FCP.

opcode	PUSH GUI DATA ($7E_{16}$)
operand[0]	source_plug
operand[1]	sub-function
operand[2]	generation_number
operand[3]	
operand[4]	status
operand[5]	indicator
operand[6]	element_id
operand[7]	
operand[8]	
operand[9]	

Figure 33 - The format of the Push GUI Data packet

Figure 33 shows the format of the Push GUI Data panel subunit packet. The opcode identifies this packet as a Push GUI Data packet. The source_plug field identifies which source plug the controller wishes the GUI data to be sent on. This source plug is specified to the controller by the target when the controller initiates communication with the target. The sub-function field may be one of two values: new and clear. When the controller wishes to obtain the GUI data for an element, it uses the new sub-function and places the id of the element into the element_id field. If for some reason, the controller wishes to cancel the request for data, it uses the clear sub-function, specifying which element it no longer wishes to receive data for using the element_id field. Both the generation number and status fields are set to FF_{16} by the controller. The indicator field is used to specify if the controller wishes the target to return all the elements that are nested hierarchically in the element it is currently trying to obtain data for. For example, if the element that the controller wishes to obtain is a panel, then the indicator field can be set so

that the target will return just the panel, or it can be set to return the panel and all the elements contained within the panel too.

The GUI data specified by the controller is returned to the controller by the target along the asynchronous connection (or in this case, via and FCP packet). However, the AV/C Panel Subunit specification mandates that the target returns a panel subunit packet reply to the controller too. This packet is simply the packet that the controller sent, except that the status and generation_number fields are changed to reflect the status of the panel subunit and the current generation number. The generation number specifies the current generation of the GUI. If GUI elements change in time, each change corresponds to an increment in the generation number. It was decided that the GUI placed on the AV/R would be static (since all the ranges and choices are predefined and never change) and so for the TV-AV/R system, the generation number is always 0. The status of the panel subunit is one of the following:

- no error – the Push GUI Data command is accepted with no errors
- preparation – the panel subunit is preparing a GUI data packet to be sent
- source plug busy – the source plug is transmitting a GUI data packet
- no element – the element specified by the controller does not exist
- not connected – the panel subunit's source plug is not connected to the unit's asynchronous plug
- not owner – the controller is not the owner of the specified source plug
- cancelled – the specified data transmission is cancelled
- not transmitting – there are no packets being transferred (i.e. nothing to cancel)
- any other error – there is some other internal panel subunit error

The element_id field is four bytes long. The fist two bytes specify the type of element (panel, link, scroller, etc.) and the second two bytes are a 16-bit unique identifier. Some special cases exist – for instance, if the element type is "Panel" and the id is FF_{16}, then a request is being made for the root panel. In this case, no GUI data is transferred – the actual unique id of the root panel is returned in the reply packet – say 00. The controller now knows the unique id of the root panel and sends another Push GUI Data packet asking for the GUI data of the element with type "Panel" and id 00. This time the GUI data of the panel is sent back to the controller.

Once the controller receives the data for all the panels, it displays the root panel to the user (see § 4.7.2 for more details). Thereafter, when the user presses a panel link button, the new panel is displayed to the user.

4.6.4 User Configuration Command Messages

Figure 30 shows how user configuration commands are handled by the DHIVA. The incoming AV/C packet contains the id of the element the user is interacting with and any arguments that need to be conveyed (for example, if the user changes the value of a slider, the user configuration command message will contain the id of the slider as well as the value the user has moved the slider to). In transform 3.1, the element is found in the panel subunit structures and its corresponding "host command" is obtained (this host command is specified when building the GUI – see § 4.3.1). If the element is valid, an OK AV/C message is transmitted back to the controller. The argument of the configuration command is then used to modify the host command and it is passed to transform 3.2. In transform 3.2, the command is converted to the correct serial form and transferred to the device (the AV/R) via the serial host interface. This is how the actual action is performed.

Figure 34 shows the format of the User Action command. The source_plug and generation_number are as for the Push GUI data command (see previous section). The element_id is the type and unique id of the element that the user is currently interacting with. The action_type specifies what action id being performed and any action-specific data is transferred in the action_specific_data field. For example, if the user changes a slider's value, the new value is sent in the action_specific_data field. The action types specified are select, press, release, set_value (used when a slider is changed), enter_data, choose_list (used when a scroller is changed), select_item and select_element.

opcode	USER ACTION ($7F_{16}$)
operand[0]	source_plug
operand[1]	generation_number
operand[2]	
operand[3]	element_id
operand[4]	
operand[5]	
operand[6]	
operand[7]	action_type
operand[8]	action_specific_data
...	
...	

Figure 34 - The format of the User Action packet

It must be noted that the controller (the TV) has no way of knowing if the command has succeeded or not – the OK AV/C message from the target (the AV/R) confirms only that the configuration request is syntactically correct.

4.7 The ControllerApp

4.7.1 Features of the ControllerApp

The final portion of the AV/C AV/R system to consider is the ControllerApp. The ControllerApp is a program that resides on the controller device. The ControllerApp performs several functions:

- initiates communications with the target
- obtains GUI information from the target
- displays GUI to user
- receives user input
- transmits user actions to the target
- receives state change information from the target and update GUI to reflect these changes

The ControllerApp must, therefore, firstly be able to interface with the IEEE1394 network. The computer (the controller device) is connected to the IEEE1394 network via an IEEE1394 *network interface card* (NIC). The computer becomes a node on the IEEE1394 network. The program interfaces to the NIC via a *device driver.* The device driver (written by Digital Harmony) provides the program with Application Programming Interfaces (APIs). The APIs provide abstraction away from the actual networking protocol (which the driver handles). The driver takes care of connecting the computer to the IEEE1394 network automatically. The program can then use the simple function calls defined by the APIs to send FCP packets and receive FCP acknowledgements on the IEEE1394 network. The relation between the program (the ControllerApp), the device driver, the NIC and the IEEE1394 network is shown in Figure 35.

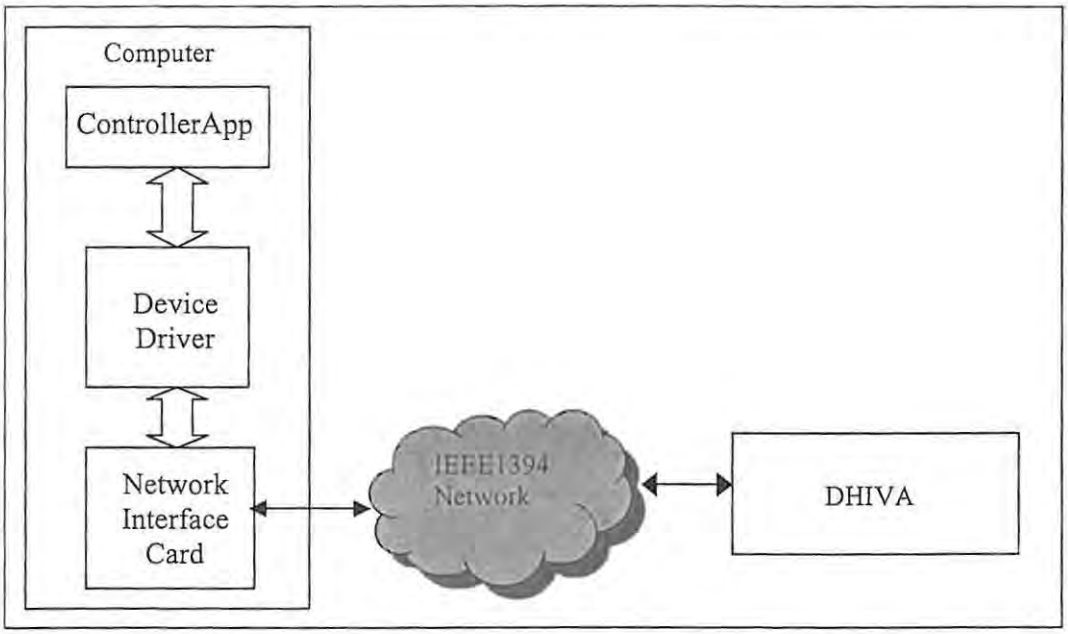

Figure 35 - The Device Driver interacts between the IEEE1394 Network and the ControllerApp

Secondly, the ControllerApp must be able to issue and receive AV/C packets – and specifically panel subunit packets. Thirdly, the ControllerApp must be able to display the GUI elements dynamically - the GUI is built at run time from the GUI data received from the target (the DHIVA and the AV/R). Fourthly, the ControllerApp must be able to interpret user inputs and issue corresponding AV/C panel subunit user action commands.

The ControllerApp is written in Visual C++, since this environment allows all of the above features to be implemented. It allows the device driver to be used to interface with the IEEE1394 network (the driver APIs are written in C++). It allows the panel subunit logic to be programmed in order to send and receive panel subunit packets and also allows dynamic GUIs to be built at run time. Furthermore, Visual C++ is an object oriented environment and allows rapid development of visual and object oriented applications.

4.7.2 ControllerApp Design

The design of the ControllerApp conforms to the following sequence:

- Use case diagram (to show the broad functionality of the system)
- Scenarios (to set out clearly what events occur in the system)
- Sequence diagrams (to show interactions between classes)
- Object Model (to show the classes with their attributes and member functions)

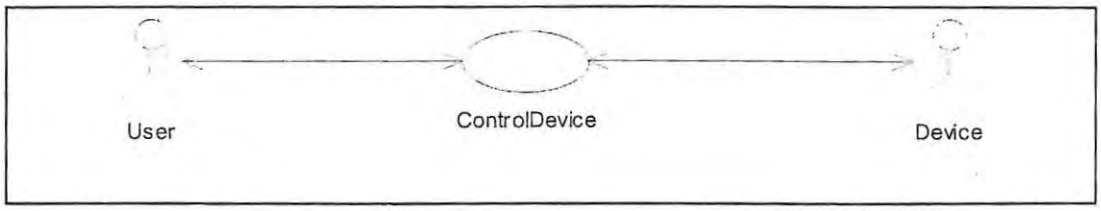

Figure 36 - The Use Case Diagram for the ControllerApp

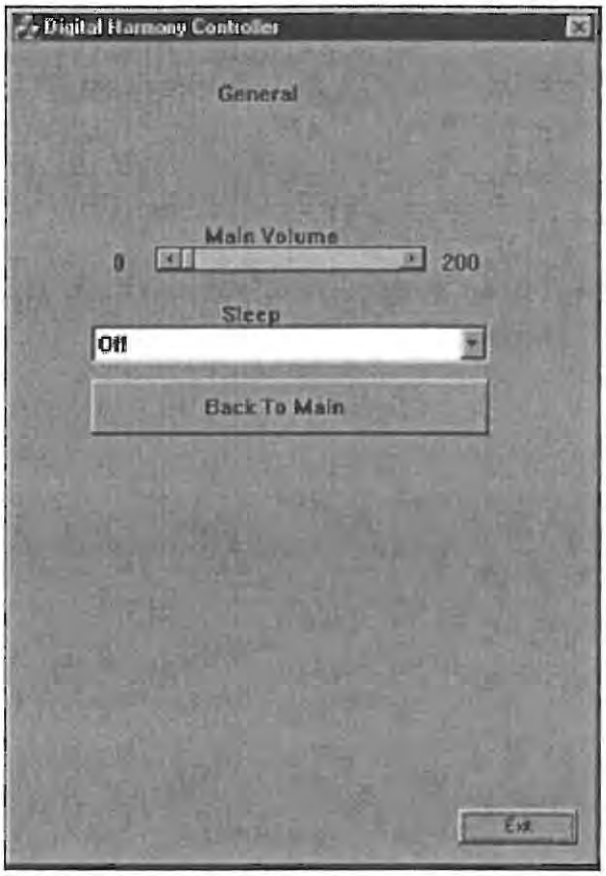

Figure 37 - An example panel for the AV/R system

Figure 36 shows the Use Case Diagram for the ControllerApp. The core function of the ControllerApp is controlling the device. There are two actors that influence the ControllerApp – the user and the device. The ControllerApp responds to stimuli received from these two actors.

The panels accessible to the user depend on the configuration that the manufacturer of the target device specifies in XML. An example of a panel for the AV/R system is shown in Figure 37. The panel's caption is "General" and it contains a slider (called "Main Volume", ranging from 0 to 200), a scroller (called "Sleep" currently showing "Off" is selected) and a panel link labelled "Back to Main". If the user slides the thumb of the slider, the main volume of the device this GUI

represents is changed. Changing the currently selected sleep option will set the sleep timer. Pressing the panel link will cause this panel to be replaced by the "Main" panel.

There are several scenarios for the ControllerApp. Scenario 1 shows how the ControllerApp initiates communications with the target. Scenarios 2 and 3 show how the ControllerApp deals with retrieving Panel data from the target. Scenario 4 shows how the ControllerApp displays (or renders) the GUI. The remaining scenarios show how the ControllerApp interacts with the user.

- Scenario 1: Startup. The user starts the program. The program scans the network for all attached devices.
- Scenario 2: Get Panel Data. The controller obtains all the panel data from the target. Each panel retrieved contains the id's of all the elements it contains, as well as its name.
- Scenario 3: Get Element Data. The controller iterates through all the elements in the panel, obtaining the element's data from the target by requesting passing the id of the element to the target. Each panel link refers to another panel, and if the panel being referred to by the link is not yet retrieved, the controller retrieves the panel.
- Scenario 4: Render Panel. The Controller displays the current panel.
- Scenario 5: Handle Link Press. The controller detects that the user has pressed a panel link. The controller gets the name of the panel that is being referred to by the link, clears the current panel off the screen and displays the new panel.
- Scenario 6: Handle Slider Change. The controller detects that the user has changed the value of a slider. It notes the new value and the id of the slider. It then conveys a user action command (using the new value and slider id) as arguments to the target.
- Scenario 7: Handle Scroller Change. The controller detects that the user has changed the selection of a scroller. It notes the new selection and the id of the scroller. It then conveys a user action command (using the new selection and scroller id) as arguments to the target.

Each scenario has a corresponding sequence diagram and they appear as follows:

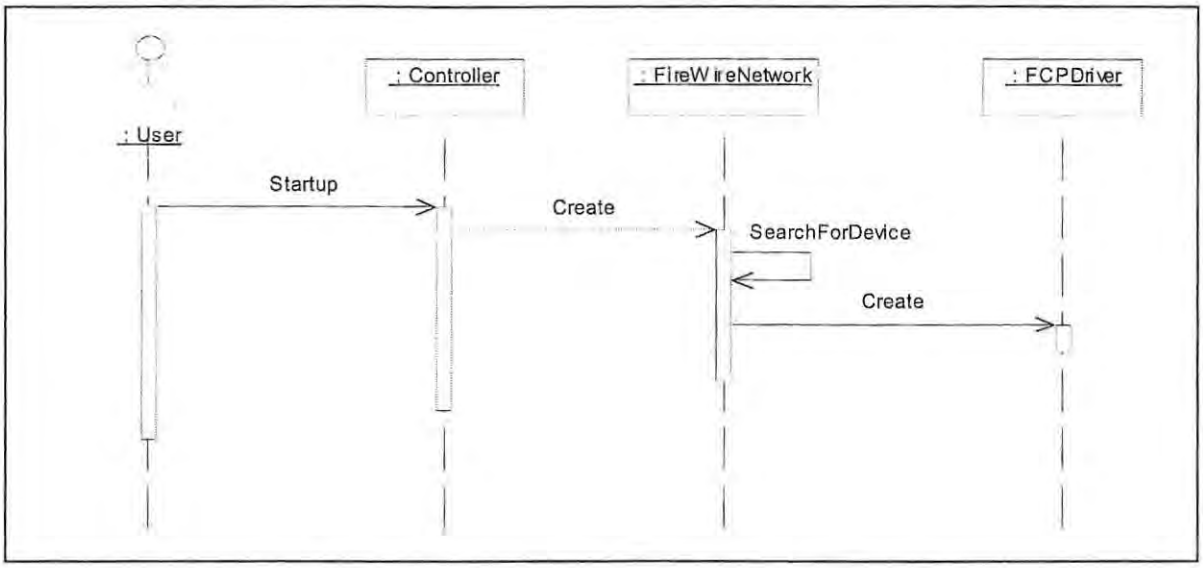

Figure 38 - The sequence diagram for the ControllerApp Scenario 1

When the FireWireNetwork object searches for devices on the IEEE1394 network, it enumerates each one and creates a handle to each one. The FCPDriver object uses the handle to address a particular node (the node with that handle) on the network.

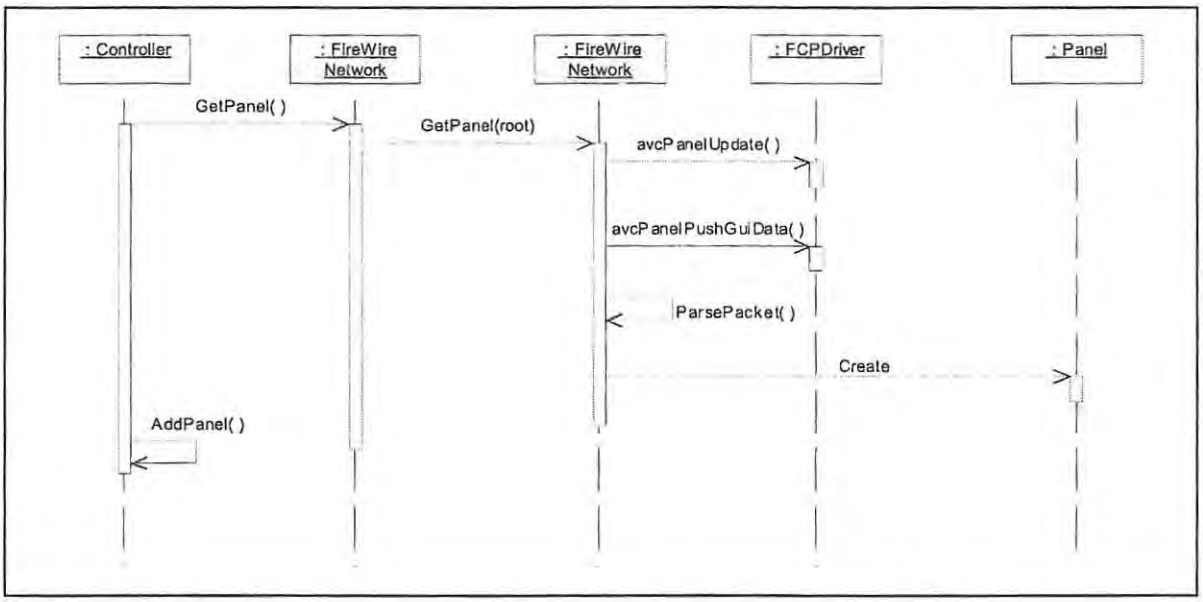

Figure 39 - The sequence diagram for the ControllerApp Scenario 2

The Controller object holds a list of panels. Since a GUI may contain several panels, and the user may switch often between these panels, the GUI data for that panel and all its elements are stored locally on the computer as they are received from the target. This prevents the

ControllerApp from having to obtain the Panel data from the target again when the user switches to that panel.

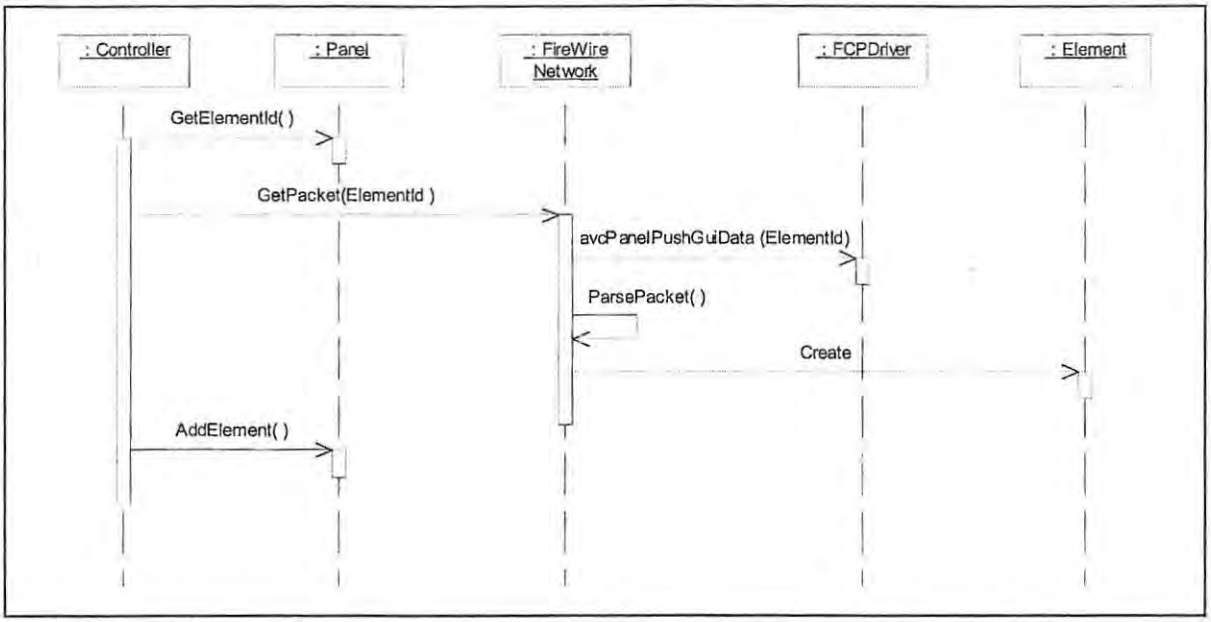

Figure 40 - The sequence diagram for the ControllerApp Scenario 3

The ControllerApp receives each panel from the target. The panel at this stage hold only the id's of the elements it contains. The ControllerApp must now retrieve each element's data from the target. As each element is retrieved, an object is created that encapsulated the functions and attributes of that element and it is stored on the computer and added to the panel's list of elements.

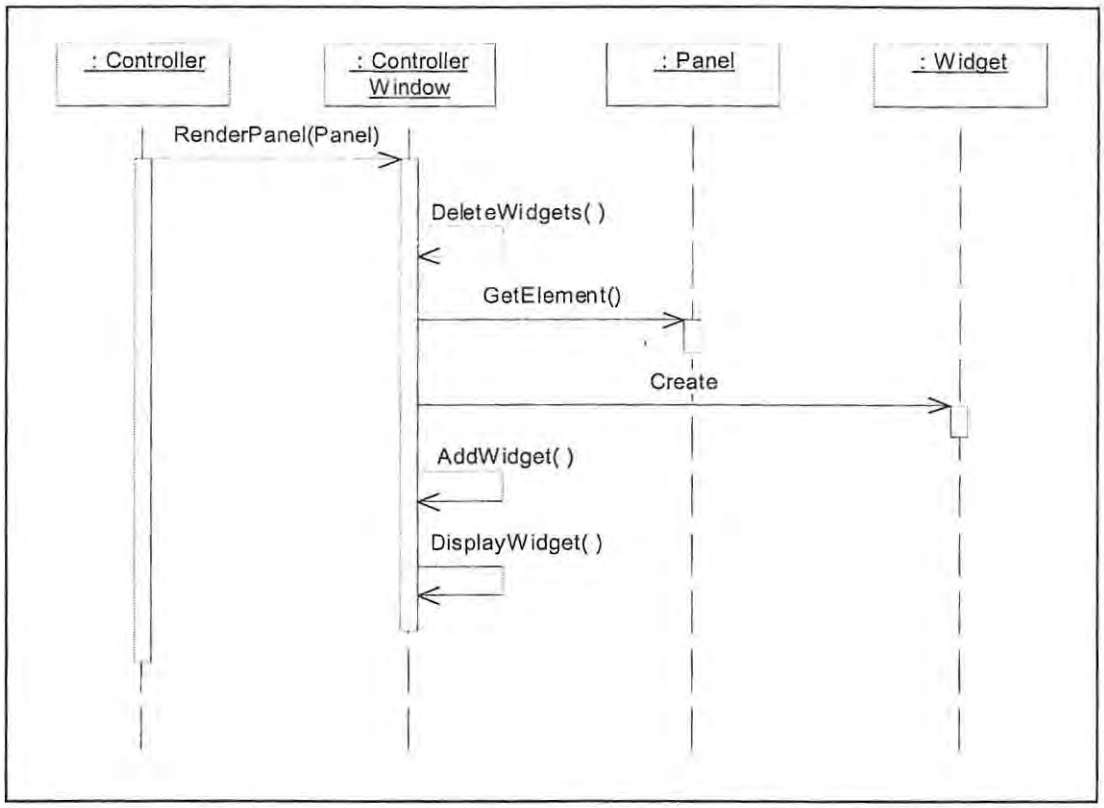

Figure 41 - The sequence diagram for the ControllerApp Scenario 4

Once all the elements in a panel have been retrieved from the target, the ControllerApp displays the panel. It iterates through each element, telling that element to create for itself an on-screen representation of itself (the Widget object) and to display that Widget. The Widgets are used to determine when and how the user is interacting with a particular element. For instance, if the element is a Scroller, the Scroller will create a drop-down box widget. The widget is displayed to the user and its choices enumerated from the Scroller's properties. When the user changes a selection, the widget reports that event to the element it is associated with so that it can then generate the corresponding panel subunit user action command. Three user action types exist – the user can press a panel link button, change a slider or change the selection of a scroller. The sequence diagrams for each of these actions follows:

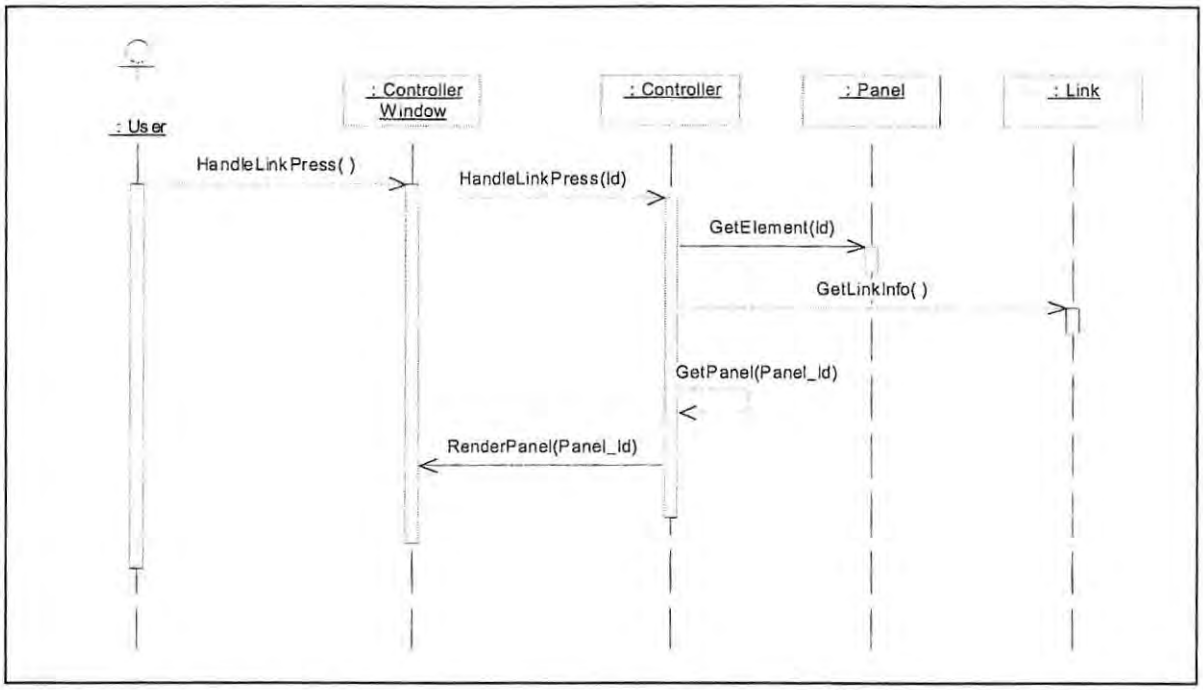

Figure 42 - The sequence diagram for the ControllerApp Scenario 5

The HandleLinkPress occurs when a user presses a panel link button. The id of the panel being linked to by the user is retrieved from the panel link's properties and the current panel is removed and the new panel is displayed.

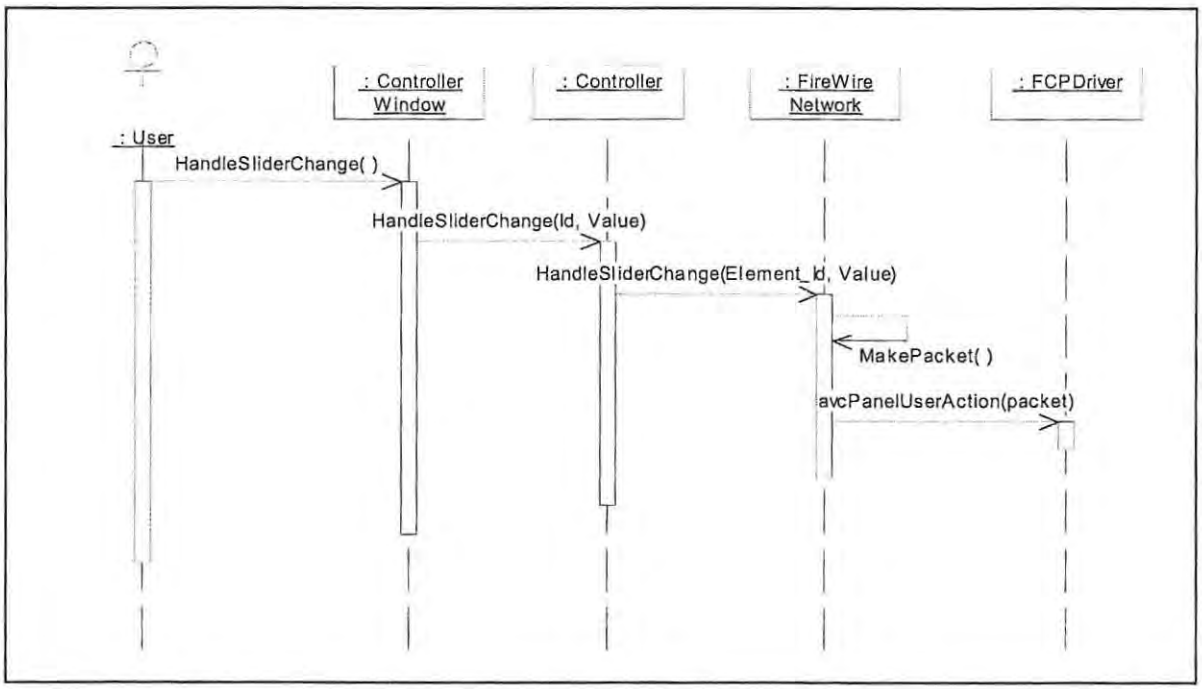

Figure 43 - The sequence diagram for the ControllerApp Scenario 6

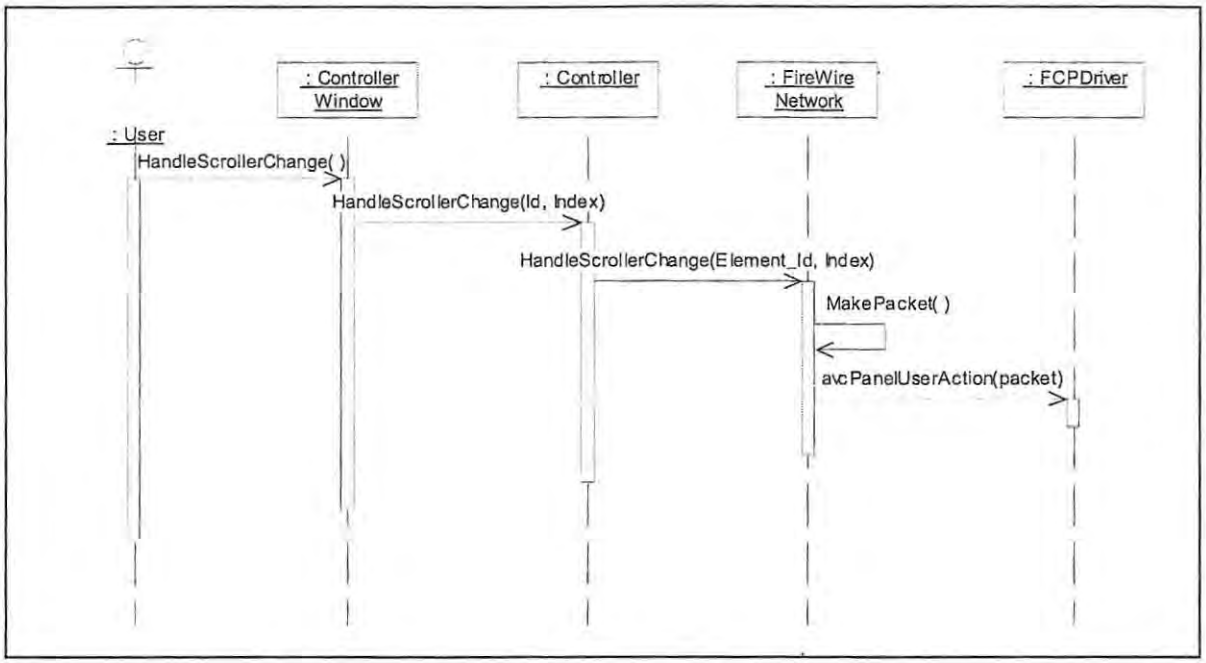

Figure 44 - The sequence diagram for the ControllerApp Scenario 7

The last two sequence diagrams (Figure 43 and Figure 44) show how the ControllerApp sends user action messages to the target when the user changes either a Slider or a Scroller respectively. The element being interacted with is determined from the Widget that the user has clicked. The elements properties include what user action command is to be sent to the target as well as what arguments (the new Slider or Scroller value) will accompany the command. The command is relayed to the target and the target then sends the appropriate serial string command to the AV/R to perform the user's command (see § 4.6.4 for more details).

The complete object model for the ControllerApp is shown in Figure 45.

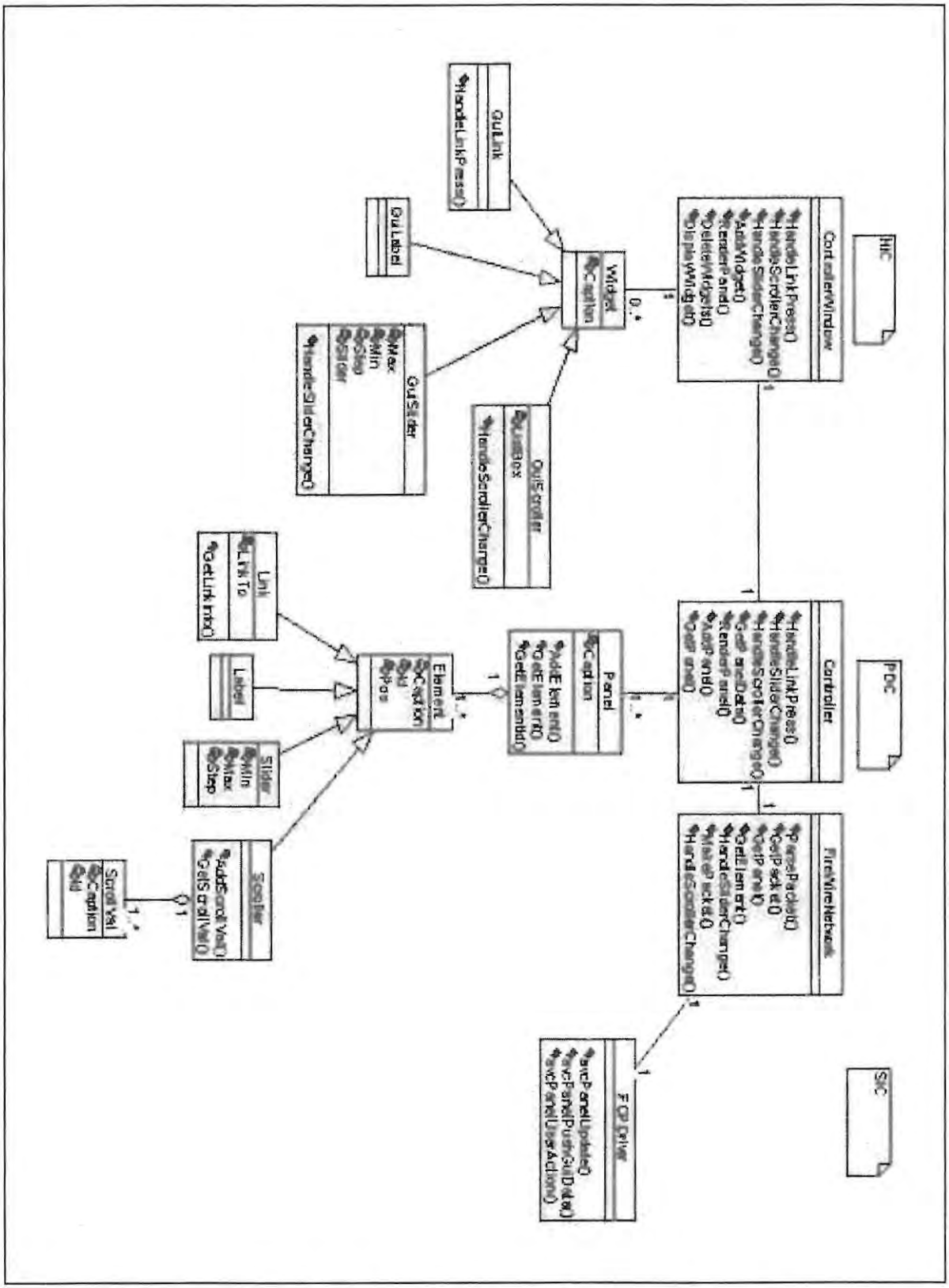

Figure 45 - The complete object model for the ControllerApp

4.8 Summary

This chapter details the implementation of a simple remote configuration system. The system remote configuration of an Audio/Video receiver (AV/R) via a TV, both on an IEEE1394 network.

The AV/R is a Yamaha RXV 1000. The AV/R itself cannot communicate with the IEEE1394 network directly. The AV/R has to be *hosted* by another device capable of interfacing with the IEEE1394 network on its behalf – the Digital Harmony Interface for Video and Audio (DHIVA). The implementation of the system described in this chapter makes use of the AV/C Panel Subunit specification to implement the TV-AV/R remote configuration system.

A PC using a Windows application simulates the TV controller for the panel subunit so that the controller can receive user input, can provide user output and is directly connected to the IEEE1394 network.

When designing the AV/C Panel Subunit implementation for the TV-AV/R system, two main goals emerged. Firstly, a working implementation had to be obtained. Secondly, a set of tools had to be developed to aid manufacturers in developing further remote configuration applications similar to this one (i.e. remote configuration systems for devices other than the TV and AV/R). Bearing these goals in mind, the work done in order to realize the remote configuration TV-AV/R system was five-fold. Each step was essential for the implementation, and a tool was created for each step. Future systems need only use the tools created in order to quickly and easily create a fully functioning remote configuration system. The steps followed were:

1. The abstraction of the GUI (the method of specifying how the GUI is laid out and how it is stored in the target) is achieved by making use of an extended Markup Language (XML) grammar. Only five graphical elements are selected, and with these elements, virtually all configuration operations can be achieved: panels (these are organizational and serve as containers for the other elements), labels (these are static text fields), links (these are buttons that link panels, i.e. clicking a link brings up the panel this link points to), sliders (these allow a user to select a value within a well-defined range) and scrollers (these allow a user to scroll through a set of several alternatives, seeing only one at a time).

The next step is to map these elements and their properties to XML notation. This is done by creating a Document Type Definition (DTD) that is able to verify the correctness of an XML GUI

file. Each of the five elements named above have corresponding XML tags arranged hierarchically.

2. A GuiBuilder tool is created. This allows manufacturers to specify the layout of a GUI graphically. The program, created in Visual Basic, can then output an XML file which corresponds exactly to the manufacturer's GUI layout.

3. The XML-GUI Parser is now created to be able to parse the XML GUI file the manufacturer has created. The parser is created using a compiler generator (Coco/R) and its function is to create C structures for the elements. These structures are then stored on the DHIVA and the DHIVA is now ready to interact with the TV.

4. The AV/C Panel Subunit is then implemented on the DHIVA in C code. The panel subunit slots into the rest of the Digital Harmony stack already present on the DHIVA. The panel subunit gets packets from the AV/C handler (which routes all AV/C messages to their intended subunits). The panel subunit on the DHIVA is able to retrieve GUI data for any given element as well as perform the user's actions by transmitting the actions to the AV/R across a serial port.

5. Finally, the ControllerApp is implemented on a PC that is connected to the AV/R via the DHIVA on an IEEE 1394 network. The ControllerApp initiates communications with the AV/R, obtains the GUI from the AV/R, displays the GUI to the user, receives user input and transmits user actions to the AV/R and receives state change information from the AV/R and updates the GUI to reflect these changes.

Chapter 5 - The UPnP Presentation Mechanism Solution to Remote Configuration

5.1 Introduction

In chapter 3, four methods of implementing remote configuration are presented – HAVi's Data Driven Interaction (DDI), Jini's ServiceUI object, AV/C's Panel Subunit and UPnP's presentation mechanism. Chapter 4 detailed the Panel Subunit solution to remote configuration for a TV-AV/R system on an IEEE 1394 network. This chapter discusses how UPnP's presentation mechanism may be used to implement remote configuration for the same TV-AV/R system.

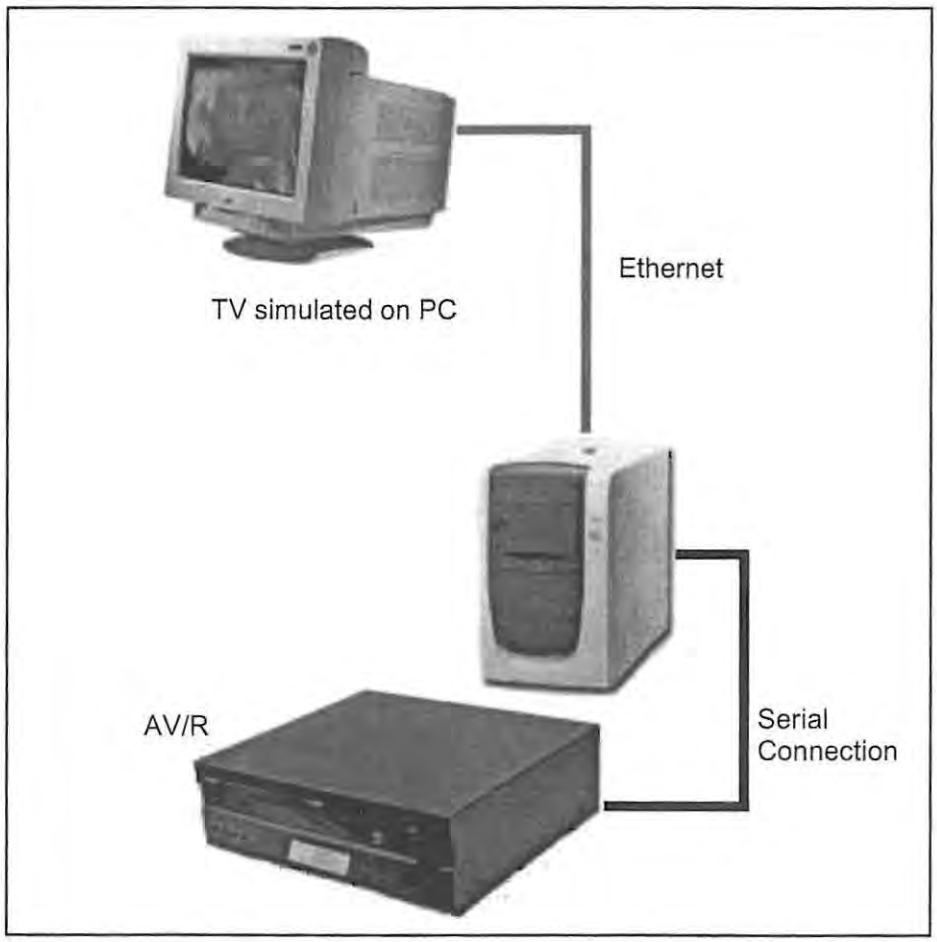

Figure 46 - The UPnP TV-AV/R system

For the UPnP presentation mechanism implementation, however, the AV/R needs to be able to communicate using IP and all the other protocols that UPnP uses. In order to simplify the system, a computer is used to host the AV/R. The computer communicates to the AV/R in the same way the DHIVA did in the AV/C Panel Subunit implementation – via a serial connection. The computer hosting the AV/R is connected to the control point (a further computer simulating the TV) by Ethernet cable. The AV/R's host computer acts as a UPnP "file server", serving the description and presentation documents to the control point when requested to do so. The system is shown in Figure 46.

5.2 The UPnP TV-AV/R

The UPnP presentation mechanism hinges on the description and presentation documents that reside on the device offering services. The control point simply retrieves these documents from the device and can immediately issue remote configuration commands or subscribe to events pertaining to the state of the device.

Figure 47 shows the components of the UPnP "file server" that resides on the AV/R. The AV/R, apart from providing the description and presentation documents, needs only to implement a utility that is able to map from UPnP messages (specifically control commands, which include remote configuration commands) to native commands that the AV/R understands. This is shown in the "Device Dependent Function Block". The three documents that will be required by the control point (the TV) are shown in the center block, while the UPnP stack is shown in the right-most block.

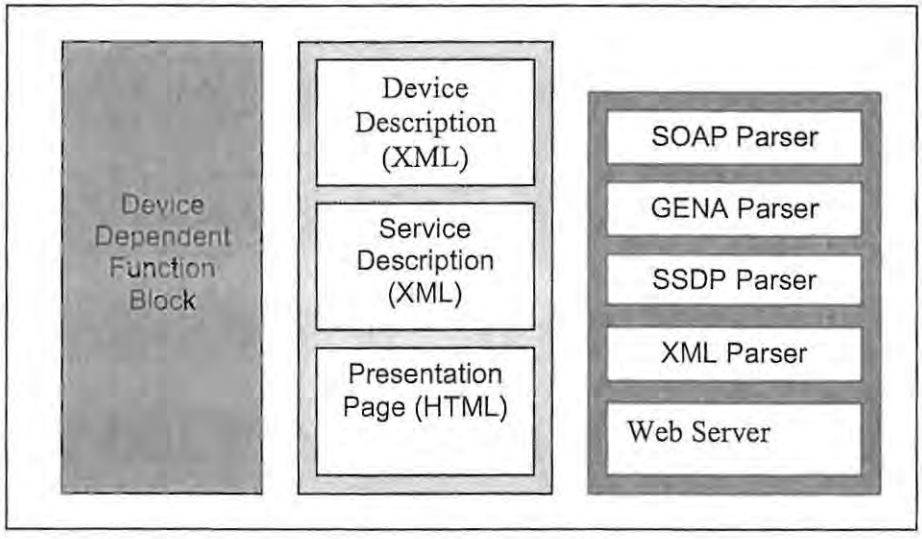

Figure 47 - Components of a UPnP Server

It was decided that implementing an entirely new UPnP stack was unnecessary, since Microsoft have already implemented such a stack in their UPnP Development Kit. The Microsoft UPnP Development Kit (available online at http://www.microsoft.com/hwdev/UPnP/) is used to implement the UPnP presentation mechanism TV-AV/R system, since it includes the Microsoft UPnP stack. The Development Kit is implemented in Visual C++. The source files are divided into modules that implement various UPnP functions. Figure 48 [36] shows the modules that are supplied with the Development Kit. The module marked "UDevice" contains device specific function prototypes. Each UPnP command is mapped to a function in this module. The functions reside in the module marked "Device". The "Device" module's functions match the device and service description that reside on the AV/R and include state variables for the device. These two modules form the "Device Dependant function block" of Figure 47.

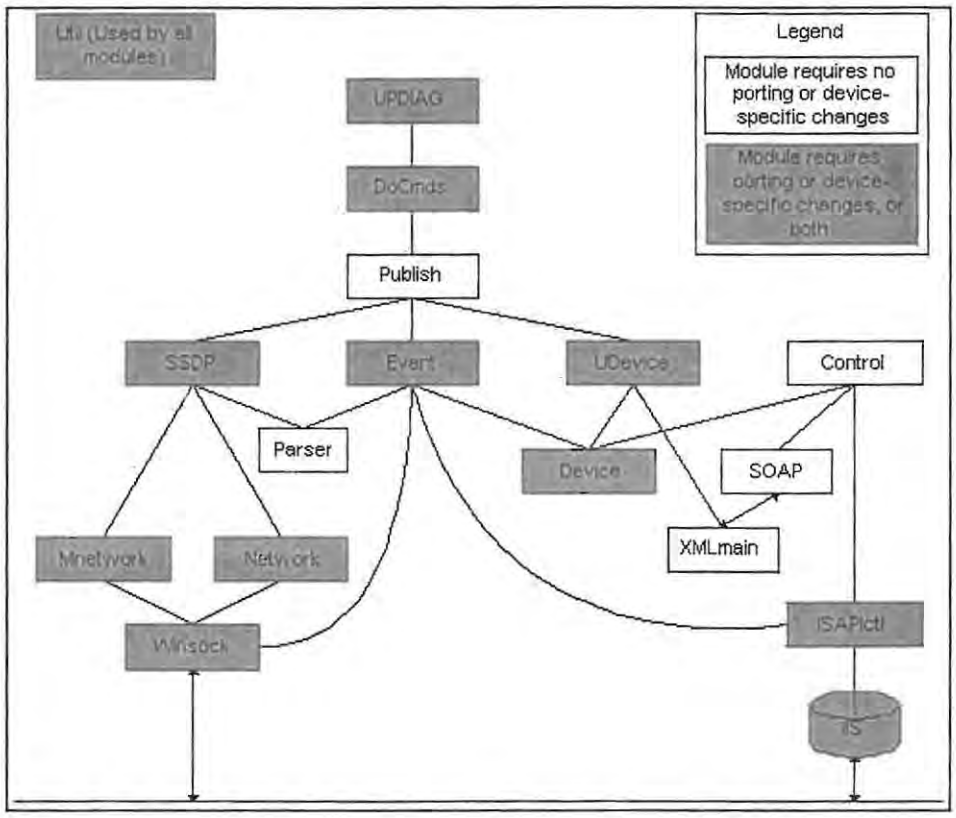

Figure 48 - The organisation of UPnP modules

5.2.1 UPnP Modules

Figure 48 shows the Microsoft UPnP Development Kit modules as they appear in the AV/R. All UPnP messages for Eventing and Controlling (see § 2.4.4 for more definitions of these terms) are processed through the Internet Information Server (IIS). The interface to the IIS is achieved

through Internet Services Application Programming Interfaces, or ISAPI. ISAPI allows web-services to be written. In the case of UPnP, the web-server on the AV/R must be able to serve the UPnP documents to the control point and receive Control and Eventing messages from the control point. The ISAPIctl module is the UPnP interface onto these APIs, and so essentially provides the interface between UPDIAG and ISAPI. Control provides services that allow a device to be manipulated. The Winsock, Network and Mnetwork modules are used to publish information about the device on the network. The Parser module allows UPDIAG to parse SSDP (Simple Service Discovery Protocol) messages, and the SSDP module is used to send the relevant SSDP messages that are required for the device and its services to remain exposed on the network. The XMLmain module is for parsing XML packets. The SOAP (Simple Object access Protocol) module handles all SOAP messages, while the Event module handles subscription to events and event notifications. The publish module is responsible for publishing the services of the device using SSDP. The UDevice module contains prototypes for all the native functions contained in the Device Module, and UDevice is responsible for reading the device and description files. The Device module contains all the native services that are published by the device. DoCmds does the actual simulation of the device, while UPDIAG handles the device initialization (it is the main program).

The Development Kit provides Visual C++ code for all the modules. Some of the modules need to be ported if the system is implemented on a non-Windows platform, but since Windows is the platform used for the implementation, no porting is necessary. Because all of the code is supplied, only minor modifications need to be made in order to implement the system for the TV-AV/R - the following must be done on the AV/R:

- provide a device description document
- provide a service description document
- provide a presentation document
- modify Device module
- modify UDevice module

Each device on the UPnP network that wishes to be remotely configured using the UPnP presentation mechanism must contain three documents. The description document is an XML document detailing features of the device, such as its name, model number and manufacturer. Also embedded in this document are the URLs of the other two documents. The service description document describes the services that are available on this device and how they are used. This document is also called the Service Control Protocol Document, or SCPD. The final document is the presentation document, which is an HTML document that specifies the user

interface to the device's services. This document uses VBScript to perform UPnP functions, and is detailed later in this chapter.

Apart from these documents, which are specific to each device, two Development Kit modules must be specific to each device. These two are the UDevice module and the Device module. Broadly, these two modules are responsible for mapping UPnP messages to native function calls and the actual native functions themselves respectively. All other modules are generalized and do not need to be modified for each device. The UDevice and Device modules are detailed later in this chapter.

The control point (the TV) needs only a browser capable of sending and receiving HTTP, SOAP, GENA and SSDP messages. The Windows Millennium operating system supports SOAP, GENA and SSDP. Microsoft's Internet Explorer uses HTTP to obtain the documents and VBScript to process SOAP, GENA and SSDP messages. Hence, since a computer is being used to simulate the TV, this Windows ME and Internet Explorer are used to implement the control point.

5.2.2 The UPnP Presentation Mechanism Process

In order for the system to work correctly, a specific sequence of operations performed between the control point (the TV) and the device (the AV/R) must be followed. Figure 49 shows the sequence of interactions diagrammatically. Firstly, the AV/R (the device offereing services) broadcasts its presence on the network (this step is not shown on the diagram). This advertisement is an SSDP message and contains a URL that points to the device's device description document. The control point then requests this XML document from the AV/R. This document is sent using HyperText Transfer Protocol (HTTP) over the network, as are all the documents sent from the AV/R to the control point. From the device description the control point obtains the URL of the service description document and requests this document from the AV/R. The control point then subscribes to any events it wishes to (events are services or action) using Simple Service Discovery Protocol (SSDP). The control point then requests the AV/R's HyperText Markup Language (HTML) presentation page from the AV/R and displays it to the user in a standard browser. Any user control or configuration commands are then transmitted from the control point to the AV/R using Simple Object Access Protocol (SOAP). The commands are parsed (using the XMLMain module) and passed to the device dependent function block (the portion of the Development Kit that includes the UDevice and Device modules), which carries out the command on the AV/R. Any events that occur on the AV/R are

transmitted from the AV/R to the control point using General Event Notification Architecture (GENA). The presentation page updates as necessary to reflect these events.

The three documents (the device description document, the service description document and the presentation document), as well as the UDevice and Device modules are detailed below.

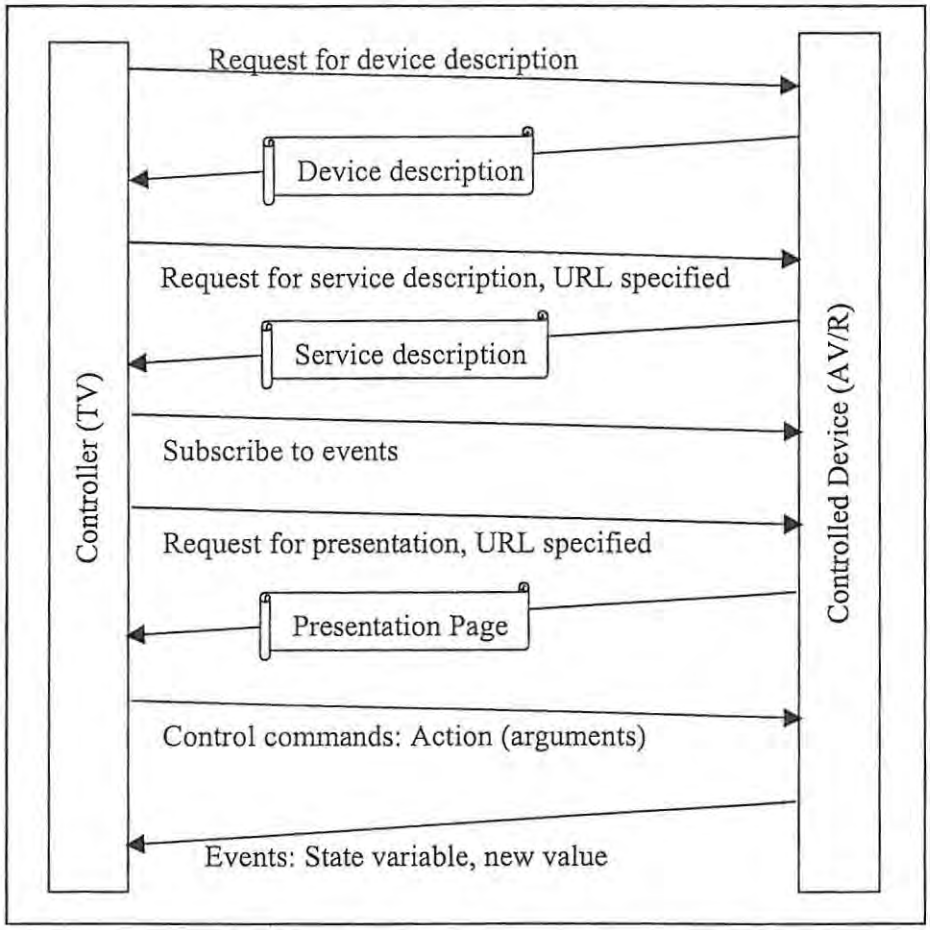

Figure 49 - Interactions between the control point (TV) and device offering services (AV/R) during a UPnP Remote Configuration session

5.2.3 The Device Description Document

The device description document is an XML document that details the device offering services itself. It contains information about the device name, model, make and manufacturer and also contains a globally unique id for the device. It contains an optional icon for the device. Importantly, this document contains URLs to all the other documents required by a control point in order to control or configure this device. Lines 29 to 38 of Listing 2 show these URLs.

The device description document is formatted according to the layout that is specified by the UPnP specification [11]. In the specification, an XML device description document is specified with placeholders written in italics. In order to create a device description document, this layout is copied and the italics placeholders are replaced by the device-specific attributes.

Line 36 tells the control point where the Service Control Protocol Document (SCPD) resides. The SCPD describes all the services that the device offers and allows the control point to subscribe to any events it wishes to.

5.2.4 The Service Description Document

The service description document is formally called the Service Control Protocol Document (SCPD). The SCPD for the AV/R is shown in Appendix D, and like the device description document is created from a template in the UPnP specification [11].

```
1.  <?xml version="1.0"?>
2.  <root xmlns="urn:schemas-upnp-org:device-1-0">
3.    <specVersion>
4.      <major>1</major>
5.      <minor>0</minor>
6.    </specVersion>
7.    <device>
8.      <UDN>uuid:8847be4e-72de-4e20-86c0-1dcbbe0f980b</UDN>
9.      <friendlyName>Yamaha AV/R</friendlyName>
10.     <deviceType>urn:schemas-upnp-org:device:avrconfig:1</deviceType>
11.     <presentationURL>../presentation/rxv1000.html</presentationURL>
12.     <manufacturer>Yamaha</manufacturer>
13.     <manufacturerURL>http://www.yamaha.com/</manufacturerURL>
14.     <modelName>RXV</modelName>
15.     <modelNumber>1000</modelNumber>
16.     <modelDescription>UPnP-RXV1000 Config Control</modelDescription>
17.     <modelURL>http://www.yamaha.com/</modelURL>
20.     <iconList>
21.           <icon>
22.          <mimetype>image/png</mimetype>
23.          <width>16</width>
24.          <height>16</height>
25.          <depth>2</depth>
```

```
26.          <url>../images/16-2.png</url>
27.        </icon>
28.      </iconList>
29.      <serviceList>
30.        <service>
31.          <serviceType>urn:schemas-upnp-org:service:avrconfig:1
32.                  </serviceType>
33.          <serviceId>urn:upnp-org:serviceId:avrconfig</serviceId>
34.          <controlURL>../control/isapictl.dll?avrconfig</controlURL>
35.          <eventSubURL>../control/isapictl.dll?avrconfig</eventSubURL>
36.          <SCPDURL>../SCPD/rxv1000scpd.xml</SCPDURL>
37.        </service>
38.      </serviceList>
39.    </device>
40. </root>
```

Listing 2 - The AV/R Device Description Document

Lines 9 – 58 show what services the device is exposing to the network. These services are actions may be performed on the device. These are the names of the UPnP commands that the control point will issue to the device when the user selects an action. When the control point (the TV) requests the SCPD from the device (the AV/R), this document is sent to the control point. The control point now knows what services the device is offering. The remainder of the file documents the state table of the services – ranges, allowed values and current values of any state variables the device wishes to publish.

The AV/R has a number of input options available. One of the remote configuration options is to allow the user to select which input is desired. The following lines of code show the state variable, called Input, is specified in the SCPD:

```
63.      <stateVariable sendEvents="yes">
64.        <name>Input</name>
65.        <dataType>string</dataType>
66.          <allowedValueList>
67.            <allowedValue>Phono</allowedValue>
68.            <allowedValue>CD</allowedValue>
69.            <allowedValue>Tuner</allowedValue>
70.            <allowedValue>CD-R</allowedValue>
71.            <allowedValue>MD/Tape</allowedValue>
72.            <allowedValue>DVD</allowedValue>
73.            <allowedValue>D-TV/LD</allowedValue>
```

```
74.            <allowedValue>CABLE/SAT</allowedValue>
75.            <allowedValue>VCR1</allowedValue>
76.            <allowedValue>V-Aux</allowedValue>
77.            </allowedValueList>
78.          <defaultValue>CD</defaultValue>
79.        </stateVariable>
```

Any time a state variable is changed by the control point (or physically on the device itself), it will generate an event to notify any interested parties (control points that have subscribed for this event notification) that it has changed. The ability for a state variable change to signal an event is determined by the attribute "sendEvents" (line 63). If this attribute is set to "yes", then any change of the variable on the device generates an event. If it is set to "no", no events will be generated for this variable. The name of the variable is specified by <name> tags (line 64). The type is specified by <dataType> tags (line 65). In this case, the names of the input options are specified as strings, and so the data type is set to "string". A list of allowed values is then specified between the <allowedValueList> tags (lines 66 – 77). Each value is specified within <allowedValue> tags. Finally, a default value is set using <defaultValue> tags (line 78). This element corresponds to a Scroller element in the Panel Subunit.

In order to manipulate this state variable and change it (as well as the physical input selected on the AV/R), the control point must be able to issue a command to change the value of the input. To do so, an action that allows manipulation of the Input state variable must be specified. The following code shows this action specification:

```
10.        <action>
11.          <name>Next_String_BoundedInput</name>
12.        </action>
```

The action appears between <action> tags. Its name is Next_String_BoundedInput. The action selects the next string of the Input variable, checking that is never goes beyond the number of allowed values for that state (it is a bounded increment). A corresponding Previous_String_BoundedInput action exists to allow the user to select the previous string.

5.2.5 The Presentation Document

The Device description document and SCPD describe the device and its services to the control point. At this stage, the control point (the TV) knows about the device offering services (the AV/R) and what services it is offering from the device and service description documents respectively. The control point must now present an interface to the user to allow the user to

make us of the services on the AV/R. In order to do this, the control must retrieve the presentation document from the AV/R. This document is an HTML file and is displayed in a browser on the TV for the user to see.

The presentation page is written in HTML and uses Visual Basic Script (VBScript) to implement the functions and handlers necessary for calling UPnP remote functions (the services that the AV/R has exposed). The document is divided into two sections – the standard HTML to display buttons and text to the user, and the VBScript section. The VBScript section is responsible for loading the device description and service descriptions and for mapping the user's actions in the browser to UPnP commands sent to the AV/R, as well as for updating the display as events that change the values of any on-screen state variables are received. The device and service descriptions must be loaded for the control point to be able to issue commands to the AV/R, since the loaded device contains the references to the remote functions.

5.2.6 The Presentation Process

The control point must display an interface to the AV/R to the user. To do so, it must retrieve all the information concerning the device and its services (contained within the device and service description documents) and display the presentation page to the user. The presentation page makes use of the device and service descriptions, and so it is vital that the control point can reference these documents during presentation. A listing of the presentation page used for the UPnP TV-AV/R system is shown in Appendix E. Figure 14 (page 62) shows what the presentation page for the UPnP TV looks like.

In order to load the device (that is, to create an internal logical representation of the device), the control point creates a reference to the device description document as shown in the following lines of code:

```
200. Dim AvrDesc
201. Set AvrDesc = CreateObject("UPnP.DescriptionDocument.1")
202. AvrDesc.Load("../description/rxv1000.xml")
```

The variable AvrDesc now is a reference to the device description. The CreateObject function creates a blank structure representing a UPnP device. A method called "Load" is defined that allows the device description document specified in the argument to be parsed and populate the blank structure. This object is defined in the UPnP DLLs (Dynamic Link Libraries) that link into Internet Explorer (the browser being used to display the presentation page). These DLLs come with the Microsoft Millennium operating system.

Next, the controller creates a reference to the root device of the AV/R's description. Device descriptions may be nested hierarchically (in a similar idea to the subunits of the Panel Subunit - see § 2.4.5.3) – and the root device will contain references to further sub-devices (if they exist). Thereafter, this reference makes attributes of the device, such as its name (AVRDevice.FriendlyName) and its type (AVRDevice.Type) available to the browser. The following lines of code show how a further UPnP DLL method, "RootDevice", is used to do this:

```
208. Dim AvrDevice
209. Set AvrDevice = AvrDesc.RootDevice
```

In order for the control point to map the user's actions to UPnP commands, the event handler that handles events generated by the user (such as clicking a button) must be linked to the services that the AV/R is advertising in its service description. The following lines of code show how the browser does this using the UPnP DLL function "Services" (used to create a reference to the services of the AV/R) and "AddCallBack" (used to add a callback for events). The "GetRef" function is used to obtain a handle to the event handler.

```
228. Dim AvrControlService
229. set AvrControlService=AvrDevice.Services("urn:upnp-
     org:serviceId:avrconfig")
230. AvrControlService.AddCallback GetRef("eventHandler")
```

At this point, the presentation page is initialized and can be displayed to the user. Now the first section of the presentation page comes to the fore, where HTML is used to specify functions that are used to invoke UPnP actions. For example, a button may be mapped to a function (the function is written in VBScript) so that when a user clicks the button, the function is called. The following code shows how a button is specified in HTML. The button is labeled "Next" and appears under a label with the caption "Input" – in other words, the button, when pressed, selects the next input option of the AV/R. The button, when pressed, invokes a function called "NextInput":

```
23. <INPUT type="button" onclick="NextInput()" value="Next">
```

Below, the function "NextInput" is shown:

```
124. function NextInput()
125.     Dim inArgs(0)
126.     Dim outArgs(0)
127. AvrControlService.InvokeAction "Next_String_BoundedInput", inArgs,
     outArgs
```

```
128. end function
```

The action invoked by this function is "Next_String_BoundedInput" and maps exactly to an action that is specified in the SCPD file of the AV/R (line 11 of Appendix D – UPnP AV/R SCPD). The function call has no input or output arguments.

When state variables of the AV/R change, the events are sent to the TV. A call-back function is specified in lines 97 – 116 to handle these event messages. The code for this function is contains the following case statement:

```
97.  Sub eventHandler(callbackType, svcObj, varName, value)
98.  'the intervening lines of code are commented out
105.   If (callbackType = "VARIABLE_UPDATE") Then
106.     select case varName
107.           Case "Input"          Input.innerText = value
108.           Case "SoundMode"      SoundMode.innerText = value
109.           Case "SoundField"     SoundField.innerText = value
110.           Case "Sleep"          Sleep.innerText = value
111.           Case "SpeakerA"       SpeakerA.innerText = value
112            Case "SpeakerB"       SpeakerB.innerText = value
113.           Case "Volume"         MainVolume.innerText = value
114.     end select
115.   End If
116. End Sub
```

The event handler accepts four arguments: the call-back type (the only one that the control point is interested in as far as UPnP goes is "Variable_Update" as seen in line 105), the service object, the variable and the value. The service object is of no interest here. The case statement (lines 106 – 114) assigns value to the on-screen value that corresponds to the variable name. In this manner, all the on-screen variables are always kept up to date.

5.2.7 Modifying the Device and UDevice Modules

The device and service description documents provide the control point (the TV) with enough information to call the correct remote function corresponding to a user's interaction with the presentation page. For instance, by pressing the "Next Input" button presented on the TV, the user communicates the intention to change the currently selected input to the next input option. The control point must now convey this command to the AV/R in order for it to be carried out. The presentation page provides the map from the on-screen user actions to UPnP control commands. The AV/R must now map the UPnP command to a native command that will

actually perform the operation requested by the user (in this case, select the next input option on the AV/R). To do this, the AV/R uses two of the Microsoft UPnP Development Kit's modules, namely the Device and UDevice modules (shown in Appendices F and G respectively).

The "Device" module is where the native, device-specific function calls reside. This module is specific for each UPnP device. For the case of the AV/R in the TV-AV/R system, the Device module contains several functions that simply issue serial commands along the serial connection to the AV/R. These serial commands will perform the configuration operations the user specifies. The map from UPnP commands to these native functions resides in the UDevice module, and so the UDevice module must also be different for each UPnP device. The Device module also holds a structure that holds the AV/R's current state.

There are very close relationships between the Device and UDevice modules and the service description document of the AV/R. The state variables and actions that appear in the service description document are exactly mapped to internal state variables and functions in the Device module. The UDevice module contains prototypes of the functions that appear in the Device module and a map from UPnP actions (as specified in the service document) to these prototypes.

UPnP commands from the TV come to the AV/R in the form of Simple Object Access Protocol (SOAP) commands. These are formatted using XML tags and when they are recognized as SOAP commands by the ISAPIctl module, the commands are passed to the SOAP module. The SOAP module then parses the command (with help from the XMLMain module) and hands the commands to the Control module. The Control module then passes the commands to the UDevice module which maps the SOAP command to a native call. The native call is then made using the Device module and the command is carried out. Any changes that occur to the state variables of the AV/R are then broadcast to interested parties (control points which have subscribed to event notification corresponding to changes in the state variables involved) on the network.

The structure that the Device module uses to hold the state of the AV/R is shown in the following lines of code (from Appendix F):

```
22. struct AVRConfig
23. {
24.    DWORD Input;
25.    DWORD SoundMode;
26.    DWORD SoundField;
```

```
27.    DWORD Volume;
28.    DWORD Sleep;
29.    BOOLEAN      SpeakerA;
30.    BOOLEAN      SpeakerB;
31. };
```

DWords are double words and are defined as 16-bit integers. The variables that the AV/R's state is modeled with are its Input, Sound Mode, Sound Field, Volume and current sleep setting. Two Booleans are used to model the on and off state of speaker relays A and B.

This structure is instantiated to a variable called "InsAVRConfig" and sets the variables to initial values using the following code, setting the initial state of the device to Input 0 (an index to the string name of the input option), SoundMode and SoundField 0, Volume 50, Sleep 0 and SpeakerA and SpeakerB both to 1 (on):

```
35. AVRConfig InsAVRConfig =
36. {
37.    0,
38.    0,
39.    0,
40.    50,
41.    0,
42.    1,
43.    1
44. };
```

The native functions appear in the Device module. Each function has a corresponding action specified in the service description document. Each of the functions performs a range check on the variable being manipulated before proceeding. If the new value is within specified boundaries, the new value is set as the current value. The AV/R then returns the results of the action (if any) to the control point. The AV/R also broadcasts a message to the entire network informing all interested control points (those control points that have registered for such events) that the state variable of the AV/R has changed.

For example, consider again the function that selects the next input of the AV/R:

```
231. DWORD Do_Next_String_BoundedInput (
232.                          CHAR*          StrEventUrl,
233.                          DWORD          cArgs,
234.                          ARG*           rgArgs,
```

```
235.                            PDWORD          pArgsOut,
236.                            ARG_OUT*        rgArgsOut
237.                           )
238. {
239.  if (InsAVRConfig.Input < MAX_INPUT)
240.  {
241.        InsAVRConfig.Input++;
242.        CHAR szValue[32];
243.        sprintf(szValue, "%u", InsAVRConfig.Input);
244.        ChangeProp(StrEventUrl, "Input", szValue);
245.        SerialCommand ("Input", szValue);
246.        SubmitPropEvents(StrEventUrl, NULL);
247.  }
248.  return 0;
249. }
```

The function is named "Do_Next_String_BoundedInput" and takes several parameters. StrEventURL is the URL of the device's event source, which is used to generate event notifications. The remainder of the parameters passed to this function are used to access the input and output arguments of this function's corresponding UPnP command, but since none of the functions used for the AV/R have any incoming or outgoing arguments, these parameters are not used. The functions are all alike in structure: range checking is performed (line 239) after which the state pertinent state variable is updated (in this case, the input index is incremented). szValue is used to convert the state variable value to a string which is used by the flowing three functions (line 243). The ChangeProp function causes the state variable's new value to be placed ready for event notification in the Event module. The SerialCommand function then performs the serial command output on the serial port to the AV/R which causes the user's action to be carried out physically on the AV/R (in this case, the next Input option is selected). The SubmitPropEvents function then posts the event notification informing the network of the change of this state variable's value. This function produces a UPnP packet, which is sent to the control point (assuming that the control point has registered for this event notification, which in the TV-AV/R's case it has) and the TV is then able to update its display.

This function (like all the other native functions) is mapped to a UPnP command in a static structure in the UDevice module (called c_rgSvc, line 111 of Appendix G). Each mapping is a tuple of the UPnP command and its corresponding native function call in the Device module and looks as follows:

```
124. { "Next_String_BoundedInput", (PFNAS)Do_Next_String_BoundedInput },
```

The UPnP command is "Next_String_BoundedInput" and this name appears in the service description (line 11 of Appendix D) and in the presentation page (line 127 of appendix E). This shows the close relationship between the presentation page, the service description document and the UDevice module. The native function that this maps to is "Do_Next_String_BoundedInput" which is the name of the function in the Device module. In this manner, all the UPnP commands are mapped to native function calls.

5.3 Summary

This chapter discusses how UPnP presentation mechanism may be used to implement remote configuration for the same TV-AV/R system used in Chapter 4. A computer is used to host the AV/R on an Ethernet network, making use of IP, the underlying protocol present in UPnP. The computer communicates to the AV/R in the same way the DHIVA did in the AV/C Panel Subunit implementation – via a serial connection. The computer hosting the AV/R is connected to the control point (a further computer simulating the TV) by Ethernet cable. The AV/R's host computer acts as a "file server", serving the description and presentation documents to the control point when requested to do so.

The UPnP presentation mechanism implementation of the system hinges on the description and presentation documents that reside on the AV/R. The control point simply retrieves these documents from the device and can immediately issue remote configuration commands or subscribe to events pertaining to the state of the device.

It was decided that implementing an entirely new UPnP stack was unnecessary, since Microsoft have already implemented such a stack in their UPnP Development Kit. The Microsoft UPnP Development Kit is used to implement the UPnP TV-AV/R system, since it includes the Microsoft UPnP stack. The Development Kit is implemented in Visual C++. The source files are divided into modules that implement various UPnP functions. Most of these devices require no changes at all, with the exception of two modules, the Device and UDevice modules. The UDevice contains device specific function prototypes. Each UPnP command is mapped to a function in this module. The functions reside in the module marked "Device". The "Device" module's functions match service description actions that reside on the AV/R and include state variables for the device.

Because all of the code is supplied in the development kit, only minor modifications need to be made in order to implement the system for the TV-AV/R - the following must be done on the AV/R:

- provide a device description document
- provide a service description document
- provide a presentation document
- modify Device module
- modify UDevice module

Each device on the UPnP network that wishes to be remotely configured using the UPnP presentation mechanism must contain three documents. The description document is an XML document detailing features of the device, such as its name, model number and manufacturer. Also embedded in this document are the URLs of the other two documents required. The service description document describes the services that are available on this device and how they are used. The final document is the presentation document, which is an HTML document that specifies the user interface to the device's services. This document uses VBScript to perform UPnP functions.

The control point (the TV) needs only a browser capable of sending and receiving HTTP, SOAP, GENA and SSDP messages. The Windows Millennium operating system supports SOAP, GENA and SSDP. Microsoft's Internet Explorer uses HTTP to obtain the documents and VBScript to process SOAP, GENA and SSDP messages. Hence, since a computer is being used to simulate the TV, this Windows ME and Internet Explorer are used to implement the control point.

In order to manipulate the state variables (as well as perform the physical operations required by the user on the AV/R), the control point must be able to issue a command to change the value of any state variable. Actions that allow manipulation of the AV/R's state variable are specified in the service description document. These actions are used by the presentation page (which is the user interface displayed to the user on the TV) to perform the users commands.

There are very close relationships between the Device and UDevice modules and the service description document of the AV/R. The state variables and actions that appear in the service description document are exactly mapped to internal state variables and functions in the Device module. The UDevice module contains prototypes of the functions that appear in the Device

module and a map from UPnP actions (as specified in the service document) to these prototypes.

Each function not only corresponds to an action specified in the service description document, but also produces a UPnP packet when a state variable is altered by performing one of these actions. This packet is sent to the control point (assuming that the control point has registered for it). The TV is then able to update its display using this event notification.

Chapter 6 - Two Communication Models for Device Configuration

6.1 Introduction

Chapter 2 presents various home entertainment networking solutions – HAVi, Jini, AV/C and UPnP. Chapter 3 examines a small part of these home entertainment networking solutions – their remote configuration solutions. HAVi uses data Driven Interaction (DDI), Jini implements user adapters, AV/C uses the Panel Subunit and remote configuration can be done in a UPnP network by making use of the UPnP presentation mechanism. A comparison of these solutions is shown in section 3.5 This comparison revolves around the theoretical features of the different solutions. In other words, by simply examining the solution, a fair comparison can be drawn to show the differences in the approaches. Furthermore, the four solutions are divided into only three categories, since the HAVi and AV/C solutions are so similar.

Chapters 4 and 5 then show how two of these categories of remote configuration solution are implemented, using the AV/C Panel Subunit and UPnP presentation mechanism respectively. The availability of an AV/C stack and the UPnP Development Kit make these two solutions the easiest of the three categories to implement.

This chapter discusses in detail the features of remote configuration solutions described in this thesis, and contrasts them with the essential features of control solutions within home entertainment networks, arriving at two models of communication for configuration on home entertainment networks.

6.2 Control Solutions

Chapters 4 and 5 have all discussed remote configuration using one particular method of communication (the features of which are discussed later). However, both UPnP and AV/C have other methods of communicating in order to configure devices using control features inherent in the networking standard itself. For example, UPnP and AV/C's control features are inherent in UPnP's control mechanism (see § 2.4.4.8) and AV/C's subunit commands. These control features are discussed briefly here in order to allow a comparison of communication models.

UPnP's control mechanism and AV/C's subunit commands are very similar. Both start off with the controller device "knowing" about the services and capabilities that other devices on the network possess. They then use control commands to configure devices. Some method other than the presentation page and Panel Subunit for UPnP and AV/C respectively must be made for user interaction to be catered for.

For example, if an Audio Subunit is present on a device, a controller would know what AV/C commands to issue to the device in order to configure (and indeed operate) the device simply by knowing about an Audio Subunits as specified in the AV/C standard. In other words, the controller can configure any device *it already knows about*. If it did not know about a Disc subunit, it would not be able to configure it. The same applies to UPnP's control mechanism. UPnP device and service specifications exist (which are analogous to AV/C subunits) and control points can only configure devices (using control messages) that they already know about.

These methods could use user interfaces that map user input to the corresponding control message. In this way a controller could present a user interface for the device being configured to the user without having to obtain the interface from the device. The interface will reside in the controller and the controller will only have interfaces for the devices that it knows about.

6.3 Comparison Parameters

In order to make a comparison between the features of remote configuration and the features of control methods, there must be some properties or characteristics of the methods to compare. This section will highlight the features of the remote configuration communication model used by the systems on chapters 4 and 5 so that it may be compared and contrasted to the control communication model. It is in fact these differences that lead to the two theoretical models.

6.3.1 Device Knowledge

This feature refers to how much knowledge the home entertainment networking solution possesses of the workings and functionality of a device. If a solution knows nothing of the capabilities of a device, it will have to discover them at run time, and it may or may not be able to make use of all the device's features – especially if the device includes some new functionality that was unforeseen when the solution was defined. So new devices can be added "on-the-fly", but may not be able to operate to their full capability since the home entertainment networking

solution may not be able to facilitate other devices to interoperate with it correctly because it does not know enough about the new functionality.

For example, consider the AV/C home entertainment networking solution. Several device types are defined in the form of logical subunits. Each subunit specifies what properties the device has and what services the subunit offers. The Audio subunit [40] defines such operations as balance, surround types etc. The gamut of operations and services covered in the specification is fairly comprehensive, and so AV/C can be said to have a deep knowledge of the capabilities of the device. However, an audio device capable of some other new feature will not be able to easily use this new feature since it is not included in the subunit specification.

The UPnP presentation mechanism, however, will know nothing of the capabilities of devices on the network. Control points will have to discover the capabilities of the device dynamically. This means that the UPnP presentation mechanism can easily incorporate new devices and new device types. If the audio device with the added feature (from the example above) is connected to the network, the UPnP presentation mechanism allows control points to simply discover the added feature and it becomes immediately available to the network.

The implication of the home entertainment networking solution knowing of the capabilities of a device becomes apparent when considering configurability. If the solution is aware of the capabilities of the devices on its network, it can guarantee configuration only of those features which it knows about. However, if the capabilities of a device must be discovered dynamically, the controller has the ability to configure features of the device that are unknown to it at first.

For example, consider a CD-player on a home entertainment network. An AV/C controller (without Panel Subunit capabilities) would only be able to configure the CD-player if its logical abstraction (the Disc subunit) is programmed into the controller (i.e. the Disc subunit commands would need to be inside the controller). The controller can guarantee that it can configure the CD-player exactly to the specifications of the Disc subunit. However, what would happen if a new feature was added to the CD-player that was not in the Disc subunit specification? The same controller would not be able to configure the new feature since it does not know of the feature or any command that can configure that feature. However, a controller with Panel Subunit capabilities would be able to access and configure the new feature (even though it knows nothing of the feature beforehand) since it has to discover the features of the CD-player dynamically.

6.3.2 Location of User Interfaces

Continuing with the above CD-player example, consider the difference between a controller that has Disc subunit capabilities and a controller that has Panel subunit capabilities. The first controller would have a Disc user interface built into it that it can display to the user, while the second controller would need to retrieve and build up a user interface to the CD-player from the CD-player itself.

This presents manufacturers with a trade-off – do they manufacture controllers that are capable of discovering devices and services dynamically, having to build up user interfaces as they are retrieved from the devices being configured (which is a complicated task) or build controllers that know only about certain other devices but have predefined user interfaces built into them? Do they manufacture devices that are going to be configured with or without their own user interfaces?

6.3.3 Command Sets

Controllers that know of devices beforehand know the commands they can issue to those devices in order to configure them. For example, a CD-player is abstracted by the Disc subunit in AV/C and may have device and service specifications in UPnP. Thus only those commands that are documented in these abstractions can be issued from the controller to the device.

On the other hand, a device could contain a Panel Subunit or presentation document (for AV/C Panel Subunit and UPnP presentation mechanism respectively) that would specify to the controller which commands it is able to receive from the controller. Thus commands to configure the device are provided from the device dynamically and are not limited to any documented command set.

6.4 Two Models

The features that have been compared lead to two models of communication for device configuration. Both the AV/C Panel Subunit and UPnP presentation mechanism hinge upon the control point obtaining device and service descriptions and user interfaces from devices that it controls. Other control mechanisms, however, have device and service descriptions (and possibly user interfaces) programmed into them when they are manufactured – they can only access the devices that they already know about. In other words, control points that have very little information about the devices and services utilize a model of communication that is dynamic

(control points discover devices, services and user interfaces "on the fly" and can access any device that presents this information) while control points that are pre-programmed with information about the devices and services they will interact with are static in nature (control points can only access devices they are programmed to). The first model is called the Rendering Model since the control point acts simply as a "remote" control surface for the user – it retrieves a user interface from the device and renders it for the user, relaying the user's actions back to the device. The second model is called the Programmed Model since it is pre-programmed with the knowledge and interfaces it requires in order to allow users to interact with the devices offering services on the network. The differences between the two models are shown in Table 4.

	Rendering Model	Programmed Model
	- Little to none - Discovered "on the fly"	- Substantial for certain devices - Preprogrammed
Location of User Interfaces	On devices offering services	On controller
	Dynamic – depends on information provided by device offering services	Static – only commands programmed into controller can be used

Table 4 - Comparison of Rendering and Programmed Communication Models

6.5 Consultant vs. Postman

In order to more fully understand the differences between the two models, an analogy is useful. The analogy involves a consultant and a postman – analogous to Programmed and Rendering models respectively.

6.5.1 The Consultant

In the Programmed model, the control point acts like a consultant to the user – it knows about the device that the user is trying to configure and displays its own interface to that device to the user. When the user performs an action, it draws on its own knowledge of the device in order to issue the correct corresponding configuration command to the device.

6.5.2 The Postman

In the Rendering model, the control point acts like a postman between the device and the user. The device provides the control point (postman) with its user interface and the commands that it

can respond to. The control point then renders the interface and waits for the user's actions. It then delivers the user's actions to the device.

6.6 Advantages and Disadvantages of the Models

Manufacturers need to know exactly how much work is involved in implementing a device configuration solution. Perhaps flexibility is more important to the manufacturer than predictability, or perhaps backward compatibility is more important to the manufacturer than the ability to quickly add new devices. The two models have distinct advantages and disadvantages and as such provide the manufacturer with some sort of criteria to be able to select one particular device configuration solution over another.

6.6.1 Amount of Programming

The manufacturer must be able to ascertain how much of a particular remote configuration solution has to be implemented each time a device is manufactured. Manufacturers using the Programmed model will have to manufacture control points that know about particular devices only – and the control points will have to be programmed with device and user specifications and user interfaces for the devices they wish to control. Manufacturers using the Rendering model have only to program their control points with enough intelligence to dynamically discover device and service descriptions and user interfaces of the devices to be controlled. However, the devices to be controlled must now themselves be programmed with their device and service specifications and user interfaces.

6.6.2 Addition of New Device Types

New devices are being invented frequently. The ability of a home entertainment networking solution to be able to work on these new device types (with their new functions) is an important consideration for manufacturers. Rendering model solutions, because they simply communicate between devices and know very little about the devices, make adding new device types easier. Programmed model solutions, because they utilise knowledge of the device and its functions, can make addition of new device types complex.

6.6.3 Backward Compatibility

Manufacturers need also to consider the issue of backward compatibility. Will newer devices with newer versions of the home entertainment networking solution be able to communicate with devices that implement older versions of that solution? Rendering model solutions, again because they know so little of the devices, make backward compatibility easier.

6.6.4 Complexity

The complexity of a home entertainment networking solution affects the time it takes to implement it. This is an important concern for manufacturers. Rendering model solutions are usually more complex since they need to create user interfaces at run time and issue commands they have only discovered at run time. Programmed solutions use pre-programmed interfaces and commands and as such these solutions are usually easier to create, implement and debug.

6.6.5 Flexibility and Functionality

Rendering model solutions are more flexible than Programmed solutions since the control points discover the services and interfaces on the network on the fly. However, this flexibility does not necessarily mean that all the functionality of a device may be exploited. This is due to the fact that the configuration solution itself is unaware of the capabilities of the devices offering service, and as such may not be able to cater for the device's functionality (by not being able to allow other devices to interoperate with it). For example, consider a TV, a DVD-player and an AV/R on a home entertainment network. If the TV were using a Rendering model configuration solution, it would need to discover the capabilities of the DVD-player and AV/R on the fly. Say the DVD-player needs to establish a connection on the network in order to stream audio from itself to the AV/R. The command to do this would be provided by the device to the TV (the control point), but the TV may not know enough about the AV/R to establish the connection. On the other hand, if the TV were using a Programmed configuration solution, it would need to know about the DVD-player and AV/R beforehand, and as such is more likely to be able to perform complex operations (such as establishing a connection).

6.6.6 Cost

Cost is an important consideration for manufacturers – how much functionality and interoperability they allow their devices to have depends on the cost of the networking hardware and firmware they decide to place into their devices. Rendering model solutions are more difficult to create and will usually take longer to create. Therefore they will end up costing the manufacturer more.

6.7 Summary

This chapter discusses the implementation differences between the AV/C Panel Subunit and UPnP presentation mechanism with other device configurations solutions in terms of:

- The amount of knowledge a control point must be programmed with when it is manufactured

- The location of user Interfaces (do they reside in the controller or in the device?)
- Where command sets come from (from the device or are they pre-programmed?)

From these differences, two conceptual models are proposed – the Programmed model and the Rendering model. The Programmed model requires the control point to have knowledge about the devices it is going to control programmed into it, while the Rendering model allows devices to provide configuration information to the control points on the fly. The analogy of the consultant vs. postman is used to highlight the differences between the two models.

The advantages and disadvantages of the models are discussed according to

- the amount of programming required
- the ease of the addition of new devices types
- backward compatibility
- complexity
- flexibility and functionality
- cost

Chapter 7 - Conclusion

7.1 Home Entertainment Networking Solutions

This thesis has examined home entertainment networking from several important angles. Firstly, economically, home entertainment networking is a growing market, and much research and innovation is going to be put into this industry in the coming decade. As such, many home entertainment networking solutions exist in order to connect the plethora of intelligent home devices that exist. Manufacturers may be overwhelmed by the amount of options available, and this thesis examines some of the more popular solutions in order to provide a framework for deciding on the best solution for a given manufacturer.

This thesis also discusses the features of four primary home entertainment networking solutions – HAVi, Jini, AV/C and UPnP and how remote configuration of devices is performed in each of these standards.

7.2 Remote Configuration

Secondly, this thesis provides two implementations of a small sub function of home entertainment networking solutions – remote configuration. Remote configuration is defined as "the ability of a home entertainment networking solution to allow one device to change the configuration state of another device". It has compared various home entertainment networking solutions as to their approach to remote configuration.

HAVi makes use of Data Driven Interaction (DDI). Jini implements remote configuration using ServiceUI objects. AV/C makes use of the Panel Subunit. UPnP makes use of its presentation mechanism.

The striking similarities between AV/C Panel Subunit and HAVi's DDI is noted, and the main differences between the four methods of implemented remote configuration are highlighted in terms of User Interface type, controller type, inter-device relationship, the coupling of functionality and user interface and specification of the target.

7.3 Contributions to Device Manufacturers

In order to explore fully the challenges of device configuration on home networks, this thesis shows two implementations of remote configuration for a TV-AV/R (Audio/Video Receiver) system. The first uses the AV/C Panel Subunit and the second makes use of UPnP's presentation mechanism.

While the AV/C implementation itself is a contribution to the AV/R manufacturer, several tools are also created along the way that will allow rapid development of such a system for other devices from different manufacturers.

A Graphical User Interface (GUI) "language" is defined that allows manufacturers to abstract the GUI of the device. This "language" is specified in XML, a widely adopted meta-language. This XML allows a manufacturer to specify exactly what user interface elements exist for the UI, where they are placed, what attributes they have and how they relate to one another. Furthermore, since not all manufacturers can readily create XML files by hand, a WYSIWYG GUIBuilder program is created that allows a manufacturer to graphically lay out the GUI and get the program itself to generate the corresponding XML. This improves the speed and efficiency of the GUI creation process considerably for the manufacturer.

A parser is then built that can parse the XML GUI description and store the GUI on the target device. Implementing an AV/C Panel Subunit function block for the Digital Harmony Protocol Stack embedded in the device allows the device to interact with a controller. Any device that now implements the same stack is AV/C Panel Subunit compliant.

A ControllerApp is provided that allows a PC to simulate a TV and remotely configure the AV/R on an IEEE1394 home entertainment network.

Lastly, a working UPnP presentation mechanism implementation for remote configuration is provided. This implementation allows a PC to host the AV/R on a UPnP network. The modifications to the Microsoft UPnP Development Kit modules and necessary UPnP documents are provided.

7.4 Contributions to the Home Entertainment Networking Field

The most significant contribution of this thesis to the field of home entertainment networking is the creation of an environment for the easy generation of remote configuration solutions by using the functionality of remote configuration standards. This environment allows manufacturers to create control points with a method of configuring device parameters that are not able to be configured using other control mechanisms.

Also, two models are derived by considering the differences between the features of remote configuration and the features of inherent control mechanisms – the Programmed model and the Rendering model. These models are based on differences that include the amount of programming required on controllers and on devices, the location of user interfaces (do they reside on the controllers or devices?) and the size and positioning of command sets.

Each model has advantages and disadvantages in terms of the amount of programming required, the ease of the addition of new devices types, backward compatibility, complexity, flexibility and functionality and cost.

By placing a device configuration solution into one of the two models, the inherent strengths and weaknesses of the solution can immediately be gauged.

Appendix A – XML DTD

```
1  <!ELEMENT PanelSubunit (Panel+)>
2  <!ATTLIST   PanelSubunit
3        name          CDATA #REQUIRED >
4  <!ELEMENT Panel (Scroller | Slider | Link | Label)+>
5  <!ATTLIST   Panel
6        name          CDATA #REQUIRED
7        caption       CDATA #REQUIRED >
8  <!ELEMENT Slider (SliderMin, SliderMax, SliderStep)>
9  <!ATTLIST   Slider
10       caption       CDATA #REQUIRED
11       position      CDATA #REQUIRED >
12
13 <!ELEMENT Scroller (ScrollVal+)>
14 <!ATTLIST   Scroller
15       caption       CDATA #REQUIRED
16       position      CDATA #REQUIRED >
17
18 <!ELEMENT Label EMPTY>
19 <!ATTLIST   Label
20       caption       CDATA #REQUIRED
21       position      CDATA #REQUIRED >
22
23  <!ELEMENT Link EMPTY>
24
25  <!ATTLIST Link
26       caption       CDATA #REQUIRED
27       position      CDATA #REQUIRED
28       linkto        CDATA #IMPLIED >
29
30 <!ELEMENT SliderMin EMPTY>
31 <!ATTLIST   SliderMin
32       value         CDATA #IMPLIED
33       hostvalue     CDATA #IMPLIED
34       hostaction    CDATA #IMPLIED >
35
36 <!ELEMENT SliderMax EMPTY>
```

```
37 <!ATTLIST   SliderMax
38       value        CDATA #IMPLIED
39       hostvalue    CDATA #IMPLIED
40       hostaction   CDATA #IMPLIED >
41
42 <!ELEMENT SliderStep EMPTY>
43 <!ATTLIST   SliderStep
44       value        CDATA #IMPLIED
45       hostvalue    CDATA #IMPLIED >
46
47 <!ELEMENT ScrollVal EMPTY>
48 <!ATTLIST   ScrollVal
49       caption      CDATA #IMPLIED
50       hostaction   CDATA #IMPLIED >
```

Appendix B – XML Grammar

```
COMPILER XMLGui

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "<!" TO ">"
COMMENTS FROM "<?" TO ">"

CHARACTERS
  cr          = CHR(13) .
  lf          = CHR(10) .
  instring    = ANY - '"' - "<" - ">" - cr - lf.

TOKENS
  string      = '"' instring { instring } '"'.

PRODUCTIONS
  XMLGui = Body EOF.
  Body = "<" "PanelSubunit" Name ">" "<" Panel "<" { Panel "<" }
         "/" "PanelSubunit" ">" .
  Panel = "Panel" ((Name Caption) | (Caption Name)) ">"
         "<" Component "<" { Component "<" }
         "/" "Panel" ">".
  Component = Slider | Label | Link | Scroller.
  Label = "Label" ((Caption Position) | (Position Caption)) "/" ">".
  Link = "Link" ((Caption Position) | (Position Caption)) LinkInfo "/" ">".
  Scroller = "Scroller" ((Caption Position) | (Position Caption)) ">"
         "<" ScrollerInfo "/" "Scroller" ">".
  ScrollerInfo = ScrollVal "<" { ScrollVal "<" }.
  ScrollVal = "ScrollVal" ((Caption HostAction) | (HostAction Caption)) "/"
  ">".
  Slider = "Slider" ((Caption Position) | (Position Caption)) ">"
         SliderInfo SliderInfo SliderInfo "<" "/" "Slider" ">".
  SliderInfo = "<" (SliderMin | SliderMax | SliderStep).
  SliderMin = "SliderMin" ValCombo ValCombo ValCombo "/" ">" .
  SliderMax = "SliderMax" ValCombo ValCombo ValCombo "/" ">".
  SliderStep = "SliderStep" ValCombo ValCombo "/" ">".
  ValCombo = (Value | HostValue | HostAction).
```

```
    Name = "name" "=" string.
    Caption = "caption" "=" string.
    Position = "position" "=" string.
    Value = "value" "=" string.
    HostValue = "hostvalue" "=" string.
    HostAction = "hostaction" "=" string.
    LinkInfo = "linkto" "=" string.
END XMLGui .
```

Appendix C – XMLGui.atg

```
COMPILER XMLGui  $XCN
/* GUI XML to binary image parser - Colin Dembovsky, 2000 */

#include <string.h>
#include <stdio.h>

char SourceName[256];
static FILE *bfile;
int check1, check2, check3, checka, checkb, checkc;

//#define DEBUG
//#define DEBUGQ
#include "qstructs.h"

IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "<!" TO ">"
COMMENTS FROM "<?" TO ">"

CHARACTERS
  cr          = CHR(13) .
  lf          = CHR(10) .
  instring    = ANY - '"' - "<" - ">" - cr - lf.

TOKENS
  string      = '"' instring { instring } '"'.

PRODUCTIONS
  XMLGui =
  (. char BimName[256]; int i;
     strcpy(BimName, SourceName);
     i = strlen(BimName)-1;
     while (i>0 && BimName[i] != '.') i--;
     if (i>0) BimName[i] = '\0';
       strcat(BimName, ".bim");
     if ((bfile = fopen(BimName, "w")) == NULL) {
       fprintf(stderr, "Unable to open binary output file %s\n", BimName);
```

```
      exit(EXIT_FAILURE);
    } .)
  Body EOF (. fclose (bfile); .) .

  Body = (. char psname[STRLEN]; TPanelSubunit* ps, *r; TPanel* p; .)
  "<" "PanelSubunit" Name<psname> (. ps = CreatePanelSubunit (psname); .)
  ">" "<" Panel<p> (. AddPanel (ps, p); .) "<"
  { Panel<p> (. AddPanel (ps, p); .) "<" } "/" "PanelSubunit" ">"
  (. if (Successful()) { SavePanelSubunit (ps);
       FreePanelSubunitMem (ps);
       fprintf (stderr, "Success!");}
       fclose (bfile); .) .

  Panel<TPanel* &p> = (. char pname[STRLEN], c[STRLEN]; .)
  "Panel" ((Name<pname> Caption<c>) | (Caption<c> Name<pname>))
  (. p = CreatePanel (pname, c); .) ">"
      "<" Component<p> "<" { Component<p> "<" } "/" "Panel" ">".

  Component<TPanel* &p> = (. TSlider* s; TLabel* L; TLink* Li; TScroller*
sc; .)
    Slider<s> (. AddSlider (p, s); .)
  | Label<L> (. AddLabel (p, L); .)
  | Link<Li> (. AddLink (p, Li); .)
  | Scroller<sc> (. AddScroller (p, sc); .) .

  Label<TLabel* &L> = (. char c[STRLEN], p[STRLEN]; .)
  "Label" ((Caption<c> Position<p> ) | (Position<p> Caption<c> ))
  (. int pos = atoi(p);
     L = CreateLabel (c, pos);  .) "/" ">".

  Link<TLink* &L> = (. char c[STRLEN], p[STRLEN], li[STRLEN]; .)
  "Link" ((Caption<c> Position<p> ) | (Position<p> Caption<c> )) LinkInfo
<li>
  (. int pos = atoi(p);
     L = CreateLink (c, pos, li); .) "/" ">".

  Scroller<TScroller* &s> = (. char c[STRLEN], p[STRLEN]; .)
  "Scroller" ((Caption<c> Position<p> ) | (Position<p> Caption<c> ))
  (. int pos = atoi(p);
     s = CreateScroller(c, pos); .)
```

```
  ">" "<"                                        :
  ScrollerInfo<s> "/" "Scroller" ">".                    .

  ScrollerInfo<TScroller* &s> = (. TScrollVal* t; .)
  ScrollVal<t>  (.  AddScrollerVal  (s,  t)  .)  "<"  {  ScrollVal<t>  (.
AddScrollerVal (s, t) .) "<" }.

  ScrollVal<TScrollVal* &s> = (. char c[STRLEN], h[STRLEN]; .)
  "ScrollVal" ( (Caption<c> HostAction<h> ) | (HostAction<h> Caption<c> ) )
  (. s = CreateScrollVal (c, h); .) "/" ">".

  Slider<TSlider* &s> = (. char c[STRLEN], p[STRLEN]; .)
  "Slider" ( (Caption<c> Position<p> ) | (Position<p> Caption<c> ))
  (. int pos = atoi(p);
       s = CreateSlider(c, pos); .)
  ">" (. checka = checkb = checkc = 0; .)
  SliderInfo<s> SliderInfo<s> SliderInfo<s> "<" "/" "Slider" ">".

  SliderInfo<TSlider* &s> = (. TSliderVal* m; .)
  "<"                                                    .
  ( SliderMin<m>  (. if (checka != 0) SemError(1003);
                     else { checka = 1; AddSliderMin(s, m); }  .)
  | SliderMax<m>  (. if (checkb != 0) SemError(1004);
                   else { checkb = 1; AddSliderMax(s, m); }  .)
  | SliderStep<m> (. if (checkc != 0) SemError(1005);
                   else { checkc = 1; AddSliderStep(s, m); }  .)
  ) .

  SliderMin<TSliderVal* &s> = (. char a[STRLEN], b[STRLEN], c[STRLEN]; int
at, bt, ct; .)
  "SliderMin"  (.  check1  =  check2  =  check3  =  0;  .)  ValCombo<a,  at>
ValCombo<b, bt> ValCombo<c, ct>
  (. if (at == 1) {
       if (bt == 2) s = CreateSliderVal (atoi(a), atoi(b), c);
       else s = CreateSliderVal (atoi(a), atoi(c), b);
     }
     if (bt == 1) {
       if (at == 2) s = CreateSliderVal (atoi(b), atoi(a), c);
       else s = CreateSliderVal (atoi(b), atoi(c), a);
     }
```

```
    if (ct == 1) {
      if (at == 2) s = CreateSliderVal (atoi(c), atoi(a), b);
      else s = CreateSliderVal (atoi(c), atoi(b), a);
    }
  .) "/" ">" .

  SliderMax<TSliderVal* &s> = (. char a[STRLEN], b[STRLEN], c[STRLEN]; int
at, bt, ct; .)
  "SliderMax" (. check1 = check2 = check3 = 0; .) ValCombo<a, at>
ValCombo<b, bt> ValCombo<c, ct>
  (. if (at == 1) {
      if (bt == 2) s = CreateSliderVal (atoi(a), atoi(b), c);
      else s = CreateSliderVal (atoi(a), atoi(c), b);
    }
    if (bt == 1) {
      if (at == 2) s = CreateSliderVal (atoi(b), atoi(a), c);
      else s = CreateSliderVal (atoi(b), atoi(c), a);
    }
    if (ct == 1) {
      if (at == 2) s = CreateSliderVal (atoi(c), atoi(a), b);
      else s = CreateSliderVal (atoi(c), atoi(b), a);
    }
   .) "/" ">".

  SliderStep<TSliderVal* &s> = (. char a[STRLEN], b[STRLEN]; int at, bt; .)
  "SliderStep" (. check1 = check2 = check3 = 0; .) ValCombo<a, at>
ValCombo<b, bt>
  (. s = CreateSliderVal (atoi(a), atoi(b), NULL); .) "/" ">".

  ValCombo<char n[], int &t> =
  ( Value<n> (. if (check1 != 0) SemError (1006);
              else { check1 = 1; t = 1; } .)
    | HostValue<n> (. if (check2 != 0) SemError (1008);
                  else { check2 = 1; t = 2; } .)
    | HostAction<n> (. if (check3 != 0) SemError (1007);
                  else { check3 = 1; t = 3; } .)
  ).

  Name<char lstr[]> = (. int i; char s[STRLEN]; .) "name" "=" string
  (. LexString (s, STRLEN);
```

```
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  Caption<char lstr[]> = (. int i; char s[STRLEN]; .) "caption" "=" string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  Position<char lstr[]> = (. int i; char s[STRLEN]; .) "position" "=" string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  Value<char lstr[]> = (. int i; char s[STRLEN]; .) "value" "=" string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  HostValue<char lstr[]> = (. int i; char s[STRLEN]; .) "hostvalue" "="
string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  HostAction<char lstr[]> = (. int i; char s[STRLEN]; .) "hostaction" "="
string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

  LinkInfo<char lstr[]> = (. int i; char s[STRLEN]; .) "linkto" "=" string
  (. LexString (s, STRLEN);
    for (i=1; i<strlen(s)-1; i++) lstr[i-1] = s[i];
    lstr[i-1] = '\0'; .) .

END XMLGui .
```

Appendix D – UPnP AV/R SCPD

```
1.  <?xml version="1.0"?>
2.  <scpd xmlns="urn:schemas-upnp-org:service-1-0">
3.
4.    <specVersion>
5.      <major>1</major>
6.      <minor>0</minor>
7.    </specVersion>
8.
9.    <actionList>
10.     <action>
11.       <name>Next_String_BoundedInput</name>
12.     </action>
13.
14.     <action>
15.       <name>Prev_String_BoundedInput</name>
16.     </action>
17.
18.     <action>
19.       <name>Next_String_BoundedSoundMode</name>
20.     </action>
21.
22.     <action>
23.       <name>Prev_String_BoundedSoundMode</name>
24.     </action>
25.
26.     <action>
27.       <name>IncreaseMainVolume</name>
28.     </action>
29.
30.     <action>
31.       <name>DecreaseMainVolume</name>
32.     </action>
33.
34.     <action>
35.       <name>IncreaseSleep</name>
36.     </action>
37.
38.     <action>
```

```
39.       <name>DecreaseSleep</name>
40.     </action>
41.
42.     <action>
43.       <name>Next_String_BoundedSoundField</name>
44.     </action>
45.
46.     <action>
47.       <name>Prev_String_BoundedSoundField</name>
48.     </action>
49.
50.     <action>
51.       <name>ToggleSpeakerA</name>
52.     </action>
53.
54.     <action>
55.       <name>ToggleSpeakerB</name>
56.     </action>
57.
58.   </actionList>
59.
60.
61.   <serviceStateTable>
62.
63.     <stateVariable sendEvents="yes">
64.       <name>Input</name>
65.       <dataType>string</dataType>
66.         <allowedValueList>
67.           <allowedValue>Phono</allowedValue>
68.           <allowedValue>CD</allowedValue>
69.           <allowedValue>Tuner</allowedValue>
70.           <allowedValue>CD-R</allowedValue>
71.           <allowedValue>MD/Tape</allowedValue>
72.           <allowedValue>DVD</allowedValue>
73.           <allowedValue>D-TV/LD</allowedValue>
74.           <allowedValue>CABLE/SAT</allowedValue>
75.           <allowedValue>VCR1</allowedValue>
76.           <allowedValue>V-Aux</allowedValue>
77.           </allowedValueList>
78.       <defaultValue>CD</defaultValue>
```

```
79.     </stateVariable>
80.
81.     <stateVariable sendEvents="yes">
82.       <name>SoundMode</name>
83.       <dataType>string</dataType>
84.         <allowedValueList>
85.           <allowedValue>Auto</allowedValue>
86.           <allowedValue>DTS</allowedValue>
87.           <allowedValue>Analog</allowedValue>
88.         </allowedValueList>
89.       <defaultValue>Auto</defaultValue>
90.     </stateVariable>
91.
92.     <stateVariable sendEvents="yes">
93.       <name>MainVolume</name>
94.       <dataType>i4</dataType>
95.         <allowedValueRange>
96.           <minimum>00</minimum>
97.           <maximum>100</maximum>
98.           <step>1</step>
99.         </allowedValueRange>
10.         <defaultValue>50</defaultValue>
101.    </stateVariable>
102.
103.    <stateVariable sendEvents="yes">
104.      <name>Sleep</name>
105.      <dataType>i4</dataType>
106.        <allowedValueRange>
107.          <minimum>00</minimum>
108.          <maximum>120</maximum>
109.          <step>30</step>
110.        </allowedValueRange>
111.      <defaultValue>0</defaultValue>
112.    </stateVariable>
113.
114.    <stateVariable sendEvents="yes">
115.      <name>SoundField</name>
116.      <dataType>string</dataType>
117.        <allowedValueList>
118.          <allowedValue>Hall</allowedValue>
```

```
119.          <allowedValue>Church</allowedValue>
120.        <allowedValue>Jazz</allowedValue>
121.          <allowedValue>Rock</allowedValue>
122.          <allowedValue>Stadium</allowedValue>
123.          <allowedValue>TV/Sports</allowedValue>
124.          <allowedValue>Sci-fi</allowedValue>
125.          <allowedValue>Adventure</allowedValue>
126.          <allowedValue>Enhanced</allowedValue>
127.          <allowedValue>Spectacle</allowedValue>
128.        </allowedValueList>
129.      <defaultValue>Rock</defaultValue>
130.    </stateVariable>
131.
132.    <stateVariable sendEvents="yes">
133.      <name>SpeakerA</name>
134.      <dataType>boolean</dataType>
135.      <defaultValue>1</defaultValue>
136.    </stateVariable>
137.
138.    <stateVariable sendEvents="yes">
139.      <name>SpeakerB</name>
140.      <dataType>boolean</dataType>
141.      <defaultValue>1</defaultValue>
142.    </stateVariable>
143.
144.  </serviceStateTable>
145. </scpd>
```

Appendix E – UPnP Presentation Page (HTML)

```
1.   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2.     <HTML>
3.       <HEAD>
4.           <TITLE>Presentation page for Controlling a UPnP Yamaha RXV1000
AV/R</TITLE>
5.       </HEAD>
6.
7.       <BODY>
8.
9.         <H3><P ID=FName>Yamaha RXV1000</P></H3>
10.
11.
12.        <H3>RXV1000 Config State Table</H3>
13.              <TABLE  BGCOLOR='#D6D7DE'  BORDER=0  VALIGN=top  ALIGN=left
CELLPADDING=1 CELLSPACING=3>
14.          <TR>
15.                        <TD  BGCOLOR='#000000'  VALIGN=center  ALIGN=center
WIDTH=60><B><FONT SIZE="2" COLOR=whitesmoke>Variable</FONT></B></TD>
16.                         <TD  VALIGN=middle  ALIGN=left  BGCOLOR='#000000'
WIDTH=60><B><FONT SIZE="2" COLOR=whitesmoke>Value</FONT></B></TD>
17.                         <TD  VALIGN=middle  ALIGN=left  BGCOLOR='#000000'
WIDTH=60><B><FONT SIZE="2" COLOR=whitesmoke>Actions</FONT></B></TD>
18.          </TR>
19.          <TR>
20.            <TD  BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Input</TD>
21.              <TD  BGCOLOR="#FFFFFF" valign="top"><P ID=Input></P></TD>
22.            <TD  BGCOLOR="#FFFFFF" valign="top">
23.               <INPUT type="button" onclick="NextInput()"      value="
Next   ">
24.           <INPUT type="button" onclick="PrevInput()"    value=" Previous
">
25.            </TD>
26.          </TR>
27.          <TR>
28.                <TD   BGCOLOR="#FFFFFF"  VALIGN=center  ALIGN=center>Sound
Mode</TD>
29.            <TD  BGCOLOR="#FFFFFF" valign="top"><P ID=SoundMode></P></TD>
30.            <TD  BGCOLOR="#FFFFFF" valign="top">
```

```
31.              <INPUT type="button" onclick="NextSoundMode()"  value="
Next   ">
32.              <INPUT type="button" onclick="PrevSoundMode()"  value="
Previous ">
33.           </TD>
34.          </TR>
35.          <TR>
36.                 <TD   BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Main
Volume</TD>
37.                              <TD     BGCOLOR="#FFFFFF"   valign="top"><P
ID=MainVolume></P></TD>
38.            <TD  BGCOLOR="#FFFFFF" valign="top">
39.               <INPUT type="button" onclick="IncVolume()"      value=" Up
">
40.             <INPUT type="button" onclick="DecVolume()"      value=" Down
">
41.            </TD>
42.          </TR>
43.          <TR>
44.                <TD   BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Sound
Field</TD>
45.                              <TD     BGCOLOR="#FFFFFF"   valign="top"><P
ID=SoundField></P></TD>
46.            <TD  BGCOLOR="#FFFFFF" valign="top">
47.               <INPUT       type="button"       onclick="NextSoundField()"
value="   Next   ">
48.             <INPUT type="button" onclick="PrevSoundField()"      value="
Previous ">
49.            </TD>
50.          </TR>
51.          <TR>
52.               <TD   BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Speaker
Relay A</TD>
53.            <TD  BGCOLOR="#FFFFFF" valign="top"><P ID=SpeakerA></P></TD>
54.            <TD  BGCOLOR="#FFFFFF" valign="top">
55.               <INPUT       type="button"       onclick="ToggleSpeakerA()"
value=" Toggle ">
56.            </TD>
57.          </TR>
58.          <TR>
```

```
59.              <TD  BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Speaker
Relay B</TD>
60.            <TD  BGCOLOR="#FFFFFF" valign="top"><P ID=SpeakerB></P></TD>
61.           <TD  BGCOLOR="#FFFFFF" valign="top">
62.              <INPUT      type="button"      onclick="ToggleSpeakerB()"
value=" Toggle ">
63.            </TD>
64.          </TR>
65.          <TR>
66.            <TD  BGCOLOR="#FFFFFF" VALIGN=center ALIGN=center>Sleep</TD>
67.            <TD  BGCOLOR="#FFFFFF" valign="top"><P ID=Sleep></P></TD>
68.            <TD  BGCOLOR="#FFFFFF" valign="top">
69.              <INPUT type="button" onclick="IncSleep()"      value=" Up
">
70.             <INPUT type="button" onclick="DecSleep()"      value=" Down
">
71.            </TD>
72.         </TR>
73.        </TABLE>
74.
75.
76.        <BR>
77.        <BR>
78.        <BR>
79.        <BR>
80.        <BR>
81.        <BR>
82.        <BR>
83.        <BR>
84.        <BR>
85.
86.  <H3> </H3>
87.
88.  <SCRIPT language=VBScript>
89.
90.
91.  ' ********************************************************
92.  ' Event handler called when the UPnP device submits events
93.  ' ********************************************************
94.
```

```
95.
96.
97.  Sub eventHandler(callbackType, svcObj, varName, value)
98.
99.  'Dim output
10.  'output = output & "varName " & varName & vbCrLf
101. 'output = output & "value " & value & vbCrLf
102. 'output = output & "svcObj " & svcObj.Id & vbCrLf
103. 'MsgBox output
104.
105.   If (callbackType = "VARIABLE_UPDATE") Then
106.     select case varName
107.           Case "Input"          Input.innerText = value
108.           Case "SoundMode"      SoundMode.innerText = value
109.           Case "SoundField"     SoundField.innerText = value
110.           Case "Sleep"          Sleep.innerText = value
111.           Case "SpeakerA"       SpeakerA.innerText = value
112            Case "SpeakerB"       SpeakerB.innerText = value
113.           Case "Volume"         MainVolume.innerText = value
114.     end select
115.   End If
116. End Sub
117.
118.
119.
120. ' ************************************************************
121. ' Button action callbacks invoke actions
122. ' ************************************************************
123.
124. function NextInput()
125.     Dim inArgs(0)
126.     Dim outArgs(0)
127.      AvrControlService.InvokeAction "Next_String_BoundedInput", inArgs,
outArgs
128. end function
129.
130. function PrevInput()
131.     Dim inArgs(0)
132.     Dim outArgs(0)
```

```
133.      AvrControlService.InvokeAction "Prev_String_BoundedInput", inArgs,
outArgs
134. end function
135.
136. function NextSoundMode()
137.     Dim inArgs(0)
138.     Dim outArgs(0)
139.         AvrControlService.InvokeAction  "Next_String_BoundedSoundMode",
inArgs, outArgs
140. end function
141.
142. function PrevSoundMode()
143.     Dim inArgs(0)
144.     Dim outArgs(0)
145.         AvrControlService.InvokeAction  "Prev_String_BoundedSoundMode",
inArgs, outArgs
146. end function
147.
148. function IncVolume()
149.     Dim inArgs(0)
150.     Dim outArgs(0)
151.     AvrControlService.InvokeAction "IncreaseVolume", inArgs, outArgs
152. end function
153.
154. function DecVolume()
155.     Dim inArgs(0)
156.     Dim outArgs(0)
157.     AvrControlService.InvokeAction "DecreaseVolume", inArgs, outArgs
158. end function
159.
160. function IncSleep()
161.     Dim inArgs(0)
162.     Dim outArgs(0)
163.     AvrControlService.InvokeAction "IncreaseSleep", inArgs, outArgs
164. end function
165.
166. function DecSleep()
167.     Dim inArgs(0)
168.     Dim outArgs(0)
169.     AvrControlService.InvokeAction "DecreaseSleep", inArgs, outArgs
```

```
170. end function
171.
172. function NextSoundField()
173.     Dim inArgs(0)
174.     Dim outArgs(0)
175.         AvrControlService.InvokeAction  "Next_String_BoundedSoundField",
inArgs, outArgs
176. end function
177.
178. function PrevSoundField()
179.     Dim inArgs(0)
180.     Dim outArgs(0)
181.         AvrControlService.InvokeAction  "Prev_String_BoundedSoundField",
inArgs, outArgs
182. end function
183.
184. function ToggleSpeakerA()
185.     Dim inArgs(0)
186.     Dim outArgs(0)
187.     AvrControlService.InvokeAction "ToggleSpeakerA", inArgs, outArgs
188. end function
189.
190. function ToggleSpeakerB()
191.     Dim inArgs(0)
192.    Dim outArgs(0)
193.     AvrControlService.InvokeAction "ToggleSpeakerB", inArgs, outArgs
194. end function
195.
196.
197. ' ********************************************************
198. ' Download the description document from the UPnP device
199. ' ********************************************************
200.     Dim AvrDesc
201. Set AvrDesc = CreateObject("UPnP.DescriptionDocument.1")
202. AvrDesc.Load("../description/rxv1000.xml")
203.
204.
205. ' ********************************************************
206. ' Get the Root Device from the description document
207. ' ********************************************************
```

```
208. Dim AvrDevice
209. Set AvrDevice = AvrDesc.RootDevice
210.
211.
212. ' ********************************************************
213. ' Output some of the device properties to the user
214. ' ********************************************************
215. Dim output
216. output = "Found: " & vbCrLf
217. output = output & "DisplayName: " & AvrDevice.FriendlyName & vbCrLf
218. output = output & "Type: " & AvrDevice.Type & vbCrLf
219. output = output & "UDN: " & AvrDevice.UniqueDeviceName & vbCrLf
220. MsgBox output
221.
222. FName.innerText =  AvrDevice.FriendlyName
223.
224.
225. ' ********************************************************
226. ' Attach the event handler to the tv control service
227. ' ********************************************************
228. Dim AvrControlService
229.           set           AvrControlService=AvrDevice.Services("urn:upnp-
org:serviceId:avrconfig")
230. AvrControlService.AddCallback GetRef("eventHandler")
231.
232. </SCRIPT>
233. </BODY>
234. </HTML>
```

Appendix F – UPnP "Device" Module Listing

```
1.  // Include all the necessary header files.
2.  #include <windows.h>
3.  #include <stdlib.h>
4.  #include <stdio.h>
5.  #include "global.h"
6.  #include "event.h"
7.  #include "util.h"
8.  #include "udevice.h"
9.
10. #define MAX_VOLUME        100
11. #define MIN_VOLUME        0
12. #define MAX_INPUT         9
13. #define MIN_INPUT         0
14. #define MAX_MODE          9
15. #define MIN_MODE          0
16. #define MAX_SLEEP         120
17. #define MIN_SLEEP         0
18. #define MAX_FIELD         9
19. #define MIN_FIELD         0
20.
21.
22. struct AVRConfig
23. {
24.   DWORD Input;
25.   DWORD SoundMode;
26.   DWORD SoundField;
27.   DWORD Volume;
28.   DWORD Sleep;
29.   BOOLEAN     SpeakerA;
30.   BOOLEAN     SpeakerB;
31. };
32.
33. /* Binary Copy of above Device */
34.
35. AVRConfig InsAVRConfig =
36. {
37.   0,
38.   0,
```

```
39.    0,
40.    50,
41.    0,
42.    1,
43.    1
44. };
45.
46. // Number of state variables.
47. static const int nStateVariables = 7;
48.
49. // String copy of state variables for initializing eventing.
50. E_PROP_LIST ePropList =
51. {
52.     nStateVariables,
53.     {
54.         {"Input",              "1",   FALSE, TRUE},
55.         {"SoundMode",          "1",   FALSE, TRUE},
56.         {"SoundField",         "10",  FALSE, TRUE},
57.         {"Volume",             "50",  FALSE, TRUE},
58.         {"Sleep",              "0",   FALSE, TRUE},
59.         {"SpeakerA",           "1",   FALSE, TRUE},
60.         {"SpeakerB",           "1",   FALSE, TRUE},
61.     }
62. };
63.
64. DWORD Do_AVR_Init    (CHAR* StrEventUrl)
65. {
66.    BOOL bResult = InitializeUpnpEventSource(StrEventUrl, &ePropList);
67.
68.    return bResult ? 1 : 0;
69. }
70.
71. DWORD Do_AVR_Cleanup (CHAR* StrEventUrl)
72. {
73.   if (!CleanupUpnpEventSource(StrEventUrl))
74.   puts ("Do_AVR_Cleanup failed to clean up the Upnp Event Source");
75.
76.   return 1;
77. }
78.
```

```
79. DWORD Do_IncreaseVolume (
80.                          CHAR*          StrEventUrl,
81.                          DWORD          cArgs,
82.                          ARG*           rgArgs,
83.                          PDWORD         pArgsOut,
84.                          ARG_OUT*       rgArgsOut
85.                         )
86. {
87.   if (InsAVRConfig.Volume < MAX_VOLUME)
88.   {
89.         InsAVRConfig.Volume++;
90.         CHAR szValue[32];
91.         sprintf(szValue, "%u", InsAVRConfig.Volume);
92.         ChangeProp(StrEventUrl, "Volume", szValue);
93.         SerialCommand ("Volume", szValue);
94.         SubmitPropEvents(StrEventUrl, NULL);
95.   }
96.   return 0;
97. }
98.
99. DWORD Do_DecreaseVolume (
100.                          CHAR*          StrEventUrl,
101.                         DWORD          cArgs,
102.                         ARG*           rgArgs,
103.                         PDWORD         pArgsOut,
104.                         ARG_OUT*       rgArgsOut
105.                        )
106. {
107.  if (InsAVRConfig.Volume > MIN_VOLUME)
108.  {
109.        InsAVRConfig.Volume--;
110.        CHAR szValue[32];
111.        sprintf(szValue, "%u", InsAVRConfig.Volume);
112.        ChangeProp(StrEventUrl, "Volume", szValue);
113.        SerialCommand ("Volume", szValue);
114.        SubmitPropEvents(StrEventUrl, NULL);
115.  }
116.  return 0;
117. }
118. DWORD Do_IncreaseSleep (
```

```
119.                                    CHAR*           StrEventUrl,
120.                          DWORD           cArgs,
121.                          ARG*            rgArgs,
122.                          PDWORD          pArgsOut,
123.                          ARG_OUT*        rgArgsOut
124.                         )
125. {
126.  if (InsAVRConfig.Sleep < MAX_SLEEP)
127.  {
128.        InsAVRConfig.Sleep+=30;
129.        CHAR szValue[32];
130.        sprintf(szValue, "%u", InsAVRConfig.Sleep);
131.        ChangeProp(StrEventUrl, "Sleep", szValue);
132.        SubmitPropEvents(StrEventUrl, NULL);
133.  }
134.  return 0;
135. }
136. DWORD Do_DecreaseSleep (
137.                          CHAR*           StrEventUrl,
138.                          DWORD           cArgs,
149.                          ARG*            rgArgs,
140.                          PDWORD          pArgsOut,
141.                          ARG_OUT*        rgArgsOut
142.                         )
143. {
144.  if (InsAVRConfig.Sleep > MIN_SLEEP)
145.  {
146.        InsAVRConfig.Sleep-=30;
147.        CHAR szValue[32];
148.        sprintf(szValue, "%u", InsAVRConfig.Sleep);
149.        ChangeProp(StrEventUrl, "Sleep", szValue);
150.        SerialCommand ("Sleep", szValue);
151.        SubmitPropEvents(StrEventUrl, NULL);
152.  }
153.  return 0;
154. }
155. DWORD Do_Next_String_BoundedSoundMode (
156.                          CHAR*           StrEventUrl,
157.                          DWORD           cArgs,
158.                          ARG*            rgArgs,
```

```
159.                          PDWORD          pArgsOut,
160.                          ARG_OUT*        rgArgsOut
161.                         )
162. {
163.  if (InsAVRConfig.SoundMode < MAX_MODE)
164.  {
165.        InsAVRConfig.SoundMode++;
166.        CHAR szValue[32];
167.        sprintf(szValue, "%u", InsAVRConfig.SoundMode);
168.        ChangeProp(StrEventUrl, "SoundMode", szValue);
169.        SerialCommand ("SoundMode", szValue);
170.        SubmitPropEvents(StrEventUrl, NULL);
171.  }
172.  return 0;
173. }
174. DWORD Do_Prev_String_BoundedSoundMode (
175.                          CHAR*           StrEventUrl,
176.                          DWORD           cArgs,
177.                          ARG*            rgArgs,
178.                          PDWORD          pArgsOut,
179.                          ARG_OUT*        rgArgsOut
180.                         )
181. {
182.  if (InsAVRConfig.SoundMode > MIN_MODE)
183.  {
184.        InsAVRConfig.SoundMode--;
185.        CHAR szValue[32];
186.        sprintf(szValue, "%u", InsAVRConfig.SoundMode);
187.        ChangeProp(StrEventUrl, "SoundMode", szValue);
188.        SerialCommand ("SoundMode", szValue);
189.        SubmitPropEvents(StrEventUrl, NULL);
190.  }
191.  return 0;
192. }
193. DWORD Do_Next_String_BoundedSoundField (
194.                          CHAR*           StrEventUrl,
195.                          DWORD           cArgs,
196.                          ARG*            rgArgs,
197.                          PDWORD          pArgsOut,
198.                          ARG_OUT*        rgArgsOut
```

```
199.                        )
200. {
201.  if (InsAVRConfig.SoundField < MAX_FIELD)
202.  {
203.        InsAVRConfig.SoundField++;
204.        CHAR szValue[32];
205.        sprintf(szValue, "%u", InsAVRConfig.SoundField);
206.        ChangeProp(StrEventUrl, "SoundField", szValue);
207.        SerialCommand ("SoundField", szValue);
208.        SubmitPropEvents(StrEventUrl, NULL);
209.  }
210.  return 0;
211. }
212. DWORD Do_Prev_String_BoundedSoundField (
213.                        CHAR*          StrEventUrl,
214.                        DWORD          cArgs,
215.                        ARG*           rgArgs,
216.                        PDWORD         pArgsOut,
217.                        ARG_OUT*       rgArgsOut
218.                       )
219. {
220.  if (InsAVRConfig.SoundField > MIN_FIELD)
221.  {
222.        InsAVRConfig.SoundField--;
223.        CHAR szValue[32];
224.        sprintf(szValue, "%u", InsAVRConfig.SoundField);
225.        ChangeProp(StrEventUrl, "SoundField", szValue);
226.        SerialCommand ("SoundField", szValue);
227.        SubmitPropEvents(StrEventUrl, NULL);
228.  }
229.  return 0;
230. }
231. DWORD Do_Next_String_BoundedInput (
232.                        CHAR*          StrEventUrl,
233.                        DWORD          cArgs,
234.                        ARG*           rgArgs,
235.                        PDWORD         pArgsOut,
236.                        ARG_OUT*       rgArgsOut
237.                       )
238. {
```

```
239.  if (InsAVRConfig.Input < MAX_INPUT)
240.  {
241.        InsAVRConfig.Input++;
242.        CHAR szValue[32];
243.        sprintf(szValue, "%u", InsAVRConfig.Input);
244.        ChangeProp(StrEventUrl, "Input", szValue);
245.        SerialCommand ("Input", szValue);
246.        SubmitPropEvents(StrEventUrl, NULL);
247.  }
248.  return 0;
249. }
250. DWORD Do_Prev_String_BoundedInput (
251.                          CHAR*          StrEventUrl,
252.                          DWORD          cArgs,
253.                          ARG*           rgArgs,
254.                          PDWORD         pArgsOut,
255.                          ARG_OUT*       rgArgsOut
256.                         )
257. {
258.  if (InsAVRConfig.Input > MIN_INPUT)
259.  {
260.        InsAVRConfig.Input--;
261.        CHAR szValue[32];
262.        sprintf(szValue, "%u", InsAVRConfig.Input);
263.        ChangeProp(StrEventUrl, "Input", szValue);
264.        SerialCommand ("Input", szValue);
265.        SubmitPropEvents(StrEventUrl, NULL);
266.  }
267.  return 0;
268. }
269. DWORD Do_ToggleSpeakerA (
270.                          CHAR*          StrEventUrl,
271.                          DWORD          cArgs,
272.                          ARG*           rgArgs,
273.                          PDWORD         pArgsOut,
274.                          ARG_OUT*       rgArgsOut
275.                         )
276. {
277.  InsAVRConfig.SpeakerA = !InsAVRConfig.SpeakerA;
278.  CHAR szValue[32];
```

```
279.  sprintf(szValue, "%u", InsAVRConfig.SpeakerA);
280.  ChangeProp(StrEventUrl, "SpeakerA", szValue);
281.  SerialCommand ("SpeakerA", szValue);
282.  SubmitPropEvents(StrEventUrl, NULL);
283.  return 0;
284. }
285. DWORD Do_ToggleSpeakerB (
286.                          CHAR*          StrEventUrl,
287.                          DWORD          cArgs,
288.                          ARG*           rgArgs,
289.                          PDWORD         pArgsOut,
290.                          ARG_OUT*       rgArgsOut
291.                         )
292. {
293.  InsAVRConfig.SpeakerB = !InsAVRConfig.SpeakerB;
294.  CHAR szValue[32];
295.  sprintf(szValue, "%u", InsAVRConfig.SpeakerB);
296.  ChangeProp(StrEventUrl, "SpeakerB", szValue);
297.  SerialCommand ("SpeakerB", szValue);
298.  SubmitPropEvents(StrEventUrl, NULL);
299.  return 0;
300. }
```

Appendix G – UPnP "UDevice" Module Listing

```
1.  // RXV1000 Device function prototypes
2.  DWORD Do_AVR_Init     (CHAR* StrEventUrl);
3.
4.  DWORD Do_AVR_Cleanup (CHAR* StrEventUrl);
5.
6.  DWORD Do_IncreaseVolume (
7.                           CHAR*          StrEventUrl,
8.                           DWORD          cArgs,
9.                           ARG*           rgArgs,
10.                          PDWORD         pArgsOut,
11.                          ARG_OUT*       rgArgsOut
12.                         );
13.
14. DWORD Do_DecreaseVolume (
15.                          CHAR*          StrEventUrl,
16.                          DWORD          cArgs,
17.                          ARG*           rgArgs,
18.                          PDWORD         pArgsOut,
19.                          ARG_OUT*       rgArgsOut
20.                         );
21.
22. DWORD Do_IncreaseSleep (
23.                          CHAR*          StrEventUrl,
24.                          DWORD          cArgs,
25.                          ARG*           rgArgs,
26.                          PDWORD         pArgsOut,
27.                          ARG_OUT*       rgArgsOut
28.                         );
29.
30. DWORD Do_DecreaseSleep (
31.                          CHAR*          StrEventUrl,
32.                            DWORD          cArgs,
33.                          ARG*           rgArgs,
34.                          PDWORD         pArgsOut,
35.                          ARG_OUT*       rgArgsOut
36.                         );
37.
38. DWORD Do_Next_String_BoundedInput (
```

```
39.                         CHAR*           StrEventUrl,
40.                         DWORD           cArgs,
41.                         ARG*            rgArgs,
42.                         PDWORD          pArgsOut,
43.                         ARG_OUT*        rgArgsOut
44.                        );
45.
46. DWORD Do_Prev_String_BoundedInput (
47.                         CHAR*           StrEventUrl,
48.                         DWORD           cArgs,
49.                         ARG*            rgArgs,
50.                         PDWORD          pArgsOut,
51.                         ARG_OUT*        rgArgsOut
52.                        );
53.
54. DWORD Do_Next_String_BoundedSoundField (
55.                         CHAR*           StrEventUrl,
56.                         DWORD           cArgs,
57.                         ARG*            rgArgs,
58.                         PDWORD          pArgsOut,
59.                         ARG_OUT*        rgArgsOut
60.                        );
61.
62. DWORD Do_Prev_String_BoundedSoundField (
63.                         CHAR*           StrEventUrl,
64.                         DWORD           cArgs,
65.                         ARG*            rgArgs,
66.                         PDWORD          pArgsOut,
67.                         ARG_OUT*        rgArgsOut
68.                        );
69.
70. DWORD Do_Next_String_BoundedSoundMode (
71.                         CHAR*           StrEventUrl,
72.                         DWORD           cArgs,
73.                         ARG*            rgArgs,
74.                         PDWORD          pArgsOut,
75.                         ARG_OUT*        rgArgsOut
76.                        );
77.
78. DWORD Do_Prev_String_BoundedSoundMode (
```

```
79.                         CHAR*           StrEventUrl,
80.                         DWORD           cArgs,
81.                         ARG*            rgArgs,
82.                         PDWORD          pArgsOut,
83.                         ARG_OUT*        rgArgsOut
84.                               );
85.
86. DWORD Do_ToggleSpeakerA (
87.                         CHAR*           StrEventUrl,
88.                         DWORD           cArgs,
89.                         ARG*            rgArgs,
90.                         PDWORD          pArgsOut,
91.                         ARG_OUT*        rgArgsOut
92.                        );
93.
94. DWORD Do_ToggleSpeakerB (
95.                         CHAR*           StrEventUrl,
96.                         DWORD           cArgs,
97.                         ARG*            rgArgs,
98.                         PDWORD          pArgsOut,
99.                         ARG_OUT*        rgArgsOut
10.                        );
101.
102. // Add SERVICE_CTL entries for each device.
103.
104. // c_cDemoSvc must be set to the number of control IDs in c_rgSvc
(below).
105. // Change this number if you add entries to c_rgSvc.
106. #define c_cDemoSvc 1
107.
108. // The table that follows is a method of associating UPDIAG control
identifiers with
109. // device-specific function calls.
110.
111. const SERVICE_CTL c_rgSvc[c_cDemoSvc] =
112  {
113.
114.    {
115.          "avrconfig",                                  //Service control
identifier.
```

```
116.            (PFNAI)Do_AVR_Init,                    //Device-specific
initialization function.
117.            (PFNAC)Do_AVR_Cleanup,                 //Device-specific
cleanup function.
118.            12,                                    //Number of actions this
service implements.
119.        {
120.            { "IncreaseVolume",
      (PFNAS)Do_IncreaseVolume                          },
121.            { "DecreaseVolume",
      (PFNAS)Do_DecreaseVolume                          },
122.              { "IncreaseSleep",
      (PFNAS)Do_IncreaseSleep                           },
123.            { "DecreaseSleep",
      (PFNAS)Do_DecreaseSleep                           },
124.              { "Next_String_BoundedInput",
      (PFNAS)Do_Next_String_BoundedInput          },
125.            { "Prev_String_BoundedInput",
      (PFNAS)Do_Next_String_BoundedInput          },
126.                                        {   "Next_String_BoundedSoundMode",
      (PFNAS)Do_Next_String_BoundedSoundMode      },
127.                                        {   "Prev_String_BoundedSoundMode",
      (PFNAS)Do_Prev_String_BoundedSoundMode      },
128.              {                           "Next_String_BoundedSoundField",
      (PFNAS)Do_Next_String_BoundedSoundField     },
129.                                      {   "Prev_String_BoundedSoundField",
      (PFNAS)Do_Prev_String_BoundedSoundField     },
130.              { "ToggleSpeakerA",
      (PFNAS)Do_ToggleSpeakerA                          },
131.            { "ToggleSpeakerB",
      (PFNAS)Do_ToggleSpeakerB                          },
132.        },
133.    },
134. };
135.
136. // Define the product name that will appear in HTTP SERVER header.
137. const CHAR c_szProduct[] = "Yamaha RXV1000";
138. const CHAR c_szProductVersion[] = "1.0";
```

References

[1] M. Weiser, R. Gold, J.S. Brown. "The origins of ubiquitous computing research at PARC in the late 1980's". *IBM Systems Journal 38*, No. 4, 693-696 (1999).

[2] I. O'Sullivan. "The IAN Information Appliance Network in Pervasive Home Computing". http://www.enikia.com/downloads/ian.pdf (1999).

[3] I.O'Sullivan. "The IEDMM" http://www.enikia.com/downloads/iedmm.pdf (1999).

[4] K.F. Eustice, T.J. Lehman, A. Morales, M.C. Munson, S. Edlund, M. Guillen. "A universal information appliance". *IBM Systems Journal 38*, No. 4, 575-601 (1999).

[5] P. Wyckoff, S.W. McLaughry, T.J. Lehman, D.A. Ford. "Tspaces". *IBM Systems Journal 37*, No. 3, 454-474 (1998).

[6] T.J. Lehman, K.H. Lee, D. Nelson, T Hodes, M.C. Munson, M Guillen, A Morales. "MoDAL (Mobile Document Application Language)". http://www.almaden.ibm.com/cs/TSpaces/MoDAL/ (1999).

[7] The ServiceUI Project. "Jini Service UI Draft Specification, Version 1.0" http://www.artima.com/jini/serviceui/DraftSpec.html

[8] "The HAVi Specification, Internal Version 1.01" (2000)

[9] IEEE. "Standard for a High Performance Serial Bus". *IEEE Std 1394-1995* (1995)

[10] G. O'Driscoll. "The Essential Guide to Home Networking Technologies". *Prentice Hall* (2001).

[11] Microsoft Corporation. "Universal Plug and Play Device Architecture, Ver 1.0". http://www.upnp.org/download/UPnPDA10_20000613.htm (2000).

[12] 1394 Trade Association. "AV/C Panel Subunit Specification 1.1". *TA Document 2001001* (2001).

[13] IEC. "Consumer and audio/video equipment – Digital Interface – Part 1: General" *IEC 61883-1* (1998)

[14] 1394 Trade Association. "AV/C Digital Command Set General Specification Version 3.0". *TA Document 1998003* (1998).

[15] 1394 Trade Association "AV/C commands for management of Asynchronous Serial Bus Connections Ver 1.0 FC1" (1999).

[16] W3C. "Extensible Markup Language (XML) 1.0 (Second Edition)" (2000)

[17] Object Management Group (OMG) Inc. "OMG Unified Modelling Language Specification Version 1.3" (1999)

[18] Pat Terry. "Compilers and Compiler Generators: An Introduction with C++". *International Thomson, London, England* (1999)

[19] Elizabeth Parks. "Multimedia Connectivity™ to Drive Networking to the Masses" http://www.parksassociates.com/inthePress/press_releases/highendentertainment.html (2000)

[20] Sun Microsystems. "Jini™ Network Technology". http://www.sun.org/jini

[21] IEEE. "Control and Status Registers (CSR) Architecture for Microcomputer Buses". *IEEE Std 1212* (1994)

[22] 1394 Trade Association "AV/C Audio Subunit Specification Ver 1.00 FC2" (1999).

[23] Y. Goland, T. Cai, et al. "Simple Service Discovery Protocol/1.0." http://www.upnp.org/draft_cai_ssdp_v1_03.txt (2000)

[24] Y. Goland, J. Schlimmer. "Multicast and Unicast UDP HTTP Messages." http://www.upnp.org/draft-goland-http-udp-04.txt (2000)

[25] D. Box, D. Ehnebuske, et al. "Simple Object Access Protocol (SOAP) 1.1. " http://www.w3.org/TR/SOAP/ (2000)

[26] J. Cohen, S. Aggarwal, Y. Goland. "General Event Notification Architecture Base: Client to Arbiter." http://www.upnp.org/draft-cohen-gena-client-01.txt (2000)

[27] Postel, J. "Internet Protocol," RFC 760, USC/Information Sciences Institute (1980)

[28] Postel, J. "Transmission Control Protocol," RFC 761, USC/Information Sciences Institute (1980)

[29] P. Mockapetris. "Domain Names – Implementation and Specification" RFC 1035, IETF (1987)

[30] R. Troll. "Automatically Choosing an IP Address in an Ad-Hoc IPv4 Network" IETF. http://www.rtroll.org/ryan/pubs/draft-ietf-dhc-ipv4-autoconfig-05.txt (2001)

[31] L. Esibov, B. Aboba, D. Thaler. "Multicast DNS" http://www.globecom.net/ietf/draft/draft-ietf-dnsext-mdns-01.html (2001)

[32] R. Droms. "Dynamic Host Configuration Protocol" RFC 1541, IETF (1997)

[33] W3C. "HTML 4.01 Specification" (1999)

[34] P. Ward, S. Mellor. "Structured development for real-time systems" (Volumes 1, 2 and 3) *Yourdon Press* (1986)

[35] 1394 TA. "AV/C Compatible Asynchronous Serial Bus Connections Version 1.0" (1999)

[36] Microsoft Corporation. file:///Library/_upnpkit_understanding_how_the_upnp_device_sample_code_is_organized.htm (2001)

[37] P. Coad, D. North, M. Mayfield. "Object Models: Strategies, Patterns and Applications, Second Edition" *Yourdon Press, Prentice Hall* (1997)

[38] P. Naur. "Report on the algorithmic language Algol 60" *Communications of the ACM*, 6(1) (1963)

[39] N. Wirth. "What can we do about the unnecessary diversity of notation for syntactic definitions?" *Communications of the ACM*, 20(11) (1977)

[40] 1394 Trade Association. "AV/C Audio Subunit Specification 1.0". *TA Document 1999001* (2000).

[41] 1394 Trade Association. "AV/C Disc Subunit General Specification 1.0". *TA Document 1998013* (1999).

[42] Digital Harmony Technologies, Inc. "Functional Specification (Preliminary). DHIVA. Digital Harmony Interface for Video and Audio (DH1 Adapter Card)" (1999)

[43] Digital Harmony Technologies, Inc. "Embedded Development Platform" (2000)

[44] P. Johansson. RFC 2734 "IPv4 over IEEE 1394" (1999)

[45] B. Nickless. Internet Draft "The IP Multicast Service Model" (2002)