Spring 5-2007

ZigBee Event System (ZES): A Multimedia Application

Bradford Andrew Russell
University of Tennessee - Knoxville

Bradford Andrew Russell
Bachelor of Science in Computer Engineering

ZigBee Event System (ZES)

A Multimedia Application

Final Report

Team Ragnar

Chris Niedzwiedz
Rick Weber
Derek Rose
Brian Sharp
Brad Russell
Justin Ellison

I) Overview

Introduction

Mission Statement

The goal of our system is to be capable of controlling wireless devices through the ZigBee wireless protocol using commands and signals from another applications or processes. These devices must act in synchronization with a given media file with the intent of providing an interactive media experience for the user.

This system could be used in trade show environments with the intent of showcasing the new products or services that a compny provides interactively to the consumer. Such a consumer would only have to approach an automated kiosk and the system would begin explaining through audio, video, and other sensory methods about the particular product. This system could also be used in stores and markets to attract potential customers to a particular product or highlight the specials of the day. Further, this system could be tied to a database and present personalized advertisments or information to a person based on their previous purchases or current items in their basket (requiring tie in with RFID).

Having this interaction with the consumer could increase sales of a particular product, or promote awareness about different things in public areas. The public are more likely to buy into products that have been target to their needs and find it less irritating than being exposed to countless advertisements about products they have no interest in. This sytem, tied with a database, could allow for such targeting. Futher, instead of having a static movie play on a loop all day is less effective than targeting such information to specific demographics.

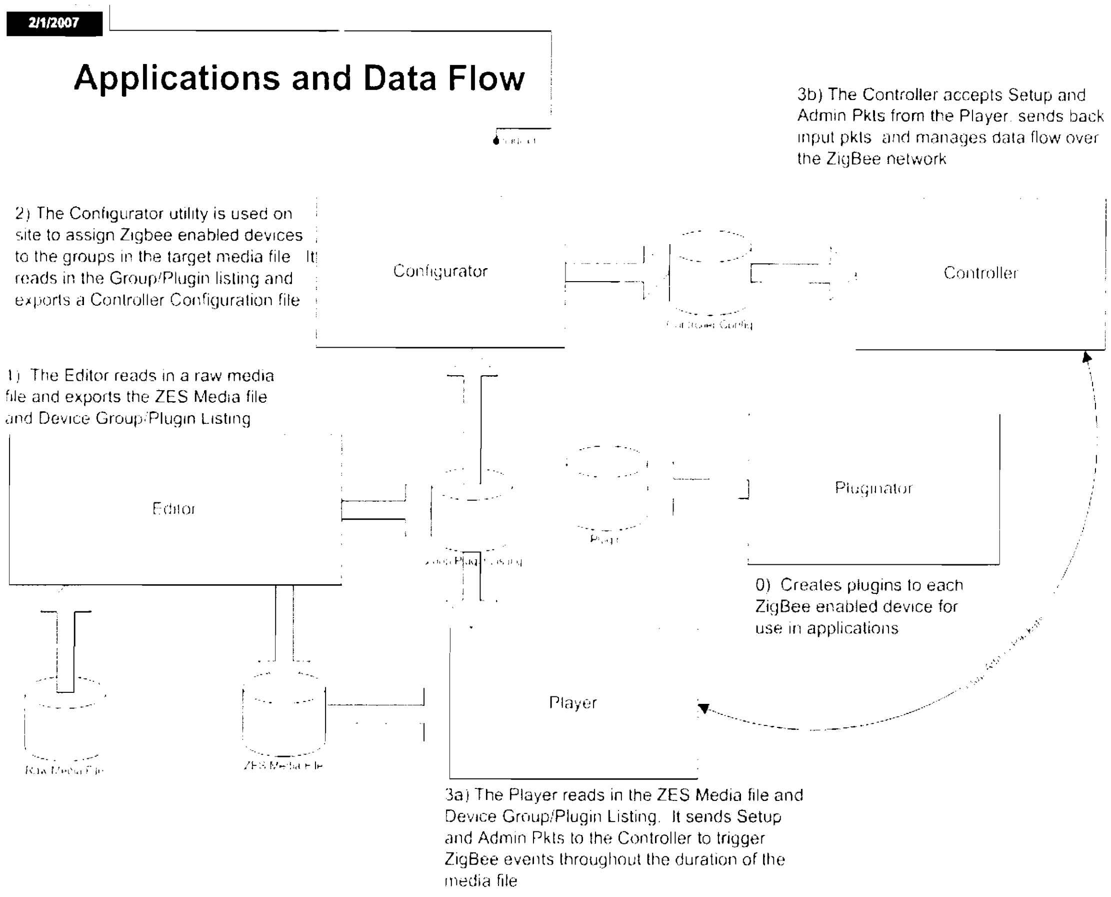
The solution to this problem is the ZigBee Event System (ZES) Framework. Intended to coordinate data transmission and synchronized events in a wireless environment, the Framework provides an interface with which to establish, manage, and communicate with a ZigBee or Embernet based wireless network. It is able to send commands to and receive data from wireless devices with which it has been preconfigured to communicate. Further, the framework is responsible for abstracting the environment from the user in such a way that they are only concerned with creation of the media file and actual hardware setup themselves.

Running on top of this framework is five separate applications, each with a unique task. The ZESEditor is a post production media editor that reads in a QuickTime Media File and allows the user to insert device commands into the media timeline. Our custom ZESPlayer then reads in these tracks during playback and sends data to the ZESController, which runs in the background and is ultimately in charge of relaying the data to and from the wireless devices. The Pluginator creates plugins to be used by the ZES Framework to provide a general way to add new functionality and new devices to the system without having to write any new code. The Configurator is run during network setup to map active wireless connections to the groups defined in the ZESEditor application. This is intended to allow for easier setup and allows for easier changing of devices from one setup to another.

Figure I-1 outlines the overall data flow in the application suite. Starting with the Pluginator, a plugin is created for any devices needing to be used in the final setup. It is left up to the devices developer to supply the plugin since they will be most familiar with the commands the device is capable of receiving. The plugin is then used in the ZESEditor to create a ZigBee enabled ZES Media File. This file is then read in by the ZESPlayer, and in conjunction with the ZESController, commands the wireless devices. These devices are associated with their ZES Track/Group before the Player and Controller are started using the Configurator tool.

Technologies

The project heavily involves Apple specific technologies, mostly Cocoa. Cocoa is the interface into Apple's operating system and its primary language: Objective C. This system was used for all the applications except the network controller, which was a C++ daemon which then took on Cocoa components to support AppleScript. Objective C applications were able to use built in frameworks to manipluate the XML data contained in much of the project's external files. The Controller turned to Xerces. References for these technologies are included at the end of this paper.



3b) The Controller accepts Setup and Admin Pkts from the Player, sends back input pkts and manages data flow over the ZigBee network

2) The Configurator utility is used on site to assign Zigbee enabled devices to the groups in the target media file. It reads in the Group/Plugin listing and exports a Controller Configuration file

1) The Editor reads in a raw media file and exports the ZES Media file and Device Group/Plugin Listing

0) Creates plugins to each ZigBee enabled device for use in applications

3a) The Player reads in the ZES Media file and Device Group/Plugin Listing. It sends Setup and Admin Pkts to the Controller to trigger ZigBee events throughout the duration of the media file

Figure I-1: Applications and Data Flow

ZigBee Wireless

ZigBee Wireless technology is based on IEEE Standard 802.15.4. It is designed to provide low power, low bandwidth wireless communication for control and sensor networks. It runs on Industrial, Commercial, and Medical bands at 868 MHz, 915 MHz, and 2.4 GHz.

The technology employs three different device types: coordinator, router, and end device. The coordinator is the head a ZigBee wireless network. The router passes data from node to node, and the end device is the least expensive and complex of the three. Intended mostly for wireless control and sensor networks, ZigBee devices are low bandwidth and low power, able to last on one battery for a duration on the order of years.

Apple QuickTime

QuickTime is Apple's media type which is capable of reading many media types such as MPEG, MP3, JPG, AVI etc. While the first iteration of the ZES Media System writes to a separate

XML file, QuickTime supports the creation of custom data tracks. While no longer in the design, the QuickTime file format was initially chosen because of its support of custom data tracks in a media file. This would permit the ZESEditor to simply add a new track with encoded information to the file and have it wrapped in one package. This is would be a requirement for a commercial application, certainly. However, it was not implemented in this design.

The QuickTime API, on the other hand, was used extensively in the Editor and Player applications since it provided an easy and straightforward interface into the media files and media playback.

ZES Framework

The ZES Framework initially consisted of three libraries which provide all the functionality to interact with ZigBee enabled devices. While originally all three of these libraries were referenced to complete the framework, the Plugins Library was left out of the final development due to a potential conflict that would have resulted with existing command standards that still need research. Plugins are currently implemented through a reader that is shared between many of the applications requiring its use. The remaining libraries have beenrefined over development to more specific tasks than were originally intended.

ZESLib

Provides functionality that permits applications to communicate with the Network Controller process. This is done through the use of the `ZESManager` class. It also provides data types for sending information and control signals to and from the two applications in the form of `SetupPacket` and `AdminPacket` data types respectively. This library is best described as providing a mediator between the application needing to control devices and the application running any network management.

ZigBeeLib

This library is what enables interaction with the actual ZigBee network. It contains `Node` and `Group` classes for logical grouping of devices. The `NetworkManager` class communicates with the network and the `NetworkStatusManager` tracks network status such as pinging devices. While the class definition exists, the `NetworkStatusManager` is not implemented in the framework. Insufficient information was available to try to run network maintenance. For instance, the Embernet light ballasts from Phillips only had limited documentation and only a single command was available for use. Before release into a commercial environment, this piece would need to be implemented as completely as possible to ensure a reliable environment.

II) Applications

ZESEditor

Motivation

The editor in the ZES Multimedia system, codenamed the ZESEditor, allows for the creation of scripts to control multimedia devices in synchronization with a movie. Using a graphical user interface (GUI) and a multimedia timeline, users can place events (key frames) on the timeline, which will be sent to their respective devices. Tracks give a convenient grouping of devices to the user that allows for convenient visualization of how sequences of events affect ZigBee enabled multimedia outputs. Using tracks and key frames, a user can rapidly and intuitively develop multimedia scripts (ZESMovie) for use with the ZESPlayer.

QuickTime

The ZES Multimedia system extensively uses Apple's QuickTime application programming interface (API) for Cocoa, as it allows for rapid development, many file formats, and a stable intuitive API. Thus, the ZESEditor heavily relies on a QuickTime framework. In doing so, many different file formats can be used with the ZESEditor, including: .avi, .mov, .mp3, .3gp, and others. In addition, since QuickTime supports a wide variety of compressors/decompressors (CODECs), the ZESEditor also supports many formats including MPEG1,2,4, H.262, H.264, Cinepak, and many more. Furthermore, the container formats and CODECs supported can be extended by the user with QuickTime components; doing so is seamless to the ZESEditor and requires no additional work than that of getting them to work with QuickTime. Through components, the ZESEditor can notably support Windows Media Player 9 and DivX. The user associates a movie to the current project, which can theoretically be of any of the aforementioned types, but is currently limited exclusively to .mov and any CODECs the container supports (nearly all of them) to simplify the design for development. The ZESEditor then adds a script that is read by the ZESPlayer that adds ZigBee functionality to the QuickTime movie.

Design

Users who have used iMovie, Final Cut Pro, or some other timeline video editor should be very familiar to the concepts of tracks and key frames. Tracks are an abstraction mechanism that allow the user to group devices. A user creates a track and associates it to a specific device type. All events that occur on that track will be sent to a single or group of devices of the selected type at the logical time it occurs on the timeline. For example, a user who has two separate light-boxes and a scent generator can make three different tracks that contain events for each device. Thus, the user will be able to control each device independently. A user can also create additional tracks that represent combinations of devices previously associated. For instance, if a user has two light boxes, A and B, the user can create one track for A, one for B, and another that controls both A and B. There is no hard limit to the number of allowed tracks in a ZESMovie, which are represented by blue rectangles that appear on the timeline.

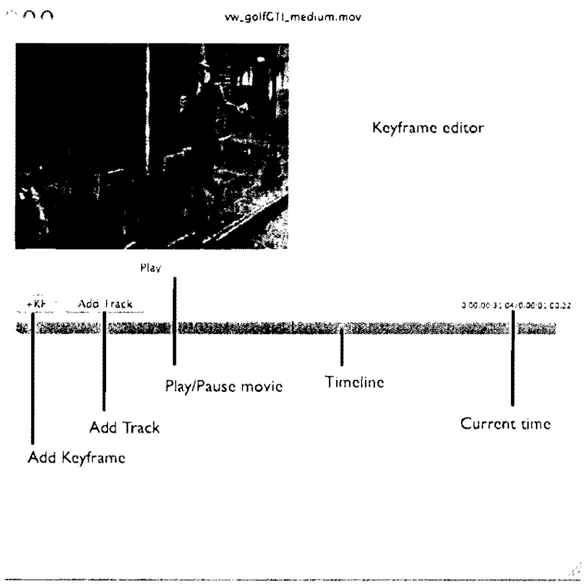
The other essential mechanism users utilize in ZESMovie creation is the key frame. The key frame represents a "significant time" in a movie file. At these key frames, users assign command outputs to be sent to ZigBee multimedia devices. For example, if a light box needs to be dimmed at 30 seconds into the movie file, the user would select the track that corresponds to that light box and add a key frame. The user would then assign the values corresponding to the command to be interpreted by the ZESPlayer during playback. Each track has its own set of key frames, meaning that the user can send events to different devices independently. If, for instance, the user wants to fade light box A to 50% at 15 seconds, light box B will be unaffected unless the user adds a keyframe to its track at the same time. Key frames are represented by green boxes, the currently selected key frame being yellow instead of green. The attributes associated with the currently selected key frame appear in the list view at the top right of the editor's user interface (UI).

When the user has finished placing tracks and key frames that represent the multimedia output events associated with the movie file, the user exports the track to a .zes file. These files are machine generated XML files that can be read by the ZESPlayer to recreate the track and event structure. This file also contains the full path filename of the movie file associated with the ZESMovie script. In its current implementation, any file format that QuickTime is capable of playing can be associated with a ZESMovie. An example ZESMovie is given below:

```
<ZESMovie movie="/yakkos_world-26-09-05.mov">
      <ZESTrack name="track1">
            <ZESKeyframe time="71813">
                  <property propertyName="fadeToLevel"
            propertyValue0="0" propertyValue1="2">
                  </property>
            </ZESKeyframe>
            <ZESKeyframe time="96099">
                  <property propertyName="fadeToLevel"
            propertyValue0="0" propertyValue1="3">
                  </property>
            </ZESKeyframe>
            <ZESKeyframe time="54839">
                  <property propertyName="fadeToLevel"
            propertyValue0="0" propertyValue1="1">
                  </property>
            </ZESKeyframe>
      </ZESTrack>
      <ZESTrack name="track2">
            <ZESKeyframe time="71813">
                  <property propertyName="fadeToLevel"
            propertyValue0="5" propertyValue1="4">
                  </property>
            </ZESKeyframe>
      </ZESTrack>
</ZESMovie>
```

The `ZESMovie` XML tag represents a `ZESMovie` script file, where its movie parameter is the actual QuickTime movie file associated with the script. `ZESTrack` tags represent tracks the user laid out in the editor's UI. The name property is currently arbitrary and sequentially generated, though this should be user definable in commercial applications. The name allows the user to more conveniently associate devices to a track in the Configurator. `ZESKeyframe` tags represent an event that occurs on the timeline. The time property is a `long long` C type used internally by the QuickTime API in referencing temporal location of the events, and is used by the ZESPlayer to trigger events. Finally, property tags represent the actual commands and values associated with an events time. A single event can have multiple properties associated with it.

The interface for creating tracks and key frames is highly intuitive and should be familiar to anyone who has previously used a timeline movie editor. To create a new ZES file in the editor that is associated with a movie, select "New" from the File menu. If a project is already loaded, any unsaved changes will be lost. To add a track, the user clicks add track and selects from a dropdown menu the type of device associated with that track. Clicking "+KF" adds a key frame to the currently selected track at the current time. The properties of the currently selected key frame appear in the keyframe editor panel in the top right of the UI. Tracks not currently selected and their keyframes are dimmed When the movie is properly edited, the user selects "Save As..." to create the ZES track. To load an existing .zes file for editing, select editing. This will discard any unsaved changes in the current project.



The types of devices a track can be associated with vary depending on which plugins are installed. Plugins are created with the Pluginator and define what parameters and how many arguments are associated with a given device type. This provides a flexible and expandable platform for generating scripts to control multimedia devices. Plugins are loaded from the directory /Library/EATON/ZESPlugins upon the ZESEditor's launch from the Finder and must be restarted upon adding new plugins. A list of available plugins appears when the user adds a track.

Currently, the ZESEditor allows for script generation and is fully functional for creating scripts. Features have been removed since the preliminary design, notably zooming and keyframe jumping. These features are not necessary for creating .ZES scripts and effort was placed elsewhere. Also, drawing routines have been updated integrated with data storage of tracks and keyframes to more easily track bugs and use less memory. In addition, this fixes memory leaks that could occur.

ZESPlayer

Motivation

In the suite of ZES applications, a necessary program to play user-created content is the ZES Media Player. This Media Player is the main tool that the user will employ to display his/her ZES Media to their audience, whoever they may be. In addition to this main role, the ZES Media Player will also serve as go-between for the information listed in the ZES Media File and the Controller process which sends data over the ZigBee network. It is the Player's job to read the data in the .zes media file, interpret it, format it, and send it to the Controller process in such a way that will allow the Controller to easily send commands to ZigBee devices. Also, future work hopes to bring the ability for the Controller process to send information back to the Player about network conditions. With this ability, the Controller could inform the Player about network latency, and the Player could issue commands earlier to make up for that circumstance.

QuickTime

The Player application is built using Apple's XCode integrated development environment. It is written in Objective C, an extremely object-oriented language. This language, particularly when used with XCode's intuitive interface, has a short learning curve and allows for rapid development. In addition, there was some experience in the development group with this language and its framework.

Within this framework, Apple has made open source much of its work with the QuickTime API. The first version of QuickTime was released in late 1991 as an addition to some already existing software. Over the years it has been through many revisions and has become very stable. A QuickTime enabled mediaplayer, such as the ZES Media Player, is capable of playing a wide range of audio and video formats including but not limited to: AIFF, CDDA, MIDI, mp3, m4a, m4b, m4p, QCELP, ULAW, ALAW, WAV, 3GPP, 3GPP2, AVI, BMP, DV, GIF, animated GIFs, H.261, H.263, H.264, JPG, Photo JPEG, JPEG-2000, MPEG-1, MPEG-2, MPEG-4, .qtz, .mov, PNG, TIFF, TGA, QTCH.

User Experience

The goal of the ZES Media Player is to be as simple and straightforward to the user as possible. Upon startup, the user selects which ZES Media File he/she would like to open using a standard Apple OS X OpenPanel. After doing so, the media plays in full screen with the cursor hidden, so that the only thing on the user's monitor will be the media. Most likely, the user will want to connect their computer to another display such as a projector or TV to allow a larger audience to view the media. A screenshot of the Player in action is show in Figure 1.

Figure 1: Movie playing in full screen with progress bar shown. This is a screenshot of everything on the monitor.

At certain points during the movie, as indicated in the ZES Media File, the Player will cause ZigBee devices to perform whatever operation that they are supposed to do at the point. This may be turn on or off a lightbox, or start dispersing some scent throughout the room. If the user wishes to see at what point in the movie he/she is, or to skip the movie to a particular point, they can push CTRL+ENTER to display the progress bar and make the cursor reappear. Repeating this command will remove them both. The user also has the ability to step through the movie frame-by-frame, stop, pause, load a new movie, or to skip to a certain point in the movie. This last action is executed by showing the progress bar and then clicking on the point in the movie he/she wishes to skip to. A diagram displaying user I/O is given in Figure 2.

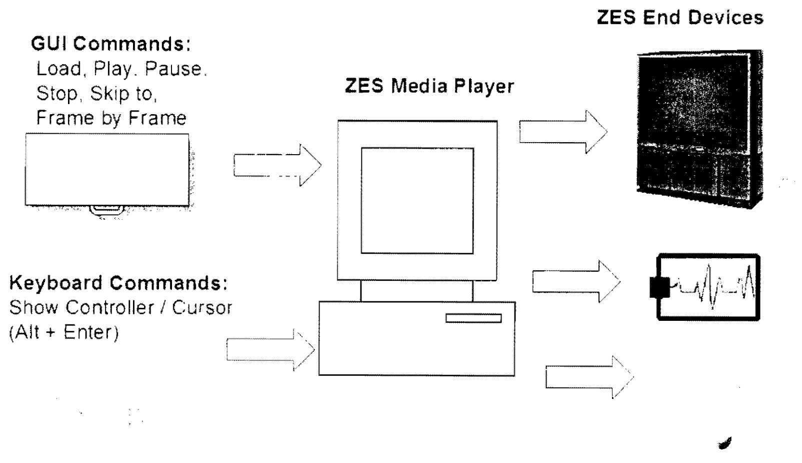
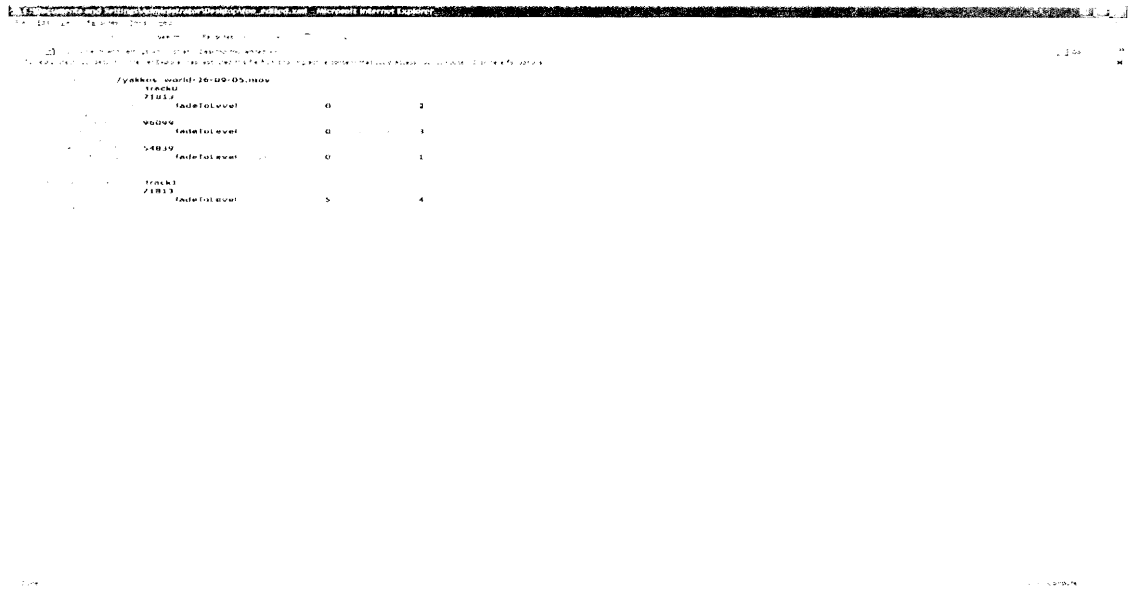


Figure 2: Overall user interaction with ZES Media Player.

<u>Program Level I/O</u>

While to the user the ZES Media Player might seem simple, what is actually going on behind the scenes is quite a bit more complicated. First of all, the Player must read in the .zes media file which is actually in XML format. This file contains a reference to the movie,which the player then opens and displays in full screen. An example .zes media file is shown in Figure 3.



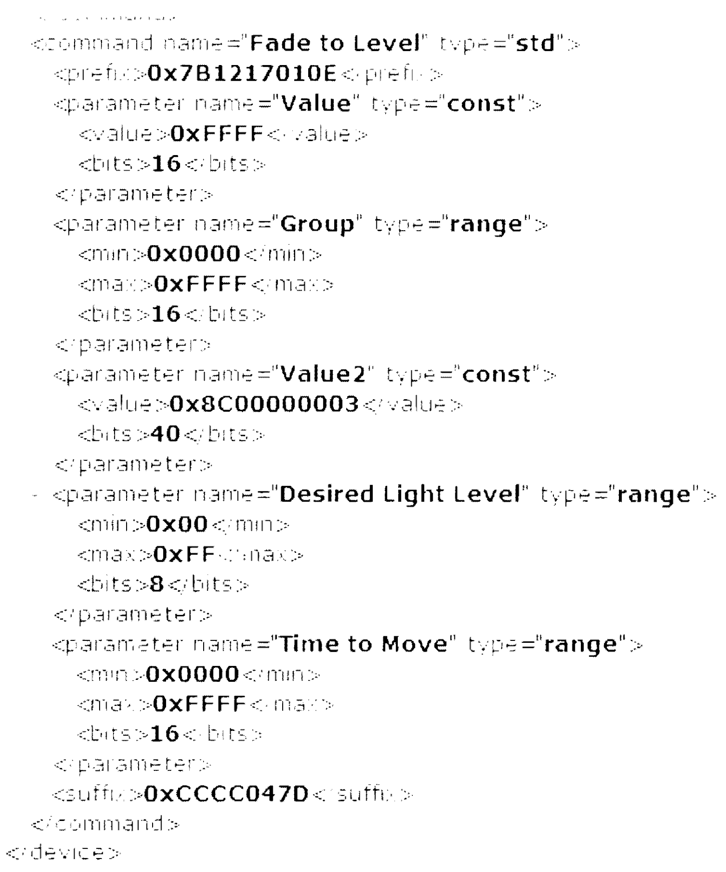
```
/yakkos_world-26-09-05.mov
    track0
    71013
        fadeToLevel        0        2
    96099
        fadeToLevel        0        3
    54839
        fadeToLevel        0        1

    track1
    71813
        fadeToLevel        5        4
```

Figure 3: Sample .zes media file. Note that the movie "/yakkos_world-26-09-05.mov" is given as the root of the XML tree and would be the movie file that the Player displays.

Each trackname references a particular type of device and the player opens the corresponding plugin .plg file which is also in XML format. The plugin file is used to translate generic command names given in the .zes file to the appropriate string that will be sent to the controller process and eventually on to the ZigBee end device. For example, the Phillips Equos Fluorescent Lamp Ballasts contain a long prefix and suffix to the actual values that govern how the light will display. This suffix and prefix information is listed in the .plg file. So when the Player parses the .zes media file and finds a track called Phillips Equos Fluorescent Lamp Ballasts it looks in the associated .plg file for the actual hex command that should be sent to the controller process.

Figure 4 shows an example .plg plugin file.



```
  <command name="Fade to Level" type="std">
    <prefix>0x7B1217010E</prefix>
    <parameter name="Value" type="const">
      <value>0xFFFF</value>
      <bits>16</bits>
    </parameter>
    <parameter name="Group" type="range">
      <min>0x0000</min>
      <max>0xFFFF</max>
      <bits>16</bits>
    </parameter>
    <parameter name="Value2" type="const">
      <value>0x8C00000003</value>
      <bits>40</bits>
    </parameter>
  - <parameter name="Desired Light Level" type="range">
      <min>0x00</min>
      <max>0xFF</max>
      <bits>8</bits>
    </parameter>
    <parameter name="Time to Move" type="range">
      <min>0x0000</min>
      <max>0xFFFF</max>
      <bits>16</bits>
    </parameter>
    <suffix>0xCCCC047D</suffix>
  </command>
</device>
```

Figure 4: Portion of .plg file that gives details about the command "fadeToLevel" for Phillips Equos Fluorescent Lamp Ballasts. Note the prefix and suffix fielas.

The .zes media file also contains triggers that are commands for devices and that are time stamped relative to the movie to allow for synchronization.

Along with reading and using the .zes file created by ZES Editor, the Player also interacts with the Controller process through the use of sockets. After selecting the .zes file, the Player reads in all the triggers throughout the movie and informs the Controller of all the ZigBee events that will happen throughout the movie. When the controller has taken this information in, it lets the

Player know that it can begin playing media. Whenever it gets close to a trigger in the movie, it sends out a packet that tells the Controller which event is supposed to occur and in how many milliseconds it is supposed to happen. In a sense, the Player tells the Controller "execute command #7 in 2 seconds". Upon further development, it is planned that the Controller will be able to respond to network conditions by letting the Player know about round trip times. The Player could then react by changing the above example command to "execute command #7 in 5 seconds". A diagram showing Player-Controller communication is given in Figure 5.

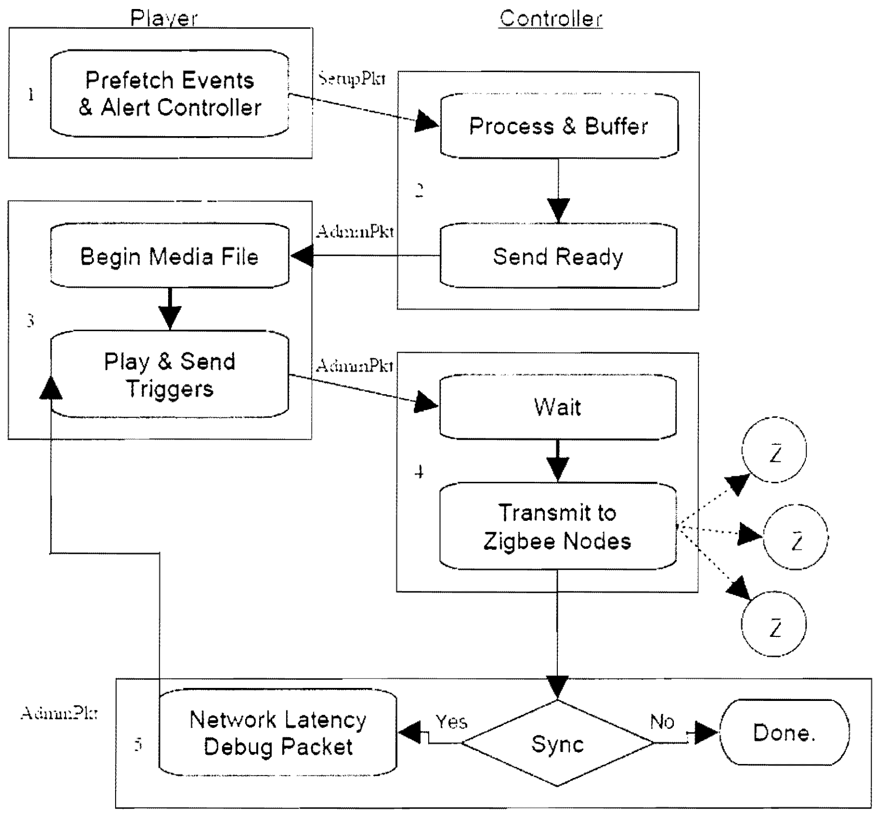
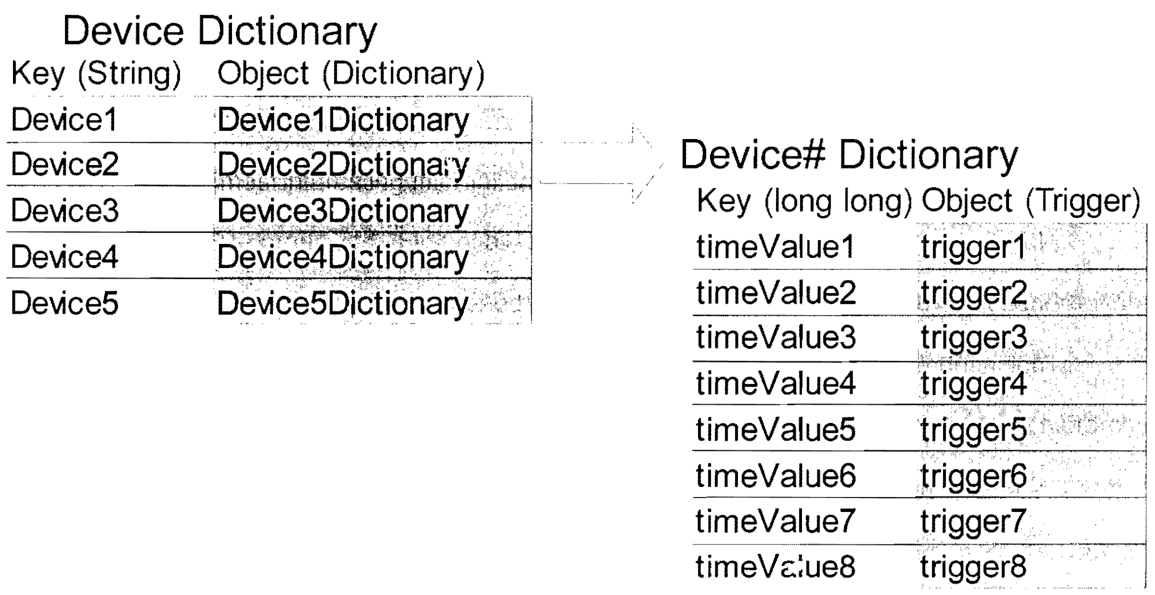


Figure 4: Player-Controller communication.

Internal Data Structure

XML was chosen as the format of choice for input/output files between ZES processes due not only to its flexibility, but also to the fact that Apple has an excellent XML Parser library at the developer's disposal. The event driven XML parser, when invoked, essentially has a callback each time that a particular string is encountered. For example, the .zes media file in Figure 3 activates a callback function when the "ZESMovie" tag is encountered. At that time, the Player copies the string in the tag and loads the movie that is indicated by the following string, inthis case "/yakkos_world-26-09-05.mov".

By far the most interesting internal data structure in the Player application is the way in which the triggers are organized. First of all, it should be pointed out that there is a useful `NSDictionary` structure that when queried with a key, returns an object. Both object and key are generics. Relating this back to real worlddictionaries, if you want to find a given definition (a particular object), then you look up the word (key) in the dictionary. At the top of the data hierarchy is the Device Dictionary which is keyed on trackname(a string) and its objects are `NSDictionary`s themselves. Each of these `NSDictionary`s is keyed on timeValue. a long long int, and its objects are triggers which contain information about the event that some ZigBee device should be doing at that given timeValue. Figure 5 displays the data structure more clearly.


Device Dictionary

Key (String)	Object (Dictionary)
Device1	Device1Dictionary
Device2	Device2Dictionary
Device3	Device3Dictionary
Device4	Device4Dictionary
Device5	Device5Dictionary

Device# Dictionary

Key (long long)	Object (Trigger)
timeValue1	trigger1
timeValue2	trigger2
timeValue3	trigger3
timeValue4	trigger4
timeValue5	trigger5
timeValue6	trigger6
timeValue7	trigger7
timeValue8	trigger8

Figure 5: Diagram of how Triggers are organized. The Device Dictionary is top level. When queried with a trackName, it returns a Dictionary. This dictionary is keyed on the time in the movie the trigger is supposed to fire and its object is a trigger object.

The long long int called timeValue now requires some explanation. QuickTime keeps up with times using a class called `QTTime`. This class has two members:a `long long int` called timeValue and an `int` called timeBase. The timeBase is a number that represents how many frames occur in one second for a particular movie. This varies from movie to movie. Some HD movies play at a timeBase as high as 600 fps while others play as slow as 10. The timeValue represents the current frame number. So for instance, with a timeBase of 600, if the current timeValue is 900, then 1.5 seconds have elapsed at this point during the movie. The default setting for the timeOffset for events is two seconds. This means that the Player will send out a command to the Controller two seconds before it's actually supposed to occur in the movie. The time that this packet is supposed to be sent out is then:

timeValueWhenTriggerShouldBeSent = triggerTime – 2/timeBase

However, it is impossible to catch every time that a frame passes in a movie. So it would be impossible to catch the exact timeValue when the trigger should be sent out. To solve this problem a timer was setup that fires every 0.1 seconds. Although it is not guaranteed to fire at exactly this rate consistently, it is close enough for this application. When the timer fires, a routine is called that gets the current time from the movie. It then checks every trigger in all the dictionaries to see if there is 2 seconds or less before that trigger needs to be sent out. If so, it continuously tries to send the trigger until the Controller receives it.

<u>Remaining Work</u>

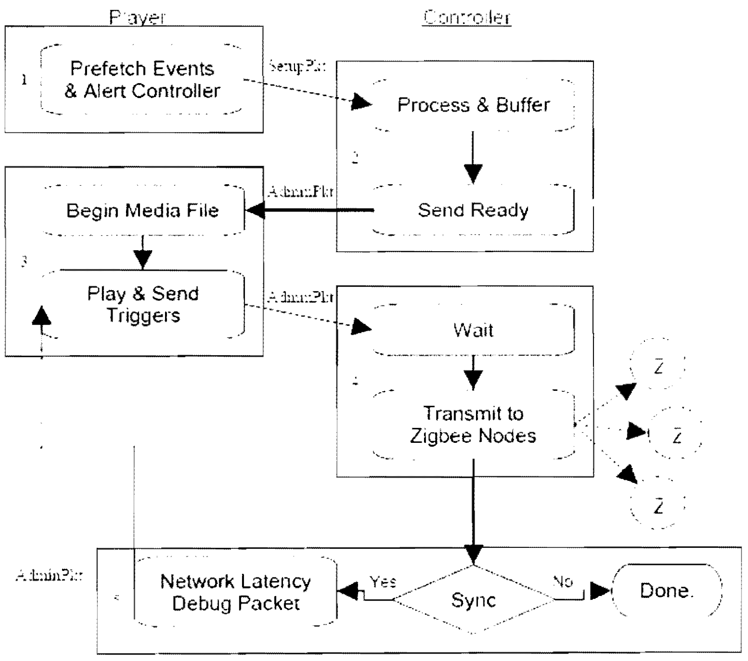
At the current development point, only a few things still need to be fleshed out. Right now, some data involving the plugins is hardcoded into the Player. Once the plugin XML parser is completed, the Player will utilize this class to read in the appropriate data from the .plg files that are referenced in the .zes file. Also, it would be favorable if some other process, probably AppleScript was able to activate the Player. If this was worked out, then input from an external device such as a motion detector would be able to start the Player, play the desired movie, and send out triggers as desired. Further enhancements may enable the Player to play one movie and send triggers until some external event occurs at which time it would reset and begin to play another movie with its associated triggers. All of these goals are realizable in a short amount with the right background information.

Network Controller

Overview

The Network Controller is responsible for managing the wireless ZigBee network and is implemented in a modular fashion. Its main functions are to maintain the network, send packets from the Player to the ZigBee nodes, and receive inputs from nodes and relay the message to the Player. The three interfaces of the Controller are communication with the player, sending packets to output nodes to cause a remote event, and receiving an input based on a remote event. The application is multi-threaded and uses the ZigBeeLib heavily.

The wireless basis for this project is the ZigBee wireless standard, which is based on IEEE standard 802.15.4 and is governed, at least in part, by the ZigBee Alliance. ZigBee is aimed at short-distance communication for low-power embedded systems. It is capable of operating on three frequencies and has a lower data rate, so it is not meant for large packets or streaming data. ZigBee features acknowledgment packets to ensure reliable data transfer and has secure network capabilities. One of the most important features is the ability to avoid other wireless networks, including Wi-Fi, allowing a layered approach to the wireless domain. Unfortunately, the Philips ballasts used do not provide these features and other devices were unavailable.



Communication between the Controller and the Player begins with setup packets sent from the Player. These packets are sent before the video begins playing to allow all outgoing packets to be queued. The packets include everything necessary to set up the ZigBee packet and also include a buffer identification number, as issued by the Player, which will trigger the send of that packet in the queue. Once all the packets are queued up, the Controller relays back to the Player that it is ready and the media file may now be started. As the file plays, trigger packets are sent to the Controller containing the buffer ID of the packet to be sent and the offset time to wait before sending that packet. To handle this offset, another queue of packets which have been triggered and are waiting the offset time is created. This is necessary because the offset is handled through an alarm.

When the first trigger is received, an alarm is set for the indicated offset. As more triggers are received, they are added to the priority queue with the `bufferID` and the actual time the packets should be sent. As the priority queue is sorted, if the top packet in the queue changes, the timer is reset to the remaining time beforethat packets transmission. This is always done by comparing the time the packet should actually be sent to the current time. The offset time value is lost after the initial setup, and triggers all become relative to the actual time of day. The `sendpacket()` function is called when an alarm is raised and it immediately sends the packet in the `bufferQueue`. It then checks the next packet and sends all packets that are overdue in order to keep up when many packets must be sent rapidly. If the network is lagging more than half a second, it is flagged and a statement is printed with how far behind the program is. After all necessary packets are sent, the timer alarm is set for the next packet to be sent.

The media file specifies how many packets to initially queue and maintain thereafter. The Player must wait for all packets to be queued before initiating media playback, as describedabove. Should the Controller not be able to keep up with the requests of the Player, a Network Latency Packet has been defined but not implemented. It could be used for some adjustment in the playback or offset times, but this was beyond the scope of this project.

The `motionDetect()` function is the only implementation of ZigBee input used in the project. It uses a motion detector that has been made into a ZigBee node by Eaton Electrical. TI ZigBee development boards are used on the receiving end to interface with the Controller application. The group had no control over the operation of the motion detector so some adjustments had to be implemented. Sometimes when motion is detected, the device will send three separate packets that motion was detected, each a few seconds apart. This prompted the addition of a temporary disable to keep these repeat packets from being incorrectly interpreted. The time for this value can be adjusted and fifteen seconds yielded the best results over an extended period. Threads had to be used to accomplish this, as the alarm is already in use elsewhere in the Controller. In addition, another disable exists to allow the player to turn motion detection on or off at will.

Some problems were experienced in relaying detected motion to the Controller. The development board correctly shows when the packet is received, but the message read by the application does not conform to the standards set in the development board's documentation. Moreover, the data read is seemingly random and differs from computer to computer. The problem is believed to be either in the USB to Serial conversion cable or in the development board itself. To work around this, whenever data is read from the motion detector, it is interpreted as motion. The only thing that requires consideration is that the development board must be turned on before the Controller application is run.

The output nodes currently being used are light ballasts made by Phillips which operate on Embernet. The Controller sends packets of node-dependent information when specified. For the light ballasts, the information includes values for fade-to-value and fade-over-time. A fade value of zero causes an instantaneous change and there is a relatively significant delay to turn the lights on, especially when they are cold, or not currently in an 'on' state. To lessen network traffic, nodes are assigned in groups. A group may consist of only a single node or a large number of nodes and each node may be a member of multiple groups. The `addGroup()` function is used to create a new group, and these groups are referenced by a name, which is a string. `addNode()` is used to create new nodes and `addToGroup()` is used to add these nodes to groups. This setup allows one node to be added to multiple groups and also for each group to have multiple nodes. The advantage of multiple nodes in a group is the capability of artificial grouping. In a real environment, the device necessary to reprogram and group ballasts may not be available. The Controller allows a new group to be created with all the different nodes available and when a packet is meant for such a group, traversal of the group is done and the packet is sent to each node individually. This is a convenience, but it should not be abused, as it creates more network traffic. Nodes and groups can also be removed, but there isn't currently a known need for this. The ability to send to a single node directly is also implemented, but this also does not have a currently-defined purpose.

Originally, the status of the network was to be maintained by the Controller. This is still a worthwhile endeavor, but was unable to be completed in this project. The return packets from the light ballasts are unknown and time was not available to attempt to reverse engineer them. Even still, the only error found outside of the light ballasts being off has been a buffer overflow, and network status does not currently have a way of dealing with this. For the motion detector, even a heartbeat request to the development board just to see if the serial connection is active yields no response, even when data can be read when motion is detected. This section of the project ended up not being feasible but should be one of the areas of emphasis should the project go forward.

The shortcomings of the Controller ended up being more related to hardware limitations than anything else. The lack of a significant number of ZigBee nodes available made development with flexibility hard. Only the light ballasts on Embernet are currently supported and it would take a fair amount of work to handle the adjustments necessary to be cross-compatible. Hopefully the ZigBee Alliance will result in a more defined path for the handling of such things. The input works very well inside of what it is designed to do, but it is extremely inflexible and has minimal functionality.

Future improvements should be completed in several areas. The core of the Controller is graceful and efficient in the way packets are queued, sorted, and triggered. The setup of groups and nodes is much the same. Additions to the interface with the player should be made so the Controller receives more information about how to send a packet to a type of node, rather than having it hard coded for Embernet as it is now. As additional input nodes become available, they should be implemented and tested to achieve more robust functionality. The network status is a big consideration going forward and should be looked at extensively. Some internal error checking could also be done to proactively adjust situations that could cause buffer overflows in the light ballasts or other performance degradation.

The difficulties experienced were not what were expected to take the most time. The motion detection was very time consuming in the beginning just to get the three devices to communicate at all. After that it was not much of a problem. The assumption had been made going in that it would be easy to handle a lot of packets with varying offset times, and not much thought had been given as to how that would work. The implementation of this was not trivial and it was difficult to trace the cause of results that differed from what was expected since errors appeared only when the internal timer reflected a specific number of milliseconds. Seemingly spontaneous errors resulted during this phase of programming. The slow response time of the ballasts required careful choosing of time offsets and end results to see if things actually were working.

The Controller handles the most unpredictable part of the project. Developers of the Player, Editor, and Pluginator can demand the requirements to use their products, but the ZigBee side is open to all future developments of the technology and thus will require adjustment as new devices are created in order to remain compatible. As an overview, the Controller must maintain the ZigBee network, send packets with node-dependent information to the output nodes, receive input packets from the remote event-triggered nodes, and communicate with the Player to assure on-time and reliable exchange of information between all of the above.

Pluginator

Motivation and Design

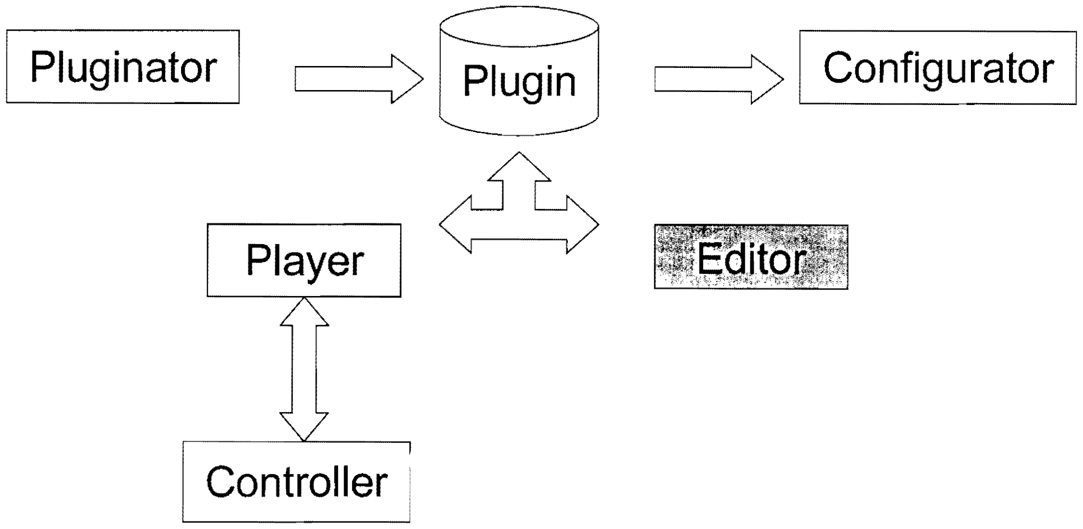
The Pluginator's goal is to allow the end-users of our program suite to quickly and easily add new devices to the system by providing a simple interface for plugin creation. A primary target audience for the Zigbee Event System is in advertising, and if someone working on a trade show setup wants to add a new and unique device to their exhibit, such as a scent generator, they shouldn't need an in-depth integral knowledge of how our particular tools work internally.

We decided the easiest way to make the ZES extensible was to use a plugin based system. Device details are fed to the Editor, Player, and Configurator through a standalone container. The Editor uses this information to allow the user to add commands in the form of key frames to tracks within the media file. The Player uses the plugin to format the commands that it sends to the Controller in the setup packets, and the Configurator uses the setup information embedded in the plugin to trigger devices to activate for detection. Plugins are therefore modular and provide a standard communication framework for the components in our system.

To make the plugin files easy to read and well structured, we chose to use XML tags to describe a particular device and its commands. The Pluginator is an end-user application which provides a graphical interface to create these XML plugins, where the primary design goal has been a quick learning curve. The plugin structure is reflected in the design of the GUI. Each plugin has device properties, a command list, and a parameter list for each command.

The command list should be able to store an unlimited amount of commands, though each plugin should have an associated setupOn and setupOff command. These commands are to be used by the Configurator to switch devices On and Off for the detection phase. For every command, both prefixes and suffixes can optionally be defined for the case when an introductory or trailing bitstring needs to be defined in a packet. Parameters are tied to commands and provide the structure for Zigbee (or Embernet) packet formatting. Parameters can either be constant values, which are set when the plugin is created and inserted into the packet by their respective position within the parameter listing, or range values, where the plugin creator sets up a minimum and maximum acceptable value. This range gives flexibility to users when creating their timeline within the Editor by providing keyframes with modifiable attributes.

Prefix and suffix values are used in combination with the parameter values to format the packet that the controller will eventually send. While the player reads the timeline, it takes the key frames added by the Editor and formats a packet with the information from the plugin. It then encapsulates this payload in the setup packet that is sent to the controller to initiate a command sequence.



Plugin Interaction with System Components

<u>Implementation</u>

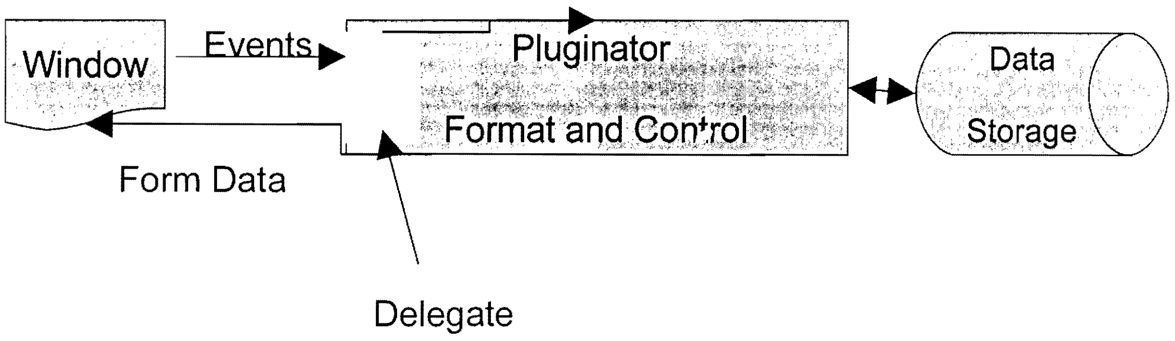
Xcode paired with Cocoa and Objective-C was used to build the Pluginator project. The GUI consists of three main sections: device properties, command list, and associated parameters. An `NSTableView` object was chosen to display the commands, as it supports scrolling natively and would be ideal for listing a large number of commands. We subclassed the object to make a child object which retained the same properties and allowed us to add functionality to the table. Commands can be added with a button and deleted simply by selecting the unwanted commands and removing them. When the program is started, the bare essentials for a plugin are added to the forms in the table and parameter list.

In Cocoa, the radio buttons are actually `NSButtonCell` objects and the traditional exclusive selection seen in radio buttons but must be implemented in software. When the user clicks on the command, the respective parameter list is pulled up in the drop-down combo box below the list. There are also fields for each parameter, including the const or range type selection, the min and max values (max is unselectable when the const type is chosen), and the number of bits to allocate for each parameter. The number of bits indicates the padding that the player should use when constructing the data payload to send to the controller.

Both the `NSTableView` and (optionally) `NSComboBox` use informal protocols for populating their forms with data. These `DataSource` protocols are simply a set of methods that an object acting as a data storage center should provide. The object is set by the `NSTableView` and `NSComboBox` objects upon their waking in "awakeFromNib". The functions inside the `DataSource` object provide the references to the objectstrings that are used to fill the command and parameter lists. For the table, the data request occurs at program start and any time an additional rowis added. For the combo box, the data source needs to be signaled when the table selection changes so it can provide the appropriate data when the user clicks the dropdown arrow to expand the list. This type of signaling will be discussed more later and is derived from the action event that occurs in every `NSControl` object. Because the data in the table and combo box is so entwined, we chose to use the same data source for each object and implemented the appropriate methods for each.

Inside the `DataSource` object, several `NSMutableArrays` are used to store the command name strings, prefix and suffix values, and setup status. There is also an array of `NSMutableArrays` which holds the parameters and their fields for each command. The setupOn and setupOff properties are stored simply through a set of integer values which are the indices of the command list items currently tagged as setup.

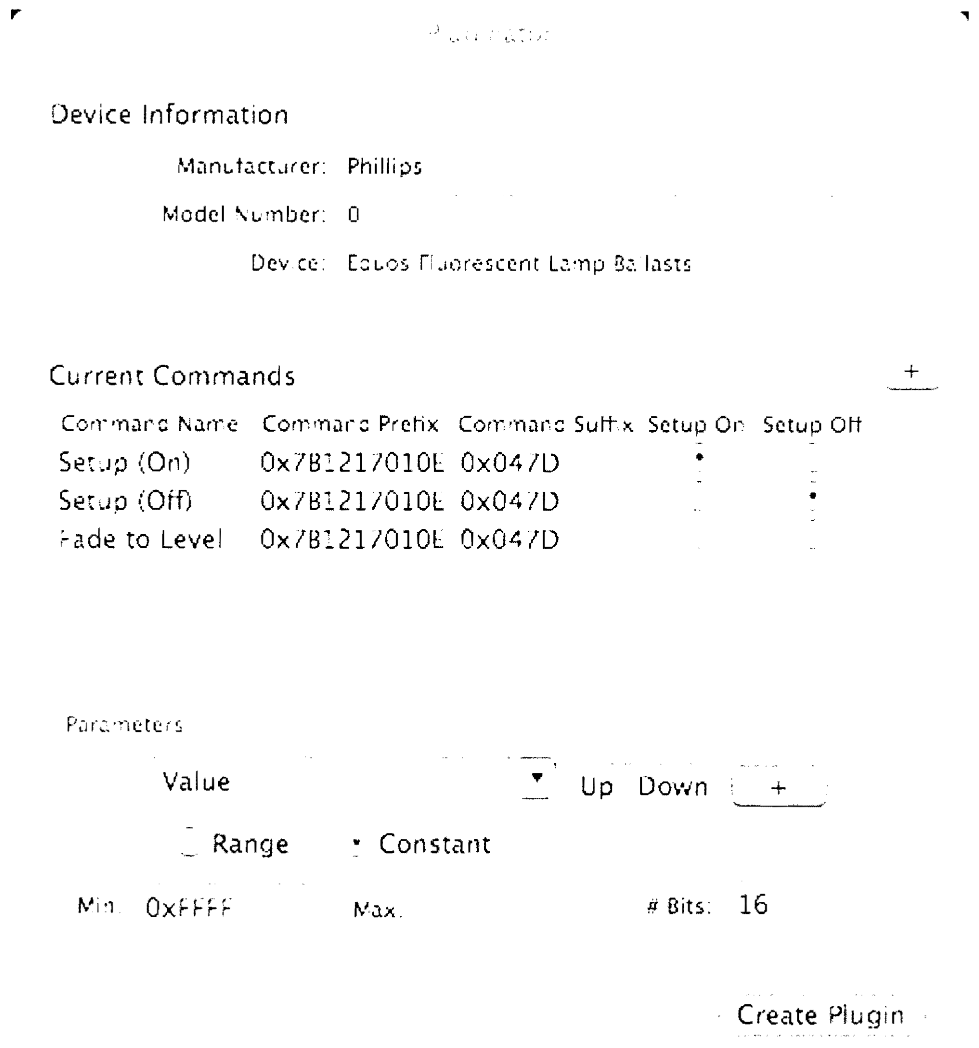
Cocoa works with forms on the windows through wirings, data sources, delegates, and action/target combinations. Wiring to the text fields describe their outlet in the main object which coordinates and houses the individual form objects within the windows. This provides one way data passing to fields themselves, as the `NSTextField` object inherits the "setStringValue" method from one of its parent classes, `NSControl`. The data source provides data to the forms inside the list and combo box in a similar way, except these are triggered automatically by the `NSTableView` object itself. Delegates provide a way for objects to pass messages back to handling objects which tell them how to act when an event occurs, such as the user clicking on a new row in the table. In this particular case, the `DataSource` object acts as the delegate message receiver to alter the parameters sent to the combo box. Another delegate method is called when the text within the parameter `NSComboBox` is edited, allowing the Pluginator to check the parameter strings for duplicates before they are updated within the storage object. Delegate methods are optional, so the receiving object can choose which messages to respond to. The figure below diagrams the delegate to object interaction within the Pluginator.



Data and Event Passing in the Pluginator

Events notifying when a change occurs for the `NSTextField`'s are handled by the target/action combination, where an action method is called in the target object when the text box loses focus. These methods are also inherited from the `NSControl` class, and in the Pluginator they are set up in the main object which uses mutators tied to the `DataSource` object. As the user updates the forms, the appropriate changes are made inside the `DataSource`. Moderate error checking is performed on input values with the `strtol` function, which returns 0 on an error and sets the errno global error flag. This flag must also be checked as 0 is a valid input. Internally, the values are still stored as strings to allow for any length field but they are parsed as if they are longs.

A screenshot of the Pluginator set up with the light ballast plugin is shown below:



Device Information

Manufacturer: Phillips

Model Number: 0

Device: Equos Fluorescent Lamp Ballasts

Current Commands

Command Name	Command Prefix	Command Suffix	Setup On	Setup Off
Setup (On)	0x7B1217010E	0x047D	●	
Setup (Off)	0x7B1217010E	0x047D		●
Fade to Level	0x7B1217010E	0x047D		

Parameters

Value

Range ● Constant

Min: 0xFFFF Max: # Bits: 16

When the user requests to save the plugin, this data is molded into an XML tree object with references to leaf nodes stemming from the root node. A file location is chosen from a save dialog box created with the `NSSavePanel` object. This dialog prompts users to select a file with the plugin extension, *.plg.

A portion of the plugin for the light ballasts is shown below to clarify the XML layout:

```
<device name="Equos Fluorescent Lamp Ballasts"
manufacturer="Phillips" model="0">
    <command name="Setup (On)" type="setupOn">
        <prefix>0x7B1217010E</prefix>
        <parameter name="Value" type="const">
            <value>0xFFFF</value>
            <bits>16</bits>
        </parameter>
        <parameter name="Group" type="range">
```

```
                <min>0x0000</min>
```

We were unable to develop auto detection of devices. They currently have to be manually coded into a file and then read into the program. It may also be advantageous to allow the user to select multiple groups simultaneously. The user may also like to see a breakdown of the configuration thus far, not in the form printed to the configuration file, but in a form appropriate to a graphical interface.

```
                <max>0xFFFF</max>
                <bits>16</bits>
            </parameter>
            <parameter name="Value2" type="const">
                <value>0x8C00000003</value>
                <bits>40</bits>
            </parameter>
            <parameter name="Desired Light Level" type="const">
                <value>0xFF</value>
                <bits>8</bits>
            </parameter>
            <parameter name="Time to Move" type="const">
                <value>0x0000</value>
                <bits>16</bits>
            </parameter>
            <parameter name="CRC" type="range">
                <min>0x0000</min>
                <max>0xFFFF</max>
                <bits>16</bits>
            </parameter>
            <suffix>0x047D</suffix>
        </command>
```

"setupOn" and "std" are additional valid types for commands. These plugin files can be reopened inside the Pluginator through the use of the PluginReader object. A `NSOpenPanel` dialog box prompts the user for the input file with extension .plg to open. The PluginReader object was created to allow any program within the project to import these the header and implementation files and have complete access to the plugin. It defines a class called `PlgReader` that has one simple function: `parsePlgFile`. This function takes a reference to an `NSString` object that is the full path to the plugin to be parsed. For parsing the plugin, the `PlgReader` uses the `NSXMLParser` to handle most of the dirty work. `parsePlgFile` returns a reference to a fairly complex `NSMutableDictionary` with all the necessary information neatly organized. The format of this dictionary is best described by the diagram below; the actual data type of each element is listed to the right of the object's name.

```
%Plugins (NSMutableDictionary)
  - name (NSString)
  - manufacturer (NSString)
  - model (NSString)
  - @commands (NSMutableArray of NSMutableDictioanry)
     - name (NSString)
     - type (NSString)
     - prefix (NSString)
     - suffix (NSString)
     - @parameters (NSMutableArray of NSMutableDictionary)
        - name (NSString)
        - type (NSString)
        - min (NSString)
        - max (NSString)
        - bits (NSString)
        - value (NSString)
```

As it is written, none of the elements listed above are required for the plugin parser to work properly. This is just an exhaustive list of what the parse is capable of recognizing; if the element is not a member of the plugin, then that part of the structure will simply be NULL.

Shortcomings, Limitations, and Future Considerations

The Pluginator walks the line between requiring the user to have a detailed knowledge of the device they are creating the plugin for and restraining the user when possible by providing error checking. Extensive error checking is impossible, as there are cases when only the user knows which parameters should be constant values and which should be ranges. For instance, setup commands should be filled with mostly constant parameters, though ranges must be still allowed for parameters like group. Whether such inputs should be allowed is a decision that cannot be made by the Pluginator.

At this time, it is also up to the user to set the correct number of bits for a given parameter value string. This could be fixed in the future by counting the number of digits within the value string and determining the base of the number. Decimal, octal, and hex are currently supported for input values, but due to a limitation in the `strtol` function used, additional characters outside of the base set (ie: 'G' in hex) are ignored when appended to the end of a value string passed to the function. One way around this would be to use the long return value from `strtol` and form a proper hex or octal string from this value, however this would limit the length of incoming strings. Complete error checking could be done by breaking the input string into multiple sections, checking the format for each of those, then concatenating them back together.

The Pluginator is limited in that it cannot create new plugins after the window panel is closed, which goes against the traditional Apple OS X style of creating new documents. Future work would include reinitializing the GUI to bring up the window again. Another nice feature would be to include a library with frameworks for multiple plugin types (ie: light ballasts, motion sensors, scent generators, etc) such that the user could add a set of common parameters to their own plugin and modify them as appropriate.

Potential improvements for the PluginReader would include specialized functions that return a certain criteria of data instead of the entire structure since most occurrences are not interested in the entire plugin but merely a small part. For instance, the Configurator needs to read all plugin files available on a system but is really only interested in the “setup” commands for that plugin/device. So, the `PlgReader` could have a function that would return only the “setup command” for that plugin. Although special attention has been given to memory management of the data returned, we are not 100% satisfied that all memory precautions have been taken. With more time, a better look at the expectations of memory management by Cocoa would be taken into account to ensure that all memory was properly handled. Memory usage analysis should also be performed for the data storage object within the Pluginator as well, though its usage is generally limited to creating a plugin and quitting the program, so memory leaks will most likely be handled well by OS X.

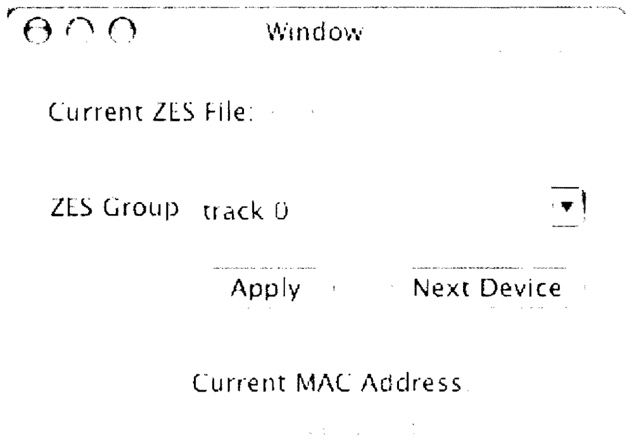
Configurator

Motivation

The motivation for this part of the project was to allow end users to more easily "configure" a ZES Multimedia setup to work properly with their physical environment. As ZES multimedia presentations grow, the task of matching physical addresses to a group or a set of groups within the ZigBee Event System will become more daunting. With the creation of the Configurator, the end user is presented with a streamlined graphical interface that allows this process to be completed in a fraction of the time. In fact, with the Configurator's auto-detection capabilities, the end user never needs to deal with a raw MAC address. Instead, the device is simply "turned on" allowing immediate visual identification.

Design

The design of the Configurator revolves around a single purpose: to map the embernet address to a group. As with any other graphical program, the first design step is determining the necessary components of its interface; thus, the interface shown below was created.
As you can see, there are just three pieces of information shown: the current file, the list of groups, and the device currently being configured Upon opening a ZES media file, the Configurator lists all the available groups in the pull-down menu. It then starts to configure the first detected device. This is accomplished by sending the "setup" command to that device; since the plugin associated



with the current device is unknown, all the known "setup" commands are sent in sequence until the device receives one and turns on. With a visual cue now available, the end user can choose which group or groups the device should be a member of. The user selects a group and hits `Apply`.

Once the device is finished being configured, the user can hit `Next Device` to configure the next device. Once all the known devices have been configured, the Configurator displays a message as such. At this point the user can either save the current configuration or start the process over again adding to previous configurations. Upon saving the configuration, a configuration file is generated. An example of this format is shown below.

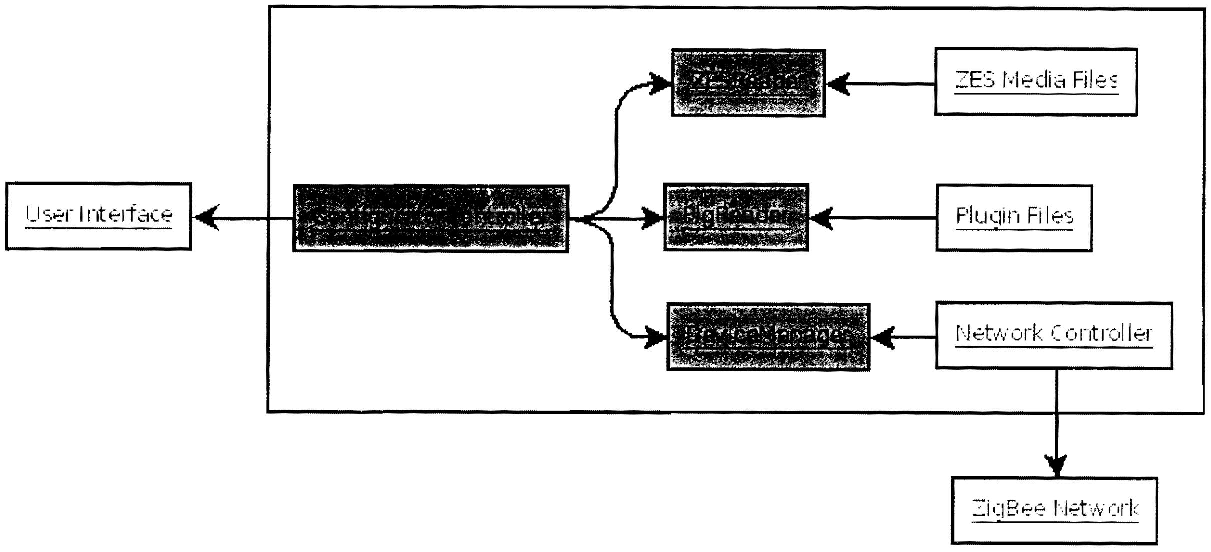
This file enables the Network Controller to send commands to the proper device during the

```
<root>
   <Group Name="LightBox1" Plugin="PhillipsLightBox"
NetworkType="Embernet" IOType="Output">
      <Node EmbernetID="1"/>
      <Node EmbernetID="2"/>
   </Group>
   <Group Name="MSensor1" Plugin="GenericMotionSensor"
NetworkType="ZigBee" IOType="Input">
      <Node ZigBeeID="8"/>
   </Group>
</root>
```

presentation. It contains all the necessary information not only for identifying the device within the multimedia presentation but also the capabilities of the device itself.

Implementation

The Configurator was created using XCode and Interface Builder that are a part of the Mac OSX standard development tools. There are four custom classes responsible for the heart of the Configurator's implementation: the `ConfiguratorController`, the `DeviceManager`, the `PlgReader`, and the `ZESReader`. The simplified model of the Configurator is shown below with these devices highlighted in red. This shows how the different components interact to create interface seen by the user.



At the heart of the program is the `ConfiguratorController`. It is basically responsible for program flow and controlling the other objects. It is also responsible for updating the user interface as its information changes. When the users choose to open a ZES file, the `ConfiguratorController` passes off control of the program to the `ZESReader`. In return, the `ZESReader` allows the user to specify which ZES file they wish to open and then proceeds to process the file accordingly. When the `ZESReader` has finished parsing the ZES file, it returns an `NSMutableArray` of `NSString`s that correspond to the available groups within that particular ZES file. The `ConfiguratorController` then checks to see if the plugins have been loaded. If they have not, it passes off control to the `PlgReader`. The `PlgReader` uses a hard coded directory and reads all the plugin files within this directory. When it has finished parsing all the plugins, the `PlgReader` returns an `NSMutableArray` of `NSMutableDictionary`s that contains the "setup" commands for each device. The `ConfiguratorController` then tells the `DeviceManager` to send "setup" commands to its first device. If the `DeviceManager` has no devices loaded, it loads the list of known devices (the details of this process are still being finalized). Otherwise it uses elements from ZigBeeLib and sends the necessary commands. As the user selects a group to be associated with a device, the `ConfiguratorController` adds the name of that group to an `NSMutableArray` contained within a larger `NSMutableDictionary` keyed on the device's unique ID.

Future Work

Before release as a commercial application, the Configurator will need to be able to perform automated network discovery. This is limited now, and unfortunately is a manual process still due to a lack of disclosure on the part of Philips, the manufacturer of our only current wireless output devices. Building this into the application will raise the utility and usefulness of the Configurator utility since right now, manual setup is still required.

III) Testing

<u>Unit Testing</u>

At the conclusion of each function and class, unit testing is performed to make sure the component works properly and is able to handle bad data. This phase of testing is concerned with ensuring that the function or class does what it needs to do and does so with a reasonable tolerance for errors. The function or class should not accept invalid data.

<u>Milestone Testing</u>

At the completion of each major part of the project, such as the Editor or Player, the application is tested to make sure it runs and properly exports the files it needs to export. Each individual application must have the correct output. They need to catch common errors in any input files they read. The communication between the Controller and the Player needs to function smoothly and efficiently.

Milestone testing done on the ZESEditor showed that there were several errors in the handling of multiple projects, resulting in rewriting some of the code. Milestone testing on the Controller and Player showed a small buffering error in the communication between them, but the situation was resolved.

<u>Final Testing</u>

Once all the applications and the framework has been completed, the entire suite of applications is tested together to ensure proper communication and process flow. The final testing included a basis for the demo to ensure that each piece of the system is functional. The final test came in several phases, each exposing different weaknesses that were subsequently corrected. Firstly, the ZESController was tested to allow interaction with AppleScript. Next, it was tested to ensure reliable communication with the Emberner ballasts. After this and the AppleScript were confirmed to be working, ZigBee input was tested, revealing that our motion detector was very unreliable. The board that reads the detector signals tends to crash.

After this was cleared up, testing with the ZESPlayer commenced, with the Player being controlled by a custom AppleScript. The player was reading infiles form the Editor, but expected a different file format. This small detail was shortly revised. The result of this series of tests, was a skeleton demonstration which provided the framework, both for the ZES files and for the AppleScript, on which we based our final presentation.

References

<u>AppleScript</u>

Apple's Developer Page

http://developer.apple.com/documentation/Cocoa/Conceptual/ScriptableCocoaApplications/SApps_

intro/chapter_1_section_1.html

<u>Cocoa</u>

Apple's Developer Page

http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/cha

pter_1_section_1.html

<u>Xerces XML</u>

Xerces-C++ Documentaton Page

http://xml.apache.org/xerces-c/apiDocs/indexhtml