Brutus 2.0 / Atlas Architecture

June 12, 2013

Broadcom Corporation • 5300 California Avenue • Irvine, CA, 92617 • Phone: 949-5300 • Fax: 949-926-5203

REVISION HISTORY

Doc Number	Revision Date	Description/Author
1.0	01/28/2012	Initial version, from D. Tokushige
1.1	06/12/2013	Updated for URSR 13.2 Release, from D. Tokushige

Section 1: Introduction

ATLAS OVERVIEW

Atlas is a demonstration application that is delivered as part of the Broadcom Set-Top Reference Software release. It offers a wide variety of functionality in an integrated architecture and provides an example of how the Set-Top Reference Software can be used to create a final product. It also provides a means to test and use Broadcom reference boards, with actual functionality depending on the platform. Atlas is not intended to be a production-quality application, but its source code can be used as the basis for real applications.

To get going quickly, refer to the *Brutus User Guide* (document number STB_Atlas-SWUM200-R) for instructions on how to configure, build, and run Brutus.

Some of the features supported in Brutus are:

- Front-end Interfaces:
  - QAM64, QAM256, QAM1024
  - QPSK DVB and DIRECTV, 8PSK
  - VSB
  - Analog RF and A/V line input
- S-Video, Composite, Component, DVI/HDMI, RF modulator, L/R audio, S/PDIF outputs supported
- Single or dual decode
- Single or dual output
- Picture-in-picture (PIP)
- Channel change
  - PSI scanning
  - Programmable channel map
- PVR
  - Single or dual record and playback
  - Time-shifting (simultaneous record and playback)
  - Single or dual channel encoder
  - Wide range of host, decoder, and STC trick modes (including Broadcom proprietary trick modes like 1x rewind)
  - DES/3DES PVR encryption and decryption
- GUI
  - Broadcom embedded windowing system and widget set
  - Anti-aliased TrueType fonts supported through FreeType
  - Image rendering (including PNG, JPEG, and BMP support)
  - Wide range of user input devices supported
  - Single- or dual-instance GUI
- On-screen status
  - Constellations
  - Front-end and back-end Status

- Playback and decode FIFO monitors

Broadcom's Reference Software team also develops numerous example applications, utilities, and test applications. They can be found in the source code tree in the following directories:

- ~~BSEAV/api/examples~~
- ~~BSEAV/api/utils~~

THE REFERENCE API SOFTWARE STACK

Brutus is the topmost layer of the Reference Software Release, a complete top-to-bottom set-top box software stack. The Reference API, shown in Reference API Software Stack, consists of four main layers: Brutus and example/utility applications, Nexus, the Porting Interface (PI), and the Operating System.

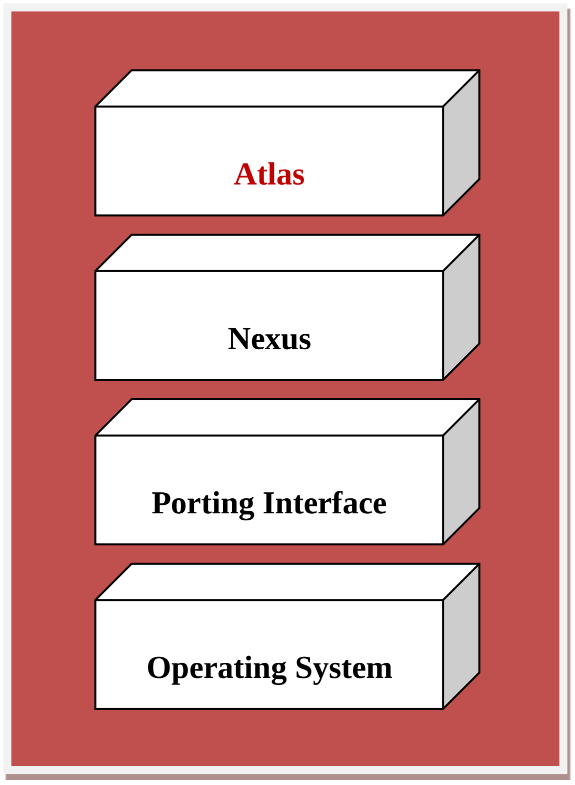


Figure 1: Reference API Software Stack

NEXUS

Nexus is a high-level, modular API for Broadcom DTV, set-top, IPTV and Blu-ray Disc boxes. The goal of Nexus is to allow Broadcom's customers to ship their products in less time and with less effort. Nexus provides easy-to-use and thread-safe interfaces that have been designed for the purpose of porting customer applications and Hardware Abstraction Layers (HALs) to Broadcom silicon.

Nexus is modular because all code lives within well-defined modules that can only interact through well-defined APIs. Nexus uses some component-based concepts found in systems like DCOM or CORBA, but Nexus is still lightweight and simple.

Nexus has a full-featured and robust synchronization model that makes it possible to build a hierarchy of modules, in which the modules have a network of complex calling and callback relationships, yet still avoid deadlock and provide maximum efficiency.

THE PORTING INTERFACE

The porting interface is a low-level C-language API that offers full control of Broadcom set-top chips. This API:

- is modularized for each block on the chip.
- has a consistent API and design methodology.
- supports interrupt safety through clearly marked ISR code.
- is portable across multiple operating systems like Linux and VxWorks.
- supports single- and multi-threading systems.
- is fully ANSI C-compliant.

On BCM7038-based platforms, the porting interface is known as “Magnum.” For previous platforms, it was referred to simply as the “Porting Interface.” This layer also includes other elements like base modules and system libraries.

THE OPERATING SYSTEM

Broadcom supports a variety of operating systems, including Linux and VxWorks. While each layer in the architecture requires some operating-system-specific code, a large majority of the code is platform-independent. The operating system consists of the following components:

- The bootloader
- The BSP (Board Support Package)
- The kernel
- The root file system (Linux only)
- The tool chain

Source code for all Linux modules can be obtained upon request.

Section 2: Brutus Functional Description

High-Level Design Goals

Brutus is designed to:

- provide a full featured application for reference software developers.
- serve as an integrated user interface application to demonstrate platform capabilities.
- integrate the many features of the set-top reference platforms into a cohesive application.
- support open source scripting language for custom application control.
- be configurable to support many of the usage modes of the set-top reference platforms.
- provide intuitive use of input devices like IR remotes.
- support all Broadcom set-top reference platforms, including both satellite and cable systems.
- run on multiple operating systems, such as Linux and VxWorks.
- run on both disk-based and diskless systems.

High Level Architecture

Graphical User Interface (GUI) Overview

The graphical user interface follows a single-threaded, cooperative, multitasking model. This means that most application work is performed during “idle time,” when the UI is done painting. Idle-time processing must usually be done in short intervals (no blocking); otherwise the UI response time is slow. Multiple threads may be required in some instances but their use shall be minimized whenever possible.

The Atlas user interface is built on top of a series of software layers. These layers include:

Layer	Description	Source Code
bwin	Minimal C based windowing system with font and image rendering support.	BSEAV/lib/bwin
bwidgets	Simple TV-friendly C based widget library based on bwin.	BSEAV/lib/bwin/bwidgets
Atlas widgets	Specialized widgets based on the basic label/button widgets offered by bwidgets	BSEAV/app/atlas/widgets

Model-View-Controller (MVC) Overview

The Atlas application is implemented using a variation on the Model-View-Controller (MVC) software design pattern which divides the application into three distinct and loosely coupled groups:

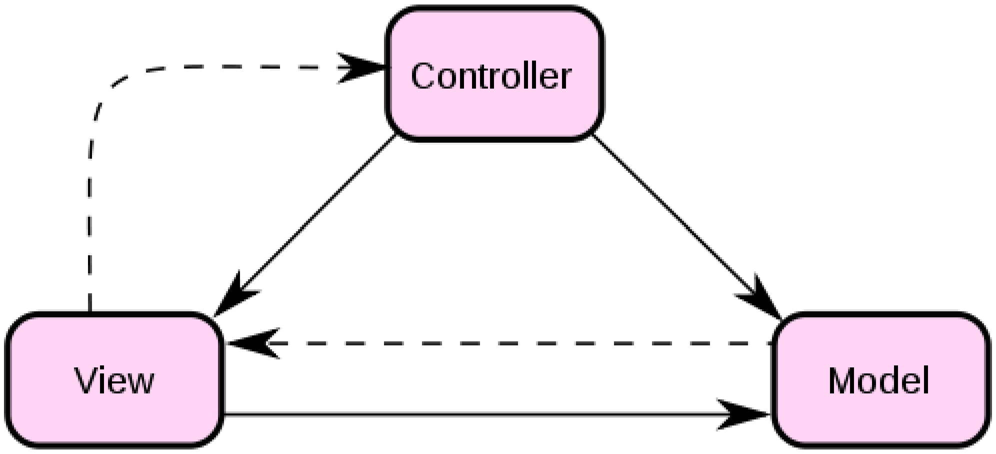


Figure 1 - Model-View-Controller. Solid lines indicate a direct association and dashed lines an indirect association (via an observer notification)

The dark lines in Figure 1 indicate a direct association from one branch of the MVC to another. In the underlying software, this equates to the ability to call public methods in the destination object instance. The dotted lines represent an indirect association in the form of an “observer notification”.

Each Model or View offers a fixed set of notifications which interested observers can register to receive. Observers can register for all notifications for a given view/model, or they can pick and choose.

- The **Model** represents the underlying data in a logical and easy to access way. It hides the underlying details of data acquisition and data storage from the View and Controller classes. Models can be directly accessed by both controllers and views to set/get data. Models can only contact views using notifications which are typically sent because of a change in data. This decouples the model from any system-wide complex decision making (controller) as well as from any presentation or user based interaction.

- The **View** represents the user’s interface to the application and the display of information. This includes graphical user interface (GUI) components such as buttons, labels, and windows. The view relies on the model for data to display, and relies on the controller for user input events such as remote control button presses. The view can be directly accessed only by the controller however this should never be done to accomplish view specific functions. Controllers can hide/show view windows or call high level initialize methods, but should never be manipulating user interface elements – this keeps the user interface decoupled from the main controllers in the application. Views receive notifications from models which triggers possible changes in the view’s presentation to the user. The view can also access models directly to retrieve state information or make minor changes. Views enact changes to the system primarily by issuing notifications to controllers.

- The **Controller** represents the business logic or “brains” of the application. It receives user input and issues instructions to the model and view. For simple user commands, the views can interact directly with models. But for more complicated or system-level tasks, the view must send a notification to the controller. Notifications are the only way to access the controllers. This keeps control and system based code confined to the controller itself. Controllers have direct access to both views and models, but accessing their lower level details is strongly discouraged

This is an example scenario showing how each part of the MVC typically interacts with one another.

1. The user uses the remote to click a button on the audio screen (view) which turns on audio dynamic range compression (DRC).

2. The screen (view) sends a notification to all interested observers (controller) indicating that audio DRC should be enabled. *The screen (view) does not care how this is accomplished nor does it know who is doing the work – it only knows what the user requested.*

3. The controller having previously registered as an observer of the screen (view) receives the audio DRC enable request and based on system state, does what is necessary to make that happen. It may have to pause a playback, stop the audio decoder, or interact with other models. Audio DRC is then enabled by calling directly into the appropriate audio decoder object (model) which will then make the change in Nexus. *The controller does not care where this request originated from or who sent it. It also does not know who will be notified after the change is made. The controller only knows how to complete the requested task at a system level.*

4. Once the audio decoder object (model) makes the change to audio DRC, it notifies all of its observers of the change. In the case of Atlas this will notify the audio screen (view) as well as the Lua scripting object (view).

5. The audio screen (view) which had previously registered to receive notifications from the audio decoder object (model) receives the audio DRC change notification and updates status in the window on screen. *The audio screen (view) does not know why or how the audio DRC setting was changed - it only knows how to react when it does.*

MODEL-VIEW-CONTROLLER (MVC) GOALS

The basic goal of the MVC architecture is to promote independent development, easy reuse and better testability of the application.

- **Supports multiple views.** An application may include several different user interfaces (such as GUI windows, command line prompt, or script engine) which both access the same data within the Model and issue similar commands to the Controller. Both the Controller and the Model should remain unchanged regardless of which type View is used.

- **Accomodates Changes.** In an application like Atlas, changes to the View can be expected to be frequent and never ending so isolating this part of the code is crucial to its long term success. Changes to the Controller should not affect the manner in which data is displayed to the user (by the View). Data storage and organization by the Model is isolated from both the Controller and the View such that they are oblivious as to how it is received and stored (the Model may even be accessing data over a network).

- **Development/Testing.** The decoupling of the MVC allows each piece to be unit developed/tested in isolation. This will improve reliability and lead to a more automated test approach. A test controller could be swapped in to provide testing functionality that the standard controller may not implement (which also helps keep test code separated of release code). Similarly, a custom GUI (view) can be used to ease development by providing additional debugging information or controls.

OBSERVERS & NOTIFICATIONS

The models and views communicate using an observer notification scheme (dotted lines in Figure 1). This keeps the Model, View, and Controller as decoupled from one another as possible.

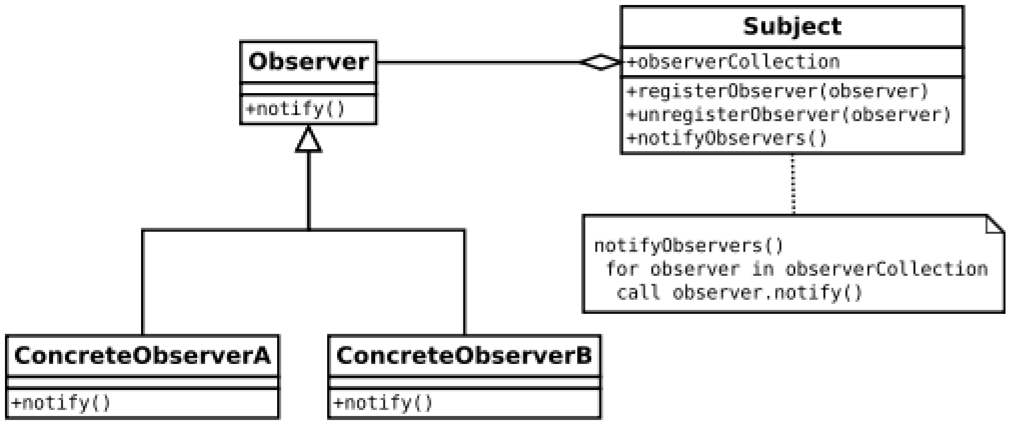


Figure 2 - Observer Pattern

In the observer notification pattern, a “subject” object maintains a list of interested “observer” objects and the notifications they are interested in. The observer objects first register with the subject object to indicate their interests. When the subject object decides to send a notification, it is automatically sent to all interested observers. Notifications can carry a data payload and a type indicator as well.

While similar in nature, notifications operate in a fundamentally different way from widget events and are not meant to replace them.

Notifications	Widget Events
Subject object provides a public registration interface.	Are private between sender and receiver.
Have an optional data payload and type value.	Do not come with any additional data.
Is broadcast to multiple registered observers.	Can bubble up the widget hierarchy allowing each parent object to handle the event. This is typically referred to as a “chain of responsibility”.

Widgets will still communicate their state using events, as their hierarchical, private nature is well suited for handling user initiated actions. The Views will then translate these events into meaningful notifications which are then sent to registered observers (controllers). The Models will use notifications to notify Views about change of state.

MVC CLASS HEIRARCHY

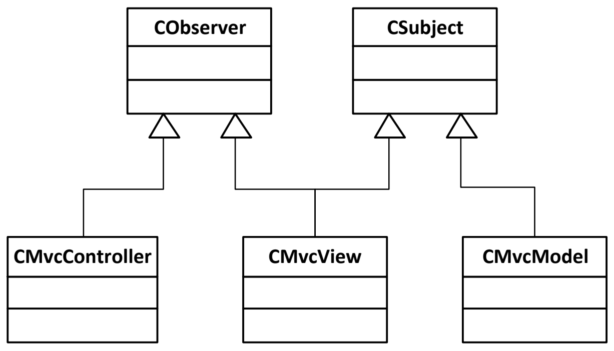


Figure 3 - MVC Class Heirarchy

The Model-View-Controller illustrated in Figure 1 - Model-View-Controller. Solid lines indicate a direct association and dashed lines an indirect association (via an observer notification)Figure 1 is implemented using the observer and subject classes as seen in Figure 2. The controller is an observer who can receive notifications from the view. The model is a subject who sends notifications to interested views. The view acts as an observer when it receives notifications from the model and also is a subject who sends notifications to the controller.

DETAILED DESIGN

VIEWS

Each GUI window will serve as a separate view in Atlas. The Lua scripting object will also be a view. There are two basic scenarios a view must handle:

1. **Widget Events.** Views hold references to all the widgets that they contain, and they listen for events in response to widget actions (such as click, keypress, etc). These events are then translated into corresponding notifications which are sent to all registered observers. The View does not know or care how these notifications are handled.

2. **Model Notifications.** Views register for notifications from associated Models (this is typically only done once at startup, although it can be done dynamically if necessary). As data and state changes within the Model, the View must be notified to relay the changes to the user where applicable. Notifications can carry a data payload so in most cases the View will not need to query the Model directly (but has that option if necessary).

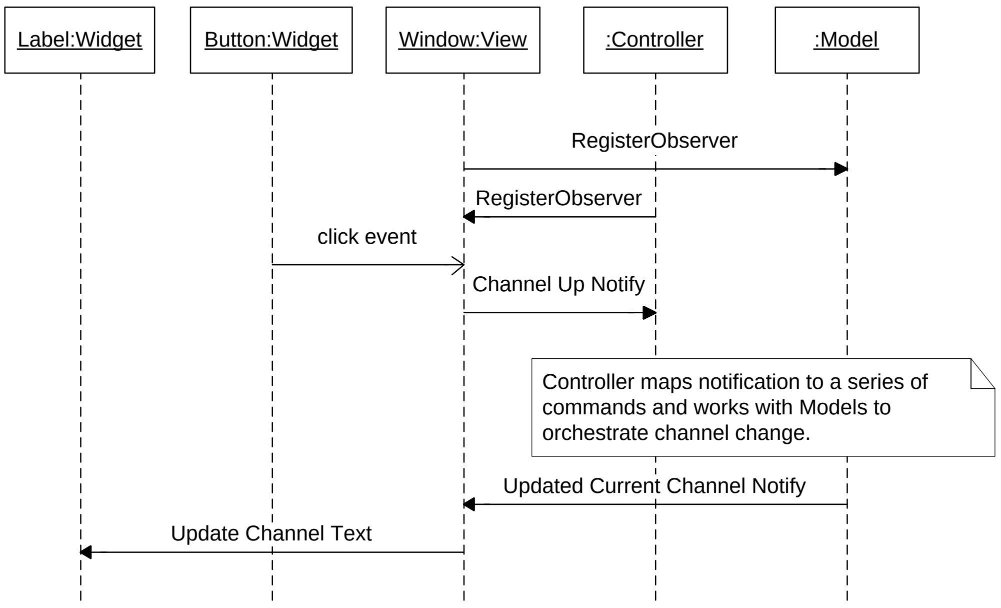

Label:Widget
Button:Widget
Window:View
:Controller
:Model
RegisterObserver
RegisterObserver
click event
Channel Up Notify
Controller maps notification to a series of commands and works with Models to orchestrate channel change.
Updated Current Channel Notify
Update Channel Text

Figure 4 - Widget Event / Model Notification Example

Screen Views

The graphical user interface is made up of a number of screens – each of which is a view in the MVC system. These screens are designed to encapsulate all their features to avoid coupling with the controller. While the controller can hide and show screens, it is largely ignorant of how each screen operates.

The Brutus user interface (UI) is designed to maximize the amount of video still visible when screens are displayed. This allows developers and chip leads to monitor video quality while making changes using the UI. Screens are designed for flexibility during testing and development so they will not resemble that of commercial set top boxes.

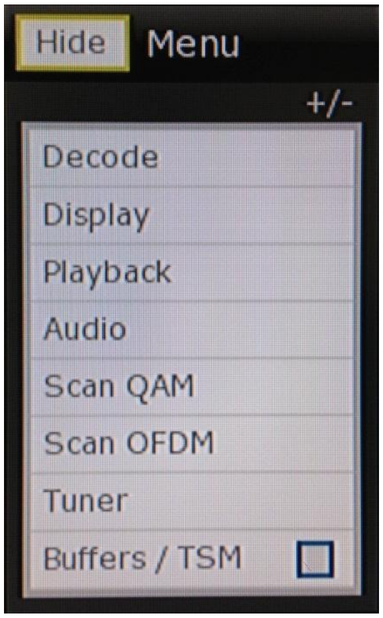
The following screens are very basic in that they use minimal graphics and are directly drawn using bwin commands. This allows for speed and minimal user interface code. Future versions of Atlas will improve the look and feel as well as add animation and help support.

Screen/Panel Class Heirarchy

Screens are always full screen and take up the entire viewing area when displayed. They can be made transparent to show underlying video, but only one screen can be displayed at a time. While this is unusual for a computer desktop, it is well suited to a TV environment without mouse pointer support. All screens will derive from a base class CScreen. Currently there is only one screen in the system – the main screen. The main screen is made up of a series of independent panels that are displayed and sometimes hidden using the buttons on the main screen. These panels are displayed in various locations around the display with the primary goal to keep as much underlying video visible as possible.

Controls on each panel can be customized using the "+/-" button to hide/show panel controls.

Main Screen



The main menu screen is displayed when the menu remote button is pressed. Most of the main screen is transparent which allows users to monitor the underlying video while changing settings. Each button on the main screen shows additional panels when clicked. The +/- button on each screen allows the user to customize which controls are visible on screen. It may also give access to additional more advanced options that are not available by default. The "Buffers/TSM" checkbox will show/hide a new, persistent menu screen which remains shown even when the menu is hidden.

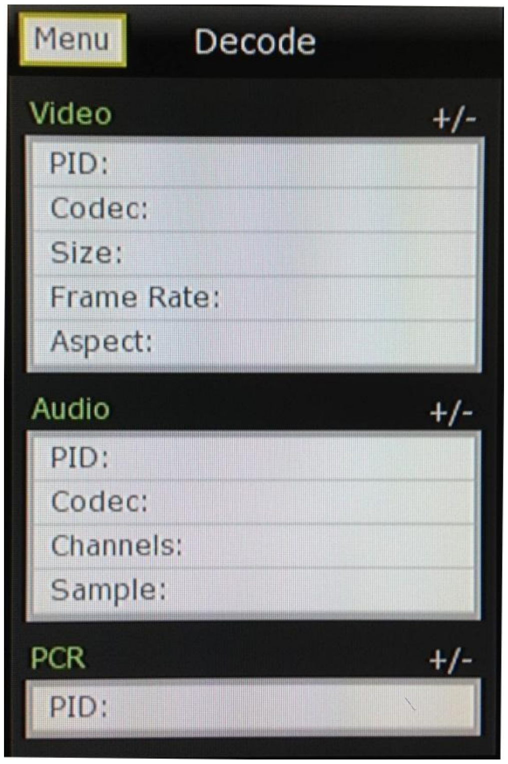
The panels are under direct control of the main screen. Notifications received by the main screen will also be propagated to the panels. Similarly, commands to hide/show the main screen will also affect each child panel.

Available options are dependent on the capabilities of the given platform. For example, channel scan buttons will only be shown for available tuners.

The main screen handles the following notifications:

- eNotify_CurrentChannel indicates the current channel has changed so the channel number panel is updated to reflect the new channel number
- eNotify_DeferredChannel indicates that the channel number has changed but tuning has not yet occurred. The channel number panel is updated.
- eNotify_RecordStarted/eNotify_RecordStopped indicate that the record state has changed so the channel number panel is updated to reflect the current record state.
- eNotify_Timeout is used to hide the channel number panel after a period of time.
- eNotify_PlaybackStateChanged indicates that playback is active and the channel number panel is updated to show “playback”.
- eNotify_ScanStarted shows the scan progress panel
- eNotify_ScanStopped hides the scan progress panel and queries user to save newly found channels to disk.
- eNotify_ChannelListChanged is received when a new channel is found during a scan and results in the number of found channels changed in the scan progress panel.

Decode Panel

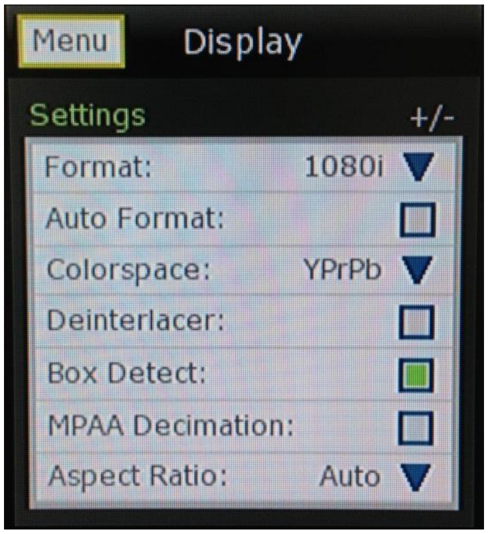


The decode menu displays decoder specific data about the current channel or playback. There are no editable fields in this menu.

The decode panel handles the following notifications:

- eNotify_VideoSourceChanged triggers and update to the video decoder status label values.
- eNotify_VideoDecodeStopped clears the video decoder status label values.
- eNotify_AudioSourceChanged triggers and update to the audio decoder status label values.
- eNotify_AudioDecodeStopped clears the audio decoder status label values.

Display Panel



Format:	1080i ▼
Auto Format:	☐
Colorspace:	YPrPb ▼
Deinterlacer:	☐
Box Detect:	🟩
MPAA Decimation:	☐
Aspect Ratio:	Auto ▼

This screen allows the user to manipulate display settings. All changes apply immediately except for “auto format” which takes effect on the next channel change.

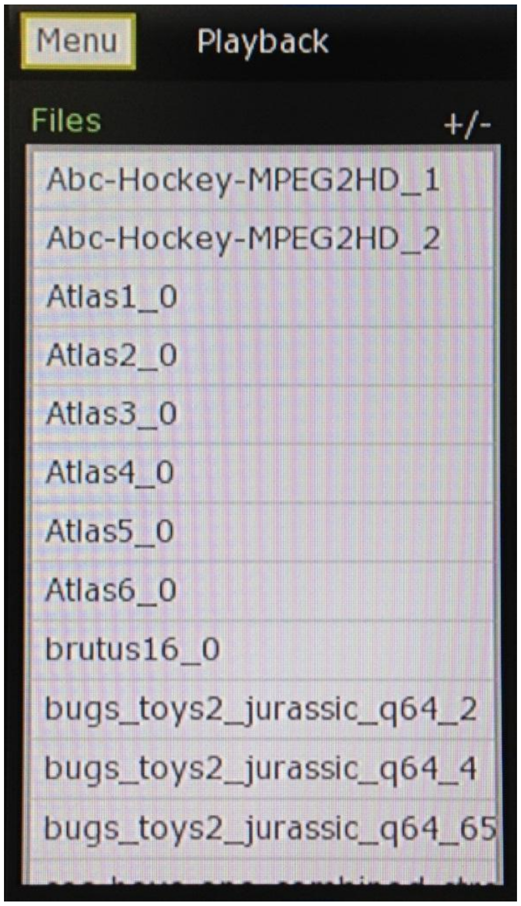
The display panel handles the following notifications:

- eNotify_Timeout occurs when display format is changed and not confirmed by the user. We will revert back to the previous format.
- eNotify_VideoFormatChanged notifies of a change in video format. If this change was the result of a notification originating from the display panel itself, we will ask the user for confirmation using a modal popup.
- eNotify_AutoVideoFormatChanged indicates that the auto format setting has changed. The checkbox control is updated to reflect the new state.
- eNotify_ColorSpaceChanged indicates that the colorspace setting has changed. The pulldown control is updated to reflect the new state.
- eNotify_MpaaDecimationChanged indicates that the MPAA decimation setting has changed. The checkbox control is updated to reflect the new state.
- eNotify_DeinterlacerChanged indicates that the deinterlacer setting has changed. The checkbox control is updated to reflect the new state.

- eNotify_BoxDetectChanged indicates that the box detect setting has changed. The checkbox control is updated to reflect the new state.

- eNotify_AspectRatioChanged indicates that the aspect ratio setting has changed. The pulldown control is updated to reflect the new state.

Playback Panel



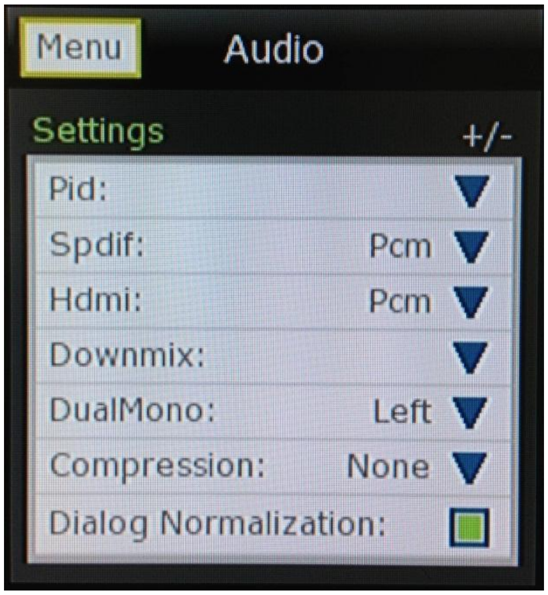
Abc-Hockey-MPEG2HD_1
Abc-Hockey-MPEG2HD_2
Atlas1_0
Atlas2_0
Atlas3_0
Atlas4_0
Atlas5_0
Atlas6_0
brutus16_0
bugs_toys2_jurassic_q64_2
bugs_toys2_jurassic_q64_4
bugs_toys2_jurassic_q64_65

This screen lists the available playback files. Selecting a file will start playback and close the menu.

The display panel handles the following notifications:

- eNotify_PlaybackListChanged causes the list of playback files to be refreshed.

Audio Panel



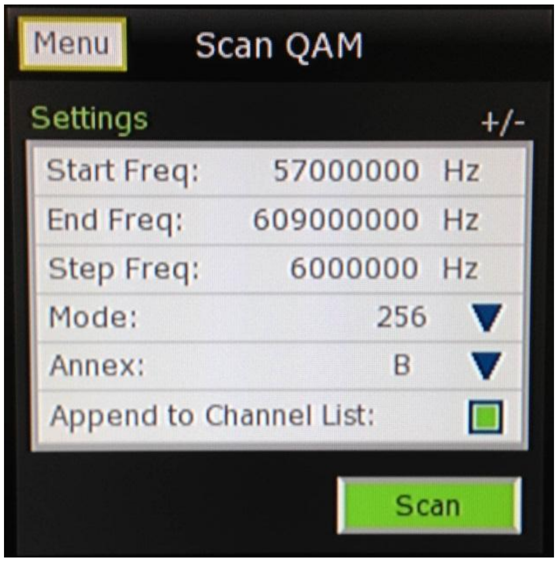
Pid: ▼
Spdif: Pcm ▼
Hdmi: Pcm ▼
Downmix: ▼
DualMono: Left ▼
Compression: None ▼
Dialog Normalization: 🟩

The audio screen allows the user to change various audio settings. Each change is applied immediately.

The audio panel handles the following notifications:

- eNotify_AudioSourceChanged causes all audio settings to refresh

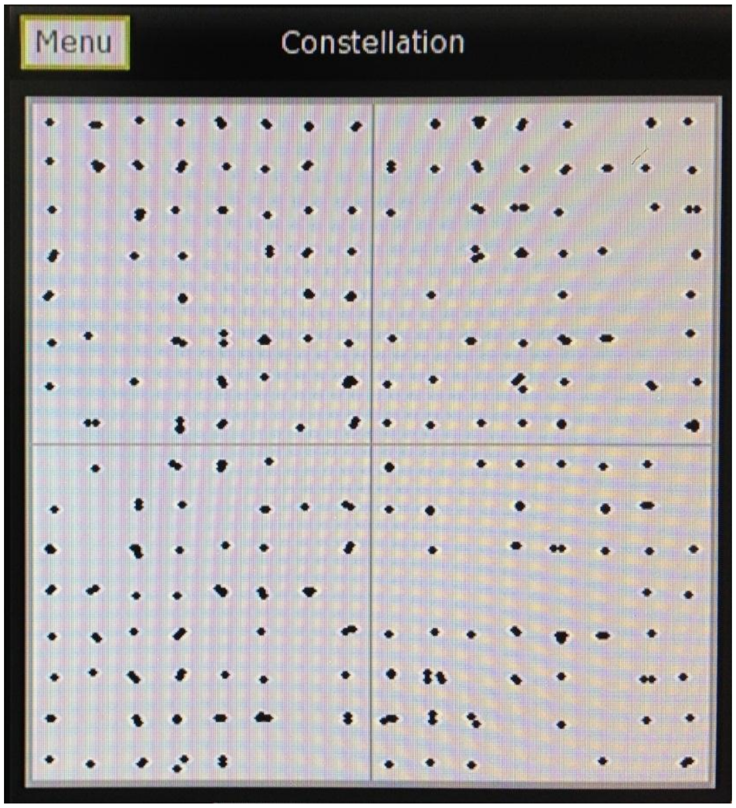
Scan Panel



Start Freq: 57000000 Hz
End Freq: 609000000 Hz
Step Freq: 6000000 Hz
Mode: 256 ▼
Annex: B ▼
Append to Channel List: 🟩

The scan screen allows the user to scan for unknown channels using the given tuner. The user can do a full replacement scan or selectively append to the channel list with newly found channels. Duplicate, encrypted, and audio only channels are not saved. Once the scan completes, the user is prompted to save the channel list to disk. If the new channel list is not saved, you can still tune to the new channels, but they will be lost after quitting Atlas.

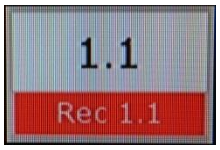
Tuner Panel



The tuner panel handles the following notifications:

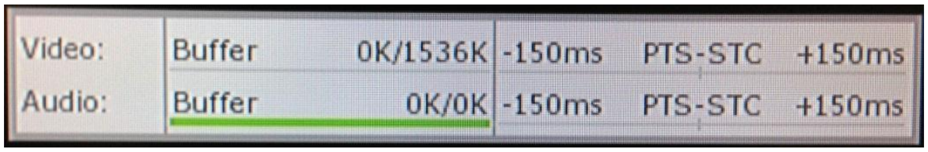
- eNotify_Timeout is triggered periodically once the panel is displayed. Each time it triggers, the constellation is updated based on data retrieved from the currently active tuner.

Channel Number Panel



The channel number screen is displayed whenever the main menu is present or for several seconds after a channel change occurs. It also shows the current channel type and what current recordings are active.

Buffers/TSM Panel



Video:	Buffer	0K/1536K	-150ms	PTS-STC	+150ms
Audio:	Buffer	0K/0K	-150ms	PTS-STC	+150ms

When displayed, the buffers screen remains on screen even when the other menu screens are hidden. It is designed to be independent of the other screens once shown. Similar to the channel number screen, the user can navigate to other menus without closing this screen.

The buffers/TSM panel handles the following notifications:

- eNotify_Timeout is triggered periodically once the panel is displayed. Each time it triggers, the buffer levels are updated based on data from the audio/video decoders and playback objects.

- eNotify_PlaybackStateChanged triggers a layout of the buffers panel controls. If playback is active, the playback buffer depth and TSM data will be displayed.

Scripting Panel

TBD

Script Engine (Lua) View

The Atlas scripting language shall be Lua (pronounced LOO-ah). Lua is a lightweight, fast, simple scripting language originally designed for embedding into larger applications. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping. It is written entirely in C and is easy to embed into C/C++ applications. It is currently in use in many major commercial applications such as Adobe Lightroom and World of Warcraft. The Lua library is only 203K and is one of the very fastest interpreted scripting languages. Lua's build times are quite low, and it is distributed under the MIT license which makes it ideal for reference software development and releases. For more information visit: http://www.lua.org.

The script engine will also serve as a view, although it will not have a graphical user interface. The script engine will interface with the controller and models in the same manner as the GUI based views. There are two ways to interact with the script engine:

- **Interactive Command Line.** Interactive shell where the user can manually input commands one by one.

- **Script File.** Load and execute script files stored on the file system. You can specify the files on the command line, run scripts using a Lua command, or select from an on-screen list through the GUI. GUI functionality will be non-responsive while a script file is executing.

The Lua scripting language shall be extended by adding a brutus library of calls. All brutus specific calls will be prefixed by "brutus." and will be accessible from both the interactive shell and scripting files.

Brutus Lua Extensions

Brutus extends the Lua scripting language to include custom commands that can be used in conjunction with standard Lua commands. Optional parameters are listed in *italics*.

brutus.channelUp()

This command tunes to the next channel in the currently loaded channel list.

brutus.channelDown()

This command tunes to the previous channel in the currently loaded channel list.

brutus.scanQam(startFreq, endFreq, bandwidth, append, mode, annex)

This command scans the given frequency range for QAM channels.

1. startFreq (Hz): frequency to begin scanning
2. endFreq (Hz): frequency to end scanning
3. bandwidth (Hz): channel bandwidth
4. *stepFreq (Hz):* default is bandwidth. This is the used to determine which frequencies to scan between the startFreq and endFreq.
5. *append (bool)*: default true:add newly found channels to channel list, if false:replace existing channel list
6. *mode (int)*: default if auto supported: NEXUS_FrontendQamMode_eAuto_64_256, otherwise: NEXUS_FronendQamMode_e256
7. *annex* (int): default:NEXUS_FrontendQamAnnex_eB

brutus.scanVsb(startFreq, endFreq, bandwidth, append, mode)

This command scans the given frequency range for VSB channels.

1. startFreq (Hz): frequency to begin scanning
2. endFreq (Hz): frequency to end scanning
3. bandwidth (Hz): channel bandwidth
4. *stepFreq (Hz):* default is bandwidth. This is the used to determine which frequencies to scan between the startFreq and endFreq.
5. *append (bool)*: default true:add newly found channels to channel list, if false:replace existing channel list
6. *mode (int)*: NEXUS_FronendVsbMode_e8

brutus.scanSat(startFreq, endFreq, bandwidth, append*)*

This command scans the given frequency range for SDS channels.

1. startFreq (Hz): frequency to begin scanning
2. endFreq (Hz): frequency to end scanning
3. bandwidth (Hz): channel bandwidth
4. *stepFreq (Hz):* default is bandwidth. This is the used to determine which frequencies to scan between the startFreq and endFreq.
5. *append (bool)*: default true:add newly found channels to channel list, if false:replace existing channel list

brutus.scanOfdm(startFreq, endFreq, bandwidth, append, mode*)*

This command scans the given frequency range for OFDM channels.

1. startFreq (Hz): frequency to begin scanning
2. endFreq (Hz): frequency to end scanning
3. bandwidth (Hz): channel bandwidth
4. *stepFreq (Hz):* default is bandwidth. This is the used to determine which frequencies to scan between the startFreq and endFreq.
5. *append (bool)*: default true:add newly found channels to channel list, if false:replace existing channel list
6. *mode (int)*: if ISDB-T:NEXUS_FrontendOfdmMode_eIsdbt, if DVB-T:NEXUS_FrontendOfdmMode_eDvbt, if DVB-T2:NEXUS_FrontendOfdmMode_eDvbt2

brutus.setVideoFormat(format)

This command requests a particular video format. Incompatible or unsupported video formats will result in TBD.

1. format (int): NEXUS_VideoFormat

brutus.setColorSpace(color)

This command sets the component video color space.

1. color space (int): NEXUS_ColorSpace

brutus.setMpaaDecimation(decimation)

This command toggles MPAA decimation.

1. decimation (bool): true or false

brutus.setDeinterlacer(deinterlace)

This command toggles Motion Adaptive Deinterlacer (MAD).

1. deinterlace (bool): true or false

brutus.setBoxDetect(boxDetect)

This command toggles Letter/Pillar box detection.

1. boxDetect (bool): true or false

brutus.setAspectRatio(aspect)

This command sets the aspect ratio.

1. aspect (int): NEXUS_AspectRatio

brutus.setAutoVideoFormat(auto)

This command toggles the auto video format setting (on channel change).

1. auto (bool): true or false

brutus.channelListLoad(filename, listname, append)

This command loads the given channel list from the given file name and either appends or replaces the current channel list.

1. filename (string): default: “channels.xml”
2. listname (string): default: “default”
3. append (bool): if true:append new channel list channels to existing channel list. Default false:replace existing channel list.

brutus.channelListSave(filename, listname, append*)*

This command saves the current channel list to the given file name and either appends or replaces the current channel list.

1. filename (string): default: “channels.xml”
2. listname (string): default: “default”
3. append (bool): if true:append current channel list channels to file channel list. Default false:replace existing channel list.

brutus.channelListDump()

This command prints out the current channel list.

brutus.playbackStart(filename, indexFilename, **path)**

This command starts playback for the given media file.

1. filename (string): filename of media file to play
2. indexFilename (string): filename of the index file
3. path (string): default:"./videos"

brutus.playbackStop(filename)

This command stops playback for the given media file.

1. filename (string): filename of media file playback to stop. You can omit this parameter and the current playback will be stopped.

brutus.playbackTrickMode(trick)

This command sets the trick mode for the currently playing file.

1. trick (string):
    a. "play" – resume normal playback
    b. "pause" – pause playback
    c. "i" – decode only I-frames
    d. "ip" – decode only I & P frames
    e. "all" – decode all frames
    f. "fa" – frame advance
    g. "fr" – frame reverse
    h. "rate(float)" – set trick mode rate (1.0 is normal speed)
    i. "host(type, *modifier*, *slowrate*)" – set host trick mode where type = I, ip, or all. Modifier = 1 for forward, -1 for reverse (I only). Slowrate = decoder repeat rate (1=1x, 2=2x, etc).
    j. "seek(pos)" – jump to position in millisecs

brutus.refreshFromDisk()

This command reloads the available playback list from disk.

brutus.recordStart(filename, path)

This command starts a recording.

1. filename (string): filename to save the new recording to.
2. Path (string): default:"./videos"

brutus.recordStop(filename)

This command stops a recording.

1. filename (string): filename associated with recording.

brutus.setAudioProgram(pid)

This command sets the current audio PID.

1. Pid (integer): audio PID number

brutus.setSpdifType(type)

This command sets the s/pdif output type.

1. type (integer):
    a. 0 = PCM
    b. 1 = Compressed
    c. 2 = Encode DTS
    d. 3 = Encode AC-3

brutus.setHdmiAudioType(type)

This command sets the HDMI output type.

1. type (integer):
    a. 0 = PCM
    b. 1 = Compressed
    c. 2 = Multichannel
    d. 3 = Encode DTS
    e. 4 = Encode AC-3

brutus.setDownmix(downmix)

This command sets the audio downmix.

1. downmix (integer):

  a. 0 = None
  b. 1 = Left
  c. 2 = Right
  d. 3 = Monomix
  e. 4 = Matrix
  f. 5 = Arib
  g. 6 = LeftRight

brutus.setDualMono(dualmono)

This command sets the audio dualmono.

1. dualmono (integer):
   a. 0 = Left
   b. 1 = Right
   c. 2 = Stereo
   d. 3 = Monomix

brutus.setDolbyDRC(level)

This command sets the Dolby dynamic range compression level.

1. level (integer):
   a. 0 = None
   b. 1 = Light
   c. 2 = Medium
   d. 3 = Heavy

brutus.setDolbyDialogNorm(level)

This command sets the Dolby dialog normalization level.

1. level (bool): on = true, off = false

brutus.setDebugLevel(module, level)

This command sets the debug output level for a given module. This allows you to change the debug output while Atlas is running. Any changes made will not survive after restarting Atlas.

1. Module (string): module name
2. Level (string):
   a. "trace"
   b. "msg"
   c. "wrn"
   d. "err"
   e. "log"

brutus.runScript(filename)

This command executes the given Lua script.

1. Filename (string): file name of Lua script to execute.

brutus.sleep(time)

This command sleeps the Lua script engine for the given period of time in milliseconds.

1. Time (integer): length of time to sleep in milliseconds.

help

This command prints basic help information.

exit

This command exits Atlas.

MODELS

The model is responsible for providing an easy to use APIs to all the data used by the application. It provides a purely functional API and does not care how its data is presented to the user. Communication with the model is accomplished using both direct calls to its API, as well as the aforementioned observer-notification mechanism. For example, Views can register to receive notifications when certain Model data changes and can also call Model APIs directly if additional information is required. Note that the model is *not* meant to be a complete layer that completely encapsulates Nexus. It provides data access and does abstract some operations on top of Nexus to where applicable. This allows for flexibility in both the view and controller while also offering a reduction in complexity.

CModelConfig

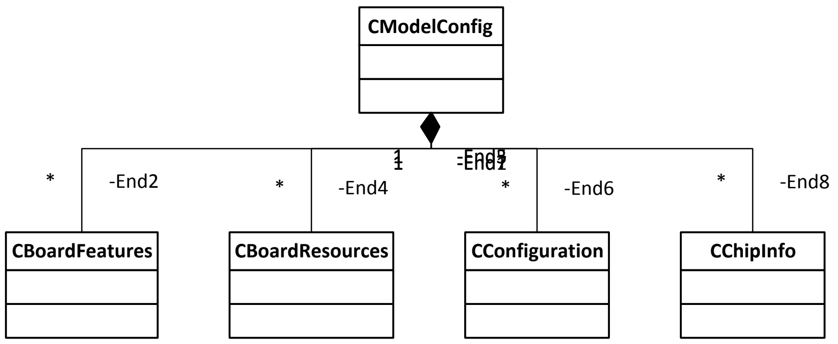


Figure 5 - CModelConfig Class Diagram

The CModelConfig object contains basic information about the board features, available resources, default configuration settings, and basic versioning for the major chips. It is a singleton class and is primarily used by objects that need access to check in/out resources and for application initialization.

CBoardResources initially owns all available resources (models) in the system. This includes objects like displays, decoders, tuners, etc. The main application, controller, and models all check out these resources when needed. This class also has the ability to reserve resources for a particular client. This is used by the main application to split resources between multiple instances of Atlas in the case of HDSD single or dual mode.

The CConfiguration object is used throughout the models to look up user defined settings. CConfiguration also interfaces with the external brutus.cfg file which allows users to override default settings. The brutus.cfg file is read at startup. ***<DTT: include link to Atlas Usage Guide here for list of available brutus.cfg options>***

CModel

The CModel object is a container for easy access to the other models in the system. While simply a container, it is still a model and as such can issue notifications if necessary. All objects stored in CModel are assumed to be checked out and valid.

CTimer

CTimer class implements a timer using the bOS scheduler. It handles translation of the timer into a Atlas notification event and ensures it synchronizes with the bwin/bwidgets main event loop. Timers are not recurring. Models/Views throughout the system use CTimer.

CModelChannel

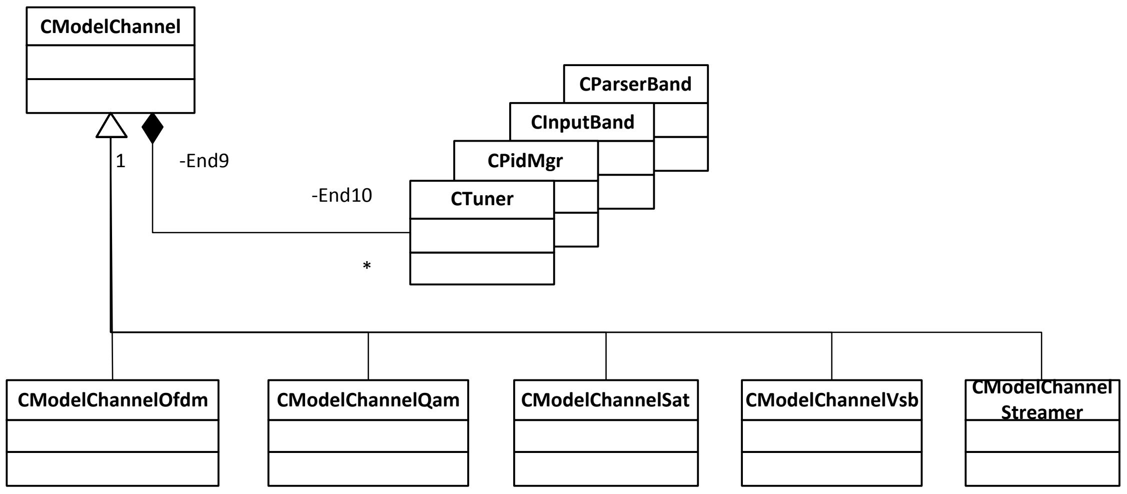


Figure 6 - CModelChannel Class Diagram

CModelChannel is an abstract base class for the various types of channels classes. It defines the common pure virtual methods used by all channel objects. Methods to persist and restore the class to disk are also supported. Derived classes can override these methods, but should still call the base class version to ensure all relevant channel data is saved or restored.

The derived classes implement channel type specific features, manage necessary resource check in/out, and orchestrate tuning. Persisting and restoring from disk is handled only for the derived class specific information.

The channel model contains the channel specific data based on type. This also includes operations on the channel itself such as tuning. Keeping tuning specifics contained within a specialized channel object will allow for better separation between sometimes significantly different tuning types (i.e. VSB vs IP tuning).

CPidMgr

A CPidMgr exists for each CModelChannel and maintains its associated list of audio, video, PCR, conditional access, and ancillary pids. Each pid manager also handles persisting and restoring pid data to/from disk. CPidMgr also make pid comparisons and copies easier with "==" and "=" operator overloads.

CModelChannelMgr

CModelChannelMgr keeps track of the currently tuned channel and maintains the list of available known channels. Channel manager is shared between multiple brutus instances since they both will share a given channel list. It can save/restore the channel list to disk in an XML formatted named "channels.xml".

CVideo

CVideo contains all the information necessary to play a video from disk. It also contains identifying information such as name and description.

CPlaybackList

The CPlaybackList class maintains a list of CVideo objects representing all the available playback files. It uses bmedia_probe to read video metadata and can save it to nfo files.

CResource

All resources managed by CBoardResources are derived from this base class. It stores reservation and checkout state and provides convenience functions to query general type and ID information. CResources correspond to actual board resources like decoder, tuners, playbacks, outputs, etc. Checking out/in CResource objects are accomplished using methods in the CBoardResource class.

Resource	Description
CAudioDecode	The CAudioDecode class wraps the Nexus audio decoder and handles the source changed callback, DTS/AC-3 encode settings, and SRS True Volume / Dolby Volume settings.
CSimpleAudioDecode	This class extends the CAudioDecode class using the Nexus simple audio decoder. It internally manages the primary/secondary audio decoders and audio outputs required by the simple audio decoder. Nexus simple audio decoder takes ownership of the decoders and outputs so the objects are checked out when opened and checked in only when CSimpleAudioDecode is closed. This class also provides methods to set audio features like downmix, dualmono, Dolby range compression, and dialog normalization.
CDisplay	Manages the Nexus display, associated video windows, and connected video outputs. CDisplay provides methods for changing outputs, graphics geometry, colorspace, decimation, deinterlacing, box detection, aspect ratio, and video format (manual/auto).
CGraphics	Manages the Nexus surface compositor, framebuffers, bwin, and bwidgets.
CVideoDecode	The CVideoDecode class wraps the Nexus video decoder and handles the source changed callback and decoder status information.
CSimpleVideoDecode	This class extends the CVideoDecode class using the Nexus simple video decoder. Nexus simple video decoder takes ownership of a decoder and video window so the objects are checked out when opened and checked in only when CSimpleVideoDecode is closed.
CRemote	This is the abstract base class that defines the basic minimal implementation required for all derived remote classes.

CIrRemote	This class is derived from the CRemote base class and implements the IR remote capabilities for the silver OneForAll remotes and black/silver Broadcom remotes. Synchronization with the bwidgets main loop is also handled here.
CUhfRemote	TBD
CPlayback	Manages a Nexus playback, associated trick modes, and file handling. The playback class has sends several notifications about playback state.
CRecord	Manages a record session where a program is saved to disk.
CTuner	Abstract base class for all tuner types. It defines the abstract virtual methods the derived tuner classes must implement. Special handling for threaded channel scan are implemented in this class, including synchronization with the bwidgets/bwin main loop for scan notifications.
CTunerQam	Implements QAM specific tuning and scanning. It is derived from CTuner and leverages the base class thread management and synchronization.
CTunerVsb	Implements VSB specific tuning and scanning. It is derived from CTuner and leverages the base class thread management and synchronization.
CTunerSat	Implements Satellite specific tuning and scanning. It is derived from CTuner and leverages the base class thread management and synchronization.
CTunerOfdm	Implements OFDM (ISDB-T, DVB-T, and DVB-T2) specific tuning and scanning. It is derived from CTuner and leverages the base class thread management and synchronization.
CVideoOutput	Abstract base class for all video output types. It defines the abstract virtual methods the derived video output classes must implement.
COutputComponent	Derived from CVideoOutput, this class implements the component output.
COutputSVideo	Derived from CVideoOutput, this class implements the s-video output.
COutputComposite	Derived from CVideoOutput, this class implements the composite output.
COutputHdmi	Derived from CVideoOutput, this class implements the HDMI output.
CAudioOutput	Abstract base class for all audio output types. It defines the abstract virtual methods the drived audio output classes must implement.

COutputSpdif	Derived from CAudioOutput, this class implements the S/PDIF output.
COutputAudioDac	Derived from CAudioOutput, this class implements the DAC output.

CONTROLLERS

The controller is home to the “business logic” of the application. It responds to user actions (from the views) and issues commands to accomplish the requested task (to models). Notifications from the View are translated into appropriate command sequences using a mapping defined within the Controller. These commands directly query and modify the Models which typically results in notifications being sent to the View for a visual update to the user.

CControl

Simple data retrieval or basic changes can be accomplished by the view directly through one of the model modules. For example, a screen can query the decode model directly to retrieve decoding statistics. The main control module handles requests for orchestrating more complicated scenarios in the system such as channel change or PVR trick modes. Typically these commands require carefully crafted interaction involving multiple model modules. Keeping the implementation details of more complicated use-cases out of the model maintains model flexibility and provides a higher level interface for views to interact with.

Sequence Diagrams

The following sequence diagrams show additional details about many of the more complex scenarios that CControl must handle.

Application Startup

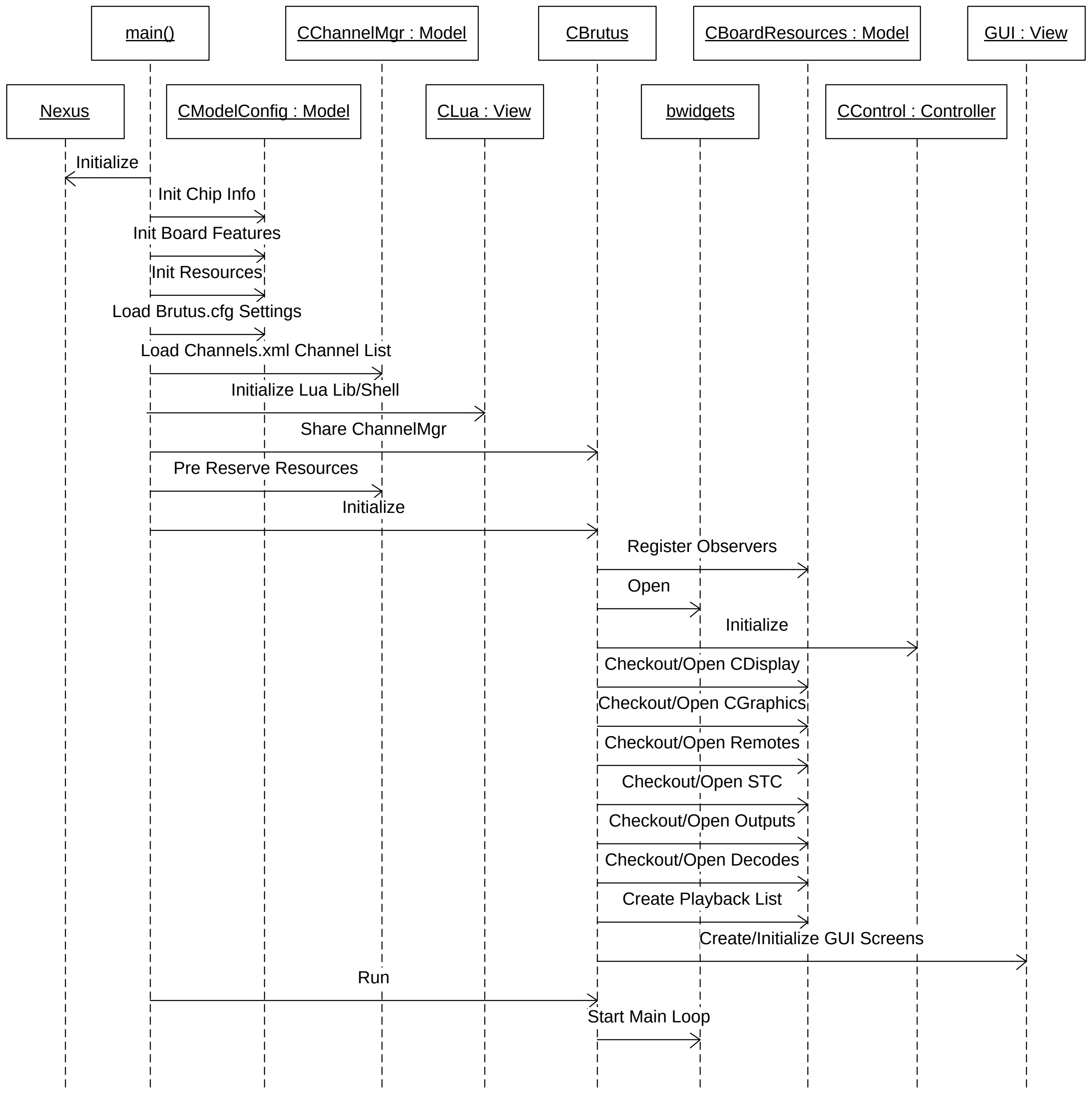


Figure 7 - Application Startup Sequence Diagram

Application startup begins with Nexus initialization and B_Os lib setup. Nexus is then used to query board information and capabilities. This data is used to create a list of available board resources in the CModelConfig. Brutus.cfg custom configuration settings are then loaded and added to the defaults in the CConfiguration object. These settings will turn features on and off throughout the application. Next, the default file containing the channel list (“Channels.xml” unless overridden in Brutus.cfg) is loaded from disk. The CChannelMgr uses this information to generate a list of CModelChannel objects.

The Lua scripting language library is initialized and the command shell is started. One or more instances of CBrutus are then created based on operating mode (Single Display Mode, HDSD Single Mode, Dual Display Mode). Resources are reserved for each instance based on mode.

Each instance of CBrutus is then initialized and their respective bwidget main loop is started. At this point the CBrutus instances are driven by incoming events from views and internal timers.

Channel Change

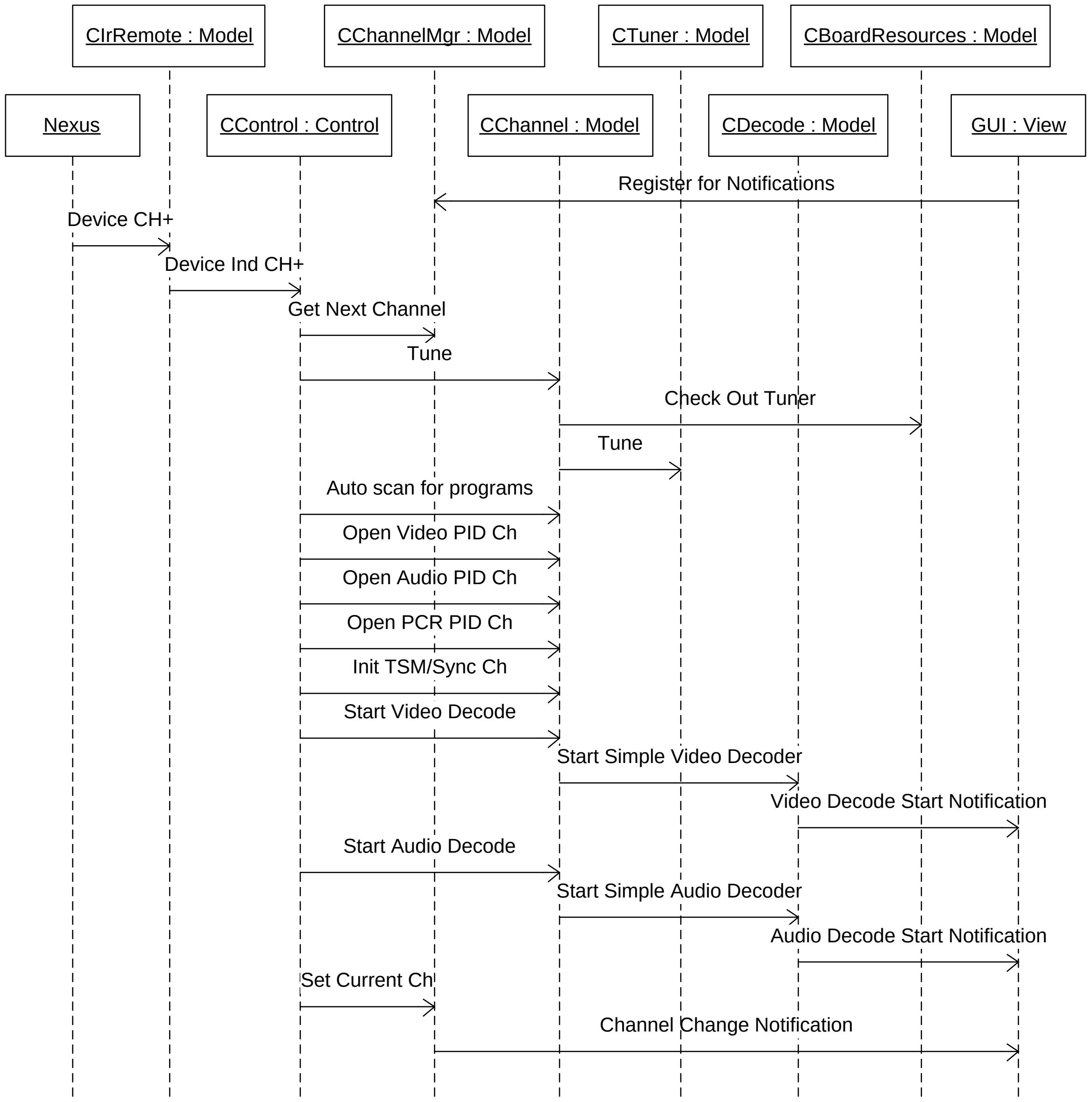


Figure 8 - Channel Change Sequence Diagram

Remote key presses are received by the various CRemote derived classes. Each remote class then synchronizes with the bwin main loop and then translates the received device specific remote code to a bwidget specific code. This bwidgets remote code is then sent to the bwidget library for processing. If the bwidget library does not consume the key, it is forwarded on to registered observers using the notification mechanism. For remote key notifications, CControl is the only observer.

In the case of a CH+ key press, CControl gets the next channel (based on current channel) from the CChannelMgr. Wraparound and channel order is maintained within the CChannelMgr. Once CControl has the next channel, it untunes the current channel and tells the next channel to tune itself. CControl is largely ignorant of what kind of channel it is accessing. All the specifics of how to tune a particular channel are hidden within based on the derived class type.

For this example, let us assume that the channel is a CChannelQam channel. It first checks out the board resources it needs such as a QAM tuner, input band, and parser band. It then directs the CTuner to do the actual tune operation. The details of the tuner operation are encapsulated within the CTuner object.

After tuner lock, CControl queries the CChannel for pids. If unavailable, CControl will retrieve PAT/PMT and add any found channels and pids. Once valid pids are retrieved, they are opened and then video/audio decoders are started. The current channel is then updated in the CChannelMgr model which in turn notifies registered observers of the change.

Channel Scan

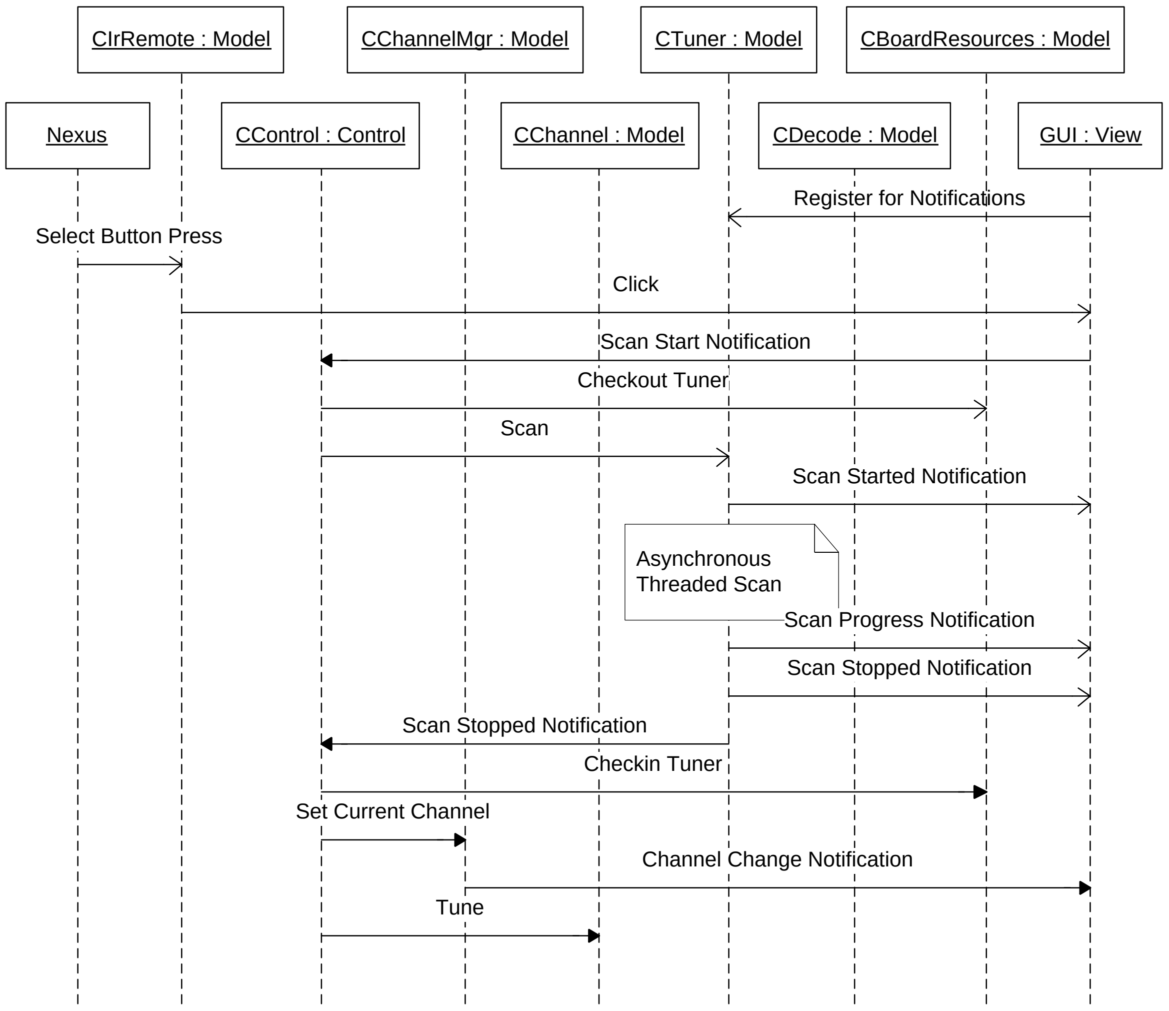


Figure 9 - Channel Scan Sequence Diagram

When the user clicks on the GUI scan button using the remote, a key press is received from Nexus. The CRemote object translates that key press into a bwidgets key press and forwards it on for bwidgets handling. The select key translates into a click event for the parent window of the widget that was clicked. This bwidgets click event is handled by the GUI view itself. The view determines in this case that the scan button was clicked and issues a notification to start a QAM scan.

CControl is a registered observer of the view and receives the scan start notification. It may need to untune the current channel and stop any associated recordings. CControl then checks out the appropriate tuner object and begins a scan. The scan itself is encapsulated within the particular tuner object since scanning is done differently depending on the particular tuner being used. This scan detail is hidden from CControl.

CTuner manages a threaded scan that does not block and allows the application to continue working while the scan is in progress. Tuner specific derived classes like CTunerQam inheirit this threading ability automatically. The scan thread communicates newly found channel objects to CControl using a callback function which in turn adds the new channel to the CChannelMgr's channel list. Scan start/progress/stop notifications are sent by the scan thread as the scan progresses.

Once the scan is stopped, CControl may need to set the current channel in the CChannelMgr (in the case where the previous channel list was replaced). It then tunes to the current channel. Updates to the GUI when the scan begins, while the scan progresses, and after it completes, are all accomplished entirely within the GUI views – Models and Control remain blissfully ignorant.

Playback

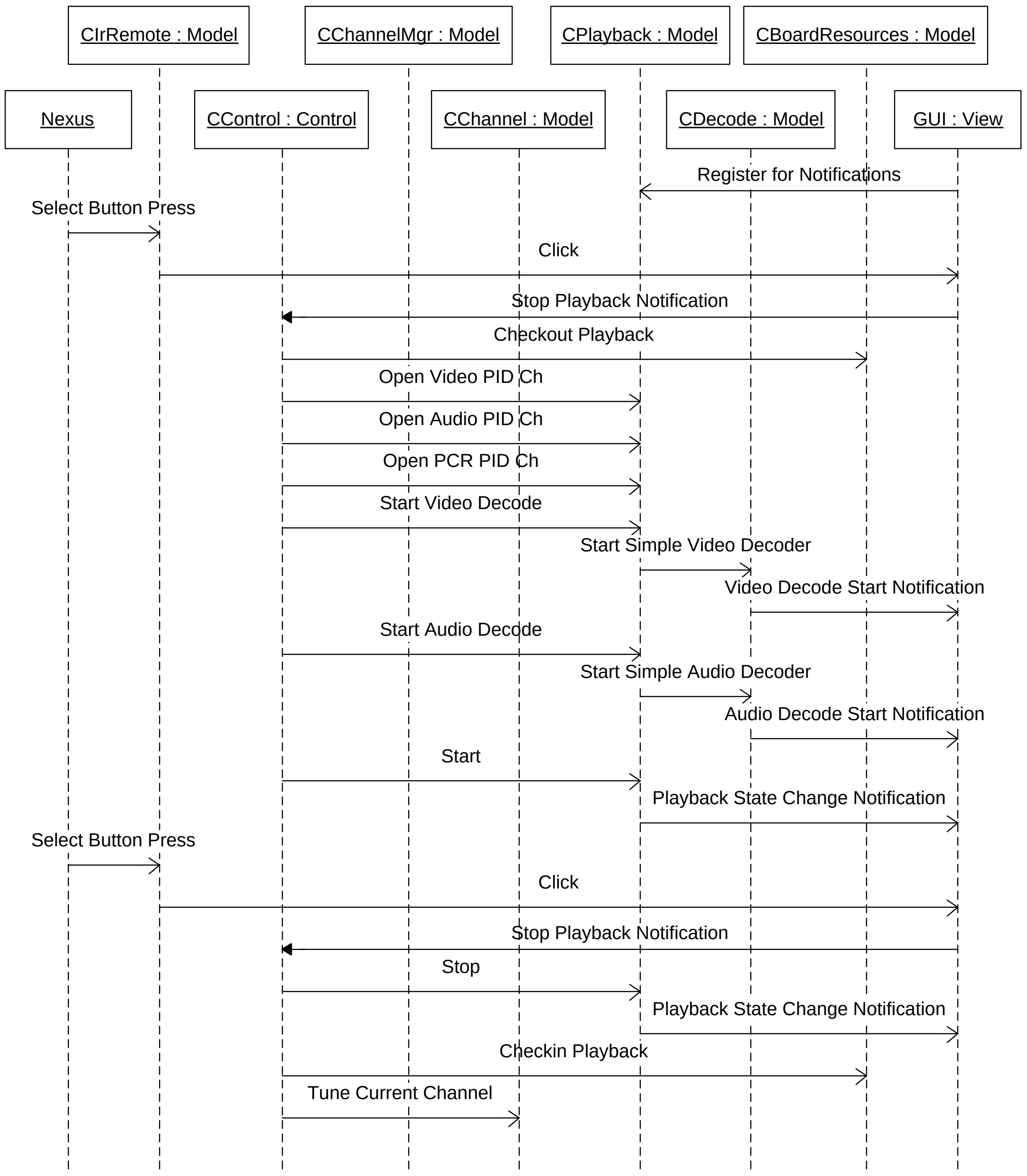


Figure 10 - Playback Sequence Diagram

The user clicks on a media file listed in the GUI playback screen. The CRemote object translates that key press into a bwidgets key press and forwards it on for bwidgets handling. The select key translates into a click event for the parent window of the widget that was clicked. This bwidgets click event is handled by the GUI view itself. The view determines in this case that the user is requesting to play a particular media file from disk and issues a notification to start a playback.

CControl is a registered observer of the view and receives the playback start notification. CControl then checks out the appropriate playback object. CPlayback is then initialized, video/audio pids are opened, decodes are started and then playback is started.

Once the playback is stopped, CPlayback is checked back into the CBoardResources. CControl then tunes to the current channel.

Recording

TBD

PVR Control

TBD

CLASS HIERARCHY

- brutus_board.h:**CChipFamily**
- brutus_board.h:**CChipInfo**
- brutus_board.h:**CBoardFeatures**
- brutus_board.h:**CBoardResources**
- brutus_cfg.h:**CConfiguration**
- brutus_os.h:**CScopedMutex**
- pid.h:**CPid**
  - pid.h:**CPidVideo**
  - pid.h:**CPidAudio**
  - pid.h:**CPidPcr**
  - pid.h:**CPidAncillary**
- video_window.h:**CVideoWindow**
- notification.h:**CNotification**

- notification.h:**CObserver**
  - mvc.h:**CMvcView**
  - mvc.h:**CMvcController**
    - controller.h:**CController**
      - control.h:**CControl**

- notification.h:**CSubject**
  - mvc.h:**CMvcView**
    - view.h:**CView**
      - brutus_lua.h:**CLua**
      - widget_base.h:**CWidgetBase**
        - widget_label.h:**CWidgetLabel**
          - screen.h:**CScreen**
            - screen_main.h:**CScreenMain**
          - panel.h:**CPanel**

- panel_audio.h:**CPanelAudio**
- panel_buffers.h:**CPanelBuffers**
- panel_decode.h:**CPanelDecode**
- panel_display.h:**CPanelDisplay**
- panel_playback.h:**CPanelPlayback**
- panel_record.h:**CPanelRecord**
- panel_scan_ofdm.h:**CPanelScanOfdm**
- panel_scan_qam.h:**CPanelScanQam**
- panel_scan_sat.h:**CPanelScanSat**
- panel_scan_vsb.h:**CPanelScanVsb**
- panel_timeline.h:**CPanelTimeline**
- panel_tuner.h:**CPanelTuner**

- mvc.h:**CMvcModel**
  - model.h:**CModel**
  - config.h:**CModelConfig**
  - channel.h:**CModelChannel**
    - channel_ofdm.h:**CModelChannelOfdm**
    - channel_ofdm.h:**CModelChannelQam**
    - channel_ofdm.h:**CModelChannelSat**
    - channel_ofdm.h:**CModelChannelVsb**
    - channel_ofdm.h:**CModelChannelStreamer**
  - channelmgr.h:**CModelChannelMgr**
  - pidmgr.h:**CPidMgr**
  - timer.h:**CTimer**
  - videolist.h:**CPlaybackList**
    - videolist.h:**CVideo**
  - resource.h:**CResource**
    - audio_decode.h:**CAudioDecode**
      - audio_decode.h:**CSimpleAudioDecode**
    - video_decode.h:**CVideoDecode**
      - video_decode.h:**CSimpleVideoDecode**
    - band.h:**CInputBand**
    - band.h:**CParserBand**
    - display.h:**CDisplay**
    - graphics.h:**CGraphics**
    - mixer.h:**CMixer**
    - remote.h:**CRemote**
      - remote.h:**CIrRemote**
    - playback.h:**CPlayback**
    - record.h:**CRecord**
    - stc.h:**CStc**
    - still_decode.h:**CStillDecode**
    - tuner.h:**CTuner**
      - tuner_ofdm.h:**CTunerOfdm**
      - tuner_qam.h:**CTunerQam**
      - tuner_sat.h:**CTunerSat**
      - tuner_vsb.h:**CTunerVsb**
    - output.h:**CVideoOutput**
      - output.h:**COutputComponent**
      - output.h:**COutputSVideo**

    output.h:**COutputComposite**
    output.h:**COutputHdmi**
output.h:**CAudioOutput**
    output.h:**COutputSpdif**
    output.h:**COutputAudioDac**

<END BRUTUS2 CHANGES>

Section 3: