# The *Galatea* Network Video Device Control System

## Version 2.5

*Daniel I. Applebaum*

Massachusetts Institute of Technology

M.I.T. Media Laboratory
20 Ames Street
Cambridge, MA 02139
danapple@media-lab.media.mit.edu

## Abstract

*Galatea* is a network transparent video device control system, providing reliable access to various video devices in a distributed network environment. This paper describes many features of *Galatea*, in addition to a description of the goals and strategies used in creating the system, and contains a manual for the C language programming interface.

## Acknowledgments

Several persons have helped in the development of Galatea. Each of their contributions is well appreciated. Hal Birkeland has made considerable improvements and fixes to much of the server implementation. The input of Ben Rubin and Paul Boutin has led to the implementation of many features in the current version. Patrick Purcell supported the early work on this system, from arranging for hardware, to consistantly pointing out bugs. Glorianna Davenport's constant encouragement and ideas were essential to the proper development of this system. Judson Harward wrote the manual pages, without which Galatea is largely incomprehensible.

Russ Sasnett provided most of the underlying device control system and device independant model for device control. These sections were part of an earlier project of his to provide network video device control. In addition, Russ has constantly added enhancements to the system, both to the device drivers, and critical portions of the server. Russ has played a principal role in the development of Galatea.

This project is supported by MIT Project Athena and the MIT Media Lab.

# 1. Introduction

## 1.1 Development Goals

Galatea was originally conceived to support the Electronic Light Table project at the MIT Media Lab. This project was an application which permitted users to retrieve images from video disks into small on-screen "slides." The intended purpose of this application was to provide users throughout the MIT campus with a simple method to access a large central bank of image data.

In order to control the centralized bank of video disk players, a client/server video control system was required. There were two such systems available for this purpose. One was created by Professor James Anderson of the MIT Architecture Department, the other by Russell Sasnett of MIT Film/Video. Although each of these systems provided most of the functionality required, neither could be used on a large enough scale to provide campus-wide distribution of video information.

During the summer of 1988, development began on a network video device control system capable of being scaled for campus-wide use. Campus-wide use did not mean that the system would be deployed on many standalone computer systems, but that there would be many systems interacting and competing for access to distributed video resources. The resources devoted to serving the whole campus were a pair of Digital Equipment Corporation VDP50’s video disk players and a BOA Source Controller II by Presentation Environments to switch the audio and video between the two players. These three units were controlled via RS232 by a DEC MicroVAX II which was connected to the campus computer network. The output of the BOA video switcher was modulated and transmitted over the MIT cable system. Any location requiring access to the video disk players needed access to both the MIT cable system and the campus computer network. Additionally, it was common for a user’s computer system to have its own local video resources. The "distributed" nature of the system stems from the need to control video devices on multiple computers without additional work by the user or the client applications. The results of these experiments with the centralized bank led to the development of the first version of Galatea.

The style of the programming interface was designed to be similar to the C language interface for the X Window System. Since many applications using video in this environment were going to be created using X, creating a similar programming interface eased the programmers burden. Galatea is also intended to be a companion to X, and works very well in that environment, although only one sample Galatea application actually requires the presence of the X Window System. Galatea is available under the same copyright restrictions as X. A person may acquire all of the source code from publicly available sources, but must maintain the MIT copyright notice on all copies.

## 1.2 Features and Use of Galatea

Galatea provides access to video devices in a distributed network environment. The system is designed in an enhanced client/server network model. A Galatea client, designed for a custom application, connects to a Galatea server on either the local or a remote machine. Once this connection is established, the client has access to the resources on the server. These can include video disk players, uncontrolled input sources and video/audio switchers. The video/audio switchers are not referenced directly, but instead controlled by the server when the client requests access to video disks or uncontrolled inputs. A *Volume Table* maintained by the server describes which discs are in which players, how the signal sources are connected to the switches and how the switches are connected together to form a switching tree, terminating in a number of server virtual outputs.

A single Galatea server can provide reliable access to many clients. The server dispatches on the incoming requests from clients, and guarantees each client a small time segment during which it has exclusive access. One client is not allowed to dominate the server, so that many clients and users can access a centralized bank of video resources without excessive delays.

### 1.2.1 Enhanced Client/Server Model

To provide additional flexibility, a Galatea server can connect to another server and utilize the resources maintained there. Once this connection is established, the resources on the secondary server are considered equivalent to local resources, except for certain system maintentance purposes. This provides to a simple client application the ability to access video resources on multiple machines without having to connect to several servers. There is only one Galatea server ever on one machine (CPU). The servers maintain the forwarding capability and have automatic systems for dealing with system crashes and network failures. Typically, a Galatea server runs on all machines on which you wish to run a Galatea client, even if the machine has no local resources. The local server has the responsibility of maintaining connections with the other remote servers. With this technique, a client program does not have to worry about a connection terminating if a remote machine crashes. The remote crash will be detected by the local server, which will then take appropriate action. Of course, a crash of the local server will bring down the client, but the local server should only crash if the entire machine goes down.

### 1.2.2 The Volume Table

A volume table is a list of video sources, each identified by name. A single volume does not necessarily correspond to a single video disk player or uncontrolled input channel. Multiple video disk players containing identical disks are combined into a single volume for client reference. Using this system, a client need only specify the name of the source that it wishes to access, and the server will deliver the video from the source most readily available. With this abstraction, there is no distinction made between resources on a local server and resources which are accessed through a chain of servers.

On the other hand, the same video device can also be a part of multiple volumes. A single volume is associated with a specific server virtual output. Since a video device can be visible on several virtual outputs, it may be part of several volumes, each of which is associated with a single virtual output. There is no ambiguity here, since a client is only passed a list of volumes which correspond to a single output, so there are never multiple volumes of the same name and type associated with the same virtual output. There may be multiple volumes with the same name if each volume is of a different type. The two types of volumes are record/play devices and uncontrolled input sources.

### 1.2.3 Server Virtual Outputs

A Galatea server is capable of serving multiple outputs, be they several channels on a cable television system or several displays on a single workstation. The identification of these outputs is site specific. In other words, the numbers used as identification for each output can be an arbitrary non-negative integer, in order to directly match cable channels or display numbers. These outputs are termed ’server virtual outputs’ since they represent the outputs of a "virtual switch" that Galatea creates out of all the physical switchers available. The configuration of the physical switchers into the virtual switch is desribed in the configuration file for galatead(8). When a client connects to a server, it claims the ability to view one particular virtual output. The client is then passed a list of volumes which are available on that output. A device can be viewable on multiple virtual outputs, if the underlying physical routing is setup appropriately.

## 1.3 Network Communication Protocols

Galatea uses two forms of network communication to provide the reliability and flexibility necessary in a distributed network environment. The first level is a client to server protocol which utilizes TCP/IP as the transport layer. This level handles commands and requests to the server for device action or queries about device state. The other communication level uses UDP transport layer for server to server communication to manage machine or network failure recovery.

### 1.3.1 Client to Server Communication

The Galatea server initially creates a TCP/IP socket for listening to requests from clients for connections. This socket is bound to a well known TCP/IP port number, usually 4001. When a client requests a connection to this socket, the server accepts this connection request by creating another socket which is used to serve that client. The initial socket is ready to be used again for accepting connections from more clients. This methodology is almost universal among TCP/IP based servers.

Once the client to server connection is established, the client may make requests to the server. The packet containing the request is structured. After the first request packet, the protocol is not clearly defined and can vary from command to command. The command request is defined by the following C language structure declaration.

```
typedef struct {
  int parm[10];
} Galatea_packet;
```

All of the integers in this packet are in network byte ordering, thus enabling hosts of different achitecture to communicate. The content of the first integer is defined to be a number indicating to the server what action should be performed. The server uses this number to branch to an appropriate routine to handle the rest of the command. The other integers in the packet are dependent on the particular command being executed. For example, many commands use the second integer to store the volume identifier for a volume action request.

If the command needs to send more data to the server than can fit in the initial command packet, additional data can be sent in a rather free-form manner. Strings and integers are passed as necessary for each command. Subroutines are provided to send and receive strings and integers.

There are two modes for specifying responses from the server. In synchronous mode, most requests to the server are acknowledged by the server. The acknowledgment takes the form of a single integer, which is either a return value that the client requested, such as the current frame number on a device, or a value indicating an error. Values above 999,999 and below -999,999 are reserved for error values. Some commands, for instance a request for statistics about the server, will return much more information than just a single integer response.

In some cases, however, it is advantageous for the return values from the server to be suppressed. This mode of operation is called asynchronous mode. Asynchronous mode is useful for *video shuttlebar* clients, which do not need responses to each request, and will operate faster if the client need not wait for the server responses. Requests which only query the server are usually not affected by asynchronous mode, since suppressing the returns in a query-only request is pointless.

After a request is handled in synchronous mode, the server blocks operation until a release command is issued from the client. This gives the client a short amount of time in which to use the results of its request. For instance, after searching to a frame on a Record Play Device, a client may digitize the frame. In order for this digitization to be reliable, the server must guarantee that no other client can manipulate any resource on the server during the digitization. Once the digitization is complete, the client sends the G_Release command to the server, freeing the server to accept commands from other clients.

### 1.3.2 Server to Server Communication

The protocol for server to server communication is designed to compensate for machine and network failures. Normally, a server can connect to another server as a regular client. The server which is acting as a client is called a "client" server, and the server which is a server for a "client" server is a "serving" server. A "client" server makes one request that a normal client never makes. This is a G_RequestNotification request. When a server receives such a command, it places the network address of the requesting client in a file on disk. By placing this information directly to disk, the server guards against spontaneous machine crashes.

Whenever a Galatea server rebuilds its internal volume table, it reads the list of hosts from disk, and notifies each host that it has rebuilt its volume table. The notification takes the form of a UDP packet containing the string "imhere". A server that receives such a UDP packet then rebuilds its own volume table. This technique propagates volume table rebuilds so that all of the servers which are affected by a change in device configuration have proper information.

When a Galatea server is shut down, it also sends out a "imhere" packet, but then does not accept new connections. This action forces "client" servers to rebuild volume tables, but since a connection can no longer be established to the "serving" server which is now shutdown, the "client" servers do not use any information about the devices on the now shutdown server.

Since a Galatea server builds its volume tables when it is first started, it sends out a UDP "imhere" packet when it is first started. This causes any "client" servers which were using the "serving" server before the machine stop to establish new connections to the now running "serving" server.

There is one common case which cannot be handled solely by "imhere" packets. This is where a network between two hosts fails, and the "client" server gets a network transmission or reception error. In this case, the "client" rebuilds its volume tables. While rebuilding, the "client" server will fail to connect to the "serving" server, and will not include the "server" server in the currently available volume table. When a "client" server fails to connect to a server, it establishes a pattern of sending query UDP packets to the server which was unavailable. These query packets contain the string "areyouthere". If a server receives a query packet, it sends an "imhere" to the host from which it received the "areyouthere" packet. This "imhere" packet will cause the querying server to rebuild its volume table, which should now include the previously unavailable server.

# 2. Glib - C Language Interface for Galatea

This is a description of Glib, version 2.5.

## 2.1 Opening a Galatea server

The first operation a Galatea client needs to perform before making use of video resources on a server is establishing a connection to the server. The action is accomplished with or GNewOpenServer() or GOpenServer():

```
Server *GNewOpenServer(hoststring)
char *hoststring;
```

**hoststring** Specifies the name of the system which is running the Galatea server to which you wish to connect and the server virtual output number you wish to use. The format for **hoststring** is "*hostname*:*output*" so that it closely matches the display name description in the X Window System.

GNewOpenServer() attempts to establish a connection to the Galatea server specified by **hoststring**. If **hoststring** is NULL, then GNewOpenServer() will attempt to use the GALATEA environment variable. If that does not exist, GNewOpenServer() will default to using "unix:0" as the **hoststring**. If the specified, or defaulted, hostname is "" or "unix" Galatea will attempt to connect to a local server, possibly using UNIX domain sockets.

GNewOpenServer() will eventually replace GOpenServer().

```
Server *GOpenServer(host, outputnum)
char *host;
int outputnum;
```

**host** Specifies the name of the system which is running the Galatea server to which you wish to connect. If **host** is NULL or "unix" Galatea will attempt to connect to a local server.

**outputnum** Specifies the number of the server virtual output which the client wishes to use. A server which is used by several several workstations or several displays on the same workstation may define independent outputs for each unit.

If a connection to the server cannot be established, a NULL is returned. Otherwise a legitimate connection has been established, and the returned structure should be used in future Glib calls.

```
Server *GOpenedHost()
```

GOpenedHost() returns the name of the host to which GNewOpenServer() or GOpenServer() has most recently either succeeded or failed in opening a connection. If no connection has yet been attempted, or an internal parsing error occurs, GOpenedHost() returns

NULL. A parsing error is probably caused by a misformatted host and output specification in the GNewOpenServer() argument or the GALATEA environment variable.

## 2.2 Obtaining information about the server

Several subroutines are provided for obtaining information about the contents of the Server structure. Clients applications should use these routines for referencing the structure, and never attempt to modify it or inspect it directly.

```
int GNumberExtensions(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

Returns the number of extensions that the server has available. Once this call is made, it is possible to interate down the list of extension names with the following command.

```
char *GExtensionName(server, which)
Server *server;
int which;
```

**server** The server connection as returned by GOpenServer().
**which** Specifies the position of an extension in the extension list.

Returns a pointer to a private area which contains the extension name of the extension at position **which** in the extension list.

```
int GNumberVolumes(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

Returns the number of volumes available on the server. The volume list is comprised of entries numbered from zero to the number of volumes minus one. After inquiring into how many volumes a server has, it is possible to interate down the volume list with the following three functions.

```
char *GVolumeName(server, which)
Server *server;
int which;
```

**server** The server connection as returned by GOpenServer().

**which** Specifies the position of a volume in the volume list.

Returns a pointer to a private area which contains the volume name of the volume at position **which** in the volume list.

```
int GVolumeIndex(server, which)
Server *server;
int which;
```

**server** The server connection as returned by GOpenServer().

**which** Specifies the position of a volume in the volume list.

Returns the index number of the volume at position **which** in the volume list.

```
int GVolumeType(server, which)
Server *server;
int which;
```

**server** The server connection as returned by GOpenServer().

**which** Specifies the position of a volume in the volume list.

Returns the volume type of the volume at position **which** in the volume list. The volume type can be one of RPD_DEV or INPUT_DEV, specifying a "Record/Play Device" or uncontrolled "Input Device," respectively.

```
int GWhichVolumeIndex(server, volume_name)
Server *server;
char *volume_name;
```

**server** The server connection as returned by GOpenServer().

**volume_name** String containing the name of the volume whose index number you wish to obtain.

Returns the index number of the specified volume on the given server. This command allows a client which knows exactly which volume it needs to find the volume index number easily. The index number of a volume is used in all server action requests.

`char *GWhichVolumeName(`**server**`, `**ind**`)`
`Server *server;`
`int ind;`

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume whose name you wish to obtain.

Returns the name of the volume whose index is **ind**. This is the reverse function of GWhichVolumeIndex().

`int GMaxLockTime(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

Returns the maximum time, in seconds, for which a client can acquire exclusive access to a server, either by a GLock() or by a GPlaySeg() in RETURN_SYNC mode. A lock which exceeds this time is silently broken by the server and the server will refuse to play a segment by GPlaySeg() which would exceed this time limit. A time limit of zero means that there is no limit on the length of a lock or segment. In a distributed environment, this is not reliable, since remote locks may be broken earlier.

`G_INT32 GRevisionNumber(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

Returns the revision number of the specified server, multiplied by ten. Since a revision number is represented as a decimal number, multiplying by ten eliminates the need to transmit floating point numbers over the network.

`int GConnection(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

Returns the connection number for the server. In UNIX, this is the file descriptor of the connection.

## 2.3 Closing the server connection

Once the client is finished accessing the resources of a server, it may close the server connection with GCloseServer().

```
GCloseServer(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

GCloseServer() closes the connection to the server and frees all the data in the server structure.

## 2.4 Controlling the switches

Once a server is opened and the client has determined which volumes are of interest, it is useful to set the server to display a particular volume of interest. The explicit switching command, GSwitch(), is used for this function:

```
int GSwitch(server, ind, mask)
Server *server;
int ind;
int mask;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume to which the switch should be made.

**mask** Bitwise or of one or more of LEFT_CHAN, RIGHT_CHAN, and VIDEO_CHAN, indicating which channel should be switched.

GSwitch() commands the server to arrange for the volume indicated to become the current source for one or more of the signal channels. Servers can typically switch left audio, right audio, and video independantly, although the independance of the two audio channels is not guaranteed. AUDIO_CHAN can be used to indicate the bitwise or of LEFT_CHAN and RIGHT_CHAN, and ALL_CHAN can be used to indicate the bitwise or of LEFT_CHAN, RIGHT_CHAN and VIDEO_CHAN. Note that switching does not explicitly determine a particular switcher to affect, as the server may have to traverse a tree of switchers to accomplish the desired result. Only the desired volume is necessary.

Many other commands can perform an implied switch for better performance. A switching mask is one of the parameters for those commands, for which a non-zero value indicates that switching should be performed. Using implied switching can increase speed of operation and reliable delivery. The only way to switch to an uncontrolled input source is GSwitch(), as none of the commands which implement an implied switch can operate on an uncontrolled input.

## 2.5 Manipulating uncontrolled input sources

As the name implies, there are no operations to be performed on an uncontrolled input source. The only command which can take the index number of an uncontrolled input source is the GSwitch() command, explained above.

## 2.6 Manipulating record/play devices

There are several methods of controlling a record/play device, ranging from variable speed play, single frame search to recording new frames. Several of these commands have a switching mask as an argument. When this mask is non-zero, it indicates to the server that after the requested operation is performed, a equivalent action to GSwitch() should be undertaken by the server. Therefore the end result of:

```
GSearch(server, ind, 20000, ALL_CHAN);
        is the same as the result of:
GSearch(server, ind, 20000, 0);
GSwitch(server, ind, ALL_CHAN);
```

The former will typically be a smoother and faster operation. Also, there is no chance that another client can access the server and disturb the arrangements in the first case, as can happen in the second. Only the results of a single command are guaranteed to be handled without another client interfering, unless GLock() is used to lock the server. The GRelease() command is used to indicate to the server that client is finished with a particular request. When a request is made to a server, the server usually maintains the results of that request until the client calls GRelease(), or the server times out on the release. The client can indicate (with GSetState()) that an asynchronous mode should be used, in which the server does not maintain results. If the client issues another command, instead of a GRelease(), that command implies a GRelease() and the server then handles any other incoming requests, not necessarily the command that acted as a GRelease(). This algorithm ensures that a single client cannot dominate a server by simply sending a stream of commands without GRelease()'s. Additionally, servers typically time out on releases in five seconds in order to allow other requests to be processed.

### 2.6.1 Playback commands

```
int GSearch(server, ind, frame_num, mask)
Server *server;
int ind;
int frame_num;
int mask;
```

| | |
|---|---|
| **server** | The server connection as returned by GOpenServer(). |
| **ind** | The index number of the volume on which you wish to search. |
| **frame_num** | The absolute frame number to which the volume should be searched. |
| **mask** | A switching mask used for an implied switch. |

GSearch() requests the server to search on the requested volume to the specified frame number. At the end of this operation, GSearch() returns the frame number which the volume is currently displaying. This command may not simply cause a search on a single video disk player. If multiple copies of a volume are available, the server will use the copy which will provide the fastest response time. This action is invisible to the client. After the search is complete, an implied switch is performed, if **mask** is non-zero.

```
int GRun(server, ind, speed, mask)
Server *server;
int ind;
int speed;
int mask;
```

| | |
|---|---|
| **server** | The server connection as returned by GOpenServer(). |
| **ind** | The index number of the volume you wish to play. |
| **speed** | The speed, in frames per second, at which the volume should be played. |
| **mask** | A switching mask used for an implied switch. |

GRun() sets the requested volume in motion at the speed indicated. The speed is specified in frames per second, and the server will attempt as well as possible to match the requested speed. Not all players can play at an arbitrary frame rate. A closest match is attempted. The actual speed of playback is reported back to the client as the return value. The speed can be positive or negative. The switching mask is handled as for GSearch(). The play continues until one of the ends of the disk is reached, or another motion command is executed.

```
int GJog(server, ind, direction, mask)
Server *server;
int ind;
int direction;
int mask;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume you wish to jog.

**direction** The direction in which the volume should be jogged.

**mask** A switching mask used for an implied switch.

A video jog is a single frame motion. This provides an easy way to step around a small portion of a video disk. The directions possible are FORWARD and REVERSE, only one of which may be specified. The new current position on the disk is returned.

```
int GStill(server, ind, mask)
Server *server;
int ind;
int mask;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume you wish to stop.

**mask** A switching mask used for an implied switch.

To stop a running volume, use GStill(). The requested volume will be stopped and the current frame number of the volume will be returned to the client.

```
int GPlaySeg(server, ind, start, end,speed, mask,
        return_flag)
Server *server;
int ind;
int start;
int end;
int speed;
int mask;
int return_flag;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume you wish to play.

**start** The starting frame number from which the segment will be played.

**end** The ending frame number to which the segment will be played.

**speed** The speed at which the segment will be played.

**mask** A switching mask used for an implied switch.

**return_flag** Value indicating whether the server should block until the end of the segment.

GPlaySeg() arranges to play a fully specified segment of video from a volume. The volume is first searched to the start frame, and playback is then assumed at the specified speed. Playback is stopped at the end frame. The implied switched in a GPlaySeg() is performed after the initial search and before the playback begins. The **return_flag** indicates to the server whether to block until the segment is complete. A client may wish to block until a segment is complete, or it may wish to simply start the segment and let it progress, while returning the client to a running state. RETURN_NOW is the flag for starting the segment and then immediately returning control back to the client. RETURN_SYNC indicates to the server that the server should block until the segment is finished. Some servers may place a limit on the length of a segment requested with RETURN_SYNC, as such a segment might interfere with the operation of other clients. GPlaySeg() normally returns the frame number at which the server returned control to the client. This may be early in the segment for a RETURN_NOW, and should be the **end** frame for a RETURN_SYNC. A client may also specify a **return_flag** of RETURN_NOW_WITH_SPEED. This requests that control should be returned to the client immediately, but that GPlaySeg() should return the current speed of the device in frames-per-second, as if from GGetSpeed() (see GRun() and GGetSpeed() for an explanation of returned speed values). If **start** is CURRENT_FRAME, then the initial search is inhibited, and the segment is played from the current frame to the **end** frame.

```
int GPlaySequence(server, numsegs, segs, return_flag)
Server *server;
int numsegs;
GSegment *segs;
int return_flag;
```

**server** The server connection as return by GOpenServer().

**numsegs** The number of segments that make up the sequence.

**segs** An array of segments to be played in order.

**return_flag** Value indicating whether the should should block until the end of the sequence.

GPlaySequence() plays a sequence of video segments specified by **segs**. Each element of **segs** is a GSegment structure which is defined as:

```
typedef struct {
  int volumenum;
  int start, end;
  int speed;
  int mask;
} GSegment;
```

This structure defines a segment similar to the segments defined in GPlaySeg(). The volume specified by **volumenum** is played, starting at frame **start**, ending at frame **end**, at speed **speed**. If **speed** is 0, then **end** is a number of microseconds for which the frame specified by **start** is displayed. The **mask** is used as the switching mask used just before the particular segment is played. The **return_flag** indicates whether the client should be blocked while the sequence is playing. RETURN_NOW indicates that the client will return as soon as the sequence starts, RETURN_SYNC indicates that the client will return when the sequence is completed. In RETURN_NOW mode, the sequence may be interupted by another client, unless the server is locked, but in RETURN_SYNC mode, the sequence can not be interupted. The RETURN_NOW_WITH_SPEED option from GPlaySeg() may not be used in a GPlaySequence() request.

GPlaySequence() will attempt to do simple optimizations to eliminate the inter-segment delay. Some pre-searching will be performed, but this routine does not guarantee "seamless" playback. If each segment is longer than the maximum search time of the video devices, and at least two devices are available, there should never be searching delays, only switching and startup delays, between consecutive segments. In addition, there is no way to overlap the audio of one segment with the video of an adjacent segment.

### 2.6.2 Record commands

Currently, recordable volumes are only considered recordable by a *Galatea* server if the volume’s device is local to the server. In other words, record requests cannot be forwarded. Also, volumes containing multiple devices are not ever recordable. *Galatea* also assumes that recordable devices are write-once, not write-many.

```
int GAlloc(server, ind, nframes)
Server *server;
int ind;
int nframes;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume on which the allocation should be made.

**nframes** The number of frames requested in the allocation.

GAlloc() allocates **nframes** contiguous frames on the volume indicated by **ind**. If the allocation succeeds, then the starting frame number of the allocated block is returned and the client may proceed to record onto any or all of the frames in the block. The allocation may fail because the device is not a recorder (FUNC_NOT_SUPPORTED), the device is too remote (FUNC_NOT_SUPPORTED), or the device does not have a sufficient number of free frames (COULDNT_ALLOC).

```
int GAllocAtFrame(server, ind, start, nframes)
Server *server;
int ind;
int start;
int nframes;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume on which the allocation should be made.

**start** The starting frame number for the requested block.

**nframes** The number of frames requested in the allocation.

GAllocAtFrame() functions similarly to GAlloc() except that the allocated block will start with the frame number specified by **start**. GAllocAtFrame() is useful when a recording is being continued from a previous session, and the new material must have a precise relationship to the previously recorded material.

```
GFree(server, ind, start, nframes)
Server *server;
int ind;
int start;
int nframes;
```

**server** The server connection as returned by GOpenServer().

**ind** The index number of the volume on which the allocation should be freed.

**start** The starting frame number to be freed.

**nframes** The number of frames requested to be freed.

GFree() frees part or all of a previously allocated block, allocated by GAlloc() or GAllocAtFrame(). A client should only attempt to free frames which were not recorded. Frames that have been recorded can not be freed, since they will not be available for subsequent record operations.

GRecord(**server**, **destvolume**, **start**, **end**, **speed**, **mask**,
**sourcevolume**, **return_flag**)

```
Server *server;
int destvolume;
int start;
int end;
int speed;
int mask;
int sourcevolume;
int return_flag;
```

**server** The server connection as returned by GOpenServer().

**destvolume** The index number of the volume on which the recording should take place.

**start** The first frame on which recording should occur.

**end** The last frame on which recording should occur.

**speed** The speed at which the destination volume should record. (This paramter is currently ignored.)

**mask** A switching mask used for an implied switch.

**sourcevolume** The index number of the volume from which the recording should take place.

**return_flag** Value indicating whether the server should block until the end of the recorded segment.

GRecord() records the video and audio from one volume onto a section of another volume. The **sourcevolume** may be either an uncontrolled input or another record/play device. The **start**, **end**, **speed**, **mask**, and **return_flag** operate the same as in GPlaySeg(). The **start**, **end**, **speed**, and **return_flag** parameters affect only the destination volume. The **mask** parameter affects the routing between the source volume and the destination volume. Before recording takes place, the parameters are checked to make sure the recording can take place. Possible failures include BAD_VOLUME if either the source or destination volume is not valid, VIOLATE_ALLOC if the specified destination segment has not been previously allocated by the client, FUNC_NOT_SUPPORTED if the specified destination volume is not a recordable volume, COULDNT_ROUTE if the output of the source volume could not be patched to the input of the destination volume, COULDNT_DO_FUNC if the record command failed, and COULDNT_FREE if the segment was recorded correctly, but could not be removed from the allocation.

```
GRecordPreview(server, destvolume, sourcevolume,mask)
Server *server;
int destvolume;
int sourcevolume;
int mask;
```

| | |
|---|---|
| **server** | The Server connection as returned by GOpenServer(). |
| **destvolume** | The index number of the volume on which the record preview should take place. |
| **mask** | A switching mask used for an implied switch. |
| **sourcevolume** | The index number of the volume from which the record preview should take place. |

GRecordPreview() allows the client to do all the routing associated with a GRecord() without actually performing the recording. The source volume is routed to the destination volume according the the switching mask and the destination volume device is placed into a record standby mode. This is useful to preview what the recording will look like, or to allow the destination volume to sync to the source volume. It may be necessary to position the destination to a recordable frame before this command is used, because some devices will not enter the record standby mode unless the device is positioned at an empty frame. The error COULDNT_DO_FUNC is returned if the record standby mode fails, possibly due to the device not being positioned at an empty frame. COULDNT_ROUTE is returned if the source volume could not be routed to the destination volume.

`GDub(`**server**`,` **destvolume**`,`**deststart**`,`**destend**`,`**destspeed**`,`
**srcvolume**`,` **srcstart**`,`**srcspeed**`,` **mask**`,` **return_flag**`)`

```
Server *server;
int destvolume;
int deststart;
int destend;
int destspeed;
int srcvolume;
int srcstart;
int srcspeed;
int mask;
int return_flag;
```

| | |
|---|---|
| **server** | The server connection as returned by GOpenServer(). |
| **destvolume** | The index number of the volume on which the recording should take place. |
| **deststart** | The first frame on which recording should occur. |
| **destend** | The last frame on which recording should occur. |
| **destspeed** | The speed at which the destination volume should record. (This paramter is currently ignored.) |
| **srcvolume** | The index number of the volume from which the recording should take place. |
| **srcstart** | The first frame on the source volume that should be recorded onto the destination. |
| **srcspeed** | The speed at which the source volume should play. |
| **mask** | A switching mask used for an implied switch. |
| **return_flag** | Value indicating whether the server should block until the end of the recorded segment. |

GDub() records a section of the contents of one RPD onto another RPD. The dub is not guaranteed to be frame accurate. In order to do the dub, the server takes the following steps:

```
Setup a route from srcvolume to destvolume using mask.
Search srcvolume to srcstart.
Search destvolume to deststart.
Prepare destvolume for recording.
Start playing srcvolume at srcspeed frames per second.
Start recording destvolume until frame destend.
Stop srcvolume.
```

Possible failures include BAD_VOLUME if either the source or destination volume is not valid, VIOLATE_ALLOC if the specified destination segment has not been previously allocated by the client, FUNC_NOT_SUPPORTED if the specified destination volume is not a recordable volume, COULDNT_ROUTE if the output of the source volume could not be patched to the input of the destination volume, COULDNT_DO_FUNC if the record command failed, and COULDNT_FREE if the segment was recorded correctly, but could not be removed from the allocation.

### 2.6.3 Other device operations

Other operations possible on a record/play device are changing the load state of the volume, changing the visibility of player frame index numbers, changing the audibility of the two audio channels, and getting the current frame number.

```
int GLoad(server, ind, on_off)
Server *server;
int ind;
int on_off;
```

**server** The server connection as returned by GOpenServer().
**ind** The index number of the volume you wish to load.
**on_off** The new load state requested.

GLoad() allows the client to manually load, unload or eject disks during runtime. This allows the user the change disks dynamically for some applications. An **on_off** value of LOAD requests that the volume should be spun up, or loaded. A value of UNLOAD requests that the volume should be spun down, or unloaded. A value of EJECT requests that the disks be spun down and the disk player lids opened. If an EJECT request is sent, the Galatea will rebuild its volume tables, since a player with an open lid is not considered available. When a Galatea server is properly configured, changing the disks in a player where multiple copies of a disk exist may cause odd results, as the server assumes that all of the players contain the named disk.

```
int GConfigure(server, ind, chan, on_off)
Server *server;
int ind;
int chan;
int on_off;
```

**server** The server connection as returned by GOpenServer().
**ind** The index number of the volume whose configuration you wish to change.

**chan** Specifies the channel or channels whose configuration should be changed.
**on_off** The new configuration state requested.

When the client needs to change the audibility of the audio channels, or change the state of the frame index numbers, a call to GConfigure() should be made for a specific volume. The **chan** parameter indicates whether the left audio channel, the right audio channel, the audio squelch mode or the index numbers should be changed by the bitwise **or** of LEFT_CHAN, RIGHT_CHAN, SQUELCH_CTRL and INDEX_CTRL. Any combination of these may be specified. The **on_off** parameter indicates the desired state of the specified channel. Values of TURN_ON and TURN_OFF can be specified.

```
int GGetFrame(server, ind)
Server *server;
int ind;
```

**server** The server connection as returned by GOpenServer().
**ind** The index number of the volume whose current position you wish to obtain.

GGetFrame() provides for the client to obtain the current position of a volume. If the user has searched maunally using GRun() commands for a particular frame, and now wishes to note the frame number, GGetFrame() can be used by the application to acquire that information. The current frame number is returned.

```
int GGetSpeed(server, ind)
Server *server;
int ind;
```

**server** The server connection as returned by GOpenServer().
**ind** The index number of the volume whose current speed you wish to obtain.

GGetSpeed() provides for the client to obtain the current speed of a volume. The current speed, in frames per second, is returned.

## 2.7 Various server functions

There are several server requests which do not specify or access particular resources on the server, but do control server behavior.

`int GRelease(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

GRelease() releases the server from maintaining the results of the previous command. If the server is not maintaining results because the library is in ASYNC_MODE mode, this command has no effect.

`int GLock(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

GLock() establishes a temporary lock on all the resources of the server. This allows a client to be guaranteed of exclusive access to all resources on the server without other client interfering. These locks are typically limited in length by servers so that other clients are not permanently locked out of a server. Any other client that attempts to access the server during a locked period is simply blocked. If the locking client does not unlock the server before the time limit is reached, the lock is silently broken by the server. The maximum lock time is available with the GMaxLockTime() call. Using locks is encouraged for clients which need to ensure that a volume of interest is completely set up for them for a short period of a time. Say, to GConfigure(), then GSearch(), then perform a still frame grab with an external processor, then unlock. GLock() is the only method to guarantee that no other client will intrude between the GConfigure() and the GSearch(). Also, if the frame grabbing board takes more time to grab than the standard release time, this mechanism provides for a slightly longer leeway after the GSearch().

(Caution: locks that must be forwarded between servers may provide less than the normal maximum lock time.)

`int GUnlock(`**server**`)`
`Server *server;`

**server** The server connection as returned by GOpenServer().

GUnlock() unlocks a server which has been locked by GLock(). Clients are encouraged to unlock a server as soon as possible, in order to allow other clients access to server resources.

```
G_INT32 GGetServerTime(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

GGetServerTime() returns the current time on the server machine in seconds since midnight, January 1, 1970. This information is useful in realizing volume scheduling systems.

```
GGetStatistics(server, stat_struct)
Server *server;
GStats *stat_struct;
```

**server** The server connection as returned by GOpenServer().

**stat_struct** A structure into which the server statistics will be placed.

GGetStatistics() is used to get information regarding the server performance. The following structure is filled in:

```
typedef struct {
int revision;
int num_requests;
int run_time;
int num_collisions;
int num_connections;
int max_connections;
int delayed_requests;
} GStats;
```

**revision** Indicates how many times the server has rebuilt its internal volume table.

**num_requests** Specifies how many client requests have been made to the server. (Includes the current GGetStatistics() request.)

**run_time** The number of seconds for which the server has been running.

**num_collisions** The number of times two client requests have arrived at the server simultaneously. **num_connections**
The total number of connections made to the server.

**max_connections** The maximum number of simultaneous connections made to the server.

**delayed_requests** The total number of client requests that were delayed because the server was busy processing another request.

```
int GCheckRevision(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

GCheckRevision() allows a client to manually check if its volume list (contained in the server structure) is up to date. Normally, an out of date volume list is indicated by the error message, VOLUME_LIST_OLD, returned by the server on most other commands. GCheckRevision() should only return NO_ERROR, VOLUME_LIST_OLD, or GIO_ERROR. NO_ERROR means that the client has an up to date volume list. VOLUME_LIST_OLD means that the client should get a new volume list with GReopenServer(). GIO_ERROR is encountered when a connection to a server dies. Unless a client is handling SIGPIPE's however, the SIG_PIPE will probably arrive first, and the client will exit.

```
Server *GReopenServer(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

GReopenServer() is used by a client to obtain a new volume list, if the server indicates that the client has an out of date volume list. This situation can occur if a secondary server crashed, or comes on-line, and the primary server rebuilds its internal volume tables to match the new availibilty of resources. A server can also be forced to rebuild its volumes tables by sending it a SIGHUP. GReopenServer() returns a pointer to a new server structure, or NULL, if there was some failure.

```
GRebuild(server)
Server *server;
```

**server** The server connection as returned by GOpenServer().

GRebuild() simply requests that the specified server rebuild its volume table.

```
GShutOff(server, up_down)
Server *server;
int up_down;
```

**server** The server connection as returned by GOpenServer().

**up_down** An integer value indicating whether the server should leave the disks in a spun up or spun down state.

GShutOff() requests that the server shut itself down, possibly spinning down the disks first. An **up_down** value of SPINDOWN will make the server spin down the disks before exiting. A value of NOSPINDOWN will leave the disks in the spinning state when the server exits.

```
GMount(server, disc_name, dev_name)
Server *server;
char *disc_name;
char *dev_name;
```

**server** The server connection as returned by GOpenServer().

**disc_name** The name of the disc being placed in a device.

**dev_name** The name of the device into which the specified disc is placed.

GMount() allows a client program to indicate to the server that the contents of a device have been replaced with a new disc. **dev_name** is the name of the local device on which the new disc, **disc_name** is to be mounted. GMount() can only be performed by a client on the same host as the server, and only on a device local to that server. Once the mount is performed, the server rebuilds its volume table so that the mount takes place immediately.

```
GUmount(server, dev_name)
Server *server;
char *dev_name;
```

**server** The server connection as returned by GOpenServer().

**dev_name** The name of the device which is to be taken off-line.

GUmount() is used to bring a local device, on the local server, off-line. The server immediately rebuilds its volume table, but ignores the now unmounted device.

```
GMounts(server, num_mounts, disc_names,dev_names, rets)
Server *server;
int num_mounts;
char **disc_names;
char **dev_names;
int *rets;
```

**server** The server connection as returned by GOpenServer().

**num_mounts** The number of mount operations to perform.

**disc_names** An array of disc names to be mounted.

**dev_names** An array of device names to be mounted upon.

**rets** An array to contain the return values from each mount operation.

GMounts() performs like GMount(), except that multiple mount operations are performed with a single request. The advantage of GMounts() is that **num_mounts** mounts can be performed with only a single ensuing volume table rebuild. If GMount() were called multiple times, a volume table rebuild would be performed for each call. Each mount performed by GMounts[] is comprised of a disc name from **disc_names** and the corresponding device name from **dev_names**. The return value of each mount is placed in the **rets** array, which should have already been allocated by the client program.

```
GUmounts(server, num_umounts, dev_names,rets)
Server *server;
int num_umounts;
char **dev_names;
int *rets;
```

| | |
|---|---|
| **server** | The server connection as returned by GOpenServer(). |
| **num_umounts** | The number of unmount operations to perform. |
| **dev_names** | An array of device names to be unmounted. |
| **rets** | An array to contain the return values from each unmount |

GUmounts() performs **num_umounts** unmount operations. The devices to be unmounted are specified in the **dev_names** array, and the return values from each unmount are placed in the corresponding entry in the **rets** array, which should already have been allocated by the client program. Once all of the unmounts are accomplished, a volume table rebuild is performed.

```
GGetMounts(server, num_mounts, mount_devs,mount_discs)
Server *server;
int *num_mounts;
char ***mount_devs;
char ***mount_discs;
```

| | |
|---|---|
| **server** | The server connection as returned by GOpenServer(). |
| **num_mounts** | Returns the number of mountable devices. |
| **mount_devs** | Returns the list of mountable devices. |
| **mount_discs** | Returns the list of discs mounted on the devices. |

GGetMounts() retrieves the list of mountable devices from the server, along with the currently mounted disc for each device. The number of mountable devices is returned in

**num_mounts**, and the arrays **mount_devs** and **mount_discs** are allocated to accommodate the returned arrays. The client program must free the arrays when finished. If a device has nothing mounted, then the corresponding disc name is returned as [nothing].

```
GRequestNotification(server, yesno)
Server *server;
int yesno;
```

**server** The server connection as returned by GOpenServer().

**yesno** An integer value indicating whether to enable notification.

GRequestNotification() should never be used by an ordinary client. It is used by servers to request that other servers notify them of volume table rebuilds. A **yesno** value of Notify indicates that notification to this host should be enabled; and value of NoNotify indicates that notification to this host should be disabled. Notification is only performed on a per-host basis, not on a per-client basis. The notification takes place over a UDP channel independent of the TCP connection used for normal communication.

## 2.8 Routines which change the action of the Galatea library

The Galatea library (Glib) can be set to act in several different modes. There are debug settings, timeouts, and asynchronous operation modes, as well as an error handling system.

```
int GSetState(debug_mode, set_time, set_sync)
int debug_mode;
struct timeval *settime;
int set_sync;
```

**debug_mode** Specifies the new debug mode for library operation.

**set_time** Specifies the new time out length on reads from the server.

**set_sync** Specifies whether the library should act synchronously or asynchronously.

GSetState() sets several modes of library operation. Debugging allows the library to print out some information as routines are called. This is mainly used in debugging the library, so the information printed may not be very consistent or useful. A value of one enables debugging and a value of zero disables debugging.

The maximum time that the library will wait for a reply from the server can be set with the **set_time** argument. If NULL, the time out is infinite. Infinite is the default library time out. If

**set_time** is non-NULL, the time contained in the timeval structure is copied and used for the new time out.

The library can also be set to use asynchonous or synchronous operation. Normally, all commands send a request to the server and then wait for a reply from the server indicating the request has been completed or that an error occured. This is called synchronous operation, since the client is kept synchronized to the server. For some applications, say a video shuttle control, acknowledgment of successful completetion is not necessary, since many commands are being sent, and single failures are not a problem. In the shuttle application, it is also unnecessary to wait for synchronization, and that wait can cause poor performance. For such applications, setting the **set_sync** argument to ASYNC_MODE will provide better performance. In this mode, many commands will report that no error occurred. For some commands, such as GReopenServer(), asynchronous mode is meaningless. Some other commands also act synchronously, even in ASYNC_MODE, such as GLock() and GCheckRevision(). The other change in ASYNC_MODE is that the server will not wait for a GRelease() from a client. A **set_sync** of SYNC_MODE returns the library to synchronized operation.

```
int GSetErrorHandler(handler)
void (*handler)(Server *, int);
```

**handler** A handler procedure to be called when errors are reported by a server.

Many Galatea commands can be met with an error response from the server. These errors are normally reported back to the client through the return value for the command. It is possible to intercept these error messages, and cause the library to invoke a client specified error handler. A very useful error handler is one that can deal with a VOLUME_LIST_OLD error, execute a GReopenServer() and then update client structures or displays before returning. Such a system reduces code duplication. The command that caused the server still returns the error code, but the main sections of the client application need not handle the error explicitly. A NULL for the **handler** procedure, causes the library to resume default action, which is to call GReopenServer() on the server which returned the error. This permits the following Galatea calls to procede normally.

The arguments to the error handler are the server on which the error was generated and the error code returned by the server. The return value from the handler is not used.

## 2.9 Error messages

Almost all Galatea library commands can cause an error to be generated on the server, if there is a fault in the function call or a device failure. The values returned are integers (32 bits). The return value of NO_ERROR is used to indicate simple completion of request which does not have an otherwise important return value. All of the actual error return values values have a value greater than or equal to LOWEST_ERROR.

A routine is provided to return a string which corresponds to a particular error:

```
char *GErrorString(error_value)
G_INT32 error_value;
```

**server** The error value as returned from other functions.

GErrorString() returns a pointer to a static string which contains a description of the specified error. The application should not manipulate the string referenced by this pointer.

The actually error return codes are as follows:

**VOLUME_LIST_OLD**
The volume list that the client is using is not up to date with the current version on the server. GReopenServer() should be called to get a new volume list.

**BAD_VOLUME** The volume index number specified in the command is not a valid volume in the server.

**FUNC_NOT_SUPPORTED**
An attempt was made to execute a function on a volume which cannot support the operation. An example of this is trying to do a frame search on an uncontrolled input device.

**BAD_COMMAND**
The server was not able to interpret the request as a reasonable command. This is probably due to a mismatched library/server combination.

**COULDNT_SEARCH**
A device error was encountered while attempting to search on a record/play device. A retry may succeed.

**COULDNT_STOP**
A device error was encountered while attempting to stop a record/play device. A retry may succeed.

**COULDNT_CHANGE_SPEED**
A device error was encountered while attempting to change the speed or direction of a record/play device. A retry may succeed.

**COULDNT_DO_FUNC**
A device error or internal server error was encountered while attemping an operation on any type of device. A retry may succeed.

**BAD_ARGUMENT**
An argument to a command was not valid.

**GIO_ERROR** A server connection has been broken. This error indicates the the server to which the client is connected has crashed or been shutdown. If a client receives this error, it should not attempt to access the Galatea server again.

**SEGMENT_TOO_LONG**
A segment requested in a GPlaySeg() or GRecord() exceeded the maximum lock time on the server. An attempt for the same segment in RETURN_NOW mode will probably succeed, but the segment may be interrupted.

**PERMISSION_DENIED**
An attempt to access a device or operation for which the user has insuffucient privleges was made. This includes attempting to load, unload, mount or unmount volumes on a remote host.

**COULDNT_ROUTE**
An attempt to switch either the viewable volume, or a recording cross-route failed due to a switcher error. A retry may succeed.

**COULDNT_ALLOC**
An attempt to allocated recordable frames failed. The recordable device has run out of available frames. This is analogous to malloc() returning NULL.

**COULDNT_FREE**
The server was unable to free the specified block.

**VIOLATE_ALLOC**
The client requested to record onto frames which are outside of any allocated segments.

**BAD_OUTPUT** The client claimed to have access to a server virtual output which does not exist.

# 3. Installing Galatea

## 3.1 Customizing the compilation process

The Galatea distribution is set up to compile on a 4.3 BSD derived system. Without any changes, Galatea should compile on Ultrix, Sun OS, Athena UNIX, and 4.3 BSD. With minimal changes to the compile configuration file, it is possible to compile Galatea for HP-UX or Interactive 386/ix.

To change the configuration to work on various systems, copy the file conf/generic to a new file whose name describes your system, ie. cmuvax. This file should then be modified to reflect your configuration. This file contains a set of make(1) macro definitions that will be substituted into makefiles with the gconfig program, which is also in the conf directory. Follow the instructions supplied as comments within each conf file to set the options you require. The conf/dirs file directs gconfig to the directories in which it should modify makefiles. If you want to create a new flag or macro, you can simply add it to the config file. Gconfig will place a copy of every macro in the config into every makefile it touches, whether or not there was a previous definition. If there was a previous definition, gconfig removes it.

To run gconfig, change working directories to conf ’cd conf’, type ’make’, then type ’gconfig’. When used without arguments, gconfig uses the ’config’ file to reconfigure the Galatea programs. If an alternate configuration file, ie. cmuvax, is required, type the name of the file as the first argument to gconfig, ie. ’gconfig cmuvax’. That invocation will use the cmuvax file as the template to configure Galatea for use on Carnegie Mellon VAX hosts.

If your system and options are already described in one of the predefined conf files (athena, hpux, inter386, or sparc), you can simply change the line "CONFIG = generic" to the name of your conf file in the top-level Makefile. Then type "make all" to automatically configure and build Galatea. Or, you can type "make all CONFIG=athena", and the build will be performed, preceeded by a configuration.

## 3.2 Tuning server parameters

There are several aspects of the server operation, mainly timing parameters, that can be adjusted for a particular site. All of these parameters are adjustable in the include file GALATEA/server/tunables.h The documentation for these parameters is in the include file.

## 3.3 Compiling the system

The Makefiles for Galatea are designed to compile the complete system without intervention. In the top level Galatea directory, typing 'make' should create the entire system. 'make install' will create the entire system, if necessary, and then install the executables, libraries and include files in common locations. This is normally /usr/local/{lib,include,bin} and /etc. If you wish to make these directories, 'make directories' will create any necessary directories that do not already exist. Typing 'make clean' will erase all of the object files and executables and typing 'make uninstall' will remove any installed files.

## 3.4 The configuration file

The Galatea server uses a configuration file to determine what the devices are connected to the local machine, what remote servers are available, and the video connectivity of the site. Each line in the file specifies a directly connected device or a remote server. Fields which are meaningless for a given device type should be filled with **?**. This configuration file is usually called /etc/galatead.conf

### 3.4.1 Representing a local videodisk player

A directly connected videodisk player has a line with the following form:

```
Type  VolumeName  Model  TtyPort  Baud  Parity  Chan
```

The fields are separated by whitespace. The values for the fields are:

| | |
|---|---|
| Type | This field should contain RPD, for Record/Play Device. |
| VolumeName | This field specifies the name of disk contained in the player. |
| Model | This field specifies the type of videodisk player. |
| TtyPort | This field specifies the port to which the videodisk player is connected. For a disk player connected to the serial port identified with /dev/ttyS0, this field would be **ttyS0**. |
| Baud | This field specifies the baud rate for communication with the videodisk player. |
| Parity | This field specifies the parity for communication with the videodisk player. This field can be **even**, **odd**, **any**, or **none**. |
| Chan | This field specifies the switch and input to which the disk player connects. This field is comma separated list of items of the form x:n-m where n is the switcher number and m is the input number on the switch. **1-3** would represent switch 1, input 3. A special case for the switcher number is **0**, which indicates that the disk player does not feed a switch, but is a direct server output. In this case, m is the server virtual output number. The x component of the item is optional and is either a **o** or **i**, standing for output or |

input. Note that if all the items in a single list must either all contain the x component, or they all must not contain that component. A device that is capable of recording can have a single item with an **i** for the x component. **i:0-3** would indicate that the recording input of the RPD is fed from server virtual output number 3.

### 3.4.2 Representing a local video routing switch

The line of configuration for a local video switch has the following form:

```
Type  SWTR-n  Model  TtyPort  Baud  Parity  Chan
```

Aside from a few differences, the representation of a switch is identical to that of a videodisk player. If the field is not noted specifically below, use the description given for local videodisk player.

| | |
|---|---|
| Type | For a local video switch, this field should be **SWTR**. |
| SWTR-n | This field specifies the switch number. **n** should be the number assigned to the switch, which is used in referencing the switch in all other device representations. For configuration with a single switch, there could be one switch line with **SWTR-1** as the SWTR-n field, although the number chosen for the switch is arbitrary. |
| Model | This field specifies which type of switcher is connected. |
| TtyPort | Normally this field is the same as for an RPD, but certain switches can be chained together on a single serial port. For these switches, the TtyPort can be of the from x:m, where x is the device number in the chain and m is the port filename as for an RPD. |
| Chan | The channel description is a field made of a comma separated list of items of the form x:y;n-m The x component specifies which of possibly several physical switch outputs this item describes. This is used when you are configuring a multiple output switcher, such as the Akai 16x16 Digital Patch Panel. The y component indicates the default input which will be patched to the specified output. The output will be set to this input whenever the switcher is reset. Switchers are reset when the players are spun down, and when the volume table is rebuilt. The n and m components are as for an RPD where n indicates the switch to which the specified output is connected and m indicates to which channel on the next switcher the specified output is connected. A value of **0** for n indicates that the specified feeds a server virtual output and does not feed another switcher. |

### 3.4.3 Representing a remote server

Any Galatea server can be configured to use the resources of another Galatea server. The local server incorporates the resources of the remote server as if they were located on the local host. Clients do not have any indication of differences between local and remote devices. To represent a remote server to the local server, use the following form of configuration file line:

```
Type  Hostname  ?  RemOut  ?  ?  Chan
```

| | |
|---|---|
| Type | For a remote server, the Type should be **FORW**. |
| Hostname | This field specifies the hostname for the server to which the local server should connect. |
| Remout | This field specifies which of the outputs on the remote machine the local server should utilize. |
| Chan | This Chan field is almost identical to the Chan field for local video switches. The differences are that there is no default input for each output and that the output number specification is not important, except that each output number must be unique. So, the form of this field is a comma separated list of items of the form x:n-m. (A later version of Galatea will permit the ommision of the output number.) Of course, if you only have the forwarding device feeding one local input, you need not specify an output number. Again, if n is **0**, then the forwarding device feeds a server virtual output. |

### 3.4.4 Representing an uncontrolled input

A Galatea server can also make use of uncontrolled input sources, such as a camera. Such a device can be represented in the configuration file with a line of the form:

```
Type  InputName ?  ?  ?  ?  Chan
```

| | |
|---|---|
| Type | The Type should be **INPUT** for an uncontrolled input device. |
| InputName | This field specifies the name of the uncontrolled input. |
| Chan | The Chan field is a comma separated list of items of the form n-m, where n is number of switcher to which the input is connected and m is the input channel number on the specified switch. If n is **0**, then the uncontrolled input feeds a server virtual output. |

### 3.4.5 Sample configuration files

For a single disk player configuration:

```
RPD   BostonDisc SONY_LDP1000A ttyS0   4800  none 0-0
```

For a fairly complex configuration, with multiple switchers, multiple disk players, and a forwarding entry:

```
# A line beginning with a # is a comment line.
# This first group feeds the little switcher (SWTR-1)
RPD   BostonDisc SONY_LDP1000A  ttyS1   4800   none o:1-1
INPUT MIT-CABLE  ?              ?       ?      ?    1-2
FORW  flotsam    ?              0       ?      ?    1-2
INPUT ColorBars  ?              ?       ?      ?    1-3
# This device feeds the AKAI directly, and takes as its input
# the 2nd output of the AKAI.
RPD   RECORDABLE PANA_3031F     ttyS4   9600   none o:3-1,i:0-1
```

```
# The little switch feeds the AKAI, and the default input is
# ColorBars
SWTR  SWTR-1     ATHENA_4x1     ttyS0   9600   none 1:3;3-2
# The big switch feeds virtual output 0 and 1, and has a
# default input of SWTR-1, note the way the line is continued
# on multiple lines.
SWTR  SWTR-2     AKAI16x16      0:ttyS3 19200  none 1:2;0-0,\
2:2;0-1
```

# Table of Contents