The interface to Pandora's box

Stuart Wray

Olivetti Research Limited

February 25, 1994

Version 2.00

1 Introduction

Pandora's box is a video processing peripheral for a workstation or PC. This document describes the model of Pandora's internal workings which is exposed to this host computer. The host sends requests to Pandora, and receives a reply to each one before sending the next request. Pandora can send asynchronous events to the host independently of these request-reply interactions. Pandora and the host can also asynchronously exchange data such as bit-map images and digitised audio. The rest of this document is arranged as follows:

- A description of Pandora's hardware.
- A description of the object based software model.
- A description of the host interface, including the format of data passing across it and between the internal objects of Pandora.
- Details of all the objects available and their operation.

2 The hardware

There is a possible confusion between the video output of the host computer, which usually drives a display, and video in the sense of a "movie". The output of the host computer which drives a display is technically a video signal, but "video" is now also used colloquially to refer to moving images on a TV screen. Pandora deals in both these types of video: it takes the output of its host computer and mixes this with digitally encoded moving pictures (or stills) then displays the resulting image on a monitor. The host computer does not have a direct connection to its display — it can only place images on the screen through Pandora.

A user of Pandora will see a screen image with some areas produced by the host computer and some areas produced by Pandora. It will sometimes not be obvious where the join is, because Pandora can show pictures with irregular edges and the host will usually be able to put still pictures on the screen independently of Pandora.

Pandora also has an attached phone and can handle digital audio. This is separate from the mechanism for handling moving pictures, but there are mechanisms

Figure 1: Pandora and host

to synchronise audio and video. Where the term “video” is used in this document it implictly means “silent video”.

The computer configuration which users of Pandora will see on their desks is shown in figure 1. Pandora takes input from a camera and digitises this so that it can be sent around the Cambridge Fast Ring local area network, stored on a winchester disc inside Pandora, and otherwise handled conveniently in a digital environment. The video output of Pandora is a mixture of digital images and the video signal from the host computer. The screen image is produced at the last moment by analogue mixing, so the video image from Pandora need not be representable in the framestore of the host computer or vice versa. For contrast, compare this with the system at MIT which uses Parallax frame capture hardware. In that system the transmission and storage of video is entirely analogue, but at the last moment this is digitised and inserted into the frame buffer of the host computer. The Parallax video hardware is thus completely host specific, unlike Pandora which can work with many hosts without modification.

The mixing of video from Pandora and the host computer is illustrated in figure 2. The image produced by a window system on the host is shown in the middle. It has two text windows and the window system has left a black area where the video

Figure 2: Mixing images

Figure 3: Inside pandora

window will appear (the reason for this is discussed below). Next on the left is the video image as it appears in the output framestore in Pandora and on the far left is the mask plane of Pandora. Where a 0 appears in this plane, the corresponding pixel will be taken from the host video image, where it is 1 the pixel will be taken from Pandora. The mask has been set to show exactly the rectangular video image placed in the frame store. It is possible to change individual bits in the mask plane, so a frame of video could be clipped to any desired shape. It would even be possible to show a video of your sweet-heart in a heart shaped window. The combined image from the host and Pandora is shown at the right of figure 2, as it would appear on the monitor screen.

The problem with the mask plane system is that a cursor in the host image which moves into a video window would disappear completely, since it too would be overlaid with the video from Pandora. The solution adopted in Pandora is to display the image from the host unless the mask plane indicates otherwise *and* the image from the host is fully black. This is the reason for the rectangle underneath the video window in figure 2 being black. To see the video from Pandora at a particular pixel you have to set the appropriate bit in the mask plane *and* make that pixel in the host's image completely black. The effect this has on the cursor is that black parts of a cursor will be transparent and show video through them, but other shades will be opaque. The cursor will still be mostly visible.

Figure 3 shows the broad internal structure of Pandora. There are four distinct modules, each controlled by a Transputer. These Transputers are connected together by their links in a tetrahedron, but the links are used only for control; data passes between the modules on a bus. This is what each module does:

- The server module is in many ways like a DMA engine. The other three modules each have FIFOs for data transfer to or from the server and these are memory mapped into the server module's address space, so to transfer data between modules in the simplest case involves a block move from one FIFO into another. The server module also has a CFR interface for communication with other Pandora's boxes. This is used for data only: all control information passes via the host (of course the host may use the same CFR for its communications, but Pandora does not know this). Pandora's permanent storage device is attached to the server module as well. Initially this will be a winchester disc, but in the future a removable magneto-optical disc may be used.

- The capture module takes a video input from a camera and digitises this frame by frame. A frame is broken up into "segments" for transmission and the results sent out via the FIFO (the format of these segments is described in a later section). If compression is required this is done segment by segment.

- The mixer module receives video streams through its FIFO, decompressing them as required and placing them in its frame buffer. The image in the frame buffer is then mixed with the video signal from the host, as described above.

- The host module directs the other three modules. It has the interface to Pandora's host computer (this is used for both control and data). The host module also has a phone connection, and Pandora's audio processing is done here.

3 The object model

This section presents the object model of the inside of Pandora. This model reflects the structure of the hardware in some ways but is rather more general, since we intend to retain the same software model across changes to Pandora's hardware.

Only one object is present when Pandora first starts running: the Factory. This object is used to create, connect, disconnect and dispose of other objects. Since the resources of Pandora are finite (internal bandwidth will often be the main constraint) the Factory can refuse a request to create a new object. Within the Factory are the prototypes of all the objects that can be created. By specifying one of these along with a few initial attributes and a name the Factory can build the required object. When the Factory deletes an object, the hardware resources allocated to the object may be recommitted elsewhere.

The Factory is also used to connect objects together, allowing streams of video and audio to pass from one object to another. The input ports and output ports of objects have types and to connect an input port to an output port their types must match. The output ports of most objects can be connected to several input ports; some objects have input ports to which several output ports can be connected. There is a maximum number of connections that can be made to an input or output port of each kind of object. Note that no objects have output ports capable of multiple connections which are type compatible with input ports capable of multiple connections. All connnections are either one-to-many or many-to-one, never many-to-many. For some kinds of object there is a maximum number of instances which the factory will create. A request can be made to the Factory to determine what these limits are.

When an object sends data to an output port its responsibility for that data ends. If the port is not connected to another object, or if the object it is connected to is not ready, the data is simply discarded. Because the model of data flow is of blowing rather than sucking, it is never an error to give data to another object, even if it cannot keep up. The object which is too slow will raise an event to complain that its input is too fast for it. Every object attempts to go as fast as it can, and the speed of a network of connected objects is set by the speed of the ulimate sources of the data flowing through the network. There is no direct feedback mechanism within Pandora to throttle back fast sources when other objects cannot keep up: instead the objects left behind must raise events and have the host send requests to the offending sources to make them go more slowly.

The mechanisms which turn camera input into its digital representation and turn this representation back into an image on a display are not single objects. Instead, each of these is put together from three objects: an input adapter, a virtual screen and an output adapter. The structure of Pandora's hardware makes it possible to build such a mechanism on the camera side with more than one output adapter, delivering several independent streams of digital video from the same original camera image. Similarly on the display side the mechanism can have several input adapters, placing several digital video streams on the display and acting as a rudimentary mixer. In practice the number of streams is likely to be one initially, but this can be checked by asking the Factory how many inputs the mixer virtual screen has, and how many outputs the capture virtual screen has.

The idea of adapters and virtual screens is illustrated in figure 4 which shows a camera adapter placing video on a virtual screen from where it is taken by an encoder adapter. It is then delivered to a decoder adapter, through a mixer virtual screen and onto the computer display itself. The video signal notionally enters the camera adapter in analogue form. The adapter is parameterised when it is built by the factory, so it knows the frame rate, line rate, and so on of the camera. Of course the hardware of Pandora restricts the range of parameters that it is sensible to allow (for example it is unlikely to be able to decode more than one type of analogue colour representation). The adapter can be instructed to place its output at a particular location on the virtual screen, and it is possible that some scaling is also available. This output is constrained to lie entirely on the virtual screen, but the place it is put can be changed.

In Pandora there can currently only be one virtual screen for capture and one for display. These virtual screens can be thought of as corresponding to the frame

Figure 4: Capturing an image

stores in the capture and mixer modules of Pandora (though the same model could be used even if frame stores were not present in the hardware). The screens are created by the Factory like all the other objects, but their size and colour encoding properties are fixed.

An encoder adapter takes images from the virtual screen and turns them into a representation suitable for digital transmission and storage. Its output is a stream of digital video "segments". An encoder adapter is parameterised when it is built by the factory: things like the type of compression to be used can be set in this way. Naturally, what it is sensible to specify will depend to a large extent upon the hardware. This adapter can be instructed to take its input from a particular place on the virtual screen and as with the camera adapter this place can be changed from time to time.

This summarises the components of the capture mechanism; the mixing mechanism feeding the computer display is similar but the other way round. It takes as input a stream of digital video segments and uses the intermediate representation on the mixer virtual screen to produce an analogue signal which will drive a monitor.

The telephone connected to Pandora appears in the object model as separate microphone, speaker and keypad objects. Microphone and speaker communicate with other objects using streams of digital audio segments, similar to digital video segments. The keypad causes events to be sent to the host whenever keys are pressed.

Pandora has three kinds of object which can be used to send data outside the box in digital form: Host, Network and Disc. Host objects connect to the asynchronous data channel described in the next section. They will most often be used for carrying digital audio but they will have too small a bandwidth to carry video, unless the frame size is very small. Network objects are used in a similar way to connect to other Pandora boxes and to fileservers over the Cambridge Fast Ring. These objects have enough bandwidth to carry video. Disc objects are connected to winchester or removable optical discs, and can be used to record and play back streams of video or audio. Disc objects may not be available in the very first Pandoras, but there will be an external video repository with some of their functions. This can be used via the Network objects.

4 Interfaces and Segments

This section describes the host interface and the structure of data segments. Initially the host interface is being implemented over a Transputer link, but a SCSI bus will be used for greater host independence. The host interface is a bidirectional serial channel which is used as three virtual channels. The abstact syntax of the communication through each channel is given below. The notation $name_{type}$ in the abstract syntax introduces a syntactic entity called *name* with type *type*.

Here are the different types:

NAME	A unique identifier (eg 'type.video.segment').
INT	A signed integer.
RAT	A signed rational number.
BYTE	An 8 bit byte.
TIME	The value of a timer, not absolute time.
ANY	Any of the above types.
SEGMENT	A self-contained part of a video or audio stream. (The format is described later).
SEQUENCE[*type*]	A sequence of zero or more items each of type *type*.

It is also possible to specify that collections of items be sent along a communication channel. A collection is a sequence of items, grouped together and delimited from previous or subsequent items. The notation for a collection is a sequence enclosed in curly brackets:

$$\{items_{\mathrm{SEQUENCE}[type]}\}$$

The concrete syntax is described in an appendix of this document. Here is what the different channels are for:

4.1 Request-reply channel

Pandora waits for the host to send a request over this channel. When the host has finished sending a request it must wait for the reply from Pandora before sending another request. It is not defined what will happen if the host attempts to send another request before Pandora replies to the first one. Don't try it.

A request from the host to Pandora has this format:

$$object_{\mathrm{NAME}} \quad request_{\mathrm{NAME}} \quad \{arguments_{\mathrm{SEQUENCE[ANY]}}\}$$

The appropriate number and type of items in the collection *arguments* depends on the *object* and the *request* that are specified. The reply from Pandora has this format:

$$reply_{\mathrm{NAME}} \quad \{results_{\mathrm{SEQUENCE[ANY]}}\}$$

The name *reply* indicates whether the request succeeded or failed (the name will be 'reply.success' if it worked, otherwise the name of an error). The collection contains such results as are returned in the case of a successful commmand, and may be empty. In the case of an unsuccessful request the collection either contains a sequence of bytes giving a human-readable text message or it is empty. The names of errors that can be returned by each object are specified in the descriptions of the individual objects in a later section.

This format will probably change in future to include a time in the reply.

4.2 Asynchronous event channel

The host must wait continually for an event on this channel. Once an event is received Pandora is free to send another one hard on its heels. There is no limit to the number of consecutive events Pandora can send to the host on this channel. Each event is triggered by some condition in an object inside Pandora, for example a buffer overflowing.

An event sent from Pandora to the host has this format:

$$object_{\mathrm{NAME}} \quad event_{\mathrm{NAME}} \quad \{message_{\mathrm{SEQUENCE[BYTE]}}\}$$

The object given in an event is the one that caused it, *event* specifies what kind of event it is and *message* is a text message for human inspection (this collection of bytes may of course be empty). The events that can be raised by objects are given with the description of the individual objects in a later section.

The host does not have to perform any particular action after an event, other than being ready to receive the next one. Objects have requests for selecting the conditions which will cause them to send an event, so they can be instructed to be quiet if necessary.

This format will probably change in future to include a time in the event.

4.3 Asynchronous data channel

The host must be ready to accept data from Pandora on this channel and Pandora must be ready to accept data from the host. It is not defined whether transmission is only one way at once, or whether data can flow in both directions at the same time.

Data traveling in either direction has this format:

$$object_{\text{NAME}} \quad data_{\text{SEGMENT}}$$

Object names in this interface have a similar rôle to port numbers in a network interface. The segment is sent to the input of *object* (if the host is sending it) or has come from the output of *object* (if the host is receiving it). If a segment of an inappropriate type is sent to an object it will raise an event on the asynchronous error channel. If the host recieves an inappropriate or unexpected segment it might ignore it or it might investigate the situation using the request-reply channel. No replies can be made directly on the asynchronous data channel.

4.4 Segment structure

A "frame" is part of a stream of video data which stands on its own, does not need the previous part of the stream for it to be meaningful and gives a consistent viewable image (though not necessarily a static image). A "segment" of video is part of a frame. It does not need previous segments, but it may not give a meaningful image on its own (for example a segment might be a single scan line). If enough segments of a frame are displayed the image seen will be meaningful, but how many this is depends on the way a frame was divided into segments. For instance in some encodings the loss of a single segment of a frame might mean that colour information is missing, or that the image is blurred. Segments may be lost while they are transmitted from one Pandora to another over the network, or because of some overcommitment of Pandora's internal capacity to transmit and process data.

The segments transmitted over the data channels and stored on discs have this format:

$$\begin{array}{l} identifier_{\text{NAME}} \\ sequence_{\text{INT}} \\ time_{\text{TIME}} \\ type_{\text{INT}} \\ \{data_{\text{SEQUENCE[ANY]}}\} \end{array}$$

The first item in a segment is *identifier*, a name which specifies the segment format and its concrete syntax. For now there is only this one format of segment (specified by the name 'name.segment.format1'), but in the future there might be other formats. As you can see, this format has a header and one data field. Only the header of the segment is fixed, and this header contains just the information which is necessary to handle the appended *data* in a fileserver or network, without knowing its meaning. Other formats of segment will only be introduced in the unlikely event that it is necessary to change this header or add more data fields at this outer level. Although at the moment there is only one kind of header there could be more than one concrete syntax for it (for example little-endian or big-endian) and each of these will have a unique identifier.

The next item in a segment is *sequence*, the number of this segment within a stream of segments (the first segment is numbered 0). Note that this is nothing to do with frame number or sequence number of segment within a frame. That information (if present) is encoded in the *data* field. The *sequence* field is there so that the mechanisms for handling segments without regard for their meaning can still detect when a segment has been lost in transit. Such lost segments may be retransmitted or just dropped on the ground.

The *type* field is an integer which specifies the format of the *data* field in the same way as *identifier* specifies the format of the whole segment. Part of the data may be further "type" information, for example the name of the compression scheme used, the size of a picture, and so on.

Each segment has a time stamp in the *time* field. This is the value of the timer in the hardware generating the segment when it is created. By replaying recorded segments at the relative times corresponding to these timer values, a fileserver can simulate the source which originally produced the segments. It need not know anything about frame rates or other such time properties of the decoding machinery.

As noted above, in the concrete syntax *data* is encoded in such a way that segments can be stored and transmitted without needing to interpret this field.

4.5 Types of segments

There are currently three types of segments: Video ('type.video.segment'), Still ('type.still.segment') and Audio ('type.audio.segment'). It is intended that Video will be able to describe most video encoding schemes, but a new type of segment may be necessary for particularly unusual schemes, since the format described here is frame oriented. Note that there are different types of segments for stills and moving video, because a still is conceptually an instantaneous picture, while a video is conceptually made up of frames which are moving pictures of short duration. As an illustration of this difference consider 18$^{\text{th}}$ century racing prints and the phenomenon of motion blur.

Motion blur is an artefact of photography. A still photograph takes a certain time to take and during this time the subject may move. A drawing takes even longer to finish, but motion blur is seldom used as an artistic device in drawing (though it is used in photography since it is so easy to achieve). Drawings and paintings are stills, pictures of something at an instant in time. Before photography was invented, artists did not know how horses legs moved when they were galloping. However, they did not draw them blurred. In 18$^{\text{th}}$ century racing prints the horses have crisp unblurred legs, drawn as the artists imagined they must be. In fact their imagination was wrong, but their decision is an example of the traditional view of the still picture as capturing a moment in time, even to the extent of inventing the scene at that instant so as to conceal the limitations of the artist's eye.

On the other hand, motion blur is an essential part of a frame of video, it is not an unwanted artefact. A frame of video is a representation of a scene as it changes through time, it is not a picture of a scene at an instant. If a video did not have motion blur it would look jumpy and artificial (as simple animation does). To improve the realism of their product, animators spend considerable effort *adding* motion blur to their crisp images.

Compression schemes will be different for stills and video. Video might be encoded in indivisible “frames” several seconds long with motion encoded explicitly. Conditional block replenishment is a bit like this. Note that it is not necessarily possible to present the moving image part way through a frame as a still image, and it is not the case that stopping a video at the end of a frame will leave a corresponding still image on the screen. Video and stills are *different* things and they might be inter-convertable only with great difficulty. It might seem that it will always be possible to make a still representation corresponding to a moment in a video, but consider a sudden flash in a video. What is the appropriate representation of this effect as a still picture?

The way in which stills and videos are constructed for viewing is also different. With a still picture, the time taken to construct it is not important, provided that it is complete at the required viewing time. For example, a painting takes a very long time to construct, but this does not matter, provided it is finished when we want to look at it. On the other hand, a video must be constructed before our eyes at a specific rate, or else its meaning to us will be changed (for example things might move too slowly). This has important consequences in the different treatment of stills and video as they are transmitted from one place to another, there being no intrinsic time constraints in the presentation of stills but very stringent constraints in the presentation of video.

Video

This type of segment gives information about the frame that it is part of, its position in that frame, and the compression scheme used for the data. This is followed by the compressed data itself.

$$
\begin{array}{l}
\mathit{frame}_{\mathrm{INT}} \\
\mathit{segments}_{\mathrm{INT}} \\
\mathit{this\ segment}_{\mathrm{INT}} \\
\mathit{frameX}_{\mathrm{INT}} \\
\mathit{frameY}_{\mathrm{INT}} \\
\mathit{pixel\ aspect}_{\mathrm{RAT}} \\
\mathit{visual}_{\mathrm{NAME}} \\
\{\mathit{compression}_{\mathrm{NAME}}\ \mathit{arguments}_{\mathrm{SEQUENCE[ANY]}}\} \\
\{\mathit{data}_{\mathrm{SEQUENCE[BYTE]}}\}
\end{array}
$$

Frames of video are formed from several segments. The field *frame* contains the number of the frame which this segment is part of, *segments* is the number of segments that this frame has been broken into (it need not be the same number from frame to frame) and *this segment* is the number of the segment within that frame. Frames are numbered from 0 and segments within a frame are also numbered from 0. The fields *frameX* and *frameY* specify the size of the frame in pixels. The aspect ratio of the pixels is given in the field *pixel aspect*.

$$\mathit{pixel\ aspect} = \frac{\text{pixel height}}{\text{pixel width}}$$

This need not be compatible with the pixel aspect ratio of the virtual screen on which the segment is shown, but if it isn't the picture will look stretched either vertically or horizontally.

The format of the pixels is specified by the field *visual* — for example it might be 'name.monochrome.8.bit'. This must be compatible with the pixel format of a virtual screen that the segment is shown on. It is probable that specific hardware will only be able to handle a small range of pixel formats. If the visuals were incompatible this would cause a type error event.

Currently four compression schemes are recognised (attempts to use other schemes will cause a type error). These are 'name.lines.null', 'name.lines.dpcm', 'name.lines.pack', 'name.lines.pack.dpcm'. The parameters taken for all these formats are the same, so only 'name.lines.null' is illustrated here:

$$\{\text{`name.lines.null'}\ \mathit{lineX}_{\mathrm{INT}}\ \mathit{startY}_{\mathrm{INT}}\ \mathit{lines}_{\mathrm{INT}}\}$$

This specifies that the following *data* is not compressed and that it is stored pixel by pixel in left to right scan lines of length *lineX*. These scan lines form the part of the frame starting from the *startY* line from the top and continuing for *lines* lines down the screen.

For the other named compression schemes, the following *data* is to be processed appropriately and when this is done the result will come out in in left to right scan lines of length *lineX*. As before, scan lines form the part of the frame starting from the *startY* line from the top and continuing for *lines* lines down the screen.

The scheme 'name.lines.pack.dpcm' uses a dpcm compressor to reduce 8 bit pixels to 4 bits of significant data, then packs two pixels per byte. It is therefore a 2:1 compression scheme. 'name.lines.dpcm' does the dpcm step without packing, so the data doesn't become any smaller. 'name.lines.pack' packs the least significant 4 bits of pairs of pixels into each byte, discarding the top 4 bits, and will produce very strange looking pictures. In practice only 'name.lines.pack.dpcm' is likely to be useful.

Still Picture

This type of segment gives information about the still that it is part of, its position in that frame, and the compression scheme used for the data. This is followed by the compressed data itself. This format is almost identical with video, but is given in full here for completeness.

$$\begin{array}{l}
\mathit{segments}_{\mathrm{INT}} \\
\mathit{this\ segment}_{\mathrm{INT}} \\
\mathit{pictureX}_{\mathrm{INT}} \\
\mathit{pictureY}_{\mathrm{INT}} \\
\mathit{pixel\ aspect}_{\mathrm{RAT}} \\
\mathit{visual}_{\mathrm{NAME}} \\
\{\mathit{compression}_{\mathrm{NAME}}\ \mathit{arguments}_{\mathrm{SEQUENCE[ANY]}}\} \\
\{\mathit{data}_{\mathrm{SEQUENCE[BYTE]}}\}
\end{array}$$

Stills are formed from several segments. The field *segments* is the number of segments that this still has been broken into and *this segment* is the number of the segment within the still. Segments within a still are numbered from 0. The fields *pictureX* and *pictureY* specify the height and width of the still in pixels. The aspect ratio of the pixels is given in the field *pixel aspect*.

$$\textit{pixel aspect} = \frac{\text{pixel height}}{\text{pixel width}}$$

This need not be compatible with the pixel aspect ratio of the virtual screen on which the segment is shown, but if it isn't the picture will look stretched either vertically or horizontally.

The format of the pixels is specified by the field *visual* — for example it might be 'name.colour.24.bit'. This must be compatible with the pixel format of a virtual screen that the segment is shown on. It is probable that specific hardware will only be able to handle a small range of pixel formats. If the visuals were incompatible this would cause a type error event.

Currently only one decompression scheme is recognised (attempts to use other schemes will cause a type error). This is 'name.lines.null':

$$\{\text{`name.lines.null'}\ \textit{lineX}_{\mathrm{INT}}\ \textit{startY}_{\mathrm{INT}}\ \textit{lines}_{\mathrm{INT}}\}$$

This specifies that the following *data* is not compressed and that it is stored pixel by pixel in left to right scan lines of length *lineX*. These scan lines form the part of the still starting from the *startY* line from the top and continuing for *lines* lines down the screen.

Audio

This type of segment specifies the length of the audio information it contains and the kind of encoding used, then specifies a (possibly parameterised) compression scheme which will decode the remaining data in the segment.

$$\begin{array}{l}
\textit{length}_{\mathrm{INT}} \\
\textit{rate}_{\mathrm{RAT}} \\
\textit{sample format}_{\mathrm{NAME}} \\
\{\textit{compression}_{\mathrm{NAME}}\ \textit{arguments}_{\mathrm{SEQUENCE[ANY]}}\} \\
\{\textit{data}_{\mathrm{SEQUENCE[BYTE]}}\}
\end{array}$$

The first field, *length*, contains the number of samples in this segment (this is the number of samples there will be after any decompression). The *rate* field specifies the number of samples per second of the decompressed data to be given to the hardware, and *sample format* names the format of these samples, for example 'name.mu.law.8.bit'. It is likely that specific hardware will only be able to handle a small range of the possible sample rates and formats. Segments that cannot be handled will be rejected with a type error event.

Currently only one named compression scheme is recognised: 'name.null':

$$\{\text{'name.null'}\}$$

This specifies that the following *data* is not compressed and can be given directly to the codec, at the appropriate rate.

It is likely that a compression scheme called 'name.silence' will be implemented in the future:

$$\{\text{'name.silence'}\ \mathit{length}_{\mathrm{INT}}\}$$

This is followed by a null *data* field, which is ignored. The decompressor generates *length* samples of simulated silence to take the place of the data that would otherwise have been there, but which was quieter than some threshold.

5 The objects

5.1 Common material

There are some things common to all objects, and they are given here to avoid repetition.

Kind

Each kind of object has a name, used when creating it. The recognised kinds are:

'kind.factory'
'kind.camera.adapter'
'kind.capture.virtual.screen'
'kind.encoder.adapter'
'kind.decoder.adapter'
'kind.mixer.virtual.screen'
'kind.display.adapter'
'kind.audio.sink'
'kind.audio.source'
'kind.keypad'
'kind.synchroniser'
'kind.joiner'
'kind.splitter'
'kind.network.sink'
'kind.network.source'
'kind.host.sink'
'kind.host.source'
'kind.file.sink'
'kind.file.source'

(The order of these names is the same as the order in which the object descriptions are presented)

Input ports, output ports and types

The description of each port of an object says how many connections can be made to it (one or many) and what its type is. The following types are defined:

'type.capture.project'	On to the capture screen
'type.capture.view'	Off the capture screen
'type.mixer.project'	On to the mixer screen
'type.mixer.view'	Off the mixer screen
'type.video.segment'	Video
'type.still.segment'	Stills
'type.audio.segment'	Audio
'type.all.segment'	Union of Video, Still and Audio
'type.any.segment'	(Polymorphic) Any of the above four

The joiner object takes video, still and audio streams as input and produces a united stream of type 'type.all.segment' as output. The splitter object reverses this. All streams of segments into Pandora are united streams, so splitters and joiners need to be used before they can be connected to display mechanisms (which only accept the simple Video, Still and Audio types).

No segment has type 'type.all.segment' or 'type.any.segment', these are the types of ports through which streams of segments flow. If an object has one port of type 'type.any.segment' then it is acceptable to connect a port to it which has any of the other segment types, including 'type.all.segment'. A port of type 'type.any.segment' effectively changes to match the type of the port it is connected to. If an object has more than one port of type 'type.any.segment', and one of these ports changes to match a connected port then they all effectively change to match, so any connections to the other ports must now match that type too. If a port of type 'type.any.segment' is connected to one of type 'type.any.segment' on another object then all ports of this type on both objects change to match whenever a port on one of them changes.

There is one more type not mentioned so far: 'type.any1.segment'. This has identical properties to 'type.any.segment', but if an object has both types of port, one of them can change type to match as described above without affecting the other one. They are entirely independent. This type is used only in the synchroniser, which has two separate polymorphic segment ports, one of type 'type.any.segment', the other 'type.any1.segment'.

Creation

Most objects require some parameters to be given to the Factory when they are created. Details of these parameters are given in the description of each kind of object in its Creation section.

Events

All objects can send these events, and some objects can send other events. See the descriptions of the individual objects.

- ‘event.internal.error’ Something has gone very badly wrong. (Pandora may be broken.)
- ‘event.host.broken’ Something has gone wrong with host communications.
- ‘event.segment.too.large’ A segment was too large for a buffer and has been discarded.
- ‘event.missing.segment’ A segment is missing (as detected by sequence numbers).
- ‘event.wrong.type.of.segment’ A segment of an unexpected type was received and discarded.
- ‘event.closed.down’ The hardware resources allocated to this object are free again.
- ‘event.log’ Logging message sent from internal layers of Pandora software.

Requests

Requests are described in the format

$$\{arguments\} \quad \Rightarrow \quad \{results\}$$

Where *arguments* is the collection of data sent with the request and *results* is the collection of data returned with a successful reply.

Replies

These are all the replies that any object can deliver. For ‘reply.success’ there will follow the results as specified in the description of each request. Otherwise there follows a string, which may give more information about the reason for failure.

- ‘reply.success’ The request worked.
- ‘reply.unknown.object’ The object named in the request does not exist.
- ‘reply.unknown.request’ The object does not implement this request.
- ‘reply.failed’ The request failed, but no error occured.
- ‘reply.request.too.large’ The arguments in the request are too big for Pandora’s buffer.
- ‘reply.malformed.request’ The number of arguments is wrong, or there is some other syntactic error in the request.
- ‘reply.host.broken’ Something has gone wrong with host communications.
- ‘reply.internal.error’ Something has gone very badly wrong. (Pandora may be broken.)

5.2 Factory

The Factory is the only object present when Pandora starts running, and is used to create, connect, disconnect and remove other objects.

Kind

'kind.factory'

Input and output ports

None.

Creation

Not applicable.

Events

Only the standard ones.

Requests

- 'request.factory.create' (Create an object)

  $$\{kind_{\mathrm{NAME}} \quad parameters_{\mathrm{SEQUENCE[ANY]}}\} \quad \Rightarrow \quad \{object_{\mathrm{NAME}}\}$$

  The kind of the new object is specified. Depending on what that kind is there may be some initialisation parameters — details are given in the description of each kind of object in the Creation section. Objects created by the factory will not raise requests until they are allowed to, using the allow event request to the factory.

  Pre-condition: *kind* must be the name of a kind of object.

  Post-condition: A new object of kind *kind* called *object* exists, and the name *object* is unique to this newly created object.

- 'request.factory.allow.event' (Allow events to be raised)

  $$\{object_{\mathrm{NAME}}\, events_{\mathrm{SEQUENCE[NAME]}}\} \quad \Rightarrow \quad \{\}$$

  The named object is allowed to raise *events*. The argument *events* can contain names which the object could never raise as events, and these are just ignored.

  Pre-condition: There is an object called *object*.

  Post-condition: The object may raise some of the named events.

- 'request.factory.forbid.event' (Stop events being raised)

$$\{object_{\text{NAME}}\, events_{\text{SEQUENCE[NAME]}}\} \;\Rightarrow\; \{\}$$

  The argument *events* specifies the names of events which the object is not to raise again until it is told to. The arguments can contain names which the object could never raise as events, and these are just ignored.

  Pre-condition: There is an object called *object*.

  Post-condition: The object will not raise any of the named events.

- 'request.factory.show.allowed.events' (What events may be raised)

$$\{object_{\text{NAME}}\} \;\Rightarrow\; \{events_{\text{SEQUENCE[NAME]}}\}$$

  This request returns the names of the events that *object* may raise.

  Pre-condition: There is an object called *object*.

  Post-condition: None.

- 'request.factory.show.forbidden.events' (What events may not be raised)

$$\{object_{\text{NAME}}\} \;\Rightarrow\; \{events_{\text{SEQUENCE[NAME]}}\}$$

  This request returns the names of the events that *object* may not raise, but which it could raise were they allowed. It is NOT a list of all the event names minus those that *object* is allowed to raise.

  Pre-condition: There is an object called *object*.

  Post-condition: None.

- 'request.factory.show.requests' (What requests can be made?)

$$\{kind_{\text{NAME}}\} \;\Rightarrow\; \{requests_{\text{SEQUENCE[NAME]}}\}$$

  This request returns the names of the requests that a *kind* of object will accept.

  Pre-condition: There is an object called *object*.

  Post-condition: None.

- 'request.factory.show.creation.parameters' (How an object was created)

$$\{object_{\text{NAME}}\} \;\Rightarrow\; \{parameters_{\text{SEQUENCE[ANY]}}\}$$

  This request returns the parameters used in the call to the factory which created this object.

  Pre-condition: There is an object called *object*.

  Post-condition: None.

- 'request.factory.delete' (Delete an object)

$$\{object_{\mathrm{NAME}}\} \;\Rightarrow\; \{\}$$

The named object is deleted and its associated resources will be reclaimed. The request returns a result immediately, and if it is successful the object will be deleted but the hardware resources may take some time to reclaim.

Pre-condition: *object* is the name of an object created by the Factory. There must be no connections to the ports of *object*.

Post-condition: No object with name *object* exists.

- 'request.factory.connect' (Connect two objects)

$$\{object1_{\mathrm{NAME}}\;\; port1_{\mathrm{NAME}}\;\; object2_{\mathrm{NAME}}\;\; port2_{\mathrm{NAME}}\} \;\Rightarrow\; \{\}$$

The two objects are specified, with the names of the ports to be joined.

Pre-condition: *object1* and *object2* exist and have ports called *port1* and *port2* respectively. One of the ports must be input and the other must be output. Both ports must not have reached their maximum number of connections. If either of the ports can connect to only one other port it must not be already connected to a port. The types of the ports must be compatible.

Post-condition: The number of connections to each port is increased by one. The objects are connected together via the ports: segments sent from one port will be received at the other port.

- 'request.factory.disconnect' (Disconnect two objects)

$$\{object1_{\mathrm{NAME}}\;\; port1_{\mathrm{NAME}}\;\; object2_{\mathrm{NAME}}\;\; port2_{\mathrm{NAME}}\} \;\Rightarrow\; \{\}$$

Two objects are specified, with the names of the ports to be disconnected.

Pre-condition: *object1* and *object2* must exist and have ports *port1* and *port2* respectively. The two objects must be connected to each other via the specified ports.

Post-condition: The number of connections to each port is decremented by one. The objects are no longer connected to one another: segments sent to whichever is the output port will not be received at the input port.

- 'request.factory.show.instances' (Maximum allowed instances)

$$\{kind_{\mathrm{NAME}}\} \;\Rightarrow\; \{instances_{\mathrm{INT}}\}$$

This request returns the number of instances of an object that the factory is able to create. Note that this is the total number, not how many more it can create at this moment. This number might change as Pandora is used due to resources being committed. For example if this enquiry says you can create x

of one kind of object and y of another this doen't mean you can do both, it means you can do either. When you have done one of them check how many of the other you can do then.

Pre-condition: There is a kind of object called *kind*.

Post-condition: None.

- 'request.factory.show.objects' (What objects are there)

$$\{\} \quad \Rightarrow \quad \{names_{\mathrm{SEQUENCE[NAME]}}\}$$

This request returns the name and kind of every existing object. The names returned are alternately object name, corresponding kind, object name, corresponding kind, and so on.

Pre-condition: None.

Post-condition: None.

- 'request.factory.show.ports' (What ports does an object have?)

$$\{kind_{\mathrm{NAME}}\} \quad \Rightarrow \quad \{ports_{\mathrm{SEQUENCE[NAME]}}\}$$

This request returns the names of the ports of the specifed *kind* of object.

Pre-condition: There is a kind of object called *kind*.

Post-condition: None.

- 'request.factory.show.port' (Details of a port)

$$\{port_{\mathrm{NAME}}\} \quad \Rightarrow \quad \{connections_{\mathrm{INT}}\}$$

This request returns details about a particular *port*. *connections* is the maximum number of connections that can be made to the port.

Pre-condition: Some object has a port called *port*.

Post-condition: None.

- 'request.factory.show.connected' (What is connected to a port)

$$\{object_{\mathrm{NAME}} \;\; port_{\mathrm{NAME}}\} \quad \Rightarrow \quad \{names_{\mathrm{SEQUENCE[NAME]}}\}$$

This request returns all the objects (and the port used) connected to *port* of the specified object. The names returned are alternately object name, corresponding port name, object name, corresponding port name, and so on.

Pre-condition: There is an object called *object*, with a port called *port*.

Post-condition: None.

- 'request.factory.show.version' (What version is this Pandora?)

  $$\{\} \quad \Rightarrow \quad \{\mathit{version}_{\mathrm{INT}} \;\; \mathit{release}_{\mathrm{INT}}\}$$

  This request returns the version of the interface to Pandora's box. If the version was 2.09, this request would return with *version* $= 2$ and *release* $= 9$.

  Pre-condition: None.

  Post-condition: None.

5.3 Camera adapter

A camera adapter takes the output from a camera and converts it frame by frame into the intermediate representation suitable for placing on the capture virtual screen.

Kind

'kind.camera.adapter'

Input and output ports

- Output (only one connection): 'port.camera.adapter.output'

  Type: 'type.capture.project'

Creation

The parameters used when the factory creates this object are:

$$\mathit{interlace}_{\mathrm{NAME}} \;\; \mathit{framerate}_{\mathrm{RAT}} \;\; \mathit{linerate}_{\mathrm{INT}} \;\; \mathit{aspect}_{\mathrm{RAT}} \;\; \mathit{visual}_{\mathrm{NAME}}$$

The parameter *interlace* can take either the value 'name.interlaced' or the value 'name.not.interlaced', and *visual* can take the value 'name.monochrome.8.bit'. The parameter *framerate* is the frame rate of the camera, *linerate* is its line rate and its pixels have aspect ratio *aspect*. The values of the parameters should match the specification of the camera attached to pandora, but currently they don't govern the behaviour of the camera at all.

Events

Only the standard ones.

Requests

- 'request.camera.adapter.set' (Set reference point, X-Y size.)

  $$\{\mathit{refX}_{\mathrm{INT}} \;\; \mathit{refY}_{\mathrm{INT}} \;\; \mathit{sizeX}_{\mathrm{INT}} \;\; \mathit{sizeY}_{\mathrm{INT}}\} \quad \Rightarrow \quad \{\}$$

This request sets the place on a virtual screen where the camera adapter places its image. The reference point (*refX*, *refY*) is the coordinate on a capture virtual screen of the top left hand corner of the rectangle where the image is placed. The rectangle is *sizeX* pixels wide and *sizeY* pixels high. Note that the top left hand pixel of the virtual screen is coordinate (0, 0) so x coordinates increase from left to right and y coordinates increase from top to bottom of the screen. If the rectangle is placed so that it is partially off the virtual screen, images might not appear even on the part that remains on the virtual screen. Only certain rectangle sizes and positions are allowed. Currently *refX* and *refY* must both be 0; *sizeX* must equal *sizeY* and can be either 512, 256 or 128. When it is created, a camera adapter is set to *size X* = 128 and *sizeY* = 128.

Pre-condition: The specified rectangle size and position is allowed.

Post-condition: None.

- 'request.camera.adapter.show' (Show reference point, X-Y size.)

$$\{\} \quad \Rightarrow \quad \{refX_{\mathrm{INT}} \;\; refY_{\mathrm{INT}} \;\; sizeX_{\mathrm{INT}} \;\; sizeY_{\mathrm{INT}}\}$$

This returns the reference point (*refX*, *refY*) of the top left hand corner of the rectangle where the camera image will be placed on a virtual screen, along with its width *sizeX* and height *sizeY*.

Pre-condition: None.

Post-condition: None.

- 'request.camera.adapter.test.card.set' (Set test card parameters.)

$$\{size_{\mathrm{INT}} \;\; refX_{\mathrm{INT}} \;\; refY_{\mathrm{INT}} \;\; rate_{\mathrm{RAT}}\} \quad \Rightarrow \quad \{\}$$

This request controls the action of the test card generator. The test card is a two by two chequer-board overlayed on the picture from the camera, where the edges of each of the four sub-squares are *size* pixels long. When a camera is created the initial value of the *size* of the test card is 0, and in this state the test card is not visible. The parameters *refX* and *refY* specify the position of the the top left hand corner of the test card relative to the reference point of the camera adapter (set with 'request.camera.adapter.set'). Note carefully that this is NOT relative to the top left hand corner of the virtual screen. Although the position of the test card is specified relative to the reference point of the camera adapter, it is not constrained to lie within the area specified in 'request.camera.adapter.set'. The default values of *refX* and *refY* when the camera adapter is created are zero.

The parameter *rate* determines the rate at which the test card inverts its chequer pattern, relative to the frame rate of the camera. The default value of *rate* is 1.0, ie the same rate as the camera.

Pre-condition: None.

Post-condition: None.

- 'request.camera.adapter.test.card.show' (Show test card parameters.)

$$\{\} \quad \Rightarrow \quad \{size_{\mathrm{INT}} \;\; refX_{\mathrm{INT}} \;\; refY_{\mathrm{INT}} \;\; rate_{\mathrm{RAT}}\}$$

  This returns the *size* of the sub-squares of the test card, the position (*refX*, *refY*) of the top left hand corner of the test card relative to the reference point of the camer adapter, and the it rate at which the pattern is inverted relative to the frame rate of the camera.

  Pre-condition: None.

  Post-condition: None.

5.4 Capture virtual screen

The capture virtual screen contains an intermediate representation of the moving images from a camera, after they have been processed by a camera adapter.

Kind

'kind.capture.virtual.screen'

Input and output ports

- Input (many connections): 'port.capture.virtual.screen.input'

  Type: 'type.capture.project'

- Output (many connections): 'port.capture.virtual.screen.output'

  Type: 'type.capture.view'

Creation

The parameters used when the factory creates this object are:

$$sizeX_{\mathrm{NAME}} \;\; sizeY_{\mathrm{NAME}} \;\; visual_{\mathrm{NAME}}$$

The parameter *sizeX* specifies the width of the capture virtual screen, *sizeY* its height and *visual* the format of the pixels on it. The values of these parameters must be one of the allowed configurations. Currently the only allowed configuration is $sizeX = 512$, $sizeY = 512$ and $visual =$ 'name.monochrome.8.bit'.

Events

Only the standard ones.

Requests

None.

5.5 Encoder adapter

An encoder adapter takes as its input the representation of a frame on the capture virtual screen and converts this into a stream of segments (possibly using some compression mechanism).

Kind

'kind.encoder.adapter'

Input and output ports

- Input (only one connection): 'port.encoder.adapter.input'

  Type: 'type.capture.view'

- Output (many connections): 'port.encoder.adapter.output'

  Type: 'type.video.segment'

Creation

The parameters used when the factory creates this object are:

$$compression_{\text{NAME}} \quad parms_{\text{SEQUENCE[ANY]}}$$

compression specifies the method of compression which the encoder is to use, and *parms* gives such further initialisation details as are necessary for setting up the compression. The values of these parameters must specify one of the allowed compression schemes. Currently these can be 'name.lines.null', 'name.lines.dpcm', 'name.lines.pack', 'name.lines.pack.dpcm'. None of these require any further parameters.

Events

Only the standard ones.

Requests

- 'request.encoder.adapter.rate' (Set relative frame rate.)

  $$\{rate_{\text{RAT}}\} \quad \Rightarrow \quad \{\}$$

  This request sets rate at which the encoder takes frames from the virtual screen relative to the frame rate at which the camera puts them in. The default value of *rate* is 1.0, ie the same rate as the camera.

Pre-condition: None.

Post-condition: None.

- 'request.encoder.adapter.set' (Set reference point, X-Y size.)

$$\{refX_{\mathrm{INT}}\ \ refY_{\mathrm{INT}}\ \ sizeX_{\mathrm{INT}}\ \ sizeY_{\mathrm{INT}}\} \quad\Rightarrow\quad \{\}$$

This request sets the place on a virtual screen from where the encoder adapter takes its image. The reference point ($refX$, $refY$) is the coordinate on a capture virtual screen of the top left hand corner of the rectangle from where the image is taken. The rectangle is $sizeX$ pixels wide and $sizeY$ pixels high. Note that the bottom left hand pixel of the virtual screen is coordinate (0, 0), so x coordinates increase from left to right and y coordinates increase from top to bottom of the screen. If the coordinates are such that the image is taken from a place partially off the virtual screen, the encoder adapter might not produce segments. Only certain rectangle sizes are allowed: both $sizeX$ and $sizeY$ must be multiples of 16 in the range [16, ..., 256], but $sizeX$ need not be equal to $sizeY$. When it is created an encoder adapter is set to ($refX$, $refY$) = (0, 0) and $sizeX = sizeY = 128$.

Pre-condition: $sizeX$ and $sizeY$ must be one of the allowed values.

Post-condition: None.

- 'request.encoder.adapter.scale' (Scale image)

$$\{scaleX_{\mathrm{RAT}}\ \ scaleY_{\mathrm{RAT}}\} \quad\Rightarrow\quad \{\}$$

This request scales the image taken from the virtual screen before compression and translation into segments. The image encoded into segments is x pixels wide by y high, where

$$x = \mid scaleX \mid \times sizeX$$
$$y = \mid scaleY \mid \times sizeY$$

If $scaleX < 0$ this means the encoded image will be reflected side to side, and if $scaleY < 0$ the image is reflected top to bottom. When it is created an encoder adapter has scale factors $scaleX = 1.0$ and $scaleY = 1.0$.

Pre-condition: $scaleX$ and $scaleY$ are acceptable scale factors for this encoder adapter: either $\mid scaleX \mid = scaleY = 1.0$ or $\mid scaleX \mid = scaleY = 0.5$.

Post-condition: None.

- 'request.encoder.adapter.show' (Show reference point, etc.)

$$\{\} \quad\Rightarrow\quad \{refX_{\mathrm{INT}}\ \ refY_{\mathrm{INT}}\ \ sizeX_{\mathrm{INT}}\ \ sizeY_{\mathrm{INT}}\ \ scaleX_{\mathrm{RAT}}\ \ scaleY_{\mathrm{RAT}}\}$$

This returns the reference point (*refX*, *refY*) of the bottom left hand corner of the rectangle where the encoded image is taken from a vitual screen, along with its width *sizeX* and height *sizeY* and the scale factors *scaleX* and *scaleY* applied to it before such an image is turned into segments.

Pre-condition: None.

Post-condition: None.

5.6 Decoder adapter

A decoder adapter takes a stream of segments and decodes this into a series of frames which are placed on the mixer virtual screen. This may require the use of some decompression machinery.

Kind

'kind.decoder.adapter'

Input and output ports

- Input (only one connection): 'port.decoder.adapter.input'

  Type: 'type.video.segment'

- Output (only one connection): 'port.decoder.adapter.output'

  Type: 'type.mixer.project'

Creation

The parameters used when the factory creates this object are:

$$priority_{\mathrm{INT}}$$

priority is the priority of this decoder adapter in putting its images onto the virtual screen. If images from two decoders overlap, the one with the highest priority wins in the area of overlap and it is that image which is put on the virtual screen. The image with the lower priority will only be visible where it is not covered by the image with the higher priority. Streams of video segments with any compression scheme may be given to a decoder, but if segments with an unimplemented compression scheme are supplied this is a type error and an event may be raised.

Note that (0, 0) refers to the point in the top left hand corner of the virtual screen, so x coordinates increase from left to right and y coordinates increate from top to bottom of the screen.

Events

- 'event.input.too.fast' The segments are arriving at 'port.decoder.adaprter.input' too quickly.

Requests

- 'request.decoder.adapter.priority' (Set priority)

  $$\{priority_{\mathrm{INT}}\} \quad \Rightarrow \quad \{\}$$

  This request changes the priority of the decoder. It must already have a priority, since that is one of the creation parameters, so this request can be used to hide or expose images without needing to disconnect and remake decoder adapters.

  Pre-condition: None.

  Post-condition: None.

- 'request.decoder.adapter.set' (Set reference point)

  $$\{refX_{\mathrm{INT}} \quad refY_{\mathrm{INT}}\} \quad \Rightarrow \quad \{\}$$

  This request sets the place on a virtual screen where the decoder adapter places its image. The reference point ($refX$, $refY$) is the coordinate on a capture virtual screen of the top left hand corner of the rectangle where the image is put (but see 'request.decoder.adapter.clip'). When it is created, an decoder adapter's reference point is set to (0, 0), so if the image will fit on the screen at all it will do so at the start.

  Pre-condition: None.

  Post-condition: None.

- 'request.decoder.adapter.scale' (Scale image)

  $$\{scaleX_{\mathrm{RAT}} \quad scaleY_{\mathrm{RAT}}\} \quad \Rightarrow \quad \{\}$$

  This request scales the decompressed image from the segment stream before placing it on the virtual screen. If the current image from the segment stream is $sizeX$ pixels wide and $sizeY$ high, the image placed on the virtual screen is x pixels wide by y high, where

  $$x = \mid scaleX \mid \times sizeX$$
  $$y = \mid scaleY \mid \times sizeY$$

  If $scaleX < 0$ this means the image will be reflected side to side, and if $scaleY < 0$ the image is reflected top to bottom. When it is created a decoder adapter has scale factors $scaleX = 1.0$ and $scaleY = 1.0$.

  Pre-condition: $scaleX$ and $scaleY$ are acceptable scale factors for this decoder adapter, so either $\mid scaleX \mid = scaleY = 1.0$ or $\mid scaleX \mid = scaleY = 2.0$.

  Post-condition: None.

- ‘request.decoder.adapter.clip’ (Clip image)

  $$\{\mathit{clipRefX}_{\mathrm{INT}}\ \ \mathit{clipRefY}_{\mathrm{INT}}\ \ \mathit{clipSizeX}_{\mathrm{INT}}\ \ \mathit{clipSizeY}_{\mathrm{INT}}\} \ \Rightarrow\ \{\}$$

  This request specifies the clipping area for the decoder adapter relative to the reference point set with ‘request.decoder.adapter.set’. Note carefully that this clipping is NOT relative to the top left hand corner of the virtual screen.

  Pre-condition: None.

  Post-condition: None.

- ‘request.decoder.adapter.show’ (Show reference point, etc.)

  $$\{\} \ \Rightarrow\ \begin{array}{c}\{\mathit{priority}_{\mathrm{INT}}\ \ \mathit{refX}_{\mathrm{INT}}\ \ \mathit{refY}_{\mathrm{INT}}\ \ \mathit{clipRefX}_{\mathrm{INT}}\ \ \mathit{clipRefY}_{\mathrm{INT}}\\ \mathit{clipSizeX}_{\mathrm{INT}}\ \ \mathit{clipSizeY}_{\mathrm{INT}}\ \ \mathit{scaleX}_{\mathrm{RAT}}\ \ \mathit{scaleY}_{\mathrm{RAT}}\}\end{array}$$

  This returns the parameters of the decoder. The priority of this decoder is given first, then the reference point (*refX*, *refY*) of the top left hand corner of the rectangle where the decoded image is placed on a virtual screen. The scale factors *scaleX* and *scaleY* are applied to the image as it is placed on the virtual screen, and it is clipped to fit in an area *clipSizeX* by *clipSizeY* whose top left hand corner is at ($\mathit{refX} + \mathit{clipRefX}$, $\mathit{refY} + \mathit{clipRefY}$) on the virtual screen.

  Pre-condition: None.

  Post-condition: None.

5.7 Mixer virtual screen

The mixer virtual screen contains an intermediate representation of the moving images to be displayed on the screen, before they are taken by the display adapter.

Kind

‘kind.mixer.virtual.screeen’

Input and output ports

- Input (many connections): ‘port.mixer.virtual.screen.input’

  Type: ‘type.mixer.project’

- Output (only one connection): ‘port.mixer.virtual.screen.output’

  Type: ‘type.mixer.view’

Creation

The parameters used when the factory creates this object are:

$$\mathit{sizeX}_{\mathrm{NAME}} \quad \mathit{sizeY}_{\mathrm{NAME}} \quad \mathit{visual}_{\mathrm{NAME}}$$

The parameter *sizeX* specifies the width of the mixer virtual screen, *sizeY* its height and *visual* the format of the pixels on it. The values of these parameters must be one of the allowed configurations. Currently the only allowed configuration is $\mathit{sizeX} = 512$, $\mathit{sizeY} = 512$ and *visual* = 'name.monochrome.8.bit'. When it is created, a mixer virtual screen has a mask plane which completely obscures it.

Events

Only the standard ones.

Requests

- 'request.mixer.virtual.screen.reveal' (Reveal image)

  $$\{\mathit{refX}_{\mathrm{INT}} \quad \mathit{refY}_{\mathrm{INT}} \quad \mathit{sizeX}_{\mathrm{INT}} \quad \mathit{sizeY}_{\mathrm{INT}}\} \quad \Rightarrow \quad \{\}$$

  This request sets the mask plane so that the image on part of the mixer virtual screen is revealed. The display adapter can then take it and mix it with the direct computer output. Where the images on the virtual screen are not revealed, the user of Pandora will see the output which came directly from the computer. The reference point (*refX*, *refY*) is the coordinate of the top left hand corner of the rectangle to be revealed. The rectangle is *sizeX* pixels wide and *sizeY* pixels high. The top left hand pixel of the virtual screen is coordinate (0, 0).

  Pre-condition: None.

  Post-condition: Pixels in this rectangle are available to be mixed onto the computer display device.

- 'request.mixer.virtual.screen.obscure' (Obscure image)

  $$\{\mathit{refX}_{\mathrm{INT}} \quad \mathit{refY}_{\mathrm{INT}} \quad \mathit{sizeX}_{\mathrm{INT}} \quad \mathit{sizeY}_{\mathrm{INT}}\} \quad \Rightarrow \quad \{\}$$

  This request sets the mask plane so that the image on part of the mixer virtual screen is obscured. The reference point (*refX*, *refY*) is the coordinate of the top left hand corner of the rectangle to be obscured. The rectangle is *sizeX* pixels wide and *sizeY* pixels high.

  Pre-condition: None.

  Post-condition: Pixels in this rectangle cannot be shown on the display device attached to Pandora.

5.8 Display adapter

This takes frames from the mixer virtual screen and sends them to a display.

Kind

'kind.display.adapter'

Input and output ports

- Input (only one connection): 'port.display.adapter.input'

  Type: 'type.mixer.view'

Creation

The parameters used when the factory creates this object are:

$$\mathit{interlace}_{\mathrm{NAME}} \;\; \mathit{framerate}_{\mathrm{RAT}} \;\; \mathit{linerate}_{\mathrm{INT}} \;\; \mathit{aspect}_{\mathrm{RAT}} \;\; \mathit{visual}_{\mathrm{NAME}}$$

The parameter *interlace* can take either the value 'name.interlaced' or the value 'name.not.interlaced', and *visual* can take the value 'name.monochrome.8.bit'. The parameter *framerate* is the frame rate of the display, *linerate* is its line rate and its pixels have aspect ratio *aspect*. The values of the parameters should match the specification of the display attached to pandora, but currently these parameters do not govern the configuration of this display.

Events

Only the standard ones.

Requests

- 'request.display.adapter.set' (Set reference point, X-Y size.)

  $$\{\mathit{refX}_{\mathrm{INT}} \;\; \mathit{refY}_{\mathrm{INT}} \;\; \mathit{sizeX}_{\mathrm{INT}} \;\; \mathit{sizeY}_{\mathrm{INT}}\} \;\Rightarrow\; \{\}$$

  This request sets the place on a virtual screen from where the display adapter takes its image. The reference point (*refX*, *refY*) is the coordinate relative to the a mixer virtual screen of the left hand corner of the rectangle from where the image is taken. The rectangle is *sizeX* pixels wide and *sizeY* pixels high. The bottom left hand pixel of the virtual screen is coordinate (0, 0). Only certain rectangle sizes and positions are allowed.

  Pre-condition: The specified rectangle size and position is allowed.

  Post-condition: None.

- 'request.display.adapter.show' (Show reference point, X-Y size.)

$$\{\} \;\Rightarrow\; \{refX_{\mathrm{INT}} \;\; refY_{\mathrm{INT}} \;\; sizeX_{\mathrm{INT}} \;\; sizeY_{\mathrm{INT}}\}$$

  This returns the reference point (*refX*, *refY*) of the top left hand corner of the rectangle from where the display adapter will take its image from a virtual screen, along with its width *sizeX* and height *sizeY*.

  Pre-condition: None.

  Post-condition: None.

5.9 Audio sink

This is an object which takes audio segments and plays them through the speaker of the telephone plugged into Pandora.

Kind

'kind.audio.sink'

Input and output ports

- Input (only one connection): 'port.audio.sink.input'

  Type: 'type.audio.segment'

Creation

The creation parameters are

$$format_{\mathrm{NAME}} \;\; rate_{\mathrm{INT}} \;\; buffer_{\mathrm{INT}}$$

where *format* is the name of the encoding used for the samples, *rate* is the frequency at which they expected and *buffer* is the prefered size of the buffer used by this object (in bytes). Only certain formats and rates are allowed.

Events

- 'event.input.too.fast' The segments are arriving at 'port.audio.sink.input' too quickly.

Requests

None.

5.10 Audio source

This is an object which delivers audio segments captured from the microphone of the telephone plugged into Pandora's box.

Kind

'kind.audio.source'

Input and output ports

- Output (many connections): 'port.audio.source.output'
  Type: 'type.audio.segment'

Creation

The creation parameters are

$$format_{\mathrm{NAME}} \quad rate_{\mathrm{INT}}$$

where *format* is the name of the encoding used for the samples, and *rate* if the frequency at which they are taken. Only certain formats and rates are allowed.

Events

Only the standard ones.

Requests

None.

5.11 Keypad

This is the source of keypad events from the phone.

Kind

'kind.keypad'

Input and output ports

None.

Creation

No parameters.

Events

- ‘event.keypad.cradle.up’ The phone is off its cradle.
- ‘event.keypad.cradle.down’ The phone is on its cradle.
- ‘event.keypad.r’ The R button has been pressed.
- ‘event.keypad.0’ The 0 button has been pressed.
- ‘event.keypad.1’ The 1 button has been pressed.
- ‘event.keypad.2’ The 2 button has been pressed.
- ‘event.keypad.3’ The 3 button has been pressed.
- ‘event.keypad.4’ The 4 button has been pressed.
- ‘event.keypad.5’ The 5 button has been pressed.
- ‘event.keypad.6’ The 6 button has been pressed.
- ‘event.keypad.7’ The 7 button has been pressed.
- ‘event.keypad.8’ The 8 button has been pressed.
- ‘event.keypad.9’ The 9 button has been pressed.
- ‘event.keypad.star’ The * button has been pressed.
- ‘event.keypad.hash’ The # button has been pressed.

Requests

None.

5.12 Synchroniser

THIS WILL NOT BE AVAILABLE INITIALLY.

This object buffers two streams of inputs and synchronises them with each other based on the timestamp values in their segment headers.

Kind

‘kind.synchroniser’

Input and output ports

- Input (only one connection): ‘port.synchroniser.input1’
  Type: ‘type.any.segment’
- Output (many connections): ‘port.synchroniser.output1’
  Type: ‘type.any.segment’
- Input (only one connection): ‘port.synchroniser.input2’
  Type: ‘type.any.segment’
- Output (many connections): ‘port.synchroniser.output2’
  Type: ‘type.any.segment’

Creation

$$\mathit{buffer}_{\mathrm{INT}}$$

The parameter *buffer* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- ‘event.input1.too.fast’ The segments on ‘synchroniser.input1’ are arriving too quickly.
- ‘event.input2.too.fast’ The segments on ‘synchroniser.input2’ are arriving too quickly.

Requests

None.

5.13 Joiner

THIS WILL NOT BE AVAILABLE INITIALLY.

This object combines up to one stream each of Video, Audio and Stills in to a stream of type ‘type.all.segment’.

Kind

‘kind.joiner’

Input and output ports

- Input (only one connection): 'port.joiner.video.input'

  Type: 'type.video.segment'
- Input (only one connection): 'port.joiner.audio.input'

  Type: 'type.audio.segment'
- Input (only one connection): 'port.joiner.still.input'

  Type: 'type.still.segment'
- Output (many connections): 'port.joiner.output'

  Type: 'type.all.segment'

Creation

$$\mathit{buffer}_{\mathrm{INT}}$$

The parameter *size* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- 'event.video.input.too.fast' The segments on 'joiner.video.input' are arriving too quickly.
- 'event.audio.input.too.fast' The segments on 'joiner.audio.input' are arriving too quickly.
- 'event.still.input.too.fast' The segments on 'joiner.still.input' are arriving too quickly.

Requests

None.

5.14 Splitter

THIS WILL NOT BE AVAILABLE INITIALLY.

This object takes a stream of segments of type'type.all.segment' and splits it up into three streams: one each of Video, Audio and Stills.

Kind

'kind.splitter'

Input and output ports

- Input (only one connection): 'port.splitter.input'
  Type: 'type.all.segment'
- Output (many connection): 'port.splitter.video.output'
  Type: 'type.video.segment'
- Output (many connections): 'port.splitter.audio.output'
  Type: 'type.audio.segment'
- Output (many connections): 'port.splitter.still.output'
  Type: 'type.still.segment'

Creation

$$buffer_{\mathrm{INT}}$$

The parameter *buffer* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- 'event.input.too.fast' The segments on 'splitter.input' are arriving too quickly.

Requests

None.

5.15 Network sink

This object sends a stream of segments to a particular network address and network port.

Kind

'kind.network.sink'

Input and output ports

- Input (only one connection): 'port.network.sink.input'
  Type: 'type.any.segment'

Creation

$$buffer_{\mathrm{INT}}$$

The parameter *buffer* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- 'event.input.too.fast' The segments on 'network.sink.input' are arriving too quickly.

Requests

- 'request.network.sink.set' (Set network destination)

  $$\{address_{\text{SEQUENCE[INT]}}\} \quad \Rightarrow \quad \{\}$$

  This request sets the destination to which segments will be sent on the network. The parameter *address* specifies the address on the network of the receiving machine (probably another Pandora's box). Currently this sequence must contain exactly two integers, which are respectively the CFR address of the receiving machine and the port number of the receiving process in that machine. If the receiving machine was another Pandora's box, the second number identifies a network source object in that machine.

  Pre-condition: None.

  Post-condition: The segments sent to this sink may be received by a machine at the specified address over the network.

5.16 Network source

This object receives a stream of segements on a particular network port.

Kind

'kind.network.source'

Input and output ports

- Output (many connections): 'port.network.source.output'

  Type: 'type.any.segment'

This is a change from the intended specification, to make networks usable without splitters and joiners. The type was originally 'type.all.segment'.

Creation

No parameters.

Events

Only the standard ones.

Requests

- 'request.network.source.show' (Show own network address)

$$\{\} \;\Rightarrow\; \{address_{\text{SEQUENCE[INT]}}\}$$

  This request returns *address*, which is currently exactly two elements long. The first integer is the address of this Pandora's box on the network, and the second is the network port corresponding to this object. These can be used by the host to set up a connection to another Pandora's box.

  Pre-condition: None.

  Post-condition: None.

5.17 Host sink

THIS WILL NOT BE AVAILABLE INITIALLY.

A host sink object sends its input stream of segments to the host, labeled as coming from this object.

Kind

'kind.host.sink'

Input and output ports

- Input (only one connection): 'port.host.sink.input'

  Type: 'type.any.segment'

Creation

$$buffer_{\text{INT}}$$

The parameter *buffer* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- 'event.input.too.fast' The segments on 'host.sink.input' are arriving too quickly.

Requests

None

5.18 Host source

THIS WILL NOT BE AVAILABLE INITIALLY.

A host source receives a stream of segments (labeled for this object) from the host and delivers them to its output port.

Kind

'kind.host.source'

Input and output ports

- Output (many connections): 'port.host.source.output'

  Type: 'type.all.segment'

Creation

No parameters.

Events

Only the standard ones.

Requests

None

5.19 File sink

THIS WILL NOT BE AVAILABLE INITIALLY.
A video repository on the CFR with similar facilities can be used instead.

This object will sink a stream of segments in a similar way to a network, but instead of going to a remote destination they will be available for subsequent playback.

Kind

'kind.file.sink'

Input and output ports

- Input (only one connection): 'port.file.sink.input'

  Type: 'type.any.segment'

Creation

$$buffer_{\mathrm{INT}}$$

The parameter *buffer* is the prefered size of buffer (in bytes) which is to be used by this object.

Events

- 'event.input.too.fast' The segments on 'network.sink.input' are arriving too quickly.

Requests

- 'request.file.sink.set' (Set filename)

  $$\{file_{\mathrm{SEQUENCE[INT]}}\} \Rightarrow \{\}$$

  This request sets the file in which recorded segments will be stored.

  Pre-condition: This object is in its stopped state, not already going.

  Post-condition: None.

- 'request.file.sink.start' (Start recording)

  $$\{\} \Rightarrow \{\}$$

  This request causes the input stream of segments to be recorded in the previously specified file. Any existing contents of the file will be overwritten.

  Pre-condition: The object has had a file name set up already.

  Post-condition: The object is in the going state.

- 'request.file.sink.stop' (Stop recording)

  $$\{\} \Rightarrow \{\}$$

  This request causes the input stream of segments to be discarded until the next start request. The file is closed.

  Pre-condition: The object has had a file name set up already.

  Post-condition: The object is in the stopped state.

- 'request.file.sink.show' (Show filename etc)

  $$\{\} \Rightarrow \{status_{\mathrm{NAME}}\ file_{\mathrm{SEQUENCE[INT]}}\}$$

  This request returns the status of the object, either 'name.going' or 'name.stopped' and the name of the file it is currently associated with. If no file has been set up this is the null string.

  Pre-condition: None.

  Post-condition: None.

5.20 File source

THIS WILL NOT BE AVAILABLE INITIALLY.
A video repository on the CFR with similar facilities can be used instead.

This object behaves somewhat like a remote Pandora supplying a stream of segments over the network. It may provide facilities such as playback at a different speed to the original recording.

Kind

'kind.file.source'

Input and output ports

- Output (many connections): 'port.file.source.output'

  Type: 'type.all.segment'

Creation

No parameters.

Events

Only the standard ones.

Requests

- 'request.file.source.queue.set' (Queue file selection)

  $$\{\mathit{file}_{\mathrm{SEQUENCE[INT]}}\} \quad \Rightarrow \quad \{\}$$

  This request queues the action "change to *file*" for execution by the file source when it finishes all the actions it currently has queued.

  Pre-condition: None.

  Post-condition: *file* is selected as the current file.

- 'request.file.source.queue.seek' (Queue a seek)

  $$\{\mathit{time}_{\mathrm{TIME}}\} \quad \Rightarrow \quad \{\}$$

  This request queues the action "seek to *time* in the current file" for execution by the file source when it finishes all the actions it currently has queued.

  Pre-condition: A change file action must have been queued previously.

  Post-condition: None.

- ‘request.file.source.queue.wait’ (Queue a wait)

$$\{time_{\mathrm{TIME}}\} \;\; \Rightarrow \;\; \{\}$$

This request queues the action “wait for *time*” for execution by the file source when it finishes all the actions it currently has queued.

Pre-condition: None

Post-condition: None.

- ‘request.file.source.queue.play’ (Queue a play)

$$\{time_{\mathrm{TIME}}\} \;\; \Rightarrow \;\; \{\}$$

This request queues the action “play the current file for *time*” for execution by the file source when it finishes all the actions it currently has queued. The file is played from the place that the last “seek” or “play” left it. If no seek or play has been executed since a “change file” the file is played from the beginning.

Pre-condition: A change file action must have been queued previously.

Post-condition: None.

- ‘request.file.source.abandon’ (Abandon queued actions)

$$\{\} \;\; \Rightarrow \;\; \{\}$$

This request flushes the queue of actions pending execution and stops the currenly executing action as well.

Pre-condition: None.

Post-condition: If a file was selected, it remains selected. Selections which were queued but not executed have no effect.

- ‘request.file.source.show’ (Show the state)

$$\{\} \;\; \Rightarrow \;\; \{time_{\mathrm{TIME}} \;\; file_{\mathrm{SEQUENCE[INT]}}\}$$

The result *time* is the total time that queued actions will take to execute, and *file* is the name of the currently selected file, or null if there is no currently selected file.

Pre-condition: None.

Post-condition: None.

Appendix: Concrete syntax

This appendix gives the concrete syntax of segments and the protocol for communicating over the link with the host. Note that the matter of big or little-endian formats for integers and other larger than byte sized fileds has still to be resolved.

Link protocol

There are two streams to be multiplexed onto the link from the host to Pandora. These are assigned stream numbers as follows:

Requests	= 1
Asynchronous data from the host	= 2

There are three streams to be multiplexed onto the link from Pandora to the host. These are assigned stream numbers as follows:

Replies to requests	= 3
Asynchronous events	= 4
Asynchronous data to the host	= 5

The traffic in both directions is packets as follows:

BYTE stream number
BYTE concrete syntax version number (currently 0)
BYTE reserved
BYTE reserved
INT32(little-endian) bytes to follow
(Only positive numbers and zero allowed).
[bytes to follow] BYTE the contents of the packet

The packets of the concrete syntax correspond exactly to requests, replies, events and so on. For example, requests cannot be sent in several packets. Replies will not be sent in more than one packet. The streams should be regarded not as streams of bytes but as streams of packets.

Segments and other types

The implementation of the various types is as follows:

NAME	INT32. 32 bit twos complement signed integer. (see the list of names below).
INT	INT32. 32 bit twos complement signed integer.
RAT	INT32. 32 bit integer, i, viewed as two 16 bit twos complement signed integers, m and l, where $i = (m \times 2^{16}) + l$ This represents the rational number $r = \frac{m}{l}$
BYTE	BYTE. An 8 bit byte.
TIME	INT32. 32 bit twos complement signed integer. Time in milli-seconds. 31 bits (ie only positive times) gives about 3 weeks before overflow.
ANY	Whatever type is actually chosen.
SEGMENT	The format is just the abstract syntax mapped into concrete syntax using the definitions in this section
SEQUENCE[*type*]	All the items in the sequence are placed end to end, with no word alignment.

The concrete syntax of the collection

$$\{items_{\text{SEQUENCE}[type]}\}$$

is a count followed by that number of bytes:

INT32 byte count
[byte count] BYTE all the items in the sequence placed end to end

So the concretization of

$$\{items_{\text{SEQUENCE}[type]}\}$$

is identical with the concretization of

$$length_{\text{INT}} \quad items_{\text{SEQUENCE}[type]}$$

where *length*= the number of bytes in the concretization of the sequence *items*.

Names

This section contains the integers corresponding to all valid names.

User defined objects

The range #0000 to #0FFF is reserved for user defined object names.

Predefined object names

Predefined object names are in the range #1000 to #1FFF.

'object.factory'	#1000

Kinds of objects

Object kinds are in the range #2000 to #2FFF.

'kind.factory'	#2000
'kind.camera.adapter'	#2001
'kind.capture.virtual.screen'	#2002
'kind.encoder.adapter'	#2003
'kind.decoder.adapter'	#2004
'kind.mixer.virtual.screen'	#2005
'kind.display.adapter'	#2006
'kind.audio.sink'	#2007
'kind.audio.source'	#2008
'kind.keypad'	#2009
'kind.synchroniser'	#200A
'kind.joiner'	#200B
'kind.splitter'	#200C
'kind.network.sink'	#200D
'kind.network.source'	#200E
'kind.host.sink'	#200F
'kind.host.source'	#2010
'kind.file.sink'	#2011
'kind.file.source'	#2012

Port names

Port names are in the range #3000 to #3FFF.

'port.camera.adapter.output'	#3000
'port.capture.virtual.screen.input'	#3001
'port.capture.virtual.screen.output'	#3002
'port.encoder.adapter.input'	#3003
'port.encoder.adapter.output'	#3004
'port.decoder.adapter.input'	#3005
'port.decoder.adapter.output'	#3006
'port.mixer.virtual.screen.input'	#3007
'port.mixer.virtual.screen.output'	#3008
'port.display.adapter.input'	#3009
'port.audio.sink.input'	#300A
'port.audio.source.output'	#300B
'port.synchroniser.input1'	#300C
'port.synchroniser.input2'	#300D
'port.synchroniser.output1'	#300E
'port.synchroniser.output2'	#300F
'port.joiner.video.input'	#3010
'port.joiner.audio.input'	#3011
'port.joiner.still.input'	#3012
'port.joiner.output'	#3013
'port.splitter.input'	#3014
'port.splitter.video.output'	#3015
'port.splitter.audio.output'	#3016
'port.splitter.still.output'	#3017
'port.network.sink.input'	#3018
'port.network.source.output'	#3019
'port.host.sink.input'	#301A
'port.host.source.output'	#301B
'port.file.sink.input'	#301C
'port.file.source.output'	#301D

Type names

Type names are in the range #4000 to #4FFF.

'type.capture.project'	#4000
'type.capture.view'	#4001
'type.mixer.project'	#4002
'type.mixer.view'	#4003
'type.video.segment'	#4004
'type.still.segment'	#4005
'type.audio.segment'	#4006
'type.all.segment'	#4007
'type.any.segment'	#4008
'type.any1.segment'	#4009

Event names

Event names are in the range #5000 to #5FFF.

'event.internal.error'	#5000
'event.segment.too.large'	#5001
'event.missing.segment'	#5002
'event.wrong.type.of.segment'	#5003
'event.closed.down'	#5004
'event.input.too.fast'	#5005
'event.input1.too.fast'	#5006
'event.input2.too.fast'	#5007
'event.video.input.too.fast'	#5008
'event.audio.input.too.fast'	#5009
'event.still.input.too.fast'	#500A
'event.keypad.cradle.up'	#500B
'event.keypad.cradle.down'	#500C
'event.keypad.r'	#500D
'event.keypad.0'	#500E
'event.keypad.1'	#500F
'event.keypad.2'	#5010
'event.keypad.3'	#5011
'event.keypad.4'	#5012
'event.keypad.5'	#5013
'event.keypad.6'	#5014
'event.keypad.7'	#5015
'event.keypad.8'	#5016
'event.keypad.9'	#5017
'event.keypad.star'	#5018
'event.keypad.hash'	#5019
'event.host.broken'	#501A
'event.log'	#501B

Request names

Request names are in the range #6000 to #6FFF.

'request.factory.create'	#6000
'request.factory.allow.event'	#6001
'request.factory.forbid.event'	#6002
'request.factory.show.creation.parameters'	#6003
'request.factory.delete'	#6004
'request.factory.connect'	#6005
'request.factory.disconnect'	#6006
'request.factory.show.instances'	#6007
'request.factory.show.objects'	#6008
'request.factory.show.port'	#6009
'request.factory.show.connected'	#600A
'request.factory.show.version'	#600B
'request.factory.verbatim'	#600D
'request.camera.adapter.set'	#600E
'request.camera.adapter.show'	#600F
'request.encoder.adapter.set'	#6010
'request.encoder.adapter.scale'	#6011
'request.encoder.adapter.show'	#6012
'request.decoder.adapter.set'	#6013
'request.decoder.adapter.scale'	#6014
'request.decoder.adapter.show'	#6015
'request.mixer.virtual.screen.reveal'	#6016
'request.mixer.virtual.screen.obscure'	#6017
'request.display.adapter.set'	#6018
'request.display.adapter.show'	#6019
'request.network.sink.set'	#601A
'request.network.sink.show'	#601B
'request.network.source.set'	#601C
'request.network.source.show'	#601D
'request.file.sink.set'	#601E
'request.file.sink.start'	#601F
'request.file.sink.stop'	#6020
'request.file.sink.show'	#6021
'request.file.source.queue.set'	#6022
'request.file.source.queue.seek'	#6023
'request.file.source.queue.wait'	#6024
'request.file.source.queue.play'	#6025
'request.file.source.abandon'	#6026
'request.file.source.show'	#6027

'request.factory.show.allowed.events'	#6028
'request.factory.show.forbidden.events'	#6029
'request.factory.show.requests'	#602A
'request.factory.debug.init'	#602B
'request.decoder.adapter.priority'	#602C
'request.decoder.adapter.clip'	#602D
'request.encoder.adapter.rate'	#602E
'request.factory.show.ports'	#602F
'request.camera.adapter.test.card.set'	#6030
'request.camera.adapter.test.card.show'	#6031

Reply names

Reply names are in the range #7000 to #7FFF.

'reply.success'	#7000
'reply.unknown.object'	#7001
'reply.unknown.request'	#7002
'reply.failed'	#7003
'reply.request.too.large'	#7004
'reply.malformed.request'	#7006
'reply.host.broken'	#7007
'reply.internal.error'	#7008

Other names

These are miscellaneous names used as values for object properties and so on (for example the name of a compression scheme, etc). Miscellaneous names are in the range #8000 to #8FFF.

'name.lines.null'	#8000
'name.null'	#8002
'name.silence'	#8003
'name.not.interlaced'	#8004
'name.interlaced'	#8005
'name.monochrome.8.bit'	#8006
'name.going'	#8007
'name.stopped'	#8008
'name.segment.format1'	#8009
'name.lines.dpcm'	#800A
'name.lines.pack'	#800B
'name.lines.pack.dpcm'	#800C