A Video Processing Framework in \textit{occam-\pi}

Carl Ritson

March 23, 2006

Abstract

The \textit{occam-\pi} language under development at the University of Kent provides many new and powerful features over traditional \textit{occam}. This report details a video processing framework developed to explore using these features for multimedia applications.

Video filters and processors map to software processes, with data transferred between them using channels, forming an inherently parallel network. Mobile data types make the transfer of large volumes of data efficient, and mobile channel ends provide the basis for dynamic reconfigurable systems.

Demonstration applications including a video player, the design of which has been simplified by \textit{occam-\pi} constructs, are presented. Further to this, there is a discussion of preliminary benchmarks which reveal that the framework has comparable overhead to a system programmed using traditional methods.

1 Introduction

This report details a video processing framework written in the \textit{occam-\pi} language. Its design derives from the author’s experiences developing for and using a number of open source video processing tools [1, 2] and applications [3, 4]. Through this research the author has sought to test the idea that \textit{occam-\pi} is not only suitable for use in handling multimedia data, but has features desirable when developing software components for doing so.

A video processing framework, or more generally a multimedia framework, is an API and supporting functions which facilitate the interaction and transfer of data between multimedia-handling software components. Almost all video processing systems are constructed within such a framework, forming pipelines or layers of distinct software components which communicate.

Software components with standardised interfaces can be easily modelled using \textit{OO} (Object Orientated), and as a consequence most existing frameworks are written in \textit{OO} languages. AviSynth [1] (C++), the DirectShow Filter Graph [5] (C++) and Kamaelia [6] (Python) are examples of this, while other frameworks such as GStreamer [7] use an object system (GLib [8]) on top of a procedural language (C).

Communication between components is either implemented by direct calls to methods upon other components [1], or by interaction with buffers [5, 7] (often called pins or pads), shared between adjacent components. The latter of these two approaches can be directly parallelised, where as the former requires the addition of buffers between parallel components. The method call model, without parallelism, is often preferred for interactive applications, where a control component pushes or pulls data to the user, as it is easier to reason about the data currently being presented to the user.

Neither of these approaches simplify the design process, particularly in the presence of parallelism. For example, in order to create a circular ring of filters using method calls, one component must act as the initiator, so that the system does not descend into infinite recursion. It is also often difficult to reason about the correctness of system produced using such methods. The \textit{occam-\pi} language provides mechanisms and design patterns which would seem well-suited to solving these problems.

The transputer processor [9] for which the \textit{occam} language was originally developed has previously been used for multimedia [10], and \textit{occam}-like languages have been developed which allow the dynamic specification of multimedia systems [11]. However such research predates the \textit{occam-\pi} language and the desirable features [12, 13] it adds for handling multimedia. Hence \textit{occam-\pi}’s potential in this area is as yet unexplored.

In the framework presented in this report, \textit{occam} channels, built on the mathematically rigorous CSP [14], provide the standardised interface between components, and allow parallelism without the explicit introduction of buffers. The networks of processes produced model the communicating software or hardware components of a traditional video processing system. Mobile data types and channel ends, introduced in \textit{occam-\pi}, are used to build efficient and dynamic networks of processes, and two modes of operation, push and pull, are explored.

Sections 2, 3 and 5 detail the protocols and data types within the framework, and sections 4 and 6 describe their application. Sections 7 and 8 give details of other functionality so far developed, and conclusions are drawn in section 9 with some thoughts for future work in sections 10 and 11.

2 Streaming - P.MM

At the heart of the framework is a single stream protocol \textit{P.MM} (Protocol MultiMedia), which carries video and audio along with untyped “packet” data and commands using a “push” data flow model. All data elements are declared mobile [12]. If mobile data types were not used then data would need to be copied between communicating processes, something inappropriate for video where the data-rate will typically exceed 20MiB/sec (for
digitised PAL or NTSC). Using mobile data types, the pipeline of processes can be extended with no significant decrease in performance.

```plaintext
-- MOBILE keywords omitted for brevity
 PROTOCOL P.MM
  CASE
    init; TRACK.INFO; []BYTE
    packet; TID; PKT_HDR; []R.DESC; []BYTE
    video; TID; VF.DESC; []BYTE
    audio; TID; AF.DESC; []BYTE
    flush
    purge
    skip
    end

init signals the start of a stream.
packet carries untyped media frames, typically compressed video or audio.
video carries a single video frame.
audio carries an audio frame of variable length.
flush instructs the receiver to output all ready buffers, then forward the flush command. This is necessary for the ideas presented in sections 3.4 and 4.
purge instructs the receiver to clear its internal state without generating any output, and prepare for new data. The purge command is forwarded when the receiver is ready for new input. Like flush, this is used in sections 3.4 and 4.
skip instructs the receiver to do nothing. Unlike flush and purge it is not forwarded. This command is used to build zero-place buffers (see section 3.5).
end indicates the end of the stream; no data will follow. This provides a form of graceful termination [15]. On receipt the receiver should terminate after outputting any ready buffers (like a flush) and propagating the end message, unless it is restartable (such as pluggable.stream.input.end in section 6).

See appendix A for a full description of the P.MM protocol.

2.1 P.MM Usage Contract

The expected sequence of messages on a channel of P.MM is encapsulated in the following regular expression:

```plaintext
  skip*
  (init
   (packet|video|audio|flush|purge|skip)*)?
end
```

In summary, this means that the only certain event is the end of the stream, which can happen without prior initialisation. The skip command is permitted before initialisation to aid in the creation of zero-place buffers (see section 3.5).

Additionally, it is assumed there will not be a one-to-one mapping between input and output for processes implementing P.MM - a process may buffer as much or as little data as needed. The effect of this is that after sending an init, purge or end message to a process, it must not be assumed that the next output message will be of that type. CODECs in particular require this form of internal buffering.

2.2 Timing - TID

Each elementary type (packet, video, audio) is associated with a TID data structure (Temporal IDentifier). A TID structure describes the position of a packet or frame within the timeline of the stream. This is done via the timecode field which is an offset in nanoseconds from a fixed point, typically the beginning of the stream. Nanoseconds are employed to allow the framework to manipulate data from Matroska [16] without loss of timing resolution; however, microseconds would be sufficient for all present media formats. A duration in milliseconds is also stored, although has limited uses.

```plaintext
DATA TYPE TID
  PACKED RECORD
    INT64 timecode:
    INT duration:
```

Traditionally multimedia systems identify frames by their number in sequence from the beginning of the stream, or using SMPTE timecodes [17, 18] which combine time and frame number offsets. This means that a stream is expected to have a fixed number of frames per unit time. In contrast the framework presented here identifies frames purely based on time. There are three significant reasons for this:

1. When combining different streams together it is more efficient to have a single common timeline to work with, rather than many sets of sample number and rate pairs which must be normalised. Although this at first this normalisation seems trivial, without timecode-based identification there is no way to synchronise and temporally manipulate streams without first knowing their respective sample rates - limiting the ways in which a system can dynamically adapt to new streams.

2. Any fixed sample rate system can be represented in a purely timecode based system, assuming the timecodes have sufficient resolution and range.

3. A timecode-based system can represent streams with variable sample rates. Sample rates themselves are an artifact of analogue electronics which need not apply directly to digital data. As the world moves toward purely digital production and delivery of media content (HDTV, IPTV and digital end-to-end mastering), it is my expectation that variable frame rate material will become the norm.

2.3 Aside on VFR (Variable Frame Rate)

Although variable sample rate audio is uncommon, mixed frame rate video content is already in wide spread circulation. In the
production of NTSC television content and DVDs, it is common to use “pull-down” techniques to mix frame rates of media from different sources (typically 23.976fps and 29.970fps). These mixed content streams can be represented in the digital domain by using a higher frame rate which is a common multiple to all the source rates (typically 119.880fps), and introducing so called “drop frames” were no actual frame data exists. This technique is however, only applicable where there is a common multiple between the frame rates. A timecode based system can overcome this problem.

VFR allows the time and rate of change to be free of quantisation. Modern video CODECs are based around coding change - there is no need to code a new frame if nothing has changed. This is typically dealt with using drop frames, so a static image becomes a constant stream of “no change” messages. However with VFR there would be no output at all, with potential bandwidth and disk space savings. Video scenes requiring smooth motion can have high rates of change, and other scenes lower rates. As the stream is not quantised, the actual changes can be placed at the most visually pleasing points in time - allowing acceptably smooth motion at lower frame rates, with further potential bandwidth and disk space savings.

Even though the stream will eventually be quantised by an output device, most LCD displays run in excess of 60Hz which is between two and three times the typical underlying rate of change, and high end systems often surpass 100Hz. This gives good scope for coding motion in a more visually pleasing way with less frames. Furthermore, if an unclocked interface were available it could be used to display VFR content without quantisation.

In summary, it seems clear that VFR has benefits over CFR there. Although there has not yet been much research into exploiting such benefits, the inevitability of variable rate content is coming to be recognised [19, 20].

2.4 Flexibility - TRACK.INFO

The TRACK.INFO data structure is heavily based on the “Track” descriptor in the Matroska [16] media container format. The Matroska format is designed to be able to hold an arbitrary number of media tracks of any type, and thus provided much inspiration for developing a flexible framework. Earlier versions of the TRACK.INFO were almost exact mirrors of the equivalent Matroska structure; however, the design has now been refined. The “TRACK” element of the name of this structure is itself a Matroska legacy; “STREAM” would be equally suitable. See appendix B for a full description of the TRACK.INFO data structure.

3 Interactivity - P.MM.CTL/SEEKABLE

The following protocols act as a request/response pair and extend the commands in the stream P.MM protocol to provide interactive facilities through a “pull” model. This pull model sacrifices full parallel processing; only a single request is outstanding at a given time. It is intended for interactive applications where filling the pipeline with data is undesirable (due to the increase in end-to-end latency that results). Pre-roll buffer processes can be added to keep the pipeline full and restore parallel processing, if so desired.

3.1 Feedback - P.MM.CTL

P.MM.CTL is the request protocol. The process “pulling” data sends a single request and waits for a response.

```pro
PROTOCOL P.MM.CTL
CASE
  init
  next
  seek; INT; INT64
  purge
end
```

init requests the TRACK.INFO structure and setup data for the track.

next requests the logically next packet, video or audio frame in the stream.

seek requests that the stream be repositioned to a new timecode in the INT64. The INT value is a constant describing how to handle cases where the exact timecode requested can not be reached, which is almost inevitable. If set to SEEK.CLOSEST then the closest match will be picked. SEEK.BACKWARD requests the closest point not after the specified timecode, and SEEK.FORWARD the closest point not before the specified timecode. In a typical video player, SEEK.BACKWARD will be used for rewind, and SEEK.FORWARD for fast-forward, in order to give the behaviour that a user would expect.

purge requests that the process clear all internal buffers. This request does not generate a response and is simply propagated to the preceding process.

end requests that the process terminate, and that it request the same of its preceding processes. Like purge, this does not generate a response.

3.2 Simplified P.MM - P.MMSEEKABLE

P.MMSEEKABLE is a simplified P.MM which carries responses. The data messages, packet, video and audio have the same meaning as in P.MM.

```pro
-- MOBILE keywords omitted for brevity
PROTOCOL P.MM.SEEKABLE
CASE
  init; TRACK.INFO; []BYTE
  packet; TID; PKT.HDR; []R.DESC; []BYTE
  video; TID; VF.DESC; []BYTE
  audio; TID; AF.DESC; []BYTE
  seeked; INT64
end
```

P.MMSEEKABLE is a simplified P.MM which carries responses. The data messages, packet, video and audio have the same meaning as in P.MM.
**init** no longer represents the start of stream, it simply describes the properties of the track and is the response to an init request.

**seeked** is sent in response to a seek request, and carries the time-code of the new stream position.

**end** means the end of the stream has been reached, but does not indicate that the component has terminated.

### 3.3 P.MM.CTL/SEEKABLE Usage Contract

The expected sequence of requests on a P.MM.CTL channel is as follows:

\[(init|next|seek|purge)*\] end

With the following request response pattern:

- **init** => (init | end)
- **next** => (packet | video | audio | end)
- **seek** => (seeked | end)
- **purge** => ()
- **end** => ()

### 3.4 Encapsulation

Through careful design of both P.MM and P.MM.CTL/SEEKABLE, it has been possible to write a reusable wrapper around P.MM processes to present them as P.MM.CTL/SEEKABLE processes: seekable.wrapper.

1. At startup the wrapper requests an init message and other data from the preceding process, and feeds it to the wrapped process until an init message is produced. This init message is stored and used to respond to init requests from the successor process.

2. On a next request the wrapper makes next requests to the preceding process, feeding data to the wrapped process until output is produced. The output is forwarded to the successor. If an end response is encountered from the preceding process then the wrapped process is sent a flush message which causes it to output any available data. Should the wrapper receive a flush message from the wrapped process, it returns end to the successor.

3. On a seek request, the wrapped process is sent a purge. The wrapper waits for purge to be emitted by the wrapped process, discarding any intervening output. At the same time, the seek request is forwarded to the preceding process, and its response returned to the successor when the purge of the wrapped process is complete.

4. A purge request is handled much like seek without any response to the successor.

5. An end request is forwarded to the preceding process and sent to the wrapped process. Once the wrapped process outputs end, then the wrapper terminates. This is the only point at which the wrapped process will receive an end message.

### 3.5 The zero.place.buffer

The seekable wrapper and many other processes in the framework depend on the zero.place.buffer which is made possible by the skip message in the P.MM protocol. The zero.place.buffer sends a skip to its output, and sends a message to its ready channel only when it is accepted. Using this method it reports the readiness to receive of a process without buffering any data. This simple technique has proved very useful, although the name “zero-place buffer” could quite rightly be considered an oxymoron. It is a variation of the standard occam “requester” and overwriting buffer patterns.

### 4 ovp - The occam-\(\pi\) Video Player

This section explores the development of the occam-\(\pi\) video player, ovp, using the framework so far presented.

#### 4.1 Basic Player

A basic video player could use the network shown in Figure 2 for files containing two tracks (variable input files are treated in section 5). The specific decoder and output types can be inferred from the init messages. The network post-init messages might look like Figure 3. This is in effect a form of runtime typing, and as such there is no guarantee that the given network of components will function together; section 10 discusses this further.

Data will flow through the network as the output processes consume it (at a rate depending on the timecodes in TID structures of the frames). The network will act as a pipeline, with the decoders running in parallel with the outputs. This will continue until an end message is received, which will flush and shut down the network. This gives us linear playback.

![Figure 1: Process network for the seekable.wrapper.](image1)

![Figure 2: Process network for a simple occam-\(\pi\) video player.](image2)
Figure 3: Simple occam-π video player after init messages have passed through the network.

Figure 4: Process network for a seekable occam-π video player.

4.2 User Control

For non-linear playback with pausing, we need to modify the network; an initial solution is presented in Figure 4. The “User Control” process repositions the input file in response to user seek requests. The purpose of the “Flow Control” processes is less obvious. As the network buffers data, a pause or change in track position will take time to filter through from the demuxer to the outputs. This is not desirable if we want the network to appear responsive to user input. The flow controls are thus used to drain the input and decoding parts of the network during a pause or seek, and purge the outputs, meaning the user does not have to wait for the pipeline to empty before new data is presented.

One significant issue with this design is that it requires the temporal position of both streams to be the same after a seek or pause, otherwise there will be skew between them. This is not something we can guarantee. After a pause we will have introduced a skew proportional to the size of any output buffers (audio output devices always have buffers). For seeking there is no guarantee that the resolution of positioning data will be the same for all tracks. The timeline resolution of video will typically be several seconds, and audio hundreds of milliseconds. Therefore after a seek the tracks will most likely be out of sync.

This problem can be resolved by making two changes:

1. When unpausing or seeking, decide a new position for all streams and distribute this to the flow controls, which then discard data before the new position.

2. Synchronise the outputs, something so far avoided.

4.3 Output Synchronisation

In order to synchronise the output processes, they are broken down into three parts, as per Figure 5.

The embedded output device process acts in a pass-through manner. The device manager monitors the position of the output device by the timecodes of its output stream and delays frames appropriately using the clock process. The clock process converts the KRoc environment’s microsecond timers to nanosecond timecodes. Given the KRoc environment’s sub-microsecond communication times, reading the time via requests to separate processes should not lead to any significant inaccuracies, although it could be inefficient when used on a large scale.

The device manager starts in an unsynchronised state, and requests that the clock synchronise when it receives a timecoded message, providing the timecode as the synchronisation point. On receipt of a purge message, the device manager resets the clock and returns to an unsynchronised state. Synchronising all the outputs of the network is now done by synchronising their respective clocks (Figure 7).

Whenever a clock receives a synchronise or reset request, it forwards this to the “Clock Sync” process. The clock sync process in turn initiates a synchronisation cycle (if one is not already running), which resets all other connected clocks. A clock which has been reset interrupts any pending alarms and refuses to accept new alarm requests until it has been resynchronised.

Once all clocks are attempting to synchronise and have presented the clock sync process with the desired timecode, the sync process picks the earliest timecode and associates it with a point in KRoc environment time. This association is the synchronisation
point and is returned to all the clocks. All clocks thus acquire the same mapping of KRoC timer offset to timecode. This process is very much like a barrier. In practice the sync process returns a synchronisation point slightly earlier than that requested, allowing some propagation time.

It is foreseeable that a synchronisation mechanism similar to this could be extended to work across multiple distributed hosts, allowing the synchronisation of multiple distributed output devices - a topic for future research.

4.4 Pull Model

The design ideas so far presented only employ the P.MM protocol. While these designs do in practice work, they are overly complex; as a side-effect of the input process driving the network, changes to the flow must be applied in two places. It makes more sense to have the process receiving user requests drive the network and hence be able to respond directly to user requests. For this we can use the P.MM.CTL/SEEKABLE protocols.

Figure 8 shows the operating process network for the complete occam-π video player. The “Play Control” process requests data from the inputs via the decoding pipelines, and passes it to the outputs, which are synchronised as previously described. The channel between the play control and the clock sync process is used to inform the clock sync process how many processes should be synchronised (some tracks may have come to an end and will not need synchronising). The seekable decoders are simply decoder processes inside seekable wrappers as explained in section 3.4. A flow path exists from each output back to the play control process to allow user commands input via the device, for example from an X11 window, to control playback.

An advantage of this design, lying in the fact that input tracks are considered separate streams and only share the common factor of the play control process, is that tracks need not come from the same source. This means that audio from one file could be combined with video from another file without a complex backend synchronising the input processes. A “multi-play” mode in the occam-π video player uses this feature to play the video from any number of files simultaneously, synchronised in the same way as a pair of audio and video tracks would be.

Another advantage of the pull model is that by adding filters which intercept requests and distribute them over input sources, many separate input files could be arbitrarily combined into a single track (Figure 6). This idea has not yet been implemented.

5 Dynamism and Reconnectivity - CT.IO.CTL

The following protocols provide a means to interrogate a component process about its connectivity as an alternative to compile-time passing of channels as parameters. Along with interrogating the process, channel ends can be “plugged” and “unplugged” to create connections between processes at runtime. To facilitate this a P.MM channel is placed in a CT.MM mobile record, and a pair of P.MM.CTL and P.MM.SEEKABLE channels are placed in a CT.MM.SEEKABLE channel type [13]. This method of building channel ends is an artifact of the occam-π type system; see section 11 for some further discussion of this.

This interactive interrogation is the primary method of dealing with file input processes. Until a file has been opened and its header parsed, it is not known what tracks it provides and their details. Hence it makes sense to select the tracks to be used at runtime. While it would be possible to provide file input processes with fixed sets of channel parameters which are satisfied at runtime (these existed in earlier versions of the framework), it makes more sense to implement these on top of this more generic architecture.

Processes implementing this interface are expected to operate in two modes, “started” and “stopped”. A stopped process can have its connectivity interrogated and changed, whereas a started process can only be queried about its mode or stopped. It is expected that processes initially start in the stopped mode, as their connectivity is undefined.
5.1 P.IO.CTL.RQ

P.IO.CTL.RQ is the request component of the protocol.

```
PROTOCOL P.IO.CTL.RQ
CASE
  inputs
  outputs
  start
  stop
  status
  end
plug.mm.i; INT64; CT.MM?
plug.mm.o; INT64; CT.MM!
plug.mms.i; INT64; CT.MM.SEEKABLE!
plug.mms.o; INT64; CT.MM.SEEKABLE?
unplug.i; INT64
unplug.o; INT64
```

**inputs and outputs** are used, respectively, to request details of the inputs and outputs of the process.

**start and stop** are used to change the mode of the process.

**status** queries the present mode of the process.

**end** closes down the IO.CTL interface; the process will terminate when all of its connected streams terminate.

**plug.* and unplug.* messages** are used to plug and unplug channel ends, with the INT64 specifying the track number.

5.2 P.IO.CTL.RE

P.IO.CTL.RE is the response component.

```
-- MOBILE keywords omitted for brevity
PROTOCOL P.IO.CTL.RE
CASE
  inputs; []; INT; []; BOOL; []; TRACK.INFO
  outputs; []; INT; []; BOOL; []; TRACK.INFO
  plugged
  started
  stopped
  error; INT
unplugged.mm.i; INT; CT.MM?
unplugged.mm.o; INT; CT.MM!
unplugged.mms.i; INT; CT.MM.SEEKABLE!
unplugged.mms.o; INT; CT.MM.SEEKABLE?
```

**inputs and outputs** return information describing the input and output connectivity of the process.

[] INT is an array of flags detailing the types of connections (stream or seekable), and whether a given connection must be plugged before the process can be started. For a process which filters data rather than originates it, the inputs and outputs will need to be plugged before the process can be started.

[] BOOL is an array of flags indicating whether a connection is plugged or unplugged. This is separate from the other flags as it changes during operation, where as the others typically do not.

[] TRACK.INFO is an array of the details of each connection. For inputs this will typically be a partial specification. An exact specification of what inputs the process will handle is not usually known in advance as most processes work over a range of data. Quite how to perform this partial specification is not truly solved; the framework currently leaves unspecified values as zero.
The dvplug application (Figure 9) is a demonstration of digital video input, an encoding process, and the reconnectivity components of the framework. It responds both to user input and external events and demonstrates a possible real world application for the reconnectivity within the framework.

### 6 dvplug - Digital Video Hot-plugging

The dvplug application (Figure 9) is a demonstration of digital video input, an encoding process, and the reconnectivity components of the framework. It responds both to user input and external events and demonstrates a possible real world application for the reconnectivity within the framework.

On start-up a clocked test.card source is wired through the control process (record.ctl) to the output. A dv1394.plug.detector process monitors the IEEE1394 [21] bus of the host and reports the presence of new devices to the player.spawner.

On connection of a DV [22] device (such as a video camera), the player.spawner forks a set of processes to take input from the new device, decode it and remove the audio stream. Using the CT.IO.CTL protocol, it stops the pluggable.stream.input.end, disconnects the test.card source from it and plugs in the new device's network. It then restarts the pluggable.stream.input.end and stores the test.card source channel end for later use.

While the DV input network is connected, the player.spawner ignores any messages about new devices. Using a status.tap (see section 7), it detects when the stream from the DV input network terminates (pluggable.stream.input.end does not propagate end messages). When such termination occurs the player.spawner disconnects the channel end for the terminated DV input network and reconnects the test.card source. Then after restarting the network it returns to monitoring events from the dv1394.plug.detector.

In parallel to this, the record.ctl process transfers data from the pluggable.stream.input.end to the output and responds to record commands from the user. On receipt of a record command, it forks a recording network, and starts copying any received messages to it in addition to output. When the user requests the end of recording, the recording network is sent an end message, and the channel end to it discarded.

### 7 Status Reporting Backend

Every distinct component developed for the framework takes a SHARED CT.STATUS! channel end as a parameter. This provides a means for processes to output debugging information, errors and other events in a safe and uniform manner. The channel end is manipulated using a set of “status.” prefixed processes (see appendix C).

The status backend is implemented as a automatically growing n-way tree. Node creation requests are passed to the root which forks off the new status.node processes before passing connections to them back up the tree where they are “wired in”. As messages pass down the tree to the root, they pick up the tags of all the nodes they pass through. By feeding the output of the root to a terminal it is easy to monitor the execution of the application network (particularly if status.debug calls are well placed). An example network instantiation is shown in Figure 10.

Another method for implementing a similar backend would be to use a flat structure with a root process which grows and shrinks an array of channel ends as nodes are created and destroyed. However such an approach would be inefficient as a result of resizing and ALTing over the root array. It would also not be able to tag messages in a tree fashion as the presented implementation does, and would complicate process-to-process monitoring.
Figure 9: Process network for the dvplug application.

Figure 10: Example instantiation of the status backend.
Other than user monitoring of the application, the status network can be used by processes within the network to monitor each other. This is done by inserting status.tap processes which copy, to an extra channel, a given subset of the messages passing through them. This mechanism is used in the dvplug application (see section 6) to detect termination, via an EOS event, of DV input networks.

At present the tags within the backend are the strings passed to status.startup; this can lead to name collisions in large networks. However as status.node forking is done at the root, there should be no problem allocating each node a unique number as it is forked. This could be used in place of or in addition to the textual name - an area for improvement in future versions of the framework.

8 Processes Developed for the Framework

This section gives a summary of the processes that have been developed for the framework during this research and highlights particular features of interest they have. A significant number of components contain C code to wrap and facilitate the use of external libraries. This non-occam code is interfaced to using the mechanisms described in [23] and [24]. It is foreseeable that external libraries. This non-occam code is interfaced to using the mechanisms described in [23] and [24]. It is foreseeable that most of this could be replaced with SWIG wrappers [25], generated as described in [26].

8.1 Input

avi.input
An input process for Microsoft’s Audio Video Interleave [27] format, one of the most commonly-used container formats for multimedia data. It consists of a large C component with an occam-π wrapper, and is connected via a CT.IO.Ctl interface with CT.MMSEEKABLE tracks.

mkv.input
An input process for the Matroska [16] container format. Much like avi.input it consists of a large C component with an occam-π wrapper and is connected via a CT.IO.Ctl interface with CT.MMSEEKABLE tracks.

input.file
Attempts to start avi.input and mkv.input on a given file, and uses whichever can parse it. This avoids having to directly specify a demuxer for a file, or infer one from its extension. If neither process can parse the file, then a null.input process is invoked, which behaves like a file with no tracks. This could be extended to support more formats in the future.

dv1394.input
A P.MM input process for DV [22] (Digital Video) over IEEE1394 [21], which allows input from digital video cameras and tape decks. It outputs packets containing the DV packets from the IEEE1394 bus. It consists of a C wrapper around open source IEEE1394 libraries, and a occam-π wrapper around the C wrapper. The occam-π code performs lossy buffering, as DV transmissions are isochronous and should be read regardless of whether the output is blocked. If a device address (GUID) is specified this process will terminate on removal of that device.

dv1394.plug.detector
Wrapping the same libraries as dv1394.input, this detects the connection and disconnection of DV capable devices to the IEEE1394 ports of the host system. It reports the port number and address of new or removed devices. This process has no termination mechanism at present.

tcp.input/tcp.output
Act as a server and client pair allowing a P.MM channel to be carried over a TCP socket. These are written entirely in occam-π and use the KRoC socket library. Although serialisation of the P.MM protocol is very simple, this explicit mechanism of network channels is likely to be superseded by pony (formerly KRoC.net [28]) or similar.

test.card
Produces a stream of video frames based on a TRACK.INFO structure given. For YV12, I420 and YUY2 colour spaces the frames will contain a test pattern and text containing a frame number; other colour spaces will simply be blank. This source can run in both a clocked mode, where data is produced on a regular clock (derived from track[default.duration]), or unclocked. Written in occam-π, it calls draw.string functions which are presently wrappers to C code.

8.2 Decoding

All the decoders and encoders run CPU intensive library calls in blocking mode. In the KRoC environment this allows the network to run in parallel with the library calls, distributing load over available CPUs.

dv.decoder
Decodes a P.MM stream of DV packets using a C wrapper around the Quasar DV CODEC [29] and produces a “complex” output stream of both audio and video frames (DV packets code both together).

ffmpeg.decoder
Wraps the ffmpeg [30] library, which contains code for decoding most known video and audio compression formats. While there is no code to utilise the audio components of ffmpeg, there are stubs to add it. Output is in the form of the native colour space of the video CODEC used, which is typically I420.

mad.decoder
Decodes a P.MM stream of packet MPEG audio using MAD [31]. The MAD library supports decoding of channels separately with up to 24 bits of precision, however output is presently only in the form of interleaved 16 bit data.
This process rewrites timecodes, as there is not a one-to-one mapping between packets and audio frames.

**xvid.decoder**
Decodes a P.MM stream of packet MPEG4 [32] video, using a C wrapper around the XviD library [33]. In practice this provides the same functionality as the ffmpeg.decoder, but supports fewer input types. Output is always in the YV12 colour space.

**decoder**
Uses the init message of a P.MM stream to invoke one of dv.decoder, ffmpeg.decoder or mad.decoder depending on the format of the track. This should be used instead of directly specifying a decoder.

**seekable.decoder**
Employs the seekable.wrapper (see section 3.4) to allow the decoder process (and the processes it invokes) to operate over P.MM.CTL/SEEKABLE channels.

### 8.3 Encoding

**xvid.encoder**
Encodes a P.MM stream of video frames into MPEG4 video packets using the Xvid library [33]. A range of quality and bitrate parameters are available.

### 8.4 Output

It is important to note that the alsa, libao and x11 processes act as filters, not sinks. They pass on frames once they have been output to the underlying hardware. This design allows outputs to be chained and is taken advantage of by the device.manager to calculate delay characteristics. This “chaining” technique is commonly used by broadcast video hardware.

**alsa.output**
This process provides sound output using the ALSA sound system [34], and is preferred to libao.output as the amount of buffering can be explicitly controlled. The control messages flush and purge have the expected effect on the audio device, allowing the network to clear the hardware buffers. Output is clocked as the output device consumes it.

**libao.output**
Wrapping the libao library [35] to provide audio output, this process has much higher compatibility with different audio systems than alsa.output, but lacks any control of buffering. Hence it may buffer several seconds worth of audio data and flush and purge control messages do not have predictable results. Output is clocked as the output device consumes it.

**x11.output**
Uses moderately-sized C and occam-π components to display video on the X11 window system. Unlike the existing XRaster and SDLRaster functionality provided by the KRoC distribution, this process employs the Xv (XVideo) extension where available to perform hardware accelerated colour space conversion and scaling. In the event that the Xv extension is not available, the video data is internal converted to a suitable colour space for the host system. Hence this process supports output in colour spaces other than the RGB colour space provided by XRaster and SDL-Raster. An additional internal process captures input events from the X11 window and converts these to message on a P.USER.CMD channel. No output clocking is performed.

**device.manager**
Employs a user supplied pair of clock channels to synchronise and clock data passing through an output device process. For video, the output delay is calculated and used to pre-roll the data.

**output**
Much like the decoder process, this process uses the init message of a P.MM channel to invoke an appropriate output device and attach a device.manager process to it.

**mkv.writer**
Writes out a single P.MM stream to a file in a very simple form of the Matroska [16] container format. Written entirely in occam-π, it internally uses an automatically expanding chain of processes for encapsulating elements in the EBML (binary XML) format Matroska is based on. It should not require much work to extend this process to handle an arbitrary number of inputs and be pluggable via the CT.IO.CTL interface. See Figure 11 for a process network overview.

### 8.5 Filters

**complex.to.av**
Filters a complex stream of audio and video frames (as produced by dv.decoder) into separate channels of P.MM, one for audio and one for video.

**frame.drop**
Drops packets and frames in accordance with a given ratio. This is useful for handling DV data when the host CPU is not fast enough to decode it at full rate.

**sanitise.timecodes**
Realigns the timecodes in a P.MM stream so that they begin at zero. This is useful when recording DV streams, which are timecoded based on the host clock and do not begin at zero.

**smpte.timestamp**
Adds an SMPTE timecode display [17, 18] to the top-left of video frames in the YV12, I420 and YUY2 colour spaces. This is used to provide an on-screen display (OSD) for the occam-π Video Player (see section 4).

### 9 Conclusions

Through this research it has been shown that the occam-π language is suitable for developing systems for handling and (to a
lesser extent) manipulating video data. The design process for
the demonstration applications, in particular the occam-π video
player, has been simplified significantly by occam-π language
constructs. On top of this, CSP and other concepts underpinning
the occam-π language have made the systems developed easy
to visualise and reason about despite their high degree of par-
allelism. While the reconnectivity model presented still requires
more research, it has been effective in harnessing occam-π’s mo-
bile channels for building dynamic systems.

Preliminary profiling using a kernel embedded profiler [36] sug-
uggests that there is no significant overhead from using the present
framework. Framework code (“ovp” in Figure 12) accounts for
only 5% of execution time of the occam-π Video Player, with
the remainder spent in external libraries decoding (libavcodec)
and copying data (libc). Profiling a popular open source video
player [37], MPlayer [3] (Figure 13), the number of CPU cy-
cles used by the framework is directly comparable. Although
as MPlayer uses the same buffers end-to-end it avoids expenses
expensive copy operations which account for 15% of the overall execution
time of ovp. It should however be noted that this copying over-
head only applies when a display output is used and could be
overcome by providing a buffer loop.

Overall, these results suggest that the occam-π language is po-
tentially very efficient and effective for this type of application
when its unique features such as mobile data types are exploited
correctly.

10 Future Work

This initial work into using occam-π for video processing opens
up many avenues for future research.

As the worth of any particular frameworks lies in the functional-
ity it provides, one clear expansion of this work would be to de-
velop more software components for the framework as it stands.

The present framework is lacking any significant filtering compo-
nents, hence adding simple video operations (crop, scale, merge,
split) would be a logical next step. Following this it would be
interesting to explore writing noise reduction filters, spatial and
temporal, using the occam-π language, as these are the most
common filtering operations performed on a video stream.

Given that a real-world application of the framework might be
video surveillance, it would be useful to provide processes for
monitoring sets of video streams and reporting changes. Pro-
cesses of this kind could also foreseeably be used in the control
of robots and other embedded devices.

Extending the input and output capabilities of the framework is
also an interesting area, in particular replacing the C compo-
nents of the avi.input and mkv.input processes with occam-
π. These components were largely written in C as the language is
more suited to handling complex data structures than the present
occam-π, but this does not mean that occam-π implementations
are impractical. In all likelihood a serious exploration into the
representation of complex data structures in occam-π could lead
to important language developments.

One major area left largely unsolved in the present framework
design is safe reconnectivity. Reconnections need to be made
both sane (video colour space consistent, etc) and deadlock-free.
Although sanity is not guaranteed for static networks, the issue
is mitigated by the runtime typing of components. This however
breaks down when a components input can potentially change
type mid-stream (as a result of a reconnection).

The potential for deadlock in the reconnectivity model as pre-
sented manifests if processes are not stopped in the correct order
(in the direction of data flow). This is not an issue in small net-
works where a single process is controlling the reconnection, but
in a large system it is foreseeable that many different processes
may be modifying the network in parallel, and this is where the
potential for deadlock lies.

It would be preferable that any solutions to these problems
be simple, as to minimise developer workload. It is my ex-
pectation that, as P.MM processes can be wrapped to provide
P.MM.CTL/SEEKABLE functionality, reconnectivity will best be
implemented in a form of wrapper, concealing the inherent com-
plexity from component developers. It would also be my desire

12
Another major unsolved area is error handling. The framework at present attempts to conceal and suppress errors: an error will typically cause termination of a filter or conversion of it to a null filter which does nothing. To be most useful, the framework should provide a means for replacing a failed component at runtime. One possibility is an exception-like model, where the failed process throws an exception providing its present state (channel ends, track information, etc). A control process catches this exception, makes any changes required, then invokes a replacement using the state information.

11 Thoughts on the ocaml-π language

During the process of this research I’ve had a number of thoughts on possible ways in which the ocaml-π language and its implementation might be improved. This section outlines two of these ideas and how they could be used to improve the framework presented.

11.1 Inherently Mobile Channel Ends

The ocaml-π language may benefit from a different syntax and model for mobile channel ends in order to enhance its flexibility. In the present model mobile channel ends must explicitly be created via a mobile record channel type definition. This explicit distinction between non-mobile and mobile ends means that the \texttt{rxyb?}, \texttt{scr!} and \texttt{err!} channels of the initial process cannot migrate around a network, or be handed to forked processes.

One foreseeable solution is to consider all channels mobile. The definition of a channel is seen as the definition of two channel end variables, which are automatically allocated and associated. Referencing the channel without a direction specifier references its present state (channel ends, track information, etc). A control process catches this exception, makes any changes required, then invokes a replacement using the state information.

The assignment statement demonstrates the advantages of this method. It means we can place pairs of unconnected channels in the same channel variable, something which at first seems confusing but is in fact very useful. However, this fundamental change breaks the strict aliasing rules around which ocaml is designed and hence would seem unwise. Instead we might have separate channel and channel end variables.
A channel variable holds two ends which will always be connected, and can only be assigned to and from other channel variables, but can be split into channel ends as required. Channel end variables can be assigned from other channel end variables of the same directionality.

```plaintext
-- Channel variables
CHAN P.STUFF a, b:
-- Channel end variables
CHAN P.STUFF to.server!, from.client?:
SEQ
  -- a = DEFINED(a), b = DEFINED(b)
  -- to.server! = UNDEFINED, from.client? = UNDEFINED
  a := b
  -- a = DEFINED(b), b = UNDEFINED
  -- to.server! = UNDEFINED, from.client? = UNDEFINED
  to.server!, from.client? := a
  -- a = UNDEFINED, b = UNDEFINED
  -- to.server! = DEFINED(b!), from.client? = DEFINED(b?)
```

Since we are effectively treating channels and channel ends as variables which can be assigned (in a safe manner via the mobile model), it should be easy to embed them in data records. Rather than have explicit channel types, we could instead use data records. Any data record type containing channels would gain two sub-types accessed via directionality specifiers like the present channel types, however it could also hold data. A language construct could be provided for “wiring” two such records together, or it may be performed at allocation time. Another approach would be to provide a construct for splitting records into send and receive “ends”, making the associated data immutable, and allowing the ends to be communicated independently.

Ultimately it is the bundling of data with channel ends which would be most useful in extending the framework outlined in this report. Channel bundles could hold the track initialisation and typing data, allowing them to be tested for compatibility directly rather than through channel communications (which might block).

If channel ends were to become implicitly mobile, it could possibly be argued that all data become implicitly mobile, and channel ends simply become variables within such a completely mobile system. Although the implications of such a system have yet to be explored, additional operators or increased compiler analysis would likely be required in order to determine if data should be copied or moved.

### 11.2 Extended Output Rendezvous

There are a number of cases where it would be beneficial to be able to write data to a channel only when a reader is present. A good example of this is user implemented timers, where we do not want stale values to be sent to the reader.

This is what we could write:

```plaintext
PROC user.timer(CHAN INT out!)
  WHILE TRUE
    INT val:
    SEQ
      read.some.clock(val)
      out ! val
  :
```

In practice we could use an ALT output guard if such a construct existed; however, I propose instead we write something like this:

```plaintext
PROC user.timer(CHAN INT out!)
  WHILE TRUE
    INT val:
    SEQ
      read.some.clock(val)
      OUTPUT val :
```

We can already mimic this behaviour:

```plaintext
PROC user.timer(BARRIER b, CHAN INT out!)
  WHILE TRUE
    INT val:
    SEQ
      SYNC b
      read.some.clock(val)
      out ! val :
```

This construct is different from an ALT output guard in that it commits the sender to send; however, it does not commit what they will send. It might foreseeably be used to build output guards through auxiliary processes.

```plaintext
PROC out.guard(CHAN BOOL ready!, CHAN DATA in?, out!)
  ...
  OUT !
  ----
  SEQ
    ready ! TRUE
    in ? data
    OUTPUT data
  :
```

This functionality could be used to improve the synchronised timers in the framework, and perhaps solve some of the problems of reconnectivity (see sections 5 and 10).

### References


Glossary

Where appropriate definitions are derived from Wiley [40].

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODEC</td>
<td>COder / DECoder pair</td>
</tr>
<tr>
<td>colour space</td>
<td>Method of representing colour images</td>
</tr>
<tr>
<td>chrominance</td>
<td>Colour difference component</td>
</tr>
<tr>
<td>field</td>
<td>Odd or even numbered lines of an interlaced video sequence</td>
</tr>
<tr>
<td>HDTV</td>
<td>High-Definition Television</td>
</tr>
<tr>
<td>interlaced</td>
<td>Video data represented as a series of alternating fields</td>
</tr>
<tr>
<td>inter-frame</td>
<td>A video frame coded using temporal prediction or compensation</td>
</tr>
<tr>
<td>intra-frame</td>
<td>A video frame coded without temporal prediction</td>
</tr>
<tr>
<td>IPTV</td>
<td>Internet Protocol Television</td>
</tr>
<tr>
<td>key frame</td>
<td>Intra-frame on which subsequent or preceding inter-frames depend</td>
</tr>
<tr>
<td>MPEG</td>
<td>Motion Picture Experts Group</td>
</tr>
<tr>
<td>NTSC</td>
<td>National Television System Committee, used in reference to the analog</td>
</tr>
<tr>
<td>PAL</td>
<td>Phase-Alternating Line, used in reference to the the analog television</td>
</tr>
<tr>
<td>planar format</td>
<td>A colour space where frames are divided into planes</td>
</tr>
<tr>
<td>plane</td>
<td>A slice of chrominance or colour information</td>
</tr>
<tr>
<td>RGB</td>
<td>Red/Green/Blue colour space</td>
</tr>
<tr>
<td>YUV</td>
<td>Luminance, Blue chrominance, Red chrominance colour space</td>
</tr>
</tbody>
</table>

A P.MM

Case

init; MOBILE TRACK.INFO; MOBILE []BYTE

packet; MOBILE TID; MOBILE PKT.HDR; MOBILE []R.DESC; MOBILE []BYTE

video; MOBILE TID; MOBILE VF.DESC; MOBILE []BYTE

audio; MOBILE TID; MOBILE AF.DESC; MOBILE []BYTE

flush

purge

skip

end

init signals the start of a stream.

TRACK.INFO describing the stream’s properties, such as type, width, height (see section 2.4 for details).

[]BYTE carries a variable quantity of untyped setup data. This typically contains a binary structure used by a specific decoder. Ideally this data would reside in the TRACK.INFO structure; however, nested mobiles are not capable of this at the time of writing, and a fixed length field would be unsuitable as the initialisation data could be several hundred kilobytes (Ogg Vorbis [41]).

packet carries untyped media frames, typically compressed video or audio.

TID carries the timing of the frames (see section 2.2 for details).

PKT.HDR holds flags indicating whether the data is a key frame and whether it is the logical progression of data already presented or not. Deviation from logical progression will occur when an input file is repositioned (“seeked”) or if data is lost. In a temporally coded video stream [40], frames are typically dependent on the data preceding them, hence deviation from logical order must be explicitly handled if decoding errors are to be avoided.

[]R.DESC is a variable size array of Range DESCriptor structures, indicating the location of frames in the following []BYTE array. The order of the descriptors is the logical order of the frames they describe. There will typically be one for video and many for audio. Multiple frames are common for audio as audio frames are usually “laced” together in container formats to avoid wastage due to their small size in comparison to metadata overheads. As an artifact of this process, there will only be one piece of timing data per-group of laced frames, hence rather than attempt to generate per-frame timing data in the demuxer, it is best left to the decoder where the true timing characteristics of the data are known.
Figure 14: Overview of planes, pitches, offsets and preambles as stored in a VF.DESC data structure.

[]BYTE is the data for the frames. By using range descriptors, the actual frame data can be dispersed and non-consecutive; this is desirable for two reasons:

1. It increases efficiency by allowing an input process to load an entire logical stream block as a packet. The stream block is simply parsed to create the range descriptors before being sent out from the input process.

2. It allows padding at the end of frames which is required for many CODECs that operate on word sized chunks rather than bytes. All file demuxers developed for the framework pad this field with a pre-defined quantity of null bytes.

video carries a single video frame.

TID carries the timing of the frame (see section 2.2 for details).

VF.DESC is a Video Frame DESCriptor structure, which describes where in the following []BYTE array to find the video pixels and the respective line widths (“pitches”) of each plane. Preambles are also present, indicating where video data begins within a line. These values and offsets are measured in bytes, rather than pixels, allowing manipulation of frame data without knowledge of the pixel structure. Non-planar formats will use only the first element of each array, where as planar forms will use as many as there are planes. Figure 14 illustrates the use of each field in the data structure.

DATA TYPE VF.DESC -- Video Frame DESCriptor
PACKED RECORD
[4]INT offsets:
[4]INT pitches:
[4]INT preambles:
:

[]BYTE contains the untyped video frame pixel data and padding.

audio carries an audio frame of variable length.

TID carries the timing of the frame (see section 2.2 for details).

AF.DESC is a Audio Frame DESCriptor structure, which contains a single range descriptor indicating where in the []BYTE the samples for this frame reside.

[]BYTE contains the untyped audio samples for this frame and padding. Due to the relative length of an average audio sample (44.1kHz ≈ 0.02ms) in comparison to a video frame (25fps ≈ 40ms), it is inefficient to pass around single samples (the smallest atomic type) like it is video frames [42]. Hence an audio frame carries zero or more audio samples. This approach does not create any loss of timing information as it would do for video, as the relative presentation time of a given sample can be calculated from the TID timecode and the sampling frequency of the track. If variable frequency audio [20] support were to be required then all samples within a frame would need to be of the same frequency, and the AF.DESC extended to carry the frequency of the frame.

flush instructs the receiver to output all ready buffers, then forward the flush command. This is necessary for the ideas presented in sections 3.4 and 4.
purge instructs the receiver to clear its internal state without generating any output, and prepare for new data. The purge command is forwarded when the receiver is ready for new input. Like flush, this is used in sections 3.4 and 4.

skip instructs the receiver to do nothing. Unlike flush and purge it is not forwarded. This command is used to build zero-place buffers (see section 3.5).

end indicates the end of the stream; no data will follow. This provides a form of graceful termination [15]. On receipt the receiver should terminate after outputting any ready buffers (like a flush) and propagating the end message, unless it is restartable (such as pluggable.stream.input.end in section 6).

B TRACK.INFO

DATA TYPE TRACK.INFO
PACKED RECORD
    INT64 number:
    INT64 uid:
    INT type:

    [32]BYTE codec.id:

    REAL64 duration:
    INT default.duration:

    INT interlaced:
    INT csp:
    INT planes:
    INT width:
    INT height:

    INT fmt:
    REAL64 freq:
    REAL64 output.freq:
    INT channels:
    INT bit.depth:

number and uid uniquely identify a track within a given set of tracks. For example, an input process may provide many tracks each identified by their respective track number, rather than their parse order or array index. It should have been noted that the uid field is not actively used in any elements of the framework as presented.

type is a constant TRACK.VIDEO or TRACK.AUDIO, or their union TRACK.COMPLEX (a track of both audio and video).

codec.id stores a null-padded string identifying the CODEC used to encode or required to decode the track (if it is compressed). For example “V_MPEG2” indicates MPEG2 [43] compressed video. This is the same convention used by Matroska.

duration is the duration of the track in seconds, if known. It is a real number allowing fractional seconds. It should be negative if the duration is unknown.

default.duration is the duration between frames in nanoseconds. For CFR tracks the duration of all frames will be equal to this value; for VFR this will be the average.

interlaced a flag indicating if the video is interlaced.

csp and planes indicate the colour space and number of planes of a video track, where the colour space is a pixel format constant. A typical example of values might be VIDEO.CSP.YV12 and 3.

width and height indicate the width and height of the video image in pixels. No separate handling of aspect ratio data exists at present.

fmt is an audio format constant, e.g. AUDIO.FMT.U16.LE. These constants encode a sample type, resolution in bits, and endianness.

freq is the sampling frequency of an audio track in Hertz.

output.freq is used by SBR (Spectral Band Replication [44]) CODECs to replicate the high frequency information of an audio stream.
channels indicates the number of audio channels, e.g. two for stereo. (There is no reason that separate audio channels could not be separate tracks, although such a technique is not used anywhere in the framework at present.)

bit.depth is a legacy Matroska element and has been superseded by data in fmt.

C Status Processes

status.started(RESULT SHARED CT.STATUS! status, VAL []BYTE name)
Indicates that a module called name has started. Internally this requests that the status network start a new tag with the given name. The new tag is a new channel end, which replaces the existing one (via a RESULT keyword, which is not shown). The new channel end points to a status.node which stores the tag name. Since all events will now pass through the new status.node they will pick up the tag it stores. A STATUS.STARTED event is then generated via status.event, with the data payload being a timestamp.

status.shutdown(RESULT SHARED CT.STATUS! status)
This sends a STATUS.SHUTDOWN event, then reverses the process of status.started, by requesting the channel end for the network upstream of the connected status.node. It informs the status.node of this, which will shut down if it has no other children.

status.error(RESULT SHARED CT.STATUS! status, VAL BOOL fatal, VAL []BYTE msg)
Indicates an error has occurred, taking a message detailing the error and a flag indicating whether the error is fatal or not.

status.debug(RESULT SHARED CT.STATUS! status, VAL INT level, VAL []BYTE msg)
Sends a debug message at the given level, where higher levels increase visibility.

status.event(RESULT SHARED CT.STATUS! status, VAL INT type, data)
Sends an arbitrary event: RUNNING, STOPPED, INPUTS.CHANGED, OUTPUTS.CHANGED or EOS (End Of Stream). The data parameter is an INT64 which is typically used as a timestamp for the event, or the track number in the case of EOS.