Design Overview of MPEG Decoder Architecture

Todd Brunhoff
North Valley Research Inc.

Version 1.0
Tuesday, January 16, 1996

Abstract

The new decoder architecture currently under development at NVR is designed to solve several problems encountered in the previous decoder implementation. The latter grew out of research at Tektronix Laboratories and licensed by NVR at its beginning in 1992. By 1995, it had become very clear that redesigning the decoder architecture from the ground up could solve many intractable problems with the older architecture, including higher quality through validation, higher performance through data-flow analysis and broader adaptability through the use of "object" interfaces. This paper describes the new architecture in detail.

Problem Statement

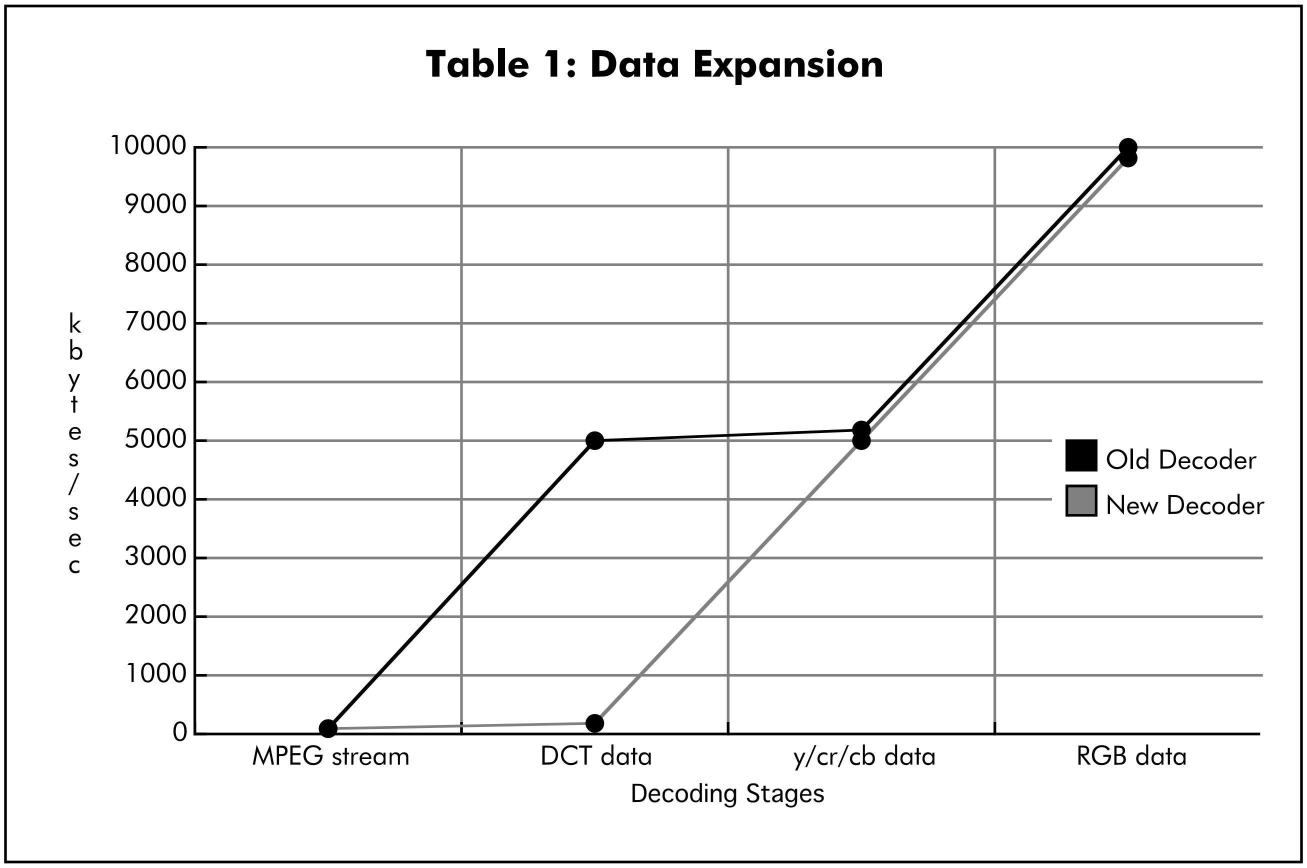
Several problems exist with the old decoder architecture (called ODA in this paper) that cannot be fixed. In particular, performance, adaptability, testability and maintainability. While the old decoder is a good one, it cannot be made better in these areas without significant changes in the basic design. The new decoder (called NDA in this paper) is an attempt to fix all of these problems; most are detailed below.

Performance

In ODA, the performance bottleneck was seen to be the performance of individual instructions such as the penalty paid with write-through memory caches in the SparcStation 2 that cause a single instruction to stall when a value is written to memory. This caused the evolution of the code to move toward code that packed values into 32 or 64 bit values before writing it out to memory (see vmpegdecomp/copyblock.c, lines 37-51). Little attention was paid to locality of reference for data and text.

Most new computer architectures employ a write-back cache that does not stall the processor on a memory write, so the corresponding code no longer needs such concessions and in fact, may cause slower performance. (see dct/idctMotion.c, lines 402-423).

An incomming MPEG data stream is about 150 kbyte/sec. As it is decompressed, it expands in data rate to about 5 mbytes/sec in the y/cr/cb form and may continue to expand out to an additional 12 mbytes/sec for 32-bit rgb data. The ODA did not consider this in the least. Note that DCT coefficients are expanded to a full 32 immidiately after decoding the coefficient from the MPEG stream (see vmpegdecomp/block.c in the ReconstructIntraCoeff() macro call on line line 214). This is illustrated in table 1.



The NDA tries to improve this in a couple of ways:

- delay data expansion as long as possible. The DCT coefficients are expanded during block-decoding only to 16-bits in a runlength encoded list (see the ComposeDctCoeff() macro call in mpeg1vdec/mpeg1vslice.c on line 575).
- all coefficients and motion vectors are decoded into run-length arrays (see the DctMbFlags, DctBlock and DctCoeff structures defined in dct/nvrIdct.h). This keeps slice-decoding instructions in the cpu cache while decoding an entire slice (this is mpeg1vdec/mpeg1vslice.c). Then the DCT and motion compensation instructions are kept in the cpu cache when the data is converted from a run-length data array to the 5mb/sec y/cr/cb data (this in in library/dct).

The ODA tried to realize performance by using constants for the width and height of a video picture (see vmpegdecomp/decoder.h, lines 33-35). This caused many problems because some streams could not be decoded. The NDA uses a general picture description that is not limited in size (see the YcImage structure defined in imgdpy/imageDpy.h).

Finally, scheduling in the ODA was done for every picture (see vmpegdecomp/display.c, lines 694-819). This is unnecessary because the system load will normally not change every 30th of a second. Instead, scheduling in the NDA is done every half second or so: a simple weight is derived and given to the video decoder to decide which frames are decoded and which are skipped (see the

BalanceSchedule() function in mpegsched/mpegsched.c; see mpeg1vdec/mpeg1vdecode.c lines 386-412).

Testability and Adaptability

An important flaw in the ODA is the large, monolithic design. All aspects of the decoding process are all handled by a single code base: control, display, MPEG syntax, inverse DCT, motion compensation, audio handling. This means that the code has a natural tendency to become intertwined and fragile. One change in any part may cause the entire decoder to change.

The NDA addresses this by separating the important parts of the decoding process into separate subsystems with a well defined interface. This means that testing modules can be fitted for each subsystem; development on one subsystem can proceed independent of another as long as semantics do not change; the subsystems can be combined in different ways to make new products.

Overall Design

The NDA comprises many subsystems, some independent, some interdependent. When choosing how to divide the subsystems, lines were drawn around major functionality, such as scheduling and MPEG syntax, as well as around performance bottlenecks such as inverse DCT computation and file I/O. Figure 1 illustrates the grouping of the various subsystems in a typical application.

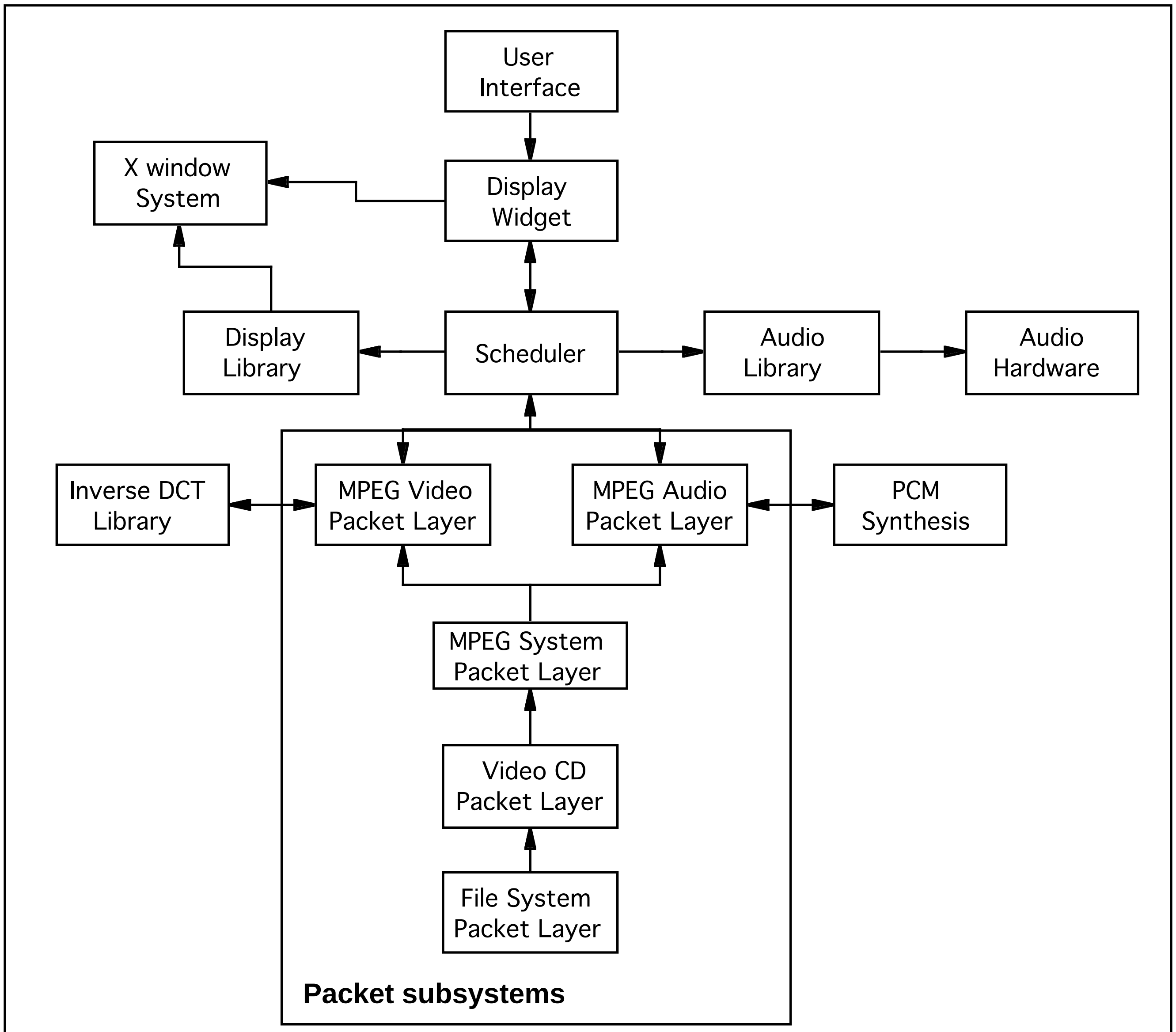


Figure 1: Typical Application Structure

Below is a brief description of each subsystem, how it interacts with other subsystems, and where to find its source code.

Packet Library

This subsystem is a grouping for architecture. It provides the basis for packet delivery between "packet layers". For example, the video packet layer recieves packets delivered from the system layer which in turn recieves its packets from a media layer. The basic services used by all packet layers are defined in packet/nvrPacket.h and implemented in packet/nvrPacket.c. The implementation for smaller packet layers, such as the system stream packet layer and the Video CD packet layer, are kept in the packet directory. The table below describes all the packet layers and the location of the source code.

Packet Layer	*Files*	*Description*

MPEG System Packet Layer	`packet/mpeg1Packet.c`
	`packet/mpeg1Packet.h`
	`packet/mpeg1PacketI.h`
	`packet/mpeg1Proto.h`
	`packet/mpeg1sys.c`
	`packet/nvrMpegConf.c`
	`packet/nvrMpegConf.h`
		This implements the MPEG 1 system stream as a packet layer. See below for more detail.

File System Packet Layer	`packet/fsPacket.c` `packet/fsPacket.h`	This is a media layer that reads files in a file system and delivers them to layers above. It is always at the lowest layer.
Network System Packet Layer	`packet/netPacket.c` `packet/netPacket.h`	This is a media layer that will make network connections and deliver packets to layers above. It is always at the lowest layer.
Video CD Packet Layer	`packet/vcdPacket.c` `packet/vcdPacket.h`	This is a media layer that will take packets from a media layer, strip off the CDROM header and trailer, and deliver packets to layers above. It expects a media layer below it.
MPEG Video Packet Layer	`mpeg1vdec/*`	This implements the MPEG 1 video syntax decoding as a packet layer. See below for more detail.
MPEG Audio Packet Layer	`mpeg1adec/*`	This implements the MPEG 1 audio syntax decoding as a packet layer. See below for more detail.

The packet library provides two main structural definitions:
- a packet layer (see PacketLayer structure in nvrPacket.h)
- a packet datum (see PacketData structure in nvrPacket.h)

The PacketLayer defines the member functions associated with a packet layer and the PacketData describes the packet data passed between layers. Each unique kind of packet layer provides a single globally accessible function: the open function. All other functions are typically declared static and are only exposed through the member functions that appear in the PacketLayer structure.

The bottom-most layer in a packet system (like that in figure 1) is called a "media" layer. Currently, only two media layers exist: the network and the file system packet layer. All other layers must have another layer below them. Conversely, the media layers must always occupy the lowest layer; they may not have another layer below them.

Packet Layer Usage

Figure 2 shows a simplified usage by software outside of the packet subsystem. Code like this can be found in mpegsched/mpegsched.c:MpsProcess(). The ReadNextPacket() function (defined in packet/nvrPacket.c) takes any packet layer, descends to the lowest layer and calls its deliver function (a member function in the PacketLayer structure). The media layer then obtains a packet and delivers it to the packet layer (or layers) immediately above it. Each layer in turn parses the incomming packet and passes it to the next higher layer or layers. Once it reaches the top layer (such as a video or audio layer), it is simply put on a queue.

Eventually, control returns to the caller and the *process* function of each layer can be called. Only the video and audio packet layers have *process* functions. These perform all of the parsing of the elementary stream and producing pictures and PCM samples

```
Mpeg1PacketInfoPtr mpegInfo;
InternalReasonPtr irp;
PacketLayerPtr elayer;

if (NULL != (irp = ReadNextPacket(mpegInfo->media))) {
    if (irp->hint == IR_Eof) {
        Note that eof has occurred
    } else {
        Deliver Reason
    }
}
for (elayer = mpegInfo->list; elayer; elayer = elayer->sibling) {
    while (irp == NULL && elayer->queue) {
        if (NULL != (irp = (*elayer->ops.process)(elayer))) {
            Deliver Reason
            break;
        }
    }
}
```

Figure 2

Packet Layer Configuration

Building a set of packet layers into a usable system can be very complex, because the code that does this task must know how the layers fit together in a useful way. Currently this is handled by the functions defined in packet/nvrMpegConf.c. Initially, the packet system is opened with a call to OpenMpeg1PacketStreams() (see mpegdemux/mpegdemux.c:OpenNewSequence() for example code). This function returns information about the how many video and audio streams are present. Following this, the final construction of the packet layer is accomplished with a call to ConfigureMpeg1Streams() (see mpegdemux/mpegdemux.c:SelectSequenceChannels() for an example.

MPEG System Packet Layer

Its purpose is to handle the decoding of the MPEG 1 system syntax. This takes incomming system stream packets from a packet layer below it and delivers elementary video and audio packets to the Video and Audio packet layers above it. It is not necessary for the packets delivered to this layer to be aligned on packet boundaries.

The PacketData structure contains a timecode member that the system packet layer will fill in with the PTS value associated with the current packet. This timecode is used by the video and audio layers to mark the pictures and PCM samples they deliver with the proper timecode values.

Video Packet Layer

This subsystem accepts elementary video packets from a system packet layer or a media layer below it and decodes the MPEG 1 video bitstream syntax. It produces a list of dct blocks with motion vectors that are decoded by the Inverse DCT Library. In addition, the pictures that are produced by this packet layer are delivered via callbacks to its caller, which is typically the scheduler.

Audio Packet Layer

This subsystem accepts elementary audio packets from a system packet layer or a media layer below it and produces arrays of dct coefficients that are decoded by the Audio PCM sample synthesis subsystem.

Inverse DCT Library (see library/dct/*)

This takes the dct and motion information given to it by the Video Packet Library and produces y/cr/cb images that are delivered to the scheduler via callbacks and in turn delivered to the display subsystem. It is very reliable. This would need to be optimized for the x86 architecture.

Audio PCM sample synthesis

This takes the audio dct information and produces PCM samples. It is virtually unchanged from our old decoder and is very reliable. At this time, this code is part of the Audio Packet Layer (see mpeg1adec/mpeg1aSynthFilter.c).

Video display subsystem (see library/imgdpy/*)

This takes the y/cr/cb images from the scheduler and displays them. This subsystem consists of two parts: device-independent and device-dependent. There are two implementations for the device-dependent part right now: X11 and MacOS. The X11 part is based on code from the ODA.

Audio presentation subsystem (see library/audio/*)

This subsystem consists of two parts: device-independent and device-dependent. The device-independent portion does things like provide abstractions to the outside world and performs resampling. The device-dependent part takes pcm samples and delivers them to the hardware. We have four working implementations: Solaris 1, Solaris 2, SGI and MacOS. This has be unchanged for about a year.

Scheduler (see library/mpegsched/mpegsched*)

This is what I am currently working on. It has turned out to be completely device- and operating system independent. It relies on the layer above it to configure the Audio presentation and video display subsystem, and simply delivers data to these at the correct time. It depends on a defined interface for checking the time of day (see common/nvrClock*) and balances the video and audio dynamically. There are two interesting cases for a scheduler:

1. the decoder is running faster than real-time
2. the decoder is running at or slower than real-time

The latter is the most common case because few computers are able to decode an MPEG stream in real time. The former will currently cause problems because if the scheduler gets ahead of delivery, then its collection of undelivered pictures and PCM samples will grow without bound. Other known problems are that it currently can handle only system streams.

Xt/Motif Widget (see library/motifUI/NvrMpegDisplay.[ch])

This is a simple implementation that handles all of the display interaction. In particular, the widget discovers all of the available depths on the machine and creates one window for each display type and maps the currently configured depth.