The Analysis and Implementation of Generic MPEG Header
And Transport Layer Decoders

by

Yuan Feng

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 11, 1999

© Copyright 1999 Yuan Feng All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.
The Analysis and Implementation of Generic MPEG Header

And Transport Layer Decoders

Submitted to the
Department of Electrical Engineering and Computer Science

May 11, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Computer technology has progressed to a point that use of video and audio is practical. An average personal computer can receive, store, manipulate and playback audio and video well enough to dramatically change multimedia communication. IBM Research’s Community-Based Customer Access Solutions Department has embarked on research to contribute to this emerging field. Initial work of the research includes the development of a system to broadcast digital video and audio streams by satellite. This paper describes a portion of that research, including: the design of digital audio and video compression technology, tools created to analyze, decode, encode and manipulate audio and transport streams encoded in MPEG compression and transport formats.

Thesis Supervisor: Madhu Sudan
Title: Associate Professor, MIT Department of Electrical Engineering and Computer Science
Acknowledgments

First and foremost, I would like to thank my entire group at IBM Research for allowing me this opportunity to work on this thesis project. Geoffrey Purdy, Edward Clarke, Robert Flavin, Perwaiz Nihal and Nobert Vogl have continually supported me throughout the design, implementation and write-up phases of this project. My manager Geoffrey, and my direct supervisor Bob tirelessly guided me through the entire project. They always had time for me whenever I needed their advice and input. Bob often grilled me on my thesis progress. Geoffrey read and commented on every draft of my thesis write-up. This thesis project would not be possible without their support, encouragement and guidance, as well as their constant reminder of my thesis deadline. Thanks to Ed, I finally understood that a picture was indeed worth a thousand words.

I would also like to thank my thesis advisor, Madhu Sudan from MIT. Without his full understanding and interest in my project, this thesis would never be complete. At every step of the process, he is there to offer me advice and help. Without his input, I would have never reaped the full benefits associated with such a thesis project. His comments helped me to fully integrate my thesis work at the end.

Tristan Savatier and Gilles Boccon-Gibod from mpegtv.com have helped me to resolve certain technical issues during the initial stages of my thesis work. Thanks to them, my thesis work has gone rather smoothly.

Many thanks to the staff members at the MIT EECS department, especially Anne, Josephina and Lydia. Their coordinated efforts and superb administrative skills allowed me to stay focused at my work.

I would also like to take a moment to salute to F.T., my car and kroll, my computer. Without their unswerving dedication, I would probably be stuck somewhere on the road between Boston and New York, or reinstalling software instead of writing this acknowledgment.

Finally, I owe my family and friends everything I have. I just hope I would have a chance to give a little back to them someday. To those of my friends who have endured this arduous ordeal with me: We have made it! Time to get ready for the next challenge.
Table of Content

Chapter 1. Introduction ........................................................................................................ 5

Chapter 2. Background ...................................................................................................... 8
  2.1 MPEG Overview ......................................................................................................... 8
  2.2 Audio Overview ......................................................................................................... 10

Chapter 3. Phase I--MPEG Audio Header Decoder .......................................................... 15
  3.1 MPEG Audio Overview ............................................................................................. 15
  3.2 Coded Audio Bit Stream Syntax ................................................................................ 19
  3.3 Design and Implementation ....................................................................................... 22
    3.3.1 Functionality Specification .................................................................................. 22
    3.3.2 Main Data Structure ........................................................................................... 22
    3.3.3 Main Routines ..................................................................................................... 23
  3.4 Simulation Results ...................................................................................................... 26
  3.5 Conclusion .................................................................................................................. 33

Chapter 4. Phase II--MPEG Transport Stream Decoder .................................................... 35
  4.1 MPEG System Layer Overview ............................................................................... 35
  4.2 Transport Stream Syntax .......................................................................................... 37
    4.2.2 Program Specific Information ............................................................................. 41
    4.2.3 Packetized Elementary Stream Syntax ............................................................... 49
    4.2.4 Adaptation Field Syntax .................................................................................... 53
  4.3 Design and Implementation ....................................................................................... 53
    4.3.1 Functionality Specification ................................................................................ 53
    4.3.2 Main Data Structure .......................................................................................... 54
    4.3.3 Main Routines .................................................................................................... 54
  4.4 Simulation Results ...................................................................................................... 60
  4.5. Conclusion ................................................................................................................. 61

Appendix A. Discrepancies .............................................................................................. 63
Appendix B. Transport Stream Adaptation Field ............................................................. 64
Appendix C. Packetized Elementary Stream ................................................................. 68
Appendix D. Related Work ............................................................................................... 73
Appendix E. Code For Phase I and II Implementation ................................................... 87
References .......................................................................................................................... 145
Chapter 1. Introduction:

The Internet has revolutionized the telecommunications world and has become the catalyst for the genesis of a new social order. The invention of the telegraph, telephone, radio and computer set the stage for this unprecedented integration of capabilities. Even though from a purely technical perspective, the Internet merely offers an alternative mechanism for the sharing of data on a worldwide scale, it really, once and for all, promises a radically new information infrastructure for collaboration and interaction between individuals and communities which is not confined to geographical locations and by other physical constraints. Given that the "Galactic Network" concept of the social interactions through networking was proposed by J.C.R. Licklider of MIT only in August of 1962, rate of deployment and social acceptance of this technology is staggering.1

Computer technology has progressed to a point that use of video and audio is practical. An average personal computer can receive, store, manipulate and playback audio and video well enough to dramatically change multimedia communication. IBM Research's Community-Based Customer Access Solutions Department embarked on research to contribute to this emerging field. Initial work of the research includes the development of a system to broadcast digital video and audio streams by satellite. This paper describes a portion of that research, including: the design of digital audio and video compression technology, tools created to analyze, decode, encode and manipulate audio

---

and transport streams encoded in MPEG compression and transport formats. MPEG (Motion Picture Experts Group) provides the current international standards for digital audio and video compression as well as the associated transport and storage formats.

Work described in this paper is organized into two phases. The first phase implements a generic audio header decoder. This decoder decomposes MPEG audio data stream and analyzes the control information in its data structure headers. Based on its analysis, the decoder further generates stream-specific information such as data duration and content type. The second phase implements a generic Transport Layer decoder. This decoder decomposes MPEG Transport Layer stream and analyzes its control structures. Based on its analysis, this decoder generates stream-specific information such as program association, stream content, elementary data type and performs reconstruction of elementary data streams. In the case when an elementary data stream is encoded in MPEG audio format, work in the first phase has a direct application on it. Chapters Two, Three and Four all devote several sections to clarify the terminology mentioned here.

Work presented in this paper differs from existing implementations of MPEG decoders and their applications in a fundamental way. Referring to the diagram below, work in this research applies to stage B at the network level, while existing MPEG decoders are applicable at stages A and C.

![Figure 1: Simplified Network Overview](image-url)
Decoders developed in this project complement existing MPEG decoders and their applications. They filter MPEG encoded data streams by decomposing and examining their control information, program association and data content. The Transport Layer decoder even performs reconstruction of elementary data streams but stops short of decoding the elementary digital video and audio data. Instead, existing MPEG decoders and their applications carry out the decoding and presentation of these elementary digital video and audio data. The following example is just one application which demonstrates the above complementary feature.

![Diagram](image)

**Figure 2: Sample Application**

While a client at stage C is receiving data from a server at stage A, it can establish a simultaneous connection with the decoders in stage B. Decoders at stage B can provide the client with control information of the associated MPEG encoded data stream along with data duration, program association and content information which then allows the client to select from and filter streams received from the server. Once the selected and filtered data stream is received and buffered at stage C, existing decoders and their applications then decode the data stream and present the decoded visual/audio information to the audience. The server at stage A can initiate a similar connection with
the decoders at stage B to monitor and analyze the transmission of its encoded data streams.

Furthermore, work in the first phase also addresses ambiguity in the standards on the decoding of header information associated with MPEG-II, Layer-III audio data streams. Refer to Appendix A for a synopsis of the observed discrepancies.

The rest of this paper is organized into the following three chapters. Chapter Two gives an overview of the MPEG technology. Chapter Three describes work related to the first phase of this project. It starts off with a presentation of the MPEG audio structure, followed by design and implementation details of the generic audio header decoder. A discussion of simulation results and future work then conclude this chapter. Chapter Four adheres to the same format as that of Chapter Three when describing work related to the second phase of this project which develops a generic Transport Layer decoder.

Chapter 2. Background

Following is a broad overview of MPEG and audio compression in general. It is meant to familiarize readers who are not technically versed in this particular subject matter before leading them through the intricacies of MPEG standards and specifics of this research.

2.1 MPEG Overview

MPEG is an acronym for a family of International Standards used for coding audiovisual information in a digitally compressed format. The name is an acronym for Moving Picture Experts Group which pioneered the efforts in developing these standards
Leonardo Chiariglione, in particular, stands out as the father of MPEG. MPEG-I is formally known as ISO/TEC Standard 11172: Coding of moving pictures and associated audio for digital storage media at up to 1.5 Mbits/s.[3] It addresses the issue of combining one or more video and audio streams with timing information to form a single stream. The resulting stream is well suited for storage and transmission. Currently, Standard 11172 has five parts: system, video, audio, conformance testing and software simulation. MPEG-II is formally known as ISO/TEC Standard 13818: Generic coding of moving pictures and associated audio information.[4] It addresses the issue of combining multiple video and audio streams with timing information to form multiple streams suitable for storage and transmission. There are two forms: Transport Stream and Program Stream. Transport Stream is designed for an error-prone environment, while Program Stream is for a rather error-free environment. Furthermore, Transport Streams can result from combining streams of different time bases, but Program Streams are composed of streams with common time bases. Standard 13818 has nine parts: system, video, audio, conformance testing, software simulation, extension for Digital Storage Media Command and Control (DSM-CC), advanced audio coding, extension for real-time interface for system decoders and conformance extensions for DSM-CC.

Finally, the MPEG-IV Standard is still under development and will be numbered as ISO/TEC Standard 14496. MPEG-IV incorporates a key technology that enables the coding of visual objects of arbitrary shape. MPEG-II visual objects are constrained to be of rectangular shape.

The gradual permeation of MPEG and other compression formats to the multimedia industry follow directly from the acceptance of digital technology. Digital
technology has gained solid footing in the industry primarily for two reasons. First, it allows economic, compact and reliable replication of complicated system components. Anyone who has come across analog design would attest to the frustration of fine-tuning one system, let alone replicating one. Second, digital technology enables the possibility of vast net saving in bandwidth in comparison to the analog solution. For example, on a CD, two channels of audio data sampled at 44.1Khz with 16 bits per sample requires a data rate of approximately 1.4 megabits per second. There is a clear need for compression, and a vast array of digital compression techniques are available. Digital visual and audio information have become an integral part of the overall multimedia environment. Along with it, digital compression becomes vital in helping to establish its efficient interchange, distribution and storage.

The major advantage of MPEG over other video and audio coding formats lies in its sophisticated compression schemes. Because of them, the compressed files are much smaller for the same audible and visible quality. Experiments reveal that even at a 6:1 compression ratio, MPEG compressed audio data is indistinguishable from the original by professional listeners. In order to lend further credence to these results, researchers intentionally selected the most difficult audio data to compress[5,12].

2.2 Audio Compression Overview

A digital representation of audio data begins with the sampling of the analog wave forms in regular and discrete intervals. Quantization refers to bit-representation of the amplitude of sampled analog wave forms. For example, an eight-bit representation has a total range of 256 steps to approximate the amplitude of an analog sample. Unless it
involves an infinite-bit representation, quantization is an approximation, so has an associated error, i.e. quantization error. Signal to Noise Ratio, SNR, is one of the standard characterization of signal quality in Digital Signal Processing, DSP[6]. It is a measure of quantization error relative to the signal strength. Observe that the same amount of quantization error has a greater SNR in relation to signals of lesser amplitude.

One of the digital compression techniques, Pulse Code Modulation, or PCM, is widely adopted to digitize and quantize audio data[7,8]. In the case of a two-bit representation, PCM specifies four evenly spaced quantization levels as well as assigning a 2-bit code word for each level.

<table>
<thead>
<tr>
<th>Level</th>
<th>code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

It then divides the range of analog samples into four segments. Quantization occurs when analog values of these samples from different segments are mapped into different code words.

A natural extension from PCM is the u-law compression method specified by CCITT Recommendation G.711[9]. Because of the even spacing in PCM, there is a greater SNR associated with samples of lower amplitude. In recognition of this fact, the u-law technique employs nonlinear quantization. Logarithmic in nature, distribution of the quantization levels are denser for lower levels. As a result, overall signal quality is much more uniform over the entire amplitude range than that achieved by PCM.
A slightly better scheme, Adaptive Differential Pulse Code Modulation (ADPCM) increases that factor by two. Instead of faithfully coding each sample as in PCM, ADPCM only encodes the difference between each sample and its predicted value. It assumes that neighboring audio samples are generally similar to each other. This assumption is reasonable when audio data contains smooth transitional tones. The extreme counter example would be sampling a sinusoidal wave form at its peaks and valleys, but this situation is very unlikely in most commercial applications. When the assumption is applicable, encoding only the difference can drastically reduce the length of code words, i.e. number of bits used to represent an analog sample. As long as the encoder and decoder both utilize the same set of rules for prediction, the difference is sufficient in reconstructing the original encoded sample value. In its simplest prediction scheme, the previous sample serves as the predictor. ADPCM further enhances compressed signal quality with the capability of adapting to signal characteristics by changing the spacing involved in quantization.

More sophisticated compression techniques rely on psycho-acoustical models for greater efficiency. Human hearing and voice range from 20 Hz to 20 KHz. The 2-4 Khz range is the most sensitive. Dynamic range is about 96dB. Further psycho-acoustical experiments[10,11] point out that several masking factors dictate the audibility of tones. Masking occurs when a tone is no longer distinguishable because of the presence of another. Frequency and amplitude are two major factors associated with masking. Frequency masking focuses on how a tone of a fixed amplitude, i.e. testing tone, is masked in the presence of a different frequency, i.e. masking tone. Here, the threshold of a masking tone is its minimum level of amplitude below which the testing tone is audible.
These thresholds vary quite a bit over the spectrum. In temporal masking, a testing tone is still masked a short period after the presence of a masking tone. Again, the exact delay in audio perceptibility is dependent on both frequency and amplitude of the masking tone. Steinmetz and Nahrstedt[12] recorded experimental results in their notes. The above observations lead to the partitioning of the audible spectrum into critical bands in order to better characterize the resolving power of the ear as a function of frequency. Finer partitioning better models the masking effects but also results in greater complexity.

Sub-band coding exploits these psychoacoustical phenomenon by designing filter banks to partition the audio data spectrum into bands approximating critical bands of human hearing. Psycho-acoustical models are then applied to sub-bands individually in order to ensure the best quantization results in light of various masking effects.

Both Layer-I and Layer-II of the MPEG audio standards employ this method. In Layer-I, the filter bank partitions the audio spectrum into 32 constant-width sub-bands. For every 32 input audio samples, a Layer-I encoder provides 32 frequency samples, one per sub-band. Each group of 12 samples from one sub-band would then receive one bit allocation along with a 6-bit scale factor if necessary. If a sample value is zero, then scale factors are omitted. A Layer-I frame then consists of 32 groups of 12 samples, i.e., 384 samples. The technical definition of a frame is a part of the audio signal that corresponds to a fixed number of audio PCM samples. More intuitively, a frame is the smallest decodable unit in an audio stream. A Layer-II algorithm enhances Layer-I by coding data in larger groups. It codes 3 groups of 12 samples from every sub-band into 1 bit allocation. Each frame has dimension of 3 by 12 by 32, or 1152 samples. Stereo
redundancy coding in MPEG audio is slightly more involved, but follows from the same logic.

Being the most complicated of the three, Layer-III extends both Layer-I and Layer-II with the incorporation of transform coding. It applies a Modified Discrete Cosine Transform (MDCT) to the filter bank outputs, thus compensating for certain filter bank deficiencies[13]. The constant-width frequency bands constructed by the filter bank do not match an ear's critical bands perfectly. The band width is too wide for lower frequencies. Thus the number of quantizer bits is insufficient for the level of noise sensitivity within the sub-band. Also, there is significant overlap between adjacent filter bands. Therefore, signals at a particular frequency can have rippling effects on several neighboring bands. The MDCT's provides better spectral resolution by further subdividing the filter bank outputs. Since there is always a tradeoff between time and frequency resolution, MDCT utilizes two block lengths, a short block of 12 samples and a long block of 36. The short length is designed for transients where better time resolution is more important. Aside from this extension, the filter bank for Layer-III is similar to that of Layer-II. Each frame contains only 1152 samples.

As a side note, there are two types of digital compression, lossless and lossy. All of the above compression methods are lossy because loss of data is tolerable when dealing with audio data. On the other hand, lossless compression methods are primarily for files whose digital makeup must be preserved, such as an executable string. Decompressing a losslessly compressed file yields the exact original. The downside, meanwhile, is the 2:1 compression ratio achieved by these methods. There are other major branches in audio lossy compression other than those discussed above. One in
particular concentrates on speech recognition. Examples of it include Linear Predictive Coding and Code Excited Linear Predictor, which attempt to best fit signals to speech models. They have little relevance to this particular thesis project and are only mentioned for the purpose of completeness.

Chapter 3. First Phase—MPEG audio header decoder

3.1 MPEG audio Overview

Figures 3 and 4 below represent the block diagrams for the generic audio encoder and decoder. Input audio samples are fed into the encoder. Mapping then converts these samples from time to frequency domain. The mapped samples are referred to as sub-band samples in Layer-I and II because these two layers employ sub-band filtering as mentioned in Section 2.2. By analogy, those in Layer-III are called transformed sub-band samples, again because of the additional MDCT involved in the mapping process in that layer. As the block diagram points out, a psycho-acoustical model generates a set of control data for the quantizer and coding process. In essence, a psycho-acoustical model attempts to describe sensitivity levels related to human hearing as qualitatively accurate as possible. The more accurate this description is, the more efficient the encoding process is. Section 2.2 briefly touched upon certain criteria essential to a psycho-acoustical model.

The bit stream data input to the decoder consists of frames. Notice how closely the decoding process mirrors the encoding process. This close matching is expected since a decoder aims to reverse the encoding process with minimal loss of information.

Figure 5 extends Figures 3 and 4 with a more in-depth look at the sub-processes within each block.
As a reminder, the term, "MPEG audio" in this research, refers collectively to ISO/TEC standards 11173-3, i.e. MPEG-I audio and 13818-3, i.e. MPEG-II audio. MPEG-II audio extends MPEG-I audio by providing three lower sampling frequencies.
MPEG Layer-III supports variable bit rate. In other words, frames from the same audio stream data can contain audio data of different bit rates. So far, most MPEG audio data examined are uniformly encoded. Most examined decoders available in public domain assume uniform bit rate within the same audio stream data. First Phase of this research takes into consideration the existence of variable bit rate encoded audio stream data.

<table>
<thead>
<tr>
<th>bitrate_index</th>
<th>MPEG-I</th>
<th>bitrate specified (kbps) for 32,48,44.1 kHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0000'</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>'0001'</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>'0010'</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>'0011'</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>'0100'</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>'0101'</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>'0110'</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>'0111'</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>'1000'</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>'1001'</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>'1010'</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>'1011'</td>
<td>176</td>
<td>176</td>
</tr>
<tr>
<td>'1100'</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>'1101'</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>'1110'</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>'1111'</td>
<td>forbidden</td>
<td>forbidden</td>
</tr>
</tbody>
</table>

Table 1: bit rates for MPEG-I and MPEG-II

<table>
<thead>
<tr>
<th>bitrate_index</th>
<th>MPEG-II</th>
<th>bitrate specified (kbps) for 16,22,05,24 kHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0000'</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>'0001'</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>'0010'</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>'0011'</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>'0100'</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>'0101'</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>'0110'</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>'0111'</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>'1000'</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>'1001'</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>'1010'</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>'1011'</td>
<td>176</td>
<td>176</td>
</tr>
<tr>
<td>'1100'</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>'1101'</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>'1110'</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>'1111'</td>
<td>forbidden</td>
<td>forbidden</td>
</tr>
</tbody>
</table>
3.2 Coded Audio Bit stream Syntax

In order to better understand the structure of a frame, let’s start with a graphical representation of it.

First phase of this research focuses on the decoding of the header section of a frame. The header section consists of the first 32 bits of a frame which are common to all layers. It contains information sufficient to enable a standard decoder to properly interpret the audio data and perform a lossy reconstruction of the original PCM audio samples. This information is also sufficient to construct additional stream-specific information.
which is of interest to this research, such as playback duration of the audio stream data.

Here are definitions to the header information:

**Syncword**—Used to identify the start of a frame, it consists of the bit string '1111 1111 1111'.

**ID**—A bit value of '1' indicates MPEG-I audio; a bit value of '0' indicates MPEG-II audio.

**Layer**—It consists of two bits which identify the layer number based on the following mapping.

- '11' Layer-I
- '10' Layer-II
- '01' Layer-III
- '00' reserved

**Protection_bit**—A bit value of '1' indicates that no redundancy has been added to facilitate error detection and concealment; a bit value of '0' otherwise.

**Bitrate_index**—The four-bit value serves as an index to the bit rate tables displayed above.

**Sampling_frequency**—Similar to the bitrate_index, this two-bit value serves as an index to the sampling frequency mapping below.

<table>
<thead>
<tr>
<th></th>
<th>MPEG-I</th>
<th>MPEG-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>'00'</td>
<td>44.1 Khz</td>
<td>22.05 Khz</td>
</tr>
<tr>
<td>'01'</td>
<td>48 Khz</td>
<td>24 Khz</td>
</tr>
<tr>
<td>'10'</td>
<td>32 Khz</td>
<td>16 Khz</td>
</tr>
<tr>
<td>'11'</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>

**Padding_bit**—It's used to adjust the mean bit rate to the sampling frequencies of 44.1 Khz in MPEG-I and of 22.05 Khz in MPEG-II. A bit value of '1' rounds the number of slots in a frame to the nearest integer, while a bit
value of '0' truncates the number. A slot in the frequency domain is very much analogous to a sample in time domain. The number of slots indicate the distance between two consecutive syncwords. Concrete examples of calculating slots are in Section 3.3.3.

Private_bit--It's a bit for private use.

Mode--The two bits serve as index to the following mapping.

- '00' stereo
- '01' joint_stereo
- '10' dual_channel
- '11' single_channel

Mode_extension--These two bits are applicable only when the mode is joint_stereo.

Layer-I, II
- '00' sub-bands 4-31 in intensity_stereo, bound=4
- '01' sub-bands 8-31 in intensity_stereo, bound=8
- '10' sub-bands 12-31 in intensity_stereo, bound=12
- '11' sub-bands 16-31 in intensity_stereo, bound=16

Layer-III
- '00' intensity_stereo off, ms_stereo off
- '01' intensity_stereo on, ms_stereo off
- '10' intensity_stereo off, ms_stereo on
- '11' intensity_stereo on, ms_stereo on

Copyright--A bit value of '1' indicates copyright protection; '0' otherwise.

Original/home--A bit value of '1' means the audio data is original; '0' otherwise.

Emphasis--A two-bit value indicates the type of de-emphasis used.

- '00' no emphasis
- '01' 50/15 microsec, emphasis
- '10' reserved
- '11' CCITT J.17
3.3 Design and Implementation

This section presents the design and implementation of the header decoder in the first phase.

3.3.1 Functionality Specification

Input: audio stream data encoded according to ISO/TEC 11172-3 and 13818-3 specifications.

Output: stream specific information including stream header information and playback duration.

3.3.2 Data Structures

Following is a list of main data structures defined for the first phase of this thesis project.

headerinfo—data structure to store the header information

MPG1_bit_rate and MPG2_bit_rate tables—tables according to Table 1 above

MPG_layer table—table according to the specification in Section 3.2

MPG_forbidden_layers table—table which contains the combinations of total bit rate and mode which are allowed for MPEG-I, Layer-II.

- free format all modes
- 32 kbits/s single_channel
- 48 kbits/s single_channel
- 56 kbits/s single_channel
- 64 kbits/s all modes
- 80 kbits/s single_channel
- 96 kbits/s all modes
- 112 kbits/s all modes
- 128 kbits/s all modes
160 kbits/s  all modes
192 kbits/s  all modes
224 kbits/s  stereo, intensity stereo, dual channel
256 kbits/s  stereo, intensity stereo, dual channel
320 kbits/s  stereo, intensity stereo, dual channel
384 kbits/s  stereo, intensity stereo, dual channel

3.3.3 Main Routines

Following is a list of main routines defined for the first phase of this research.

First, the following diagram depicts the chain of execution in the MPEG stream decoding process.
Figure 7: Phase I Decoder Flow Chart

**Buffer_file**: A routine that buffers input audio data stream.

**readbits**: A routine that reads in the number of bits specified and returns them in decimal representation.

**find_sync**: This routine scans the data stream and locates the first occurrence of a syncword. Because the buffer storing the data is of type `char`, it's necessary to take care of the two different bit-representations of 0xFFF, 0xFFF0 and 0x0FFF. Once an occurrence
of a syncword is located, find_sync then evokes authenticate_head to test if the syncword found is a real syncword.

authenticate_head: Once find_sync locates a possible syncword, this routine then verifies the authenticity of the syncword. Initial tests involve check for forbidden sampling frequencies, bit rates, and combinations of bit rates and modes. The final and most sophisticated check involves the location of consecutive syncwords based on calculation of slots. The core of the test is as follows:

```c
// Check if Layer-I
if (head->layer == 1) {
   sync_distance=48*head->bitrate*1000/head->Samp_freq; (a)
// Check if MPEG-II and Layer-III
} else if (!head->ID && (head->layer == 3)) {
   sync_distance=72*head->bitrate*1000/head->Samp_freq; (b)
} else {
// Otherwise
   sync_distance=144*head->bitrate*1000/head->Samp_freq; (c)
}
```

By definition, one slot is equal to one byte in Layers-II/III, and four bytes in Layer-I. Using N to indicate the number of slots, then the following equations hold,

\[
N = 12 \times \frac{\text{bitrate}}{\text{sampling frequency}}; \quad \text{(Layer-I)}
\]
\[
N = 144 \times \frac{\text{bitrate}}{\text{sampling frequency}}; \quad \text{(Layer-II/III)}
\]

These two equations help to derive statements (a) and (c) in the above test by equating one slot with one byte for Layers-II/III and with four bytes for Layer-I. The fact that each frame contains information for 384 samples in Layer-I and 1152 samples in Layers-II/III form the basis for the derivation of the above equations. During test runs to decode header information of some sample audio streams, however, statement (c) does not predict correctly the location of consecutive syncwords in MPEG-II, Layer-III audio data.
streams. In fact, when there is no padding, it skips every other syncword. In the case when padding bits of consecutive frames alternate from bit value '1' and '0', statement clearly would never be able to predict the location of consecutive syncwords correctly. This miscalculation, I suspect, is the culprit behind the differences in MPEG-II, Layer-III audio file sizes calculated using various commercial decoders. According to Standard 11172-3 specification, Layer-III utilizes two granules in its encoding algorithm. When extended to lower sampling frequencies, however, Layer-III only uses one granule, therefore effectively reducing the number of slots per frame by a factor of two. This detail is lost in the Standard 13818-3 specification. MPEG experts from mpg.com under Tristan's direction clarified this ambiguity. Upon authenticating the syncword, authenticate_head then decodes the header information and stores it into the headerinfo data structure.

\textit{pb\_duration}: playback duration calculates, to the accuracy of millisecond, the length of an audio data file from the first frame to the very last. The total file size along with the bit rate of a clip are sufficient to generate a rough estimate of the overall playback duration of the piece. A more accurate method utilizes \textit{find\_sync} and \textit{authenticate\_head} to first calculate the total number of non-null frames present in the clip. That information, along with the number of slots in one frame, will yield a more accurate estimate of playback duration. To take into account the occurrence of variable bit rates, \textit{pb\_duration} offers the option of checking the bit rate on a frame by frame basis.

\textbf{3.4 Simulation results}

Following contains some simulation results performed on some testing data. Presentation of each result includes three sections, samples of testing data, simulation
output and verification. Samples of testing data are in hexadecimal format. The first column specifies the location of data bytes.

\textit{funky.mp3}

\textbf{sample of testing data:}

\begin{verbatim}
0000: FFFB7054 000007D 00417000 0021100
0010: 082F0000 0555181 04958480 00BB8960
0020: AA9EC000 EBC9ACD C0000000 495032AA
0030: 2C7FACAF E6DC6E5 BBFFFF6D B692A065
0040: 545877AC A8000119 04180342 8680A070
0050: F4A5FBA9 137E250F E3FB0A8A 5B862524
0060: 8A050310 87EBB91E A440497D 70C120A0
0070: 9080300B 8174A122 84014008 18C5D020
0080: 4116810 8AE4DAE8 BA7347AA 48805116
0090: EA108922 6D0A1D39 CE68136E 09BE098A
00A0: C30E9A82 8143A702 8E159C3C 2B6D7D23
00B0: 15B6CE20 2040849C B93F9D5A 347AD901
00C0: E0B66DBD 82016201 C23E989A 9162EAC5
00D0: 64F74114 11232710 1E15ACC8 20C8C000
00E0: 31E96121 C4040000 30283132 06985F80
00F0: 0BC2DB6E 36D0C000 481904AD OB3FD0ED
0100: 1FEC41CB 4EEB8390 B8340B82 ED146120
0110: 043089BB AFAD81C0 2A01C663 84486663
0120: 57666367 36603803 80000000 000D9F0
0130: 290023C3 E157C09C 1AFFFB72 740D8001
\end{verbatim}

\textbf{simulation output:}

Input File Name: funky.mp3
input audio file size is 744626 bytes
SYNCWORD found at 0
ID is MPG-1.
layer is 3
redundancy has not been added
bitrate is 96 kbits/second
Sampling frequency is 44100 Hz
padding is not introduced
private bit is 0
mode is joint_stereo
intensity_stereo on; ms_stereo off
no copyright protection
original
type of de-emphasis used:
no emphasis
playback time is 61 secs and 72 milliseconds

\textbf{verification:}
audio frame header: 0xFFFB7054

<table>
<thead>
<tr>
<th>0xFFFF</th>
<th>synword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID: MPEG I</td>
</tr>
<tr>
<td>01</td>
<td>layer: Layer III</td>
</tr>
<tr>
<td>1</td>
<td>protection_bit: no redundancy</td>
</tr>
<tr>
<td>0111</td>
<td>bit_rate: 96 kbits/s</td>
</tr>
<tr>
<td>00</td>
<td>sampling_frequency: 44.1 kHz</td>
</tr>
<tr>
<td>0</td>
<td>padding_bit: no padding</td>
</tr>
<tr>
<td>0</td>
<td>private_bit: not used</td>
</tr>
<tr>
<td>01</td>
<td>mode: joint_stereo</td>
</tr>
<tr>
<td>01</td>
<td>mode_extension: sub-bands 8-31 in intensity_stereo, bound=8</td>
</tr>
<tr>
<td>0</td>
<td>copyright: no copyright</td>
</tr>
<tr>
<td>1</td>
<td>original: original</td>
</tr>
<tr>
<td>00</td>
<td>emphasis: no emphasis</td>
</tr>
</tbody>
</table>

sync_distance = 144 * bit_rate / sampling_frequency
= 313 or 0x139 bytes

The next syncword 0xFFFFB7274 has its padding_bit equal to 1, so the sync_distance to the next syncword is 314 or 0x13A bytes away, which is located at address 0x273.

classic1.mp3

sample data:

| 0000: | FFF37274 0D0D7C7F | 4F2FC324 000E88FE |
| 0010: | 9E1F8660 00801502 | ACAEA862 A5018613 |
| 0020: | 8B8D845A 279AE3EC | 43E31146 48CB0A12 |
| 0030: | 24C2C638 030B0221 | 605APFED 6D546943 |
| 0040: | D04F8842 6A24C977 | D47DC648 865F89D2 |
| 0050: | 3670C311 0C87A50E | 9C02090C BC08F5A1 |
| 0060: | 4D70123D EC6BEC43 | 6E65C145 B9AB0E2F |
| 0070: | ABF0C7E 00244464 | 68A6CC8EC A6CCD24D |
| 0080: | B6DA482B 4179D5B0 | 8418E094 5D410A4F |
| 0090: | BE7FEFAF 733272AF | 9FDDCA38 B28C28D2 |
| 00A0: | 78F41492 8269BF52 | 79A3B7C7 722FB918 |
| 00B0: | 449355E5 9A4723FF | F3727409 0D50A15A |
simulation result:

Input File Name: classic1.mp3
input audio file size is 138972 bytes
SYNCWORD found at 0
ID is MPG-2.
layer is 3
redundancy has not been added
bitrate is 56 kbits/second
Sampling frequency is 22050 Hz
padding is introduced
private bit is 0
mode is joint_stereo
intensity_stereo on; ms_stereo on
no copyright protection
original
type of de-emphasis used:
no emphasis
playback time is 19 secs and 33 millisecs

verification:
audio frame header: 0xFFF37274

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncword</td>
<td>0xFF</td>
</tr>
<tr>
<td>ID</td>
<td>0</td>
</tr>
<tr>
<td>layer</td>
<td>1</td>
</tr>
<tr>
<td>protection_bit</td>
<td>no redundancy</td>
</tr>
<tr>
<td>bitrate_index</td>
<td>56 kbits/s</td>
</tr>
<tr>
<td>sampling_freq</td>
<td>22.05 kHz</td>
</tr>
<tr>
<td>padding_bit</td>
<td>padding</td>
</tr>
<tr>
<td>private_bit</td>
<td>not used</td>
</tr>
<tr>
<td>mode</td>
<td>joint_stereo</td>
</tr>
<tr>
<td>mode_extension</td>
<td>intensity_stereo on, ms_stereo on</td>
</tr>
<tr>
<td>copyright</td>
<td>no copyright</td>
</tr>
<tr>
<td>original</td>
<td>original</td>
</tr>
<tr>
<td>emphasis</td>
<td>no emphasis</td>
</tr>
</tbody>
</table>

sync_distance = 72 * bit_rate / sampling_frequency + 1 = 183 or 0xB7 bytes

**15-kz-064.mp2**

sample data:

<table>
<thead>
<tr>
<th>Time (in samples)</th>
<th>Sample Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FFFD4800 443324A2 51249249 20900000</td>
</tr>
<tr>
<td>0010</td>
<td>00000009 6B029BC0 A6B931DF 4C6FE49F</td>
</tr>
<tr>
<td>0020</td>
<td>ECC73A35 EEC77E3 1DE4F7FE 35DF948F</td>
</tr>
<tr>
<td>0030</td>
<td>E3DF8EB9 7D460ECE 8BA21F9 087B4A2E</td>
</tr>
<tr>
<td>0040</td>
<td>518BE428 F8FA7E77 77775B16 C6B5AF9A</td>
</tr>
<tr>
<td>0050</td>
<td>D7CD6B5A D6B5AD6B 5AEEEEE2 B62D856B</td>
</tr>
<tr>
<td>0060</td>
<td>5F35AF9A D6B5AD6B 5AD6B5A3 56B6A78E</td>
</tr>
<tr>
<td>0070</td>
<td>E6AB61B4 F85A99B6 673C5B0A 83C3BC3A</td>
</tr>
<tr>
<td>0080</td>
<td>EAB635AD 7CD6BD6B 9AD6B5AD 6B5AD77</td>
</tr>
<tr>
<td>0090</td>
<td>E775C167 739D01CE 80E73A2E 739CE72E</td>
</tr>
<tr>
<td>00A0</td>
<td>0EEEEE9B 62D8D6B5 F35AF9AD 685AD6B5</td>
</tr>
<tr>
<td>00B0</td>
<td>AD6B5DDD DDD6C5B1 AD6B6B5 F35AD6B5</td>
</tr>
<tr>
<td>00C0</td>
<td>AD6B5AD6 BBBBBBAD 8B635AD7 CD6BE6B5</td>
</tr>
<tr>
<td>00D0</td>
<td>AD6B5AD6 B5AD0000 00002D80 034001A3</td>
</tr>
<tr>
<td>00E0</td>
<td>EDD0340D 341A0000 00000000 5B000680</td>
</tr>
<tr>
<td>00F0</td>
<td>0347DBA0 681A6834 00000000 0000B600</td>
</tr>
<tr>
<td>0100</td>
<td>0D00068F B74D034D D680000 00000001</td>
</tr>
<tr>
<td>0110</td>
<td>6C001A00 0D1F6BE81 A069A0D0 00000000</td>
</tr>
<tr>
<td>0120</td>
<td>FFFD4800 66226DB6 C001C6C8 012AAAAA</td>
</tr>
</tbody>
</table>

**simulation result:**

Input File Name: 15kz-064.mp2

30
input audio file size is 120097 bytes
SYNCWORD found at 0
ID is MPG-1.
layer is 2
redundancy has not been added
bitrate is 64 kbits/second
Sampling frequency is 32000 Hz
padding is not introduced
private bit is 0
mode is stereo
no copyright protection
not original
type of de-emphasis used:
no emphasis
playback time is 14 secs and 86 millisecs

verification:

audio frame header: 0xFFFFD4800

<table>
<thead>
<tr>
<th>0xFFFF</th>
<th>syncword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID: MPEG 1</td>
</tr>
<tr>
<td>10</td>
<td>layer: Layer II</td>
</tr>
<tr>
<td>1</td>
<td>protection_bit: no redundancy</td>
</tr>
<tr>
<td>0100</td>
<td>bit_rate: 64 kbits/s</td>
</tr>
<tr>
<td>10</td>
<td>sampling_frequency: 32 kHz</td>
</tr>
<tr>
<td>0</td>
<td>padding_bit: no padding</td>
</tr>
<tr>
<td>0</td>
<td>private_bit: not used</td>
</tr>
<tr>
<td>00</td>
<td>mode: stereo</td>
</tr>
<tr>
<td>00</td>
<td>mode_extension: sub-bands 4-31 in intensity_stereo, bound=4</td>
</tr>
<tr>
<td>0</td>
<td>copyright: no copyright</td>
</tr>
<tr>
<td>0</td>
<td>original: no original</td>
</tr>
<tr>
<td>00</td>
<td>emphasis: no emphasis</td>
</tr>
</tbody>
</table>

sync_distance=144 * bitrate / sampling_frequency
=288 or 0x120 bytes

verd16m.mus
sample data:

0000: FFF488C4 53D24566  B6DBB7FF  FFF488C4
0010: 55555555 555555F7  DF7DF7DF  7DF7DF7D
0020: F7DF7DF7  DF7DF7DF  7DF7DF7D  F7CB63BB
0030: BDEF7BDE F7BDEF7B  DEF7BDEF  7BDF7DF7
0040: DF7D6C5B 16C5B16C  5B16C5B1  6C5B16C5
0050: B16C5B16  C5B16C5B  16C5B16DD  DEF7BDEF
0060: 7BDEF7BD EF7BDEF7  BDEFBEFB  EFBE62BD
0070: BB62D8B6  DB862D8B  B62D8B62  DB62D8B8
0080: 62D8B62D  8B62D8B8  EF7BDEF7  BDEF7BDE
0090: F7BDEF7B  DEF7DF7D  F7DF5B16  CB165B16
00A0: 16C5B16C  5B16C5B1  6C5B16C5  B16C5B16
00B0: C5B16C77  77BDEF7B  DEF7BDEF  7BDF7BD
00C0: EF7BDEF7  BEFBAD8B  62D8B62D  8B62D688
00D0: 2D862D8B  B62D8B62  D862D8B8  62D8B63B
00E0: BBDEF7BD  EF7BDEF7  BDEF7BDE  7BDF7DF7
00F0: 7D7D6C5B 16C5B16C  6C5B16C5  16C5B16C
0100: 5B16C5B1  6C5B16C5  B16C5B1D  DDEF7BDE
0110: F7BDEF7B  DEF7BDEF  7BDEFBEF  BEFBEB62
0120: D862D8B6  62D8B62D  8B62D686  2DB62D88
0130: B62D8B62  8B62D8B8  EF7BDEF7  7BDEF7BD
0140: EF7BDEF7  BDEF7DF7  DF7DF5B1  66C5B16C
0150: B16C5B16  C5B16C5B  16C5B16C  5B16C5B1
0160: 6C5B16C7  77BDEF7B  DEF7BDEF  F7BDEF7B
0170: DEF7BEFB  EFBEAD8B  62D6BB62  DB62B8B8
0180: 62D8B62D  8B62D8B8  2DB62D88  B62D88B3
0190: BBDEF7B6  DEF7BDEF  7BDEF7BD  EF7BDF7D
01A0: F7DF7D6C  5B16C5B1  6C5B16C5  B16C5B16
01B0: C5B16C5B  16C5B16C  5B16C5B1  5B16CB1D
01C0: EF7BDEF7  BDEF7BDE  F7BDEFBE  BEFBEB6B
01D0: 2D862D8B  B62D8B62  D8B62B82  62D88B62
01E0: 8B62D8B6  2D862D8B  EEF7BDEF  F7BDEF7B
01F0: DEF7BDEF  7BDF7DF7  7DF7DF5B  16C5B16C
0200: 5B16C5B1  6C5B16C5  B16C5B16  C5B16C5B
0210: 16C5B16C  7777BDEF  7BDF7BD  EF7BDF7D
0220: BDEF7BEF  BEFBEAD8  8B62B8B6  2D88B62D
0230: B62D8B62  D8B62D8B  62D88B20  8B628800
0240: FFF488C4  53D24566  B6DBB7FF  FFF488C4

simulation result:
Input File Name: verdl6m.mus
input audio file size is 100224 bytes
SYNCWORD found at 0
ID is MPG-2.
layer is 2
redundancy has been added
bitrate is 64 kbits/second
Sampling frequency is 16000 Hz
padding is introduced
private bit is 0
mode is single_channel
no copyright protection
original
type of de-emphasis used:
no emphasis
playback time is 12 secs and 31 millisecs

verification:

audio frame header: 0xFFF488C4

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF</td>
<td>syncword</td>
</tr>
<tr>
<td>0</td>
<td>ID: lower sampling frequency</td>
</tr>
<tr>
<td>10</td>
<td>layer: Layer II</td>
</tr>
<tr>
<td>0</td>
<td>protection_bit: redundancy</td>
</tr>
<tr>
<td>1000</td>
<td>bit_rate: 64 kbits/s</td>
</tr>
<tr>
<td>10</td>
<td>sampling_frequency: 16kHz</td>
</tr>
<tr>
<td>0</td>
<td>padding_bit: no padding</td>
</tr>
<tr>
<td>0</td>
<td>private_bit: not used</td>
</tr>
<tr>
<td>11</td>
<td>mode: single_channel</td>
</tr>
<tr>
<td>00</td>
<td>mode_extension: sub-bands 4-31 in intensity_stereo, bound=4</td>
</tr>
<tr>
<td>0</td>
<td>copyright: no copyright</td>
</tr>
<tr>
<td>1</td>
<td>original: original</td>
</tr>
<tr>
<td>00</td>
<td>emphasis: no emphasis</td>
</tr>
</tbody>
</table>

\[
\text{sync\_distance} = 144 \times \frac{\text{bit\_rate}}{\text{sampling\_frequency}} \\
\text{= 576 or 0x240 bytes}
\]

3.5 Conclusion

Carried out in parallel to other phases of the research, the current implementation the generic header audio decoder has been tested against stored data files only. Integration with the rest of the group project requires some additional work resolving issues associated with handling of network data. Specifics include consideration of buffer overflow and synchronization issues. Also, there needs to be an additional handler to
piece together the MPEG encoded data from packetized network traffic data. The design and implementation of this handler can simply follow the groundwork laid in the second phase of this research which deals with the assemblage of packetized data based on MPEG Systems specifications. Issues surround the adoption of `pb_duration` to handle network traffic is nontrivial as compared to those mentioned in `buffer_file`. The fundamental issue is how to differentiate one clip from another as they are played in continuous succession. Possible checks include header mapping, checking the garbage in between clips, sensing the delays in between sound clips.
Chapter 4. Phase II--MPEG Transport Stream Decoder

4.1 MPEG Systems Layer Overview

In this documentation, MPEG Systems layer refers specifically to the Systems part of the international standard, ISO/IEC 13818. It addresses the combining of one or multiple Elementary Streams of video and audio into single or multiple streams which are suitable for storage and transmission. The following diagram depicts the scope of the Systems specifications.

Figure 8: Scope of the Systems Specification

As mentioned in Section 2.1, Program Stream and Transport Stream make up the two

---

forms of system encoding in this specification. The Program Stream, PS, is analogous to ISO/IEC 11172 Systems layer. It results from combining one or more streams of PES packets which have a common time base into a single stream. PES stands for Packetized Elementary Stream. Program Stream packets can be of variable length and are for transmission/storage in a relatively error-free environment. The Transport Stream, TS, combines one or more programs of independent time bases into one single stream. Each program is made up from PES packets of Elementary Streams of a common time base. Each Transport Stream packet is of 188 bytes in length. It’s designed for error-prone environments. Notice how the Systems layer is transparent to the content carried within Elementary Streams. This level of transparency allows data transmission schemes such as MPEG over MPEG, IP over MPEG, etc. There are three major decoding operations at the Systems layer: retrieval of encoded programs from either a Transport or Program Stream; regrouping within either a Transport or Program Stream, such as outputting multiple Transport Streams, each of which contains only one program from the original Transport Stream; conversion between a Transport Stream and a Program Stream.

Phase II of this research focuses on the first case involving the retrieval of programs from a Transport Stream. The task of retrieving programs from a TS breaks down to three components:

1. Retrieval of Program Specific Information, PSI;
2. Reconstruction of Elementary Streams;
3. Retrieval of decoding control information.

The following diagram gives the overall picture of encoding, transmission and decoding at the Transport layer.
Figure 9: Transport Layer System Overview
4.2 Transport Stream Syntax

The following diagram contains the structural layout of a Transport Stream packet.

```c
transport_packet()
{
  sync_byte
  transport_error_indicator
  payload_unit_start_indicator
  transport_priority
  PID
  transport_scrambling_control
  adaptation_field_control
  continuity_counter
  if (adaptation_field_control=='10' || adaptation_field_control=='11')
  {
    adaptation_field()
  }
  if (adaptation_field_control=='01' || adaptation_field_control=='11')
  {
    for (i=0;i<N;i++)
    {
      data_byte
    }
  }
}
```

Figure 10: Transport Stream Packet Syntax

Definitions to the terminology shown above are as follows:

**Sync_byte**—This fixed 8-bit field of hexadecimal value, 0x47 indicates the start of a Transport Stream packet.

**Transport_error_indicator**—This is a 1-bit field. A value of 1 reflects the presence of error within this packet. The bit value is set outside of the scope of the Systems specification.
Payload_unit_start_indicator--This is also a 1-bit field. The payload of a Transport Stream packet may assume two forms, either as a PES packet or a PSI section. PSI stands for Program Specific Information. Refer to Section 4.2.2 for specifics. A value of 1 by the payload_unit_start_indicator means that the payload of this packet starts with the first byte of either a PES packet or a PSI section. Otherwise, a value of 0 occurs.

Transport_priority--This 1-bit field has a value of 1 when this packet has a higher priority than other packets with the same PID but with their corresponding fields set to 0. Refer to PID definition below.

PID--This is a 13-bit field whose values serve to identify the type of data stored in the packet payload. It is stream-specific. For example, each elementary stream has its unique elementary_PID. As is clear later in the PSI section, Program Association Tables and Conditional Access Tables use additional flags such as version numbers to differentiate between multiple tables of the same type.

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Program Association Table</td>
</tr>
<tr>
<td>0x0001</td>
<td>Conditional Access Table</td>
</tr>
<tr>
<td>0x0002-0x00F</td>
<td>reserved</td>
</tr>
<tr>
<td>0x0010-0x1FF</td>
<td>purposes other than those mentioned above, such as network_PID, elementary_PID, etc.</td>
</tr>
<tr>
<td>0x1FF</td>
<td>Null packet</td>
</tr>
</tbody>
</table>

Transport_scrambling_control--This 2-bit field indicates the scrambling mode of the Transport Stream packet payload. The packet header, which includes bits representing
the sync_byte to those representing the continuity_counter, and the adaptation_field are not to be scrambled.

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>not scrambled</td>
</tr>
<tr>
<td>01</td>
<td>user defined</td>
</tr>
<tr>
<td>10</td>
<td>user defined</td>
</tr>
<tr>
<td>11</td>
<td>user defined</td>
</tr>
</tbody>
</table>

**Adaptation_field_control**—This 2-bit field indicates the presence of the adaptation field as well as the payload according to the following table.

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>reserved for future use by ISO/IEC</td>
</tr>
<tr>
<td>01</td>
<td>no adaptation_field, payload only</td>
</tr>
<tr>
<td>10</td>
<td>adaptation_field only, no payload</td>
</tr>
<tr>
<td>11</td>
<td>adaptation_field followed by payload</td>
</tr>
</tbody>
</table>

**Continuity_counter**—This 4-bit field serves as a counter for packets of the same PID. It wraps around to 0 after reaching its maximum value of 15. This counter does not increment when the adaptation_field_control is set to '00' or '10'. Duplicate packets have their continuity_counters set to the same values. Values for continuity_counters of null packets are undefined.
Data_byte—This is the payload of a Transport Stream packet. N is equivalent to 184 minus the number of bytes in the adaptation_field. 184 is from 188 minus the 4-byte packet header. The payload of a Transport Stream packet can contain either data from PES packets, PSI sections, or void in the case of a null packet.

4.2.2 Program Specific Information

Program Specific Information, or PSI is one of the two types of data carried in the payload of a Transport Stream packet. It contains four table structures, Program Association Table(PAT), Program Map Table(PMT), Network Information Table(NIT) and Conditional Access Table(CAT).

<table>
<thead>
<tr>
<th>Structure Name</th>
<th>Stream Type</th>
<th>PID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Association Table</td>
<td>ITU-T Recommendation</td>
<td>0x00</td>
<td>Associates Program Number and Program Map Table PID</td>
</tr>
<tr>
<td></td>
<td>H.222.01ISO/IEC 13818-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Map Table</td>
<td>ITU-T Recommendation</td>
<td></td>
<td>Specifies PID for components of one or more programs</td>
</tr>
<tr>
<td></td>
<td>H.222.01ISO/IEC 13818-1</td>
<td>Assigned in PAT</td>
<td></td>
</tr>
<tr>
<td>Network Information Table</td>
<td>Private</td>
<td>Assigned in PAT</td>
<td>Physical network parameters such as FDM frequencies, etc.</td>
</tr>
<tr>
<td>Conditional Access Table</td>
<td>ITU-T Recommendation</td>
<td>0x01</td>
<td>Associates one or more EMM streams</td>
</tr>
<tr>
<td></td>
<td>H.222.01ISO/IEC 13818-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: PSI Table Structures

Instead of being packetized into PES packets first, these tables are segmented into sections before being inserted into the payload of Transport Stream packets. Hence, PAS, PMS, NIS and CAS correspond to Program Association Section, Program Map Section, Network Information Section and Conditional Access Section respectively. Each section can be of variable length, and starts with a PSI pointer. A PSI pointer is an 8-bit field which indicates the number of bytes following the pointer field up to but not including
the first byte of the first section that is present in the payload of a Transport Stream packet. The syntax for sections of different table structures are somewhat different from each other; each, therefore, warrants individual attention.

A PAT provides the association information between a program_number and the PID value of the Transport Stream packets which carry the program definition.

```
program_association_section()
{
  table_id
  section_syntax_indicator
  '0'
  reserved
  section_length
  transport_stream_id
  reserved
  version_number
  current_next_indicator
  section_number
  last_section_number
  for (i=0; i<N; i++)
  {
    program_number
    reserved
    if (program_number=='0')
    {
      network_PID
    } else
    {
      program_map_PID
    }
  }
  CRC_32
}
```

*Figure 11: Program Association Section Syntax*
**table_id**—This 8-bit field takes on values according to the table below.

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox00</td>
<td>program_association_section</td>
</tr>
<tr>
<td>Ox01</td>
<td>conditional_access_section</td>
</tr>
<tr>
<td>Ox02</td>
<td>Transport_Stream_program_map_section</td>
</tr>
<tr>
<td>0x03-0x3F</td>
<td>ITU-T Recommendation H.222.01</td>
</tr>
<tr>
<td></td>
<td>ISO/IEC 13838 reserved</td>
</tr>
<tr>
<td>0x40-0xFE</td>
<td>User private</td>
</tr>
<tr>
<td>0xFF</td>
<td>forbidden</td>
</tr>
</tbody>
</table>

**section_syntax_indicator**—This 1-bit field is set to ‘1’.

**section_length**—This 12-bit field specifies the number of bytes immediately following this field up until and including the CRC_32 field. The first two bits of this field should be set to ‘00’, thus yielding a maximum section_length value of 1021. The maximum number for a PAS, therefore, is 1024 bytes.

**transport_stream_id**—This 16-bit field serves to uniquely identify this stream within the network. It is user-defined.

**version_number**—This 5-bit field identifies the version of the PAT. Whenever there is a change in the program definition, the version_number should increment by 1.

**current_next_indicator**—The PAT is currently applicable if this 1-bit field has a value of ‘1’; not applicable otherwise.

**section_number**—This 8-bit field serves as a counter. Starting from 0x00, it increments by 1 with each additional section from the same PAT.

**last_section_number**—This 8-bit field specifies the number of the last section of the PAT.
**program_number**--This 16-bit field, if set to 0x00, indicates the presence of a network_PID. Otherwise, it provides a one-to-one association to the PID of the Transport Stream packet which contains the program definition.

**network_PID**--This 13-bit field specifies the PID of the Transport Stream packet which contains the NIT.

**program_map_PID**--This 13-bit field specifies the PID of the Transport Stream packet which contains the program definition.

**CRC_32**--This 32-bit field is a check for the integrity of the PAS.

A PMT contains a complete collection of program definitions for a Transport Stream. A program definition is the mapping of one program number to one program element. One program number can have multiple program elements. Examples of program elements include elementary video and audio streams.
While many fields of PMT and PAT have the same functions, `section_number` and `last_section_number` have different applications in each. In PMT, these two fields are not used and are set to 0x00. In replacement, the `program_number` field is used to identify sections. Section 4.3.3 discusses implementation issues stemming from this special case.

**PCR_PID**—This 13-bit field indicates the PID of the Transport Stream which
contains the PCR fields valid for this program. PCR, short for Program Clock Reference, provides timing information to the decoder.

**program_info_length**—This 12-bit field specifies the number of bytes for the descriptor immediately following this field. The first two bits of this field is set to '00'.

**stream_type**—This 8-bit field specifies the type of program element according to the table below. The table below is here for the sake of completeness. Even though Phase I of this research dives into decoding of MPEG header information, stream-level analysis is outside of the scope of phase II.

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>ITU-TISO/IEC Reserved</td>
</tr>
<tr>
<td>0x01</td>
<td>ISO/IEC 11172 Video</td>
</tr>
<tr>
<td>0x02</td>
<td>ITU-T Rec. H.262</td>
</tr>
<tr>
<td>0x03</td>
<td>ISO/IEC 11172 Audio</td>
</tr>
<tr>
<td>0x04</td>
<td>ISO/IEC 13818-3 Audio</td>
</tr>
<tr>
<td>0x05</td>
<td>ITU-T Rec. H.222.0</td>
</tr>
<tr>
<td>0x06</td>
<td>ITU-T Rec. H.222.0</td>
</tr>
<tr>
<td>0x07</td>
<td>ISO/IEC 13522 MHEG</td>
</tr>
<tr>
<td>0x08</td>
<td>ITU-T Rec. H.222.0</td>
</tr>
<tr>
<td>0x09</td>
<td>ITU-T Rec. H.222.1</td>
</tr>
<tr>
<td>0x0A</td>
<td>ISO/IEC 13818-6 type A</td>
</tr>
<tr>
<td>0x0B</td>
<td>ISO/IEC 13818-6 type B</td>
</tr>
<tr>
<td>0x0C</td>
<td>ISO/IEC 13818-6 type C</td>
</tr>
<tr>
<td>0x0D</td>
<td>ISO/IEC 13818-6 type D</td>
</tr>
<tr>
<td>0x0E</td>
<td>ISO/IEC 13818-1 auxiliary</td>
</tr>
<tr>
<td>0x0F-0x7F</td>
<td>ITU-T Rec. H.222.0</td>
</tr>
<tr>
<td>0x80-0xFF</td>
<td>User Private</td>
</tr>
</tbody>
</table>

Table 3: Stream Types
**elementary_PID**--This 13-bit field specifies the PID of the Transport Stream which carries the program element.

**ES_info_length**--This 12-bit field specifies the number of bytes for the descriptor associated with this program element.

**descriptor**--These are structures that extend the program definitions.

The following diagram illustrates the mapping association between PAT and PMT.

![Diagram](image-url)

*Figure 13: Mapping Association between PAT and PMT*
A Conditional Access Table associates between Conditional Access systems with their EMM streams. EMM stands for Entitlement Management Messages. These messages specify the level of authorization to the decoders. The terminology here is the same as those in PAT. Finally, a Network Information Table provides physical network information which are user-specific. Its section follows the structure of a private section. Since data carried within a NIT is private, this research does not have the necessary resources to attempt to deal with the specifics of it.

```
CA_section()
{
    table_id
    section_syntax_indicator
    reserved
    section_length
    reserved
    version_number
    current_next_indicator
    section_number
    last_section_number
    for (i=0;i<N;i++)
    {
        descriptor()
    }
    CRC_32
}

CA_descriptor()
{
    descriptor_tag
    descriptor_length
    CA_system_ID
    reserved
    CA_PID
    for (i=0;i<N;i++)
    {
        private_data_byte
    }
}
```

```
private_section()
{
    table_id
    section_syntax_indicator
    private_indicator
    reserved
    private_section_length
    if (section_syntax_indicator==0)
    {
        for (i=0;i<N;i++)
        {
            private_data_byte
        }
    }
    else
    {
        table_id_extension
        reserved
        version_number
        current_next_indicator
        section_number
        last_section_number
        for (i=0;i<private_section_length-9;i++)
        {
            private_data_byte
        }
        CRC_32
    }
}
```

Figure 14: Conditional Access Section and Private Section Syntax
Conditional Access Descriptor Syntax
4.2.3 Packetized Elementary Stream Syntax

```c
PES_packet() {
    packet_start_code_prefix
    stream_id
    PES_packet_length
    if (stream_id=program_stream_map, padding_stream, private_stream_2, EGM, EMM, program_stream_directory or DSMCC_stream, ITU-T Rec. H.222.1 type E_stream) {
        '10'
        PES_scrambling_control
        PES_priority
        data_alignment_indicator
        copyright
        original_or_copy
        PTS_DTS_flags
        ESCR_flag
        ES_rate_flag
        DSM_trick_mode_flag
        additional_copy_Info_flag
        PES_CRC_flag
        PES_extension_flag
        PES_header_data_length
        if (PTS_DTS_flags=='10' or '11') {
            PTS_DTS_block
        }
        if (ESCR_flag) {
            ESCR_block
        }
        if (ES_rate_flag) {
            ES_rate_block
        }
        if (DSM_trick_mode_flag) {
            DSM_trick_mode_block
        }
        if (additional_copy_Info_flag) {
            additional_copy_Info_block
        }
        if (PES_CRC_flag) {
            PES_CRC_block
        }
        if (PES_extension_flag) {
            PES_extension_block
        }
        for (i=0;i<N;i++) {
            stuffing_byte
        }
        for (i=0;i<N2;i++) {
            PES_packet_data_byte
        }
    } else if (stream_id=padding_stream) {
        for(i=0;i<PES_packet_length;i++) {
            PES_packet_data_byte
        }
    } else if (stream_id=padding_stream) {
        for (i=0;i<PES_packet_length;i++) {
            padding_byte
        }
    }
}

Figure 15: PES Packet Syntax

packet_start_code_prefix--This 24-bit field has a fixed value of 0x000001.
**PES_packet_length**—This 16-bit field specifies the number of bytes in a PES packet starting from the first byte immediately following this field.

**PES_scrambling_control**—This 2-bit field indicates the scrambling mode for the PES packet payload data. Refer to *Transport_scrambling_control* for mode definitions.

**PES_priority**—This 1-bit field indicates the priority of payload data carried within PES packets. A value of 1 indicates higher priority.

**data_alignment_indicator**—A value of 1 in this 1-bit field indicates that video start code or audio syncword immediately follows the packet header.

**copyright**—This 1-bit field specifies copyright protection with a value of 1 and otherwise with a value of 0.

**original_or_copy**—This 1-bit field indicates the payload data as original with a value of 1, and otherwise with a value of 0.

**PTS_DTS_flags**—This 1-bit field indicates the presence of the *PTS_DTS_block*.

**ESCR_flag**—This 1-bit field, with a value of 1, indicates the presence of the *ESCR_block*. It contains timing information for each elementary stream.

**ES_rate_flag**—This 1-bit field indicates the presence of the *ES_rate_block*.

**DSM_trick_mode_flag**—This 1-bit field indicates the presence of the *DSM_trick_mode_block* with a value of 1.

---

1. Refer to Appendix C for an expanded description.
**additional_copy_info_flag**—This 1-bit field indicates the presence of the `additional_copy_info_block` with a value of 1.

**PES_CRC_flag**—This 1-bit field indicates the presence of the `PES_CRC_block` with a value of 1.

**PES_extension_flag**—This 1-bit field indicates the presence of the `PES_extension_block` with a value of 1.

**PES_header_data_length**—This 8-bit fields indicates the number of bytes containing optional fields and stuffing in the PES packet header. The preceding eight bits contain flag fields which indicate the presence of these optional fields.

**marker_bit**—This 1-bit field has a value of 1.

### 4.2.4 Adaptation Field Syntax

```c
adaptation_field() {
    adaptation_field_length
    if (adaptation_field_length>0) {
        discontinuity_indicator
        random_access_indicator
        elementary_stream_priority_indicator
        PCR_flag
        OPCR_flag
        splicing_point_flag
        transport_field_extension_flag
        if (PCR_flag=='1') {
            PCR_block
        }
        if (OPCR_flag=='1') {
            OPCR_block
        }
        if (splicing_point_flag=='1') {
            splicing_block
        }
        if (transport_private_data_flag=='1') {
            transport_private_data_block
        }
        if (adaptation_field_extension_flag=='1') {
            adaptation_field_extension_block
        }
        for (i=0;i<N;i++) {
            stuffing_byte
        }
    }
}
```

Figure 16: Transport Stream Adaptation Field Syntax
adaptation_field_length--This 8-bit field indicates the number of bytes for the adaptation field starting from the first byte immediately after this field.

discontinuity_indicator--With a value of 1, this 1-bit field indicates either a system time-base discontinuity when the TS packet has a PCR_PID, or a continuity_counter discontinuity otherwise. A system time-base discontinuity indicates that the next TS packet containing a PCR_PID shall contain the new system time-base. A continuity_counter discontinuity occurring in a TS packet containing elementary stream data indicates that the payload of the next TS packet with the same PID starts with the first byte of an elementary access unit. An elementary access unit for video is the first byte of a video sequence header, and of an audio frame for audio. A continuity_counter discontinuity occurring in a PSI packet is followed by the transmission of new program definitions carried in the appropriate TS program map sections.

random_access_indicator--A value of 1 in this 1-bit field indicates that the next TS packet with the same PID carries the first byte of an elementary access unit for video and audio.

elementary_stream_priority_indicator--This 1-bit field has a value of 1 when indicating a higher priority for a TS packet among those of the same PID.

PCR_flag--A value of 1 in this field indicates the presence of the PCR_block. The PCR_block contains Program Clock Reference, PCR, which indicates time of arrival of certain timing fields at the input of the system target decoder.

1. Refer to Appendix B for an expanded description.
**OPCR_flag**—A value of 1 in this 1-bit field indicates the presence of the OPCR_block. The OPCR_block contains timing information which is useful for the reconstruction of Transport Streams.

**splicing_point_flag**—A value of 1 in this 1-bit field indicates the presence of the splicing_block, which specifies the occurrence of a splicing point. Splicing refers to the injection of data into existing Transport Streams. The end result is very much like TV commercials in between regular programs.

**transport_private_data_flag**—A value of 1 in this 1-bit field indicates the presence of the transport_private_data_block, which contains private data bytes.

**adaptation_field_extension_flag**—A value of 1 in this 1-bit field indicates the presence of the adaptation_field_extension_block.

4.3 Design and Implementation

This chapter discusses design and implementation details for Phase II of this research.

4.3.1 Functionality Specification

**Input:** Transport Stream data encoded according to ISO/IEC 13818-1 specifications

**Output:** PSI data and PID-specific elementary stream data

---

1. Refer to Appendix B for expanded description.
4.3.2 Main Data Structure

*TRANSPORT_PACKET*—data structure to store one Transport Stream packet

*PES_PACKET*—data structure to store one PES packet

*PSI*—data structure to store one PSI section

*PAS, PMS, CAS, NIS*—data structures to store the appropriate PSI section

*ADAPTATION_FIELD*—data structure to store the adaptation field of a TS packet

4.3.3 Main Routines

The following block diagram illustrates the scope of the implementation for Phase II of the research.
Figure 17: Scope of Implementation in Phase II

decodeTransport

Specification:

//
// function:
//   Decode a Transport packet.
// entry:
//   buffer containing 188 bytes of data; 188 is the size of one TS packet.
//   TRANSPORT_PACKET structure
This routine decodes Transport Stream packets one at a time. Each input packet is decoded and mapped into a `TRANSPORT_PACKET` structure. In the case when the user specifies a PID or PSI to decode, `decodeTransport` decodes only those packets of interest. In considering the overall implementation architecture, there were two choices. One scheme is to not differentiate the various PSI table structures in `decodeTransport`, and have a function, `reassemblePSI`, perform the differentiation. The other way is to perform the PSI-level differentiation in `decodeTransport`. While the first scheme is clean in design, the latter is much more practical since identifying PSI sections requires examining PSI-level information. While identifying data of a PES packet from that of a PSI section follows the simple procedure of checking the TS packet PID, the same procedure runs aground when attempting to further classify PSI sections. PID of a TS packet does not uniquely identify payload with PMT data. PAT contains the mapping information between TS packet PID's and those that carry PMS. To bypass this problem in `decodeTransport`, this implementation utilizes two additional parameters, `program_map_PID` and ` program_map_PID_current` within the `PSI` structure. The first field contains a list of `program_map_PID` from PAT, and the second indicates whether the list is current or not.
As an additional feature, *decodeTransport* keeps count of distribution bandwidth of different types of data present at the Transport level. This feature is scalable to include identification of data at the PES, PSI and even Elementary Stream levels. Record of such data distribution might be useful. For example, the Systems specification has compiled information on PAT bandwidth usage, and plots it against total bandwidth.

**reassemblePES:**

Specification:

```
// function:
// Assembles PES packets from segments carried by transport packets.
// entry:
// TRANSPORT_PACKET structures
// an array of PES_PACKET structures
// return:
// error
// append successful
// assemble entire packet successful
```

This routine takes in a mapped `TRANSPORT_PACKET` structure whose payload contains data from a PES packet, and maps the payload to the `packet_buf` of a `PES_PACKET` structure. More specifically, this `PES_PACKET` structure is a member of an array of such structures. The TS packet PID serves as an index into this array. Refer to comments in the attached code for exact implementation. Decoding of a PES packet commences only when a PES packet is assembled completely. In general, the `PES_packet_length` field specifies the length of the PES packet. It’s straightforward to detect the completion of assembling a PES packet using this field. Instead of allocating for the maximum packet length statically, this routine chooses to dynamically allocate memory once the packet length has been determined in `decodeTransport`. Reallocation

57
of memory is only necessary when the incoming packet is of greater length than the previous one. When the payload of a PES packet carries data from a video elementary stream, however, the \textit{PES\_packet\_length} need not be defined. To detect the completion of an assemblage in this case, current implementation utilizes the presence of the next \textit{payload\_unit\_start\_indicator}. When \texttt{reassemblePES} detects the next \textit{payload\_unit\_start\_indicator}, and the previous PES packet contains video stream data and has an unbounded length, the routine signals the completion of the assemblage.

\begin{verbatim}
decodePES:
// function:
// decodes one PES packet at a time.
// entry:
// PES\_PACKET
// PID
// return:
// error
// decode_complete
\end{verbatim}

This routine decodes a reassembled PES packet, and maps the PES packet into a \texttt{PES\_PACKET} structure. The mapping process is straightforward as is evident from the attached coding.

Specifically, this routine performs a reconstruction of Elementary Stream data carried within the payloads of PES packets. It also compiles stream-specific control information which enables proper decoding of these Elementary Streams. Limited by time and resource, interfacing with decoders at this point is beyond the scope of this research.

\begin{verbatim}
reassemblePAS, reassemblePMS, reassembleCAS:
// function:
\end{verbatim}
Reassembles a complete PAS(reassemblePAS), PMS(reassemblePMS) or CAS(reassembleCAS)

// entry:
// TRANSPORT_PACKET structure
// PSI structure
// return:
// error
// assemble Section complete
// nothing assembled, duplicate

Similar in functionality to reassemblePES, these three routines reassemble the appropriate PSI tables. Each also considers special cases such as section continuity, duplicate packets and stuffing bytes. As mentioned earlier in the syntax analysis, the section_number and last_section_number fields are all set to 0x00. The field program_number is user-defined, so cannot serve as index into the array. A combination of that with program_map_PID_count serves the purpose. Examination of current testing data reveals that all three PSI tables here are small enough to fit into one section.

decodePAS, decodePMS, decodeCAS:

// function:
// Decodes a PAT(decodePAS), PMT(decodePMS), CAT(decodeCAS) section by section.
// entry:
// pointer to PSI data structure
// pointer to PAT(decodePAS), PMT(decodePMS), CAT(decodeCAS) structure within the PSI data structure
// return:
// error
// section complete
// table complete

Similar in functionality as decodeTransport, these routines decode the appropriate PSI table structures section by section. As mentioned earlier, decodePAS updates the two parameters program_map_PID and program_map_PID_current. PAS carries both
network information as well as program map information. Even though current implementation does not consider NIT, it retains scalability by keeping track of TS packets which carry NIT sections through the above two parameters. Note, decodeCAS has not been fully implemented. CAT contains user-defined private data. Testing data obtained for this research does not contain any CAT, thus making the tasking of implementing and testing the above routine unfeasible. This research recognizes this shortcoming and addresses this issue in its architectural considerations. Refer to conclusion in Section 4.5 for more detail.

4.4 Simulation Results

The simulation results include data from the input Transport Stream and the final output. Data from intermediate stages such as PES packet data and PSI section data are also included here for reference purposes.

Final output:

PID in decimal is -7
program number 1: program_map_PID 4217
elementary PIDs for program number 1:
4130
4131
current transport packet number is 324455
current null packet number is 8184
current psi packet number is 128

Transport Stream data:

0EB0: 47400010 0000B00D 0000C100 000001F0
0EC0: 79B36D35 90FFFFFF FFFFFFFF FFFFFFFF
0ED0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0EE0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0EF0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F00: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F10: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F20: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F30: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F40: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F50: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0F60: FFFFFFFF FFFFFFFF FFFFFFFF 47507910
0F70: 0002B017 0001C100 00F022F0 0002F022
0F80: F00003F0 23F00070 B15F12FF FFFFFFFF
0F90: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FA0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FB0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FC0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FD0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FE0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0FF0: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
1000: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
1010: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
1020: FFFFFFFF FFFFFFFF 47502310 000001C0

PES data:

0000: 000001C0 0C328180 05210001 0001FFFF
0010: A0005555 44554455 55434433 446E46DB
0020: 24900000 00000000 00FAAAA B0EFA0AF
0030: AFFF7587 5869A820 8A296592 9966AE79
0040: AB9E9B2A A6DA6DAE BA2EAE9B ABAEBA6C
0050: A6CAEBA9 DAADBAFB AFAEFAF0 7DF7DF7D
0060: F7DF7DF7 DF7BDEF7 BDDDDDD 6C5B1DDD
0070: DDDDDDDD D6C6DAD8 B636DB68 62D8DB6D
0080: AD8B636D B6DB6DB5 AD6BEF44 FBD13EFB
0090: AFBEAEF1 07C20EEE EEEB62D8 EEECCEEF
00A0: 0EF0B636 D6C5B1B6 DB5B16C6 DB6D6C5B
00B0: 1BD6B6DB 6DAD6B55 E957B5A2 0471811C
00C0: 677BCEF7 8A48A43D 8F655510 29E59E60
00D0: D340A3A8 E0D06BC4 F1586C2A 42905D2E

Elementary Stream data:

0000: FFFDA000 55554455 44555543 4433446E
0010: 46DB2490 00000000 000000F AAAAB0EF
0020: A0FAAFFF 75875869 A8208A29 65929966
0030: AE79AB9E 9B2A6DA 6DAEBA2E AE9BAABE
0040: BA6CA6CA EBAADAAD BAFAFAAE FAF07DF7
0050: DF7DF7DF 7DF7DF7B DEF7BDDDDDDDDC5B
0060: 1DDDDDDDD DDDDD6C6 DAD8636 DB6B62D8
0070: DB6AD8DB 636DB6DB 6DB5AD6B EF44FBD1

4.5 Conclusion

The information already gathered here makes implementing the second of the decoding operations easy. The timing considerations such as PCR and ESCR pose a major concern though when programs are regrouped within Transport Streams.
In comparison, future work involving the third of the decoding operations, conversion between Transport Streams and Program Streams carries greater practical implication. Some group projects in the Community-Based Customer Access Solution Department involve interface with and use of MPEG drivers. The MPEG driver from VisualCircuit only decodes Transport Streams and so do the vendors surveyed in Appendix D¹. Since Program Stream presents the more desirable format for storage, life is much easier if there is a conversion implementation which gaps the bridges between the two.

Again, due to the lack of actual network data, current implementation is tested only against stored data. This research anticipated the difficulty in obtaining adequate amount of testing data, and strives to provide a framework for easy integration, modification and scalability. When the resources become available, the natural step is to develop an interface to decoders for the transmission of control information.

In conclusion, this implementation serves primarily as a learning platform to thoroughly analyze the Systems specifications while at the same time making a substantial contribution to an emerging field. In these capacities, this research has fulfilled its objectives.

¹. Refer to Appendix D for a discussion of work in other projects which are relevant to this research.
Appendix A. Discrepancies

The following example demonstrates the ambiguity associated with locating synchronization words in MPEG-II, Layer-III audio files.

A portion of a MPEG-II, Layer-III file in HEX representation:

```
0000: FFF340C4 0012414D 6C794630 00D5B9F6
0010: 002D6600 02241C2E 89A66104 1C993B22
0020: 4C980C9E 98A07002 02C06174 08112410
0030: 3C985B02 001068FE 22C2016 B3000424
0040: 993A2699 84107264 D89A6600 0838385E
0050: 934CE7F7 17900008 0302B160 44510830
0060: C0300302 E5116E28 FFF342C4 121572A6
0070: 7A559168 2120C8D4 716D193B F052C4C0
0080: 2D67536E 24E2083E AA8F934A 44A1A57F
0090: F34A0BBA BFD23464 16A4BFFF D14DD04D
00A0: CC168FFF FFE92082 8D2EA40C D2362E7F
00B0: FFFFD03 EB1E22AA FC000555 88797856
00C0: 85457227 24724B60 00ACE447 23BCB124
00D0: 67FFF340 C4181600 BEA31F8F 49247FDF
```

format: MPEG-II, layer 3
bitrate: 32kbits/s
sampling_frequency: 22050 samples/s
frame_size: 1152 samples
padding: alternates between frames; 1st frame has no padding.

first frame:

```
frame_size * bitrate / sampling_frequency = 208 bytes / frame
```

208 bytes/frame indicates 2nd frame should start at 0xD0, but the second frame actually starts at 0x68.

For an MPEG-II, Layer-III file, there is only one granule per frame which translates into 576 samples per frame. Therefore, instead of 208 bytes, each frame contains only 104 bytes. This deviation is not clearly stated in the standard specifications. Contradicting outputs of frame number and total file size by existing decoders might be caused by the above confusion.
Appendix B. Transport Stream Adaptation field

**PCR_block:**
- program_clock_reference_base
- reserved
- program_clock_reference_extension

Calculations of PCR, PCR_ext and PCR_base follow the formulas below.

\[
PCR(i) = PCR_{base}(i) \times 300 + PCR_{ext}(i)
\]
\[
PCR_{base}(i) = \left((system\_clock\_frequency \times t(i)) \div 300\right) \% 2^{13}
\]
\[
PCR_{ext}(i) = \left((system\_clock\_frequency \times t(i)) \div 1\right) \% 300
\]

\(i\) stands for the \(i\)th byte; the \(i\)th byte for the value encoded in the PCR field stands for the byte containing the last bit of the *program_clock_reference_base* field. PCR_base is in units of \(1/300\) times the system clock frequency which is 27MHz. The difference in time specified by two successive *PCR_blocks* determines the transport rate for that segment of the Transport Stream.

**OPCR_block:**
- original_program_clock_reference_base
- reserved
- original_program_clock_reference_extension

Calculations involved in the *OPCR_block* follow the exact formulas as those for *PCR_block*.

**splicing_block:**
- splice_countdown
A positive value in the `splice_countdown` field indicates number of Transport Stream packets until a splicing point is reached; a negative value indicates number of Transport Stream packets since a splicing point. Null packets, duplicates and packets without payload are not counted.

```
transport_private_data_block:
transport_private_data_length
for(i=0;i<transport_private_data_length;i++) {
    private_data_byte
}
```

This block carries private data which is not specified by ISO/IEC. The `transport_private_data_length` specifies the total number of bytes carried in this block immediately following this field.
The adaptation_field_extension_length field specifies the number of bytes in this block. A value of 1 in the ltw_valid_flag means that ltw_offset field has a valid value.

The ltw_offset field contains the upper bound in deviation between time of intended arrival of data, \( t \) and of actual arrival, \( t_i \) at the T-STD as indicated in the formula below.

\[
t_i(t) - t(i) = ltw\_offset
\]

\( ltw\_valid \) stands for Legal Time Window, and is in units of \((300/system\_clock\_frequency)\); \( i \) stands for the \( i \)th byte of data entering the T-STD.

The piecewise_rate is only defined when both \( ltw\_valid\_flag \) and \( ltw\_offset \) fields are defined. If defined, this field specifies a hypothetical bit rate based on the above \( ltw\_offset \) for Transport Stream packets of the same PID but without their associated \( ltw\_valid\_flag \) and \( ltw\_offset \) fields.
The `splice_type` field is 0 when the Transport Stream packets carry audio data. It specifies the `splicing_decoding_delay` and `max splice rate` parameters for Transport Stream packets carrying video data. The three fields of `DTS_next_AU` combine to specify the decoding time of the first access unit following a splicing point.
Appendix C. Packetized Elementary Stream

\[PTS_{DTS}\_block: \]
\[\text{if (PTS}_{DTS}\_flags=='10') \{ \]
\[PTS[32..30] \]
\[\text{marker\_bit} \]
\[PTS[29..15] \]
\[\text{marker\_bit} \]
\[PTS[14..0] \]
\[\text{marker\_bit} \]
\[\text{if (PTS}_{DTS}\_flags=='11') \{ \]
\[PTS[32..30] \]
\[\text{marker\_bit} \]
\[PTS[29..15] \]
\[\text{marker\_bit} \]
\[PTS[14..0] \]
\[\text{marker\_bit} \]
\[DTS[32..30] \]
\[\text{marker\_bit} \]
\[DTS[29..15] \]
\[\text{marker\_bit} \]
\[DTS[14..0] \]
\[\text{marker\_bit} \]
[\}]

Presentation Time Stamp indicates the time of presentation in the system target decoder. The time encoded in the field refers to the first access unit commencing in the PES packet. Decoding Time Stamp indicates the decoding time of the first access unit in the system target decoder.

\[ES\_rate\_block: \]
\[\text{marker\_bit} \]
\[ES\_rate \]
\[\text{marker\_bit} \]

ES\_rate specifies the rate at which the system target decoder receives bytes of a PES stream.
ESCR_block:
reserved
ESCR_base[32..30]
marker_bit
ESCR_base[29..15]
marker_bit
ESCR_base[14..0]
marker_bit
ESCR_extension
marker_bit

This block contains information on the Elementary Stream Clock reference. Calculations of ESCR, ESCR_base and ESCR_ext follow those of PCR, PCR_base and PCR_ext above.

DSM_trick_mode_block:
trick_mode_control
if (trick_mode_control==fast_forward) {
  field_id
  intra_slice_refresh
  frequency_truncation
} else if (trick_mode_control==slow_motion) {
  rep_cntrl
} else if (trick_mode_control==freeze_frame) {
  field_id
  reserved
} else if (trick_mode_control==fast_reverse) {
  field_id
  intra_slice_refresh
  frequency_truncation
} else if (trick_mode_control==slow_reverse) {
  rep_cntrl
} else {
  reserved
}
The trick_mode_control indicates the five following modes: fast forward, slow motion, freeze frame, fast reverse and slow reverse.

```
additional_copy_info_block:
  marker_bit
  additional_copy_info
```

This block contains additional private copyright information.

```
PES_CRC_flag_block:
  previous_PES_packet_CRC
```

This field contains the CRC value that yields a zero output after processing the data bytes of the previous PES packet exclusive of its header. The polynomial for calculation of the CRC value is:

```
x^{16} + x^{12} + x^5 + 1
```
The field `pack_header()` contains an ISO/IEC 11172 pack header or a Program Stream pack header the length of which is specified in the `pack_field_length` field. The field `program_packet_sequence_counter` provides functions similar to a continuity counter, except it is program-specific, i.e. specific to one program definition or one Program Stream. `MPEG1_MPEG2_identifier` is 1 when data is from ISO/IEC 11172 Systems Stream, or 0 when data is from Program Stream which is specific to MPEG-II.
C_STD_buffer_scale and C_STD_buffer_size are only defined when PES packets are carried within Program Streams.
Appendix D. Related Work

One of the group projects focused on the design and implementation of a computer architecture for the delivery of multimedia data from a central head end to locations scattered across the country. The design and implementation at a local site involved the development of a user-interface and a serving environment. Through the user-interface, an administrator could edit and administer presentations which consisted of MPEG multimedia data. The user-interface then conveyed these commands from the administrator to the serving environment. In this project, a MPEG driver from VisualCircuit was used to serve the data either to a TV monitor or overhead projector.

/* MPMPEG -- BlueSky, MPEG Digital Video Play
 ** written by R A Flavin, RAF Dec 98
 ** modified by Y. Feng, YF Dec 98
 **
 ** Register -- Add a selection(song) to the Jukebox Playlist
 **
 ** Change History:
 ** 10 Dec 98 RAF - New
 ** 17 Dec 98 YF -- Add mpg card control handles
 ** 18 Dec 98 YF -- Append VIDDIR only when when VID contains file name and not full file path.
 */

#define _MT

#include "granddb.h"
#include <stdio.h>
#include <string.h>

/* YF header files for access to 4Reeltime Card */
#include "resource.h"
#include <stdlib.h>
#include "4reelapi.h"

/* YF definitions for 4Reeltime Card */
#define MAXBOARD 4
#define MAXCHAN 4

/* YF needed for mpgCards */
int mpeg2=1; /* True if 4reeltime pro; False if mpeg1 4reeltime/P */

/* YF parameters for 4ReelTime Card */
int numCards; /* number of MPG cards on bus */
int curr_Card=0; /* current Card number; 4ReelTime/P must start with 0 */
int numChan;
int curr_Chan=0;

/* YF 4Reeltime Card functions */
int ReplayFile(int chan);

int vidLoopRunning = 0; /* Global flag that vidLoop thread is started*/
#define MSGSIZE 256
char sharedCmd[MSGSIZE] = "unset";
char sharedResp[MSGSIZE] = "(first pass)\n";
char sharedOID[NAMESIZE]; /* Name of current video object */
GDblk sharedNB;
struct critlock vidLoopGate; /* Sig from DVP() to vidLoop to execute a cmd*/
struct critlock vidDoneGate; /* Signal from vidLoop that cmd is done */

int findSeg(GDblk*, char*, char*);
int setState(char*, char*, char*, char*);
void vidLoop(void);
void setTellTale(char*); /* From gchttp.c, set info about thread */
int prepSock(void);
#ifdef OPERATING_SYSTEM_NT
  #define tcperrno() WSAGetLastError()
#else
  #define tcperrno() errno
#endif

/* DVP -- Interface to Digital Video Controller background thread */
long DVP(
  char *arg, /* Pointer to argument string */
  long alen, /* Length of argument string */
  char *parm, /* Pointer to parameter string */
  long plen /* Length of parameter string */
)
{
  struct GST *pts = GUgetpts();
  
}
char buf[MSGSIZE];
int rc=0;

/* Start our associated video loop thread if not already going. */
if (vidLoopRunning==0) { /* If the vidLoop func is not running, start it */
   GUinitgate(&vidLoopGate, 1); /* Initialize gate in the blocked state */
   GUinitgate(&vidDoneGate, 1); /* Initialize gate in the blocked state */
   pgs->prunthread((int(*)(char*))vidLoop, 0);
}

/* Parse the command string by appending the args to the parms */
strcpy(sharedOID, pts->wrtoid);
sharedNB = pts->wrtnb;
if (plen && parm[0] == ' ') {parm++; plen--;}; /* Kill leading space */
GXstr(sharedCmd, sizeof(sharedCmd)-1, parm, plen);
if (alen) { /* If there are user supplied args, pass them now */
   sharedCmd[plen] = ' '; /* Append a space before the args */
   GXstr(sharedCmd+plen+1, sizeof(sharedCmd)-plen-1, arg, alen);
}
GUleavegate(&vidLoopGate); /* Signal the video loop to handle this */
GUentergate(&vidDoneGate); /* Wait for vidLoop to handle the command */
strcpy(buf, sharedResp); /* Save a copy of the command response */

GError((char*)O, (GDblk*)0, buf, (long)rc, pts->cmdname);
return 0;
} /* end of DVP */

/* Defines from OutlEntry.java: */
#define expandOffset 0
#define skipOffset 1
#define stopOffset 2
/* offset 3 not used yet */

int
setState(
   char *tag, /* Which type of data to be changed */
   char *segName, /* The abbreviated attribute name of video seg to change */
   char *val, /* The value the tag is to be set to */
   char *presp /* Ptr to buffer for response message */
) {
   GDblk onb, nb;
   char *pv; long vl;
   int rc = 0;
onb.value = 0;
if (GDgetw(&onb, sharedOID, strlen(sharedOID)) != 0) {
    sprintf(presp, "Can't get write access to '%s'.
"sharedOID);
    return 1;
}

nb = onb;
if ((rc = findSeg(&nb, segName, presp)) != 0) goto bail;

GDvalue(&nb, &pv, &vl);
if (strcmp(tag, "SKIP") == 0) {
    if (pv[skipOffset] == 'R') {
        sprintf(presp, "MPEG DVP: '%s' '%s' can't be skipped, it is required.
"sharedOID, segName);
        rc = 3; goto bail;
    }
    pv[skipOffset] = val[0];
} else { /* Handle changes to 'stopAfter' */
    char c = toupper(val[0]);
    if (c == 'F') c = ' '; /* 'F'alse means no stop, use a '.' symbol*/
    else if (c == 'T') c = 'S'; /* 'T'rue means stop, use 'S' for stop*/
    pv[stopOffset] = c;
}

GDput(&onb);
bail:
    GDrelease(&onb, 1);
    return rc;
} /* end of setState */

/* Find the named video segment */
int findSeg(
    GDblk *pnb,
    char *segName,
    char *presp
) {
    char buf[80];
    char *p, *q;

    strcpy(p=buf, segName);
    while (p && *p) { /* While there are more segment name fragment... */
        if (q = strchr(p, '.')) *q++ = '0'; /* Separate name fragments */
        printf(p,...
}
if (GDselect(pnb, "SEG")==0 || GDselect(pnb, p)==0) {
    sprintf(presp, "Failed to find 'SEG.%s'in '%s'(%.%s).\n", p,
        sharedOID, segName);
    return 2;
}

p = q;  /* Point to the next segment */

}  /* end of while more name frags */
return 0;
}  /* end of findSeg */

/* vidLoop -- Controls the MPEG video decoder card */
#define DEFAULTPORT 2233

void
vidLoop(
    void)
{
    int timeToGo = 0;  /* Play time in seconds */
    GDblk nb, xnb,xxnb;
    char cmd[16], arg1[80], arg2[80];
    char currentSelection[80] = "";
    char curr_File_Name[80];  /* VIDDIR Default dir for loading vid files */
    char curr_fid[80];  /* Full filename for selected mpg file */
    char curr_File_Path[80];  /* Full path/filename for selected mpg file */
    char *pv; long vl, temp_length;
    char *presp = sharedResp;  /* Return the response in the shared msg buffer */
    int sendSock;  /* Socket on which to send time/selection msgs */
    struct sockaddr_in toSA;  /* Socket address to send time/selection to */
    int rc;
    int file_load=0;  /* YF 1 file loaded; 0 otherwise. */
    /* Assuming utilizing one channel on one card */
    /* i.e. sequential access only */
    /* YF Define queue for playback list. */
    char *p_queue[100];
    int q_index = 0;

    vidLoopRunning = 1;  /* Signal that the video loop is running */

    setTellTale("vidLoop");
    memset(&toSA, 0, sizeof(toSA));
    toSA.sin_family = AF_INET;
    sendSock = prepSock();

/* YF Initialize MPG Card */
/* Establish connection to the kernel mode driver (loaded already).* /
if (mpgOpenDriver() == driver_not_found) {
    sprintf(presp,"Open Driver failed\n");
    goto bail;
}

/* YF Check number of cards available */
mpgCards(&numCards);
if (numCards == 0) {
    sprintf(presp, "No Cards detected\n");
    goto bail;
}

printf("Init Process finished. \n");

/* YF Record file path stored under attribute: VIDDIR */
/* Note: It might be more suitable to * /
/* store different modules under */
/* separate subdirectories. */
xnb=sharedNB;
if (GDselect(&xnb,"VIDDIR")) { /* See if there is a VIDDIR */
    GDvalue(&xnb,&pv,&vl);
    temp_length=vl;
    PStr(curr_File_Dir,sizeof(curr_FileDir), &pv, &vl);
    /* Make sure that the VIDDIR ends with a directory separator */
    printf(" VIDDIR length is %d. \n", temp_length);
    if (curr_File_Dir[temp_length-1]!=DIRSEP[0]) strcat(curr_File_Dir, DIRSEP);
} else {
    curr_File_Dir[0] = '\0';/* No default video file directory */
}
printf(" VIDLOOP using VIDDIR '%s'.\n",curr_File_Dir);

/* FOREVER LOOP, HANDLE COMMANDS, INTERRUPTS FROM THE CARD */
/* and TIMEOUTS */
while (1) {
    /* Wait for 1 sec, or for someone to send us a command */
    /* This wait function is effectively like a call to GUentergate,
    ** except that it has a timeout as well. */
    if ((rc=WaitForSingleObject((HANDLE)vidLoopGate.sem, 1000))
        ==WAIT_FAILED) {
        printf("WaitForSingleObject \p failed with %d.\n", &vidLoopGate.sem,
                GetLastError());
        break;
    }
    if (pgs->msgmode[3] & 0x40) DIE; /* Cause debugger to interrupt */
debugdo( printf("vidLoop wait ended %d, %d, %s\n", rc, GetLastError(),
    rc==WAIT_OBJECT_0?"WAIT_OBJECT_0": rc==
    WAIT_TIMEOUT?"WAIT_TIMEOUT":
    (rc==WAIT_ABANDONED?"WAIT_ABANDONED":("unknown")));
    debugdo( printf("last vidLoop cmd '%s',resp '%.*s'.\n",sharedCmd,
    strlen(sharedResp)-1, sharedResp);

    /* !!! Dummy time decremter (make it an incremter for now) */
    timeToGo++;

    if (rc == WAIT_TIMEOUT) {
        /* Send the time/selection message to the client */
        /* This only supports one client at a time, for now */
        char buf[128];
        sprintf(buf+4, "%s", currentSelection);
        *(long*)buf = htonl(timeToGo);

        /* Send a UTP message to the client */
        if (toSA.sin_addr.s_addr) { /* If there is a client, send to him */
            sendto(sendSock, buf, 4+strlen(buf+4), 0,
            (struct sockaddr*)&toSA,
            sizeof(toSA));
        }
    }
    continue; /* Wait again */
}
/* rc could also be WAIT_ABANDONED or WAIT_OBJECT_0 */
/* RECEIVED A COMMAND FROM THE USER, PROCESS IT */
/* Get the command token and first two arguments as strings */
pv = sharedCmd; vl = strlen(sharedCmd);
PLstr(cmd, sizeof(cmd), &pv, &vl);
PLstr(arg1, sizeof(arg1), &pv, &vl);
PLstr(arg2, sizeof(arg2), &pv, &vl);
printf("vidLoop cmd: '%s' '%s' '%s'\n",sharedOID, cmd, arg1, arg2);

    if (strcmp(cmd, "STOP") == 0) {
        /* Signal player to stop */
        /* YF Stop here will close the file */
        if (mpgStop(curr_card, curr_channel)!=no_error) {
            printf("Error Stopping file\n");
goto cmdDone;
        }
    }

    sprintf(presp, "Stopped.\n");
/* Close Driver */
/* gmp not behaving properly after exiting forever loop */
} else if (stricmp(cmd, "EXIT") == 0) {
    goto bail;
}
else if (stricmp(cmd, "CLOSE") == 0) {
/* Stop and Close file. */
    if (mpgStop(curr_card, curr_chan) != no_error) {
        printf("Error Stopping file\n");
        goto cmdDone;
    }
    if (mpgClose(curr_card, curr_chan) != no_error) {
        printf("Error Closing file\n");
        goto cmdDone;
    }
    file_load = 0;
    sprintf(presp, "Closed\n");
}
else if (stricmp(cmd, "GO") == 0) {
/* Signal player to start/resume playing */
/* YF if file not loaded, load it first and play */
    if (!file_load) {
        printf(" mpgLoad file '%s'.\n", curr_file_dir);
        if (mpgLoad(curr_card, curr_chan, curr_file_path) == file_not_found) {
            printf("File not found at location %s\n", curr_file_path);
            goto cmdDone;
        }
        file_load = 1;
    }
/* Play/Resume file */
    if (mpgPlay(curr_card, curr_chan) != no_error) {
        printf("Error playing file\n");
        goto cmdDone;
    }
    sprintf(presp, "Playing '%s'.\n", current_selection);
/* Record client's IP address so we can send him time/selection msgs */
}
else if (stricmp(cmd, "REPLAY") == 0) {
/* YF Rewind 1/10 of a minute because clips are short. */
    if (!file_load) {
        if (mpgLoad(curr_card, curr_chan, curr_file_path) == file_not_found) {
            printf("File not found when loading\n");
            goto cmdDone;
        }
    }
file_load=1;
}
if (ReplayFile(curr_Chан)==0) {
    printf("Error replaying file\n");
goto cmdDone;
}
/* UIF signal player to rewind a minute and start again */
sprintf(presp, "Replaying.\n");

/* Set Frame */
/* Does seem to have any effect */
} else if (stricmp(cmd, "SETFRAME")==0) {
    /* hard-coded for testing purposes */
    mpgSetCurrentFrame(curr_Card, curr_Chан, 300);
sprintf(presp, "SetFrame '%s'.\n", arg1);

/* Select a piece to play now and click GO to play immediately. */
/* or behaves as mpgLoadNext when not forced to play immediately. */
} else if (stricmp(cmd, "SELECT")==0) {
    nb = sharedNB;
    /* Get the name of the video segment to start */
    if ((rc = findSeg(&nb, arg1, presp)) != 0) goto cmdDone;
    /* YF Record the file name for current selection */
    /* and append to curr_File_Path */
    xnb=nb;
    if (GDselect(&xnb, "VID")==NULL) {
        goto cmdDone;
    }
    GDvalue(&xnb, &pv, &vl);
    PLstr(currfid, sizeof(currfid), &pv, &vl);

    /* Check if fid contains explicit file path */
    /* Unix path starts with '/' NT may start with \" or 'e:\" */
    ifdef OPERATING_SYSTEM_NT
    if (currfid[0]==DIRSEP[0]
        && (currfid[1]=='.'&& currfid[2]!=DIRSEP[0])) {
    #else /* UNIX style (no disk prefix) */
    if (currfid[0] == DIRSEP[0]) {
    #endif
        curr_File_Path[0] = '\0'; /* Absolute, don't prefix with VIDDIR */
    } else { /* This is a relative fid, prefix it with the VIDDIR */
        strcpy(curr_File_Path, curr_File_Dir);
    }
    strcat(curr_File_Path, curr_fid);
}

81
printf(" selected '%s', file'%s\n",arg1, curr_File_Path);

/* Record the current selection in persistent memory */
strcpy(currentSelection, arg1);

mpgLoadNext(curr_Card, curr_Chan, curr_File_Path);

/* Prepare response */
sprintf(presp, "Selected '%s'.\n",arg1);

/* Queue files to list */
} else if (stricmp(cmd, "QUEUE")==0) {
    p_queue[q_index]=(char*)malloc(80);
    nb = sharedNB;
    /* Get the name of the video segment to start */
    if ((rc = findSeg(&nb, arg1, presp)) != 0) goto cmdDone;
    /* YF Record the file name for current selection */
    /* and append to curr_File_Path */
    xnbe=nb;
    if (GDselect(&xnbe, "VID")==NULL) {
        goto cmdDone;
    }
    GDvalue(&xnbe, &pv, &vl);
    PLstr(curr_fid, sizeof(curr.fid), &pv, &vl);

    /* Check if fid contains explicit file path */
    /* Unix path starts with '/' NT may start with '@' or 'e:\'*/
    #ifdef OPERATING_SYSTEM_NT
        if (curr_fid[0]==DIRSEP[0] && (curr_fid[1]=='.'&&& curr_fid[2]!=DIRSEP[0])) {
            /* UNIX style (no disk prefix) */
            if (curr_fid[0] == DIRSEP[0]) {
                #endif
                p_queue[q_index][0] = '0'; /* Absolute, don't prefix with VIDDIR */
            } else { /* This is a relative fid, prefix it with the VIDDIR */
                strcpy(p_queue[q_index], currile_Dir);
            }
    } else {
        strcpy(p_queue[q_index], currile_Dir);
        printf(" queued '%s', file'%s\n"arg1, p_queue);
    }

    q_index++;

    strcat(p_queue[q_index],curr_fid);
    printf(" queued '%s', file'%s\n"arg1, currile_Dir);

} else if (stricmp(cmd, "SKIP")==0) {
    if ((rc = setState("SKIP", arg1, arg2, presp)) != 0) goto cmdDone;
    sprintf(presp, "Skipping '%s'.\n",arg1);
} else if (stricmp(cmd, "STOPAFTER")==0) {
    if ((rc = setState("STOPAFTER", arg1, arg2, presp)) != 0) goto cmdDone;
    sprintf(presp, "Will stop after '%s'.\n", arg1);
}

} else if (stricmp(cmd, "TALKTO")==0) {
    /* Record the IP address and port number to send UDP time msgs to */
    struct hostent *hp;
    char *p;

    toSA.sin_port = htons(DEFAULTPORT);
    if (p = strchr(arg1, ',') ) {
        *p++ = '0';/* Null terminate host name, separate port number */
        toSA.sin_port = htons((unsigned short)atoi(p));
    }
    if (hp = gethostbyname(arg1)) {
        toSA.sin_addr.s_addr = *(long*)hp->h_addr_list[0];
    } else {
        sprintf(presp, "Can't resolve IP address of '%s'.\n", arg1);
    }
}

} else {
    sprintf(presp, "Unrecognized MPEG DVP command: '%s'.\n", cmd);
    rc = -1;
}

}/* Signal the caller that we have completed the command */
printf(" resp: '%s'.\n", presp, strlen(sharedResp)-1, sharedResp);
GULeavegate(&vidDoneGate); /* Signal caller that we are done */

} /* end of forever loop */

/* Abnormal end of vidLoop */
bail:
/* YF Free MPG Card */
if (mpgCloseDriver()!=no_error) {
    printf("Error Freeing Card\n");
} else {
    printf("Closed Driver\n");
}

vidLoopRunning = 0;
toSA.sin_addr.s_addr = 0;
return;
} /* end of vidLoop */
/* Callback routine to getting MPEG_CALLBACKMESSAGES from ReelTime card */

int
prepSock(
) {  
int sock;

/* INITIALIZE SOCKET SYSTEM (for NT) */
{ WSADATA wsaData;
  if (WSAStartup(0x101,&wsaData) == SOCKET_ERROR) {
    /* NOTE: Use version 0101 above rather than 0202, so that setsockopt
     ** will work in MCsockr. */
    printf("WSAStartup failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
  }
}

/* CREATE THE SOCKET */
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    printf("Can't open time/selection send socket, %d\n", tcperrno());
    return -1;
}

return sock;
} /* end of prepSock */

#if defined(OPERATING_SYSTEM_UNIX) &&
!defined(OPERATING_SYSTEM_NT)
FUNCLIST EPlist[] = {
    DVP ,"DVP",
    0, 0
};

long
SLinit(
struct GS *pgsarg,
char *funcname
) {
    FUNCLIST *pfunclist = EPlist;
    long (*funcptr)(void);

84
funcptr = (long (*)(void))-1; /* Set failure return code */
do { /* Find the desired function name in the list of functions */
    if (strcmp(pfunclist->funcname, funcname) == 0) {
        funcptr = (long (*)(void))pfunclist->funcptr;
        pgs = pgsarg; /* Set the local version of pgs */
        break;
    }
} while ((++pfunclist)->funcname); /* Loop till we get to end of list */
return (long)funcptr;
} /* end of SLinit */
#endif

/* YF ReplayFile
** Replays back for a minute
** Current Implementation simply calculates
** bytes to retrace back
*/
int
ReplayFile(
    int chan
)
{
    mpgStatusStruct *curr_Status;
    MPEGHeader *curr_Header;
    int replay_position;
    int temp;

    mpgStop(curr_Card, curr_Chan);

    if (mpgStatus(curr_Card, chan, curr_Status)!=no_error) {
        printf("Error obtaining status on file\n");
        return 0;
    }

    printf("mpgstatus\n");
    printf("File_size: %d\n",curr_Status->Filesize);
    printf("File_position: %d\n",curr_Status->FilePosition);
    printf("System?: %d\n",curr_Status->System);
    printf("HSize: %d\n", curr_Status->HSize);
    printf("VSize: %d\n", curr_Status->VSize);
    printf("PicRate: %d\n", curr_Status->PicRate);
    printf("BitRate: %d\n", curr_Status->BitRate);
    printf("TimeCode: %d\n", curr_Status->TimeCode);
    printf("RunTime: %d\n", curr_Status->RunTime);

    return 0;
}
printf("Frame: %d\n", curr_Status->Frame);

temp = curr_Status->BitRate * 400 * 10 / 8;
if (temp >= (int)curr_Status->FilePosition) {
    replay_position = 0;
} else {
    replay_position = curr_Status->FilePosition - temp;
}

if (mpgSeek(curr_Card, chan, replay_position) != no_error) {
    printf("Error repositioning to file location\n");
    return 0;
}

if (mpgPlay(curr_Card, chan) != no_error) {
    printf("Error replay file\n");
    return 0;
}

if (mpgGetMPEGParameters(curr_Card, curr_Chan, curr_Header) != no_error) {
    printf("error reading MPEG params\n");
    return 0;
}

printf("MPEGHeader info\n");
printf("***************\n");
printf("hsize: %d\n", curr_Header->horizontal_size);
printf("vsize: %d\n", curr_Header->vertical_size);
printf("pcreate: %d\n", curr_Header->picture_rate);
printf("video_bit_rate: %d\n", curr_Header->video_bit_rate);
printf("sampling_frequency: %d\n", curr_Header->sampling_frequency);
printf("audio_bit_rate: %d\n", curr_Header->audio_bit_rate);
printf("bitsPerSecond: %d\n", curr_Header->bitsPerSecond);
printf("framesPerSecond: %d\n", curr_Header->framesPerSecond);
printf("***************\n");

return 1;
} // end of ReplayFile
Appendix E. Code For Phase I and II Implementation

There are three files attached in this appendix, header.h, header.c and detect_transport.c. The first two are relevant to the implementation in Phase I and the last one is relevant to the implementation in Phase II.

/* header.h */
** header file for Phase I implementation, MPEG header decoder ** written by Yuan Feng, YF Aug 98 */
typedef struct headerinfo {
    int ID;       // one bit
    int layer;   // two bits
    int protection;  // one bit
    int bitrate;  // four bits
    int Sampl_freq;  // two
    int padding; // one
    int private_bit; // one
    int mode; // two
    int mode_ext; // two
    int copyright; // one
    int original; // one
    int emphasis; // two
    int frame_count;
    div_t duration; // playback time accurate up to ms
} AUDIO_HEAD;

// array element0 is layer3, element1 is layer2, element2 is layer1
int MPG_layer[3] = {3,2,1};

// row0 is layer1, row1 is layer2, row2 is layer3
// column0-14 corresponds to 0000-1110
int MPG1_bitrate[3][15] = {
    {0,32,64,96,128,160,192,224,256,288,320,352,384,416,448},
    {0,32,48,56,64,80,96,112,128,160,192,224,256,320,384},
    {0,32,40,48,56,64,80,96,112,128,160,192,224,256,320}};

int MPG2_bitrate[3][15] = {
    {0,32,48,56,64,80,96,112,128,144,160,176,192,224,256},
    {0,8,16,24,32,40,48,56,64,80,96,112,128,144,160},
    {0,8,16,24,32,40,48,56,64,80,96,112,128,144,160}};}
// not all modes are allowed for layer II
// 0 means all modes
// 1 means single_channel
// 2 means except single_channel
int MPG_layer2[15] = {0,1,1,0,1,0,0,0,0,2,2,2,2};

int MPG1_sampling_freq[3] = {44100, 48000, 32000};
int MPG2_sampling_freq[3] = {22050, 24000, 16000};

int buffer_file(char *, unsigned char *, int);

//Takes in head structure, ptr to buf, buf_size and starting pos.
//Starting pos is necessary when detection is required in the middle of a file.
//Returns byte position of the start of a sync.
int find_sync(AUDIO_HEAD *, unsigned char *, int, int);

//Takes in head struct, ptr to buf and current sync position in bytes.
//Returns 1 if verified; otherwise 0.
int authenticate_head(AUDIO_HEAD *, unsigned char *, int);

//Takes in ptr to buf, ptr to current bit position in buf and number of bits to be read.
//Returns the decimal representation of bits read.
int readbits(unsigned char *, int *, int);

//Take in buf ptr and first sync pos, return playback time.
//Returns playback time in seconds.milliseconds format.
div_t pb_duration(AUDIO_HEAD *, unsigned char*, int);

/* header.c
** written by Yuan Feng, YF Aug 98
*/
/*
/* YF header.c analyzes the header information of a given audio MPG file */
/* There are four major sub-routines: */
/*
/* buffer_file: Buffers in an input file. */
/* find_sync: Locates possible SYNCWORDS. */
/* authenticate_head: Verifies possible SYNCWORDS. */
/* If SYNCWORDS are authentic, extract header info. */
/* readbits: A generic routine to read file in bits. */
/* pb_duration: A routine to calculate playback duration of the */
/* audio file. */

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
FILE *final_result_file;

void main(
    int argc,
    char *argv[])
{
    struct stat file_stat;
    AUDIO_HEAD head1;
    int ppos;
    unsigned char *buf;
    int buf_size;
    int buf_byte_ptr=0;
    char *file_name = "raw_file";
    int time_flag = 1;

    FILE *final_result_file;

    if (argc > 1) {
        file_name = argv[1];
    }

    if ((final_result_file=fopen("final_result_file", "a+")) == NULL) {
        printf("cannot open final_result_file for writing\n");
    }

    fprintf(final_result_file, "Input File Name: %s\n", file_name);

    if (stat(file_name, &file_stat) == -1) {
        printf("cannot access file\n");
    }

    fprintf(final_result_file,"input audio file size is %d bytes\n", file_stat.st_size);

    buf_size = file_stat.st_size;

    buf = (unsigned char *) malloc(buf_size);
if (buffer_file(file_name, buf, buf_size) == 999) {
    fprintf(final_result_file, "cannot buffer file");
    return;
} // end of if

if (ppos = find_sync(&head1, buf, buf_size, buf_byte_ptr)) == 999) {
    printf("SYNC not found");
    return;
} else {
    fprintf(final_result_file,"SYNCWORD found at \%d\n", ppos);
}

if (head1.ID) {
    fprintf(final_result_file,"ID is MPG-1\n");
} else {
    fprintf(final_result_file,"ID is MPG-2\n");
}

fprintf(final_result_file,"layer is %d\n",head1.layer);

switch (head1.protection) {
    case 1:
        fprintf(final_result_file,"redundancy has not been added\n");
        break;
    case 0:
        fprintf(final_result_file,"redundancy has been added\n");
        break;
    default:
        break;
}

fprintf(final_result_file,"bitrate is %d kbits/second\n",head1.bitrate);
fprintf(final_result_file,"Sampling frequency is %d Hz\n",head1.Samp_freq);

switch (head1.padding) {
    case 1:
        fprintf(final_result_file,"padding is introduced\n");
        break;
    default:
        fprintf(final_result_file,"padding is introduced\n");
        break;
}

fprintf(final_result_file,"private bit is %d\n",head1.private_bit);
switch (head1.mode) {
    case 0:
        fprintf(final_result_file,"mode is stereo\n");
        break;
    case 1:
        fprintf(final_result_file,"mode is joint_stereo\n");
        break;
    case 2:
        fprintf(final_result_file,"mode is dual_channel\n");
        break;
    default:
        fprintf(final_result_file,"mode is single_channel\n");
        break;
}

// if joint_stereo and layer 1 and 2
if (head1.mode == 1 && head1.layer != 3) {
    switch (head1.mode_ext) {
    case 0:
        fprintf(final_result_file,"subbands 4-31 in intensity_stereo, bound is 4\n");
        break;
    case 1:
        fprintf(final_result_file,"subbands 8-31 in intensity_stereo, bound is 8\n");
        break;
    case 2:
        fprintf(final_result_file,"subbands 12-31 in intensity_stereo, bound is 12\n");
        break;
    default:
        fprintf(final_result_file,"subbands 16-31 in intensity_stereo, bound is 16\n");
        break;
    }
}

// if joint-stereo and layer 3
if (head1.mode == 1 && head1.layer == 3) {
    switch (head1.mode_ext) {
    case 0:
        fprintf(final_result_file,"intensity_stereo off; ms_stereo off\n");
        break;
    case 1:
        fprintf(final_result_file,"intensity_stereo on; ms_stereo off\n");
        break;
    case 2:
fprintf(final_result_file,"intensity_stereo off; ms_stereo on\n");
            break;
        default:
            fprintf(final_result_file,"intensity_stereo on; ms_stereo on\n");
            break;
    }
}

if (head1.copyright) {
    fprintf(final_result_file,"copyright protection\n");
} else {
    fprintf(final_result_file,"no copyright protection\n");
}

if (head1.original) {
    fprintf(final_result_file,"original\n");
} else {
    fprintf(final_result_file,"not original\n");
}

fprintf(final_result_file,"type of de-emphasis used:\n");
switch (head1.emphasis) {
    case 0:
        fprintf(final_result_file," no emphasis\n");
        break;
    case 1:
        fprintf(final_result_file," 50/15 microseconds emphasis\n");
        break;
    case 2:
        fprintf(final_result_file," reserved\n");
        break;
    default:
        fprintf(final_result_file," CCITT J.17\n");
        break;
}

if (time_flag) {
    head1.duration=pb_duration(&head1, buf, ppos);
    head1.duration.rem = head1.duration.rem*100/head1.Sampl_freq;
    fprintf(final_result_file, "playback time is %d secs and %d millisecs\n",
            head1.duration.quot, head1.duration.rem);
}

fclose(final_result_file);
return;
int
buffer_file(
    char * file_name,    //ptr to file name
    unsigned char *buf,  //ptr to buffer
    int buf_size         //buf size
) {

    FILE * mp3file;
    int numread, flag;

    //Open mp file. If not successful, exit.
    if( (mp3file = fopen( file_name, "r" )) == NULL )
        return 999;

    //If successful, read file.
    numread = fread(buf, buf_size, 1, mp3file);

    //close file
    flag=fclose(mp3file);
    return numread;
} // end of buffer_file

int
find_sync(
    AUDIO_HEAD *head,
    unsigned char *buf,
    int buf_size,
    int buf_byte_ptr    //It's the start bit position of the search
) {

    unsigned char prev, present, next, test0, test1;
    int i=buf_byte_ptr, comp0=0,comp1 = 0;
    present =buf[i];

    while (i < buf_size) {
        next = buf[i+1];
        test0 = next&0xF0;

test1=present&0x0F;
compO=(present== (unsigned char)0xFF) && (testO==(unsigned char)0xF0);
compI=(present== (unsigned char)0x0F) && (testI==(unsigned char)0xFF);

/* NOTE: Use cast to ensure that data types are matched exactly when */
/* performing operations such as comparison, bit shift, etc. */
/* Different OS implement their functionalities with different default */
/* conditions. */

if (compO || compI) {
  if (authenticate_head(head, buf, i)! = 0) {
    printf("header found at byte number %d\n", i);
    return i;
  } // end of if
} // end of if

i++;
prev = present;
present = buf[i];
} // end of while
return 999;
} // end of find_sync

/* authenticate_head verifies whether SYNCWORD found is real or fake */
int authenticate_head(
  AUDIO_HEAD *head,
  unsigned char *buf,
  int current_sync
) {

  int sync_distance, bit_pos, dd, dd_bitrate;
  unsigned char current_byte, next_byte;

  // need to implement general getbit function
  // take in pos of a char buffer, and number of bits needed
  // first bit after the sync
  if (buf[current_sync] == (unsigned char)0xFF) {
    bit_pos = current_sync*8+12;  // Conversion from byte to bits
  } else {  // Header info starts with 13th bit if FFFX
    bit_pos = current_sync*8+16;  // Otherwise, starts with 17th bit if XFFF
  } // end of if

  head-&gt;ID = readbits(buf, &amp;bit_pos, 1);

  return 0;
} // end of authenticate_head

94
if ((dd-readbits(buf, &bit_pos, 2)) == 0) {
  printf("not valid sync\n");
  return 0;
} else {
  head->layer=MPG_layer[dd-1];
}  // end of if-else

head->protection=readbits(buf,&bit_pos,1);
if ( ((dd_bitrate=readbits(buf,&bit_pos,4)) == (int)0x0F ) {  
  printf("not valid bitrate\n");
  return 0;
} else if (head->ID) {
  head->bitrate=MPG1_bitrate[head->layer-1][dd_bitrate];
} else {
  head->bitrate=MPG2_bitrate[head->layer-1][dd_bitrate];
}  // end of if-else

if ( (dd=readbits(buf,&bit_pos,2)) == (int)0x03 ) {  
  printf("not valid sampling frequency");
  return 0;
} else if (head->ID) {
  head->Sampling_freq=MPG1_sampling_freq[dd];
} else {  
  head->Sampling_freq=MPG2_sampling_freq[dd];
}  // end of if-else

head->padding=readbits(buf,&bit_pos,1);
head->private_bit=readbits(buf,&bit_pos,1);
head->mode=readbits(buf,&bit_pos,2);
head->mode_ext=readbits(buf,&bit_pos,2);  // define macro for this
head->copyright=readbits(buf,&bit_pos,1);
head->original= readbits(buf,&bit_pos,1);
head->emphasis= readbits(buf,&bit_pos,1);

if (head->layer == 2 & & head->ID == 1) {  
  if (((MPG_layer2[dd_bitrate] == 1) & & (head->mode!=3)) ||  
    ((MPG_layer2[dd_bitrate] == 2) & & (head->mode==3))) {  
    printf("Invalid combination of bitrate and mode");
    return 0;
  }
}

if (head->layer == 1) {  
  // This is crucial because for layer 1, one slot is 4 bytes.
  // If slot number is not integral number, then truncation follows.

sync_distance=12*head->bitrate*1000/head->Sampl_freq;

// This step cannot be incorporated into the above if slot number
// is not integral value. For example, 48*384/44.1->417, not 416
sync_distance=4*sync_distance;

//If higher sampling rate employed here
} else if (!head->ID && (head->layer == 3)) {
    sync_distance=72*head->bitrate*1000/head->Sampl_freq;
} else {
    sync_distance=144*head->bitrate*1000/head->Sampl_freq;
} // end of if-else

if (head->padding==1) {
    if (head->layer == 1) {
        sync_distance=sync_distance+4;
    } else {
        sync_distance++;
    } // In layer 2, 3, one slot is one byte.
} // end of if

current_byte = buf[current_sync+sync_distance]&0xFF;
next_byte = buf[current_sync+sync_distance+1]&0xF0;

if ( (current_byte==(unsigned char)0xFF) && (next_byte==(unsigned char)0x0) ) {
    return (current_sync+sync_distance); // Can't check when there is only one
    // frame, so next_byte = 0
} else {
    return 0;
} // end of if
} // end of authenticate_head

/* readbits is a simple routine to read in bits as requested by user */
int
readbits(
    unsigned char * buf,
    int *bit_ptr,
    int bits_to_read
) {
    int byte_pos, bit_pos;
    div_t result = div(*bit_ptr,8);
    unsigned char temp;
    byte_pos = result.quot;
    bit_pos = result.rem;
//get rid of bits before bit_pos
temp = buf[byte_pos] << bit_pos;

//get rid of bits after
temp = temp >> (8-bits_to_read);
*bit_ptr=(*bit_ptr)+bits_to_read;

return ((int)temp);
} // end of readbits

/* pb_duration calculates the playback time of the file */
/* It assumes that the sync_pos passed in here is a valid one */

div_t
pb_duration(
    AUDIO_HEAD *head,
    unsigned char *buf,
    int sync_pos
) {

div_t t_time;
int count = 1; // for the very last frame
int next_sync = sync_pos;

while ((sync_pos=authenticate_head(head, buf, next_sync)) != 0) {
    next_sync=sync_pos;
    count++;
}

head->frame_count=count;

if (head->layer == 1) {
    t_time = div(384*count, head->Sampl_freq);
} else if (!head->ID && (head->layer == 3)) {
    t_time = div(576*count, head->Sampl_freq);
} else {
    t_time = div(1152*count, head->Sampl_freq);
} // end of if-else

return t_time;
} // end of pb_duration
/* detect_transport.c Phase II implementation of the Transport Layer Decoder
 ** written by Yuan Feng, YF Sept 98 */

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>

typedef struct ADAPTATION_FIELD {
    int adaptation_field_length; // 8 bits

    // If adaptation_field_length > 0;
    int discontinuity_indicator; // 1
    int random_access_indicator; // 1
    int elementary_stream_priority_indicator; // 1
    int PCR_flag; // 1
    int OPCR_flag; // 1
    int splicing_point_flag; // 1
    int transport_private_data_flag; // 1
    int adaptation_field_extension_flag; // 1

    // If PCR_flag = 1;
    long int program_clock_reference_base; // 33
    int reserved_PCR; // 6
    int program_clock_reference_extension; // 9

    // If OPCR_flag = 1;
    long int original_program_clock_reference_base; // 33
    int reserved_OPCR; // 6
    int original_program_clock_reference_extension; // 9

    // If splicing_point_flag = 1;
    int splice_countdown; // 8

    // If transport_private_data_flag = 1;
    int transport_private_data_length; // 8

    unsigned char * private_data_type; // 8

    // If adaptation_field_extension_flag = 1;
    int adaptation_field_extension_length; // 8
    int ltw_flag; // 1
    int piecewise_rate_flag; // 1
    int seamless_splice_flag; // 1
int reserved_adaptation; // 5

// If ltw_flag = 1;
int ltw_valid_flag; // 1
int ltw_offset; // 15

// If piecewise_rate_flag = 1;
int reserved_piecewise_rate; // 2
long int piecewise_rate; // 22

// If seamless_splice_flag = 1;
int splice_type; // 4
int DTS_next_AU32_30; // 3
int marker_bit1; // 1
int DTS_next_AU29_15; // 15
int marker_bit2; // 1
int DTS_next_AU14_0; // 15
int marker_bit3; // 1

// For (i=0; i<N; i++);
// N is 184 minus the number of bytes in the adaptation_field;
// 184 is total number of adaptation_field and payload bytes
// after the initial header information.
unsigned char * reserved_stuffing; // 8

// For (i=0; i<N; i++);
unsigned char * stuffing_byte; // 8
} STRUCT_ADAPTATION_FIELD;

typedef struct TRANSPORT_PACKET {
    int transport_error_indicator; // 1
    int payload_unit_start_indicator; // 1
    int transport_priority; // 1
    int PID; // 13
    int old_PID;
    int transport_scrambling_control; // 2
    int adaptation_field_control; // 2
    int continuity_counter; // 4
    int pes_packet_length;
    int old_pes_packet_length; // When video stream has
        // unspecified length
    int psi_section_length;
    int psi_version_number;
79
} TRANSPORT_PACKET;

99
STRUCT_ADAPTATION_FIELD * adapt_field;

int payload_byte;
// For (i=0; i<N; i++)
unsigned char * data_byte;
} STRUCT_TRANSPORT_PACKET;

typedef struct PES_PACKET {
int packet_start_code_prefix;
int stream_id;
int PES_packet_length;
int total_payload_byte;
unsigned char PES_scrambling_control;
unsigned char PES_priority;
unsigned char data_alignment_indicator;
unsigned char copyright;
unsigned char original_or_copy;
unsigned char PTS_DTS_flags;
unsigned char ESCR_flag;
unsigned char ES_rate_flag;
unsigned char DSM_trick_mode_flag;
unsigned char additional_copy_info_flag;
unsigned char PES_CRC_flag;
unsigned char PES_extension_flag;
unsigned char PES_header_data_length;

//if PTS_DTS_flag == '10', 5 bytes of additional info
//if PTS_DTS_flag == '11', 10 bytes of information
//At this point, I don't care what they are
//When the need arises, I will specify in greater detail.
unsigned char PTS[10];

//if ESCR_flag == '1', 6 bytes of additional info
unsigned char ESCR[6];

//if ES_rate_flag == '1', 3 bytes of additional info
unsigned char ES_rate[3];

//if DSM_trick_mode_flag == '1', 4 bytes of additional info
unsigned char DSM_trick_mode[4];

//if additional_copy_info_flag == '1', 1 byte of added info
unsigned char additional_copy_info;

//if PES_CRC_flag == '1', 2 bytes of added info

100
int previous_PES_packet_CRC;

//if PES_extension_flag=='1', 24 bytes of added info
unsigned char PES_extension[24];
unsigned char *stuffing_byte;
unsigned char *PES_packet_data_byte;

//if padding_stream, then padding data_byte
int padding_byte_indicator;
unsigned char *padding_byte;

int new_packet_flag;
int packet_start_indicator;    //1 if start of PES packet; 0 otherwise
int continuity_counter;        //necessary for reassemble to keep track
unsigned char* packet_buf;
int packet_start;
int packet_end;               //points to the next available slot in packet_buf

} STRUCT_PES_PACKET;

//Program Map Section
typedef struct PMS {
    int table_id;            //8 bits
    int section_syntax_indicator;   //1 bit
    //There is a 1 bit of '0';
    int reserved_1;          //2 bits
    int section_length;      //12 bits
    int program_number;      //16 bits
    int reserved_2;          //2 bits
    int version_number;      //5 bits
    int current_next_indicator;    //1 bit
    int section_number;       //8 bits
    int last_section_number;  //8 bits
    int reserved_3;           //3 bits

    //PCR_PID indicates whether program timing
    //info is present in PMT.
    int PCR_PID;              //13 bits
    int reserved_4;           //4 bits

    //program_info_length specifies number of bytes
    //in the following descriptor field
    int program_info_length;  //12 bits
    unsigned char* descriptor_1; //8 bits each array element
int PID_array_size;

//the total bytes following can be inferred
//from section_length. Each loop contains
//the following fields.
//Given how many bytes are left for the info below,
//allocate max possible by excluding descriptor fields
//when considering number of array elements.
int* stream_type; //8 bits
int* reserved_5; //3 bits
int* elementary_PID; //13 bits
int* reserved_6; //4 bits
int* ES_info_length; //12 bits

//descriptor_2 should be an array(a) of arrays(b) of
//unsigned char. array(a) size is determined by the
//formula above. array(b) size is determined by ES_info_length.
//ES_info_length specifies how many bytes
//to buffer in for every element of descriptor_2.
unsigned char** descriptor_2; //8 bits
unsigned int CRC_32; //32 bits
unsigned char* section_buf; //buf for an entire section
                      //starting with the pointer field.
int buf_start;
int buf_end;
} STRUCT_PMS;

//Program Association Section
//Note: Each Table is split into
//sections of the following syntax.
//Each section is then packetized
//and carried in the payload of the PES packet.
typedef struct PAS {
    int table_id; //8 bits
    int section_syntax_indicator; //1 bit
    //There is a 1 bit field = '0' hardcoded.
    int reserved_1; //2 bits
    int section_length; //12 bits
    int transport_stream_id; //16 bits; user-defined
        //for identifying among multiplexes
    int reserved_2; //2 bits
    int version_number; //5 bits
    int current_next_indicator; //1 bit; 1 means current
        //0 means not yet applicable
}
int section_number;            //8 bits; like a counter, starts with 0x00
int last_section_number;       //8 bits

int PID_array_size;            // size of array for program_number,
// loops here till section_length

// Make sure to initialize network_PID and program_map_PID
// with 0. It avoids potential memory valuation error.
int *program_number;           //16 bits each

// When allocating for program_number in a PAS section,
// also do it for pmt.
STRUCT_PMS **pmt;
unsigned char *reserved;        //3 bits each

//if program_number == 0;
int *network_PID;               //13 bits each

//if program_number != 0;
int *program_map_PID;           //13 bits each
// end of loop

// assuming int is 32 bits at least.
unsigned int CRC_32;            //32 bits
unsigned char* section_buf;     // buf for an entire section
    // starting with the pointer field.
int buf_start;                  // current start and end
int buf_end;                    // of buf positions.
} STRUCT_PAS;

// Conditional Association Sable
typedef struct CAS {
    int table_id;                 //8 bits
    int section_syntax_indicator;  //1 bit

    // There is a bit hardcoded to '0' here.
    int reserved_1;               //2 bits
    int section_length;           //12 bits
    int reserved_2;               //18 bits
    int version_number;           //5 bits
    int current_next_indicator;   //1 bit
    int section_number;           //8 bits
    int last_section_number;      //8 bits
    unsigned char* descriptor;    //8 bits
    unsigned int CRC_32;          //32 bits
};
unsigned char* section_buf;  //buf for an entire section
                      //starting with the pointer field
int buf_start;
int buf_end;
} STRUCT_CAS;

//Network Information Section
//Its content is private and
//should adhere to the format of a private section
typedef struct NIS {
  int table_id;            //8 bits
  int section_syntax_indicator;  //1 bit
  int private_indicator;      //1 bit
  int reserved_1;            //2 bits
  int private_section_length; //12 bits

  //if section_syntax_indicator==0;
  unsigned char* private_data_byte_1;  //8 bits

  //else
  int table_id_extension;          //16 bits
  int reserved_2;                  //2 bits
  int version_number;              //5 bits
  int current_next_indicator;      //1 bit
  int section_number;              //8 bits
  int last_section_number;         //8 bits

  //for i=0;i<private_section_length-9;
  unsigned char* private_data_byte_2;  //8 bits

  unsigned int CRC_32;             //32 bits
  unsigned char* section_buf;      //buf for an entire section
                      //starting with the pointer field.
int buf_start;
int buf_end;
} STRUCT_NIS;

//Program Specific Information

//The max length of a section is 1024.
//When allocating memory, allocate for
//max section number as well for max
//max section length.

//Assume that the only reason to break table into
typedef struct PSI {
    int current_indicator;  //1 decodeTransport need to look for PES
    //packets containing NIT and PMT
    int pas_section_length;  //length of the current section.
    //its section_length_field plus field pointer
    //and header fields along with stuffing
    //bytes in between them if any.
    int cas_section_length;  //use of separate fields for each table
    int nis_section_length;  //in case when decoding several PIDs at
    //once.
    int pms_section_length;
}

STRUCT_PAS* pat[512];  //pat stands for program_association_table
    //which is allocated to be an array of STRUCT_PAS.
//Problem, in NIT and PMT, program_numbers(16 bits) serve as index
//instead of section_numbers(8 bits). So need to dynamically allocate
//array size.

//Because pms is identified using program_number, it is natural
//to associate pms sections with program_numbers within a pas.

STRUCT_PMS* pmt[512];
STRUCT_NIS* nit[512];
STRUCT_CAS* cat[512];

//Use just to read in program_number and program_map_PID
int* program_number;
int* program_map_PID;
int program_number_count;
int program_map_PID_count;
int program_map_PID_current;
int program_number_current;
int last_program_number_count;
int last_program_map_PID_count;
int element_PID_count;  //within every program_map_PID

int pas_counter;  //keeps track of continuity of transport
    //packets with PSI info
int nis_counter;
int cas_counter;

int pas_section_number;  //indicates section most recently read.
int nis_section_number; // resets to 0 when a table is complete.
int cas_section_number;
int pms_section_number;

int pas_last_section_number;
int nis_last_section_number;
int cas_last_section_number;

// This is derived from program_number_count;
int pms_last_section_number;

int pas_version_number;
int nis_version_number;
int cas_version_number;
int pms_version_number;

int pas_skip_section;
int cas_skip_section;
int nis_skip_section;
int pms_skip_section;

) STRUCT_PSI;

#define MAXPES_STREAM 8192 // 13-bit PID = 8192
#define TRANSPORT_PACKET_LENGTH 188 // 188 bytes
#define MAXPSI_SECTION_LENGTH 1024 // 12 bits with first two being '00'
#define MAXPSI_SECTION_NUMBER 512 // 8 bits; indexed from 0x00

// command line options
char helpmsg[] =
" ts <options> fileID --MPEG Transport Stream Decoder\n"
" Options:\n"
" -p pid --Decode only the PID specified,(default to all)\n"
"\n"
" Yuan Feng, Nov 98, 12 Nov 98 version, built __DATE__ __TIME__\n";

int verbose = 0;
extern int optind;
extern char *optarg;
int onlyPID = 0;
int badopt=0;

STRUCT_PES_PACKET *PESlist[MAXPES_STREAM];
STRUCT_PSI *PSIinfo;
int decodeTransport(unsigned char*, STRUCT_TRANSPORT_PACKET*);
int reassemblePES(STRUCT_TRANSPORT_PACKET*, STRUCT_PES_PACKET**);
int decodePES(STRUCT_PES_PACKET**, int);
int reassemblePSI(STRUCT_TRANSPORT_PACKET*, STRUCT_PSI*);
int decodePAS(STRUCT_PSI*, STRUCT_PAS**);
int decodeCAS(STRUCT_PSI*, STRUCT_CAS**);
int decodePMS(STRUCT_PSI*, STRUCT_PMS**);
int decodeNIS(STRUCT_PSI*, STRUCT_NIS**);

int getopt(int, char**, char*);

int readbits(unsigned char*, int*, int);

//returns 1 if element found, 0 otherwise
//entry: int element
//      int* list
//      int list_count
int elementinlist(int, int*, int);

/////////////////////////////////////////////////////////////////////////////////////////////////

//reassemblePAS
//function:
//      ——Reassembles a complete PAS,
//      Program Association Section.
//entry:
//      ——pointer to decoded transport packet
//      ——pointer to PSI data structure
//return:
//      ——error
//      ——assemble Section complete
//      ——nothing assembled, duplicate
/////////////////////////////////////////////////////////////////////////////////////////////////
int error; 1 nothing performed; 2 section assembly complete
reassemblePAS(
    STRUCT_TRANSPORT_PACKET* ptransport,
    STRUCT_PSI* ppsi
)
{
int pas_section_number;
int i;
int pointer_field;        //8 bits at the start of section
int bit_ptr=0;
int garbage;
int start, end;

if (ptransport->PID==0x00) {
    //PAS

    //memset in main ensures that such section_number start with 0.
    //It's used as an index into the pat array.
    pas_section_number=ppsi->pas_section_number;

    //if start of a section.
    if (ptransport->payload_unit_start_indicator==1) {

        ppsi->pas_skip_section=0;
        ppsi->pas_section_length = ptransport->psi_section_length;

        //If haven't, allocate memory for section structure
        //and initialize.
        //Note: I can instead allocate for section_length as PES_length,
        //but the max here is only about 2k, so no big deal.
        if (ppsi->pat[pas_section_number]==0) {
            ppsi->pat[pas_section_number]=(STRUCT_PAS*)
                malloc(sizeof(STRUCT_PAS));
            memset(ppsi->pat[pas_section_number],0,sizeof(STRUCT_PAS));
            ppsi->pat[pas_section_number]->section_buf=(unsigned
                char*)malloc(MAXPSI_SECTION_LENGTH);
            memset(ppsi->pat[pas_section_number]->section_buf,
                0,MAXPSI_SECTION_LENGTH);

            //If new version and previous version complete, then buffer in new version.
        } else if ((ptransport->psi_version_number!=ppsi->pas_version_number)
            && (ppsi->pas_section_number==ppsi->pas_last_section_number)) {
            if (ppsi->pas_section_number!=ppsi->pas_last_section_number) {
                printf("PAS table not complete yet\n");
                return 0;
            }
        }
        for (i=0;i<ppsi->pas_last_section_number;i++) {
            ppsi->pat[i]->buf_start=ppsi->pat[i]->buf_end=0;
        }
    }
    ppsi->pas_section_number=0;
    ppsi->pas_last_section_number=0;
    ppsi->pas_version_number=ptransport->psi_version_number;
    ppsi->program_number_PID_current=0;
    ppsi->program_map_PID_current=0;
} else {
    printf("PAS section duplicate\n");
}
ppsi->pas_skip_section=1;
    return 1;
}

} // end of if start of PAS section

if (ppsi->pas_skip_section==1) {
    return 1;
}

//Start buffering section payload into section_buf.
//Note: buf_end indicates the next available slot.
start=ppsi->pat[pas_section_number]->buf_start;
end=ppsi->pat[pas_section_number]->buf_end;

//Need to check for continuity
for (i=0;i<ptransport->payload_byte;i++) {
    ppsi->pat[pas_section_number]->section_buf[end+i]=ptransport->data_byte[i];
}
ppsi->pat[pas_section_number]->bufLend+=ptransport->payload_byte;

//if reassembling section complete, call decodePAS.
//section_length+3 bytes of overhead before and including the
//section_length field.
//< when stuffing occurs. More accurate method
//exists but doesn't matter this case.
if (ppsi->pas_section_length+3<=ppsi->pat[pas_section_number]->buf_end) {
    printf("PAS section %d assemblage complete\n",pas_section_number);
    return 2;
}

//Note:
//what if loss of packets result in section_number 1 coming before
//section_number 0? it will be wrongly assigned to array[0].
//so check for this error in decodePAS.
}
} // end of if PAS

return 1;

} // end of reassemblePAS

// decodePAS
// function:
// -- Decodes a PAT, Program Association Table, section by
//   section.
// entry:
//   -- pointer to PSI data structure
//   -- pointer to PAT structure within the PSI data structure
// return:
//   -- error
//   -- section complete
//   -- table complete

int decodePAS(STRUCT_PSI* ppsi, STRUCT_PAS** ppat)
{
    int pointer_field;
    int bit_ptr=0;
    int pas_section_number;
    int garbage;
    int program_data;
    int loop_number;
    int temp;
    int i;
    int return_flag = 1;

    pas_section_number=ppsi->pas_section_number;
    pointer_field=readbits(ppat[pas_section_number]->section_buf,&bit_ptr,8);  //8 bits for
    //pointer_field
    //Get to the start of the section
    if (pointer_field>0) {
        garbage=readbits(ppat[pas_section_number]->section_buf,&bit_ptr,pointer_field);
    }

    ppat[pas_section_number]->table_id=
    readbits(ppat[pas_section_number]->section_buf,&bit_ptr,8);  //8 bits for table_id
    //Check for reserved table_id's

    ppat[pas_section_number]->section_syntax_indicator=
    readbits(ppat[pas_section_number]->section_buf,&bit_ptr,1);  //1 bit
    if (readbits(ppat[pas_section_number]->section_buf,&bit_ptr,1)!=0) {

110
printf("Error detected, the bit should be 0\n");
return (return_flag=0);
}
//This bit must be '0'

ppat[pas_section_number]->reserved_1=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,2);  //2 bits
ppat[pas_section_number]->section_length=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,12);  //12 bits
ppat[pas_section_number]->transport_stream_id=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,16);  //16 bits
ppat[pas_section_number]->reserved_2=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,2);  //2 bits
ppat[pas_section_number]->version_number=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,5);  //5 bits
ppat[pas_section_number]->current_next_indicator=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,1);  //1 bit
ppat[pas_section_number]->section_number=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,8);  //8 bits

ppat[pas_section_number]->last_section_number=
  readbits(ppat[pas_section_number]->section_buf,&bit_ptr,8);  //8 bits
ppsi->pas_last_section_number=ppat[pas_section_number]->last_section_number;

//check for continuity: section_number, transport_packet_continuity
if (ppsi->pas_section_number==ppat[pas_section_number]->section_number &&
    ppsi->pas_last_section_number>ppsi->pas_section_number) {
  ppsi->pas_section_number++;  
  ppsi->program_number_current=0;
  ppsi->program_map_PID_current=0;
} else if (ppsi->pas_last_section_number==ppsi->pas_section_number) {
  printf("Table complete\n");
  return_flag=2;
  ppsi->program_number_current=1;
  ppsi->program_map_PID_current=1;
} else {
  printf("Sections numbers don't match\n");
  return (return_flag=0);
}

//Now read in program association table entries.
program_data=ppat[pas_section_number]->section_length-9;  //9 for CRC_32, and from
//transport_stream_id
//to last_section_number
//This must be divisible.

loop_number=program_data/4;
allocate memory

```c
temp = sizeof(int)*loop_number;
ppat[pas_section_number]->PID_array_size=loop_number;
ppat[pas_section_number]->program_number=(int*)malloc(temp);
```

Once allocated, decodePES can buffer the section payload into this array.

```c
ppat[pas_section_number]->reserved=(unsigned char*)malloc(temp);
ppat[pas_section_number]->network_PID=(int*)malloc(temp);
ppat[pas_section_number]->program_map_PID=(int*)malloc(temp);
```

reading in program info

```c
for (i=0;i<loop_number;i++) {
    ppat[pas_section_number]->program_number[i]=
        readbits(ppat[pas_section_number]->section_buf,&bit.ptr,16); //16 bits
    ppsi->program_number_count++; ppat[pas_section_number]->reserved[i]=
        readbits(ppat[pas_section_number]->section_buf,&bit.ptr,3); //3 bits
    //1 indicates no data stored
    if (ppat[pas_section_number]->program_number[i]==0) {
        ppat[pas_section_number]->network_PID[i]=
            readbits(ppat[pas_section_number]->section_buf,&bit.ptr,13); //13 bits
        ppat[pas_section_number]->program_map_PID[i]=-1;
    } else {
        ppat[pas_section_number]->program_map_PID[i]=
            readbits(ppat[pas_section_number]->section_buf,&bit.ptr,13); //13 bits
        ppat[pas_section_number]->network_PID[i]=-1;
        ppsi->program_map_PID_count++;
    }
}
```

```c
ppat[pas_section_number]->CRC_32=
    readbits(ppat[pas_section_number]->section_buf,&bit.ptr,32); //32 bits
```

Implement CRC decoder

```c
return return_flag;
}
```

// end of decodePAS

//reassemblePMS

```
// function:
// — Reassembles a complete PMS,
//   Program Mapping Section.
// entry:
// — pointer to decoded transport packet
// — pointer to PSI data structure
```

112
int reassemblePMS(
    STRUCT_TRANSPORT_PACKET* ptransport,
    STRUCT_PSI* ppsi)
) {

    int pas_section_number;
    int cas_section_number;
    int pms_section_number;

    int i;
    int pointer_field;
    int bit_ptr=0;
    int garbage;
    int start, end;

    // Each section of PMT is separate with section_number = 0x00.
    // Use program_map_PID_count instead.
    // and program number as well as index.
    // PAT is assumed to be assembled completely at this point.
    pms_section_number=ppsi->pms_section_number;

    // If start of a section.
    if (ptransport->payload_unit_start_indicator==1) {

        // Reexamine at start of every new section.
        ppsi->pms_skip_section=0;
        ppsi->pms_section_length = ptransport->psi_section_length;

        // Allocate memory for each array element.
        // Allocation of memory for the array pointer should be in main
        // after PAT is complete.
        if (ppsi->pmt[pms_section_number]==0) {
            ppsi->pmt[pms_section_number]=
                (STRUCT_PMS*)malloc(sizeof(STRUCT_PMS));
            memset(ppsi->pmt[pms_section_number],0,sizeof(STRUCT_PMS));
            ppsi->pmt[pms_section_number]->section_buf=
                (unsigned char*)malloc(MAXPSISECTION_LENGTH);
        }
memset(ppsi->pmt[pms_section_number]->
  section_buf,0,MAXPSI_SECTION_LENGTH);

// Note: version_number is always 0 for PMS
} else if ((ptransport->psi_version_number!=ppsi->pms_version_number) &&
(ppsi->pms_section_number==ppsi->pms_last_section_number)) {
  if (ppsi->pms_section_number!=ppsi->pms_last_section_number) {
    printf("current PMS section not yet buffered complete\n");
    return 0;
  }
  for (i=0;i<ppsi->pms_last_section_number;i++) {
    ppsi->pmt[i]->buf_start=ppsi->pmt[i]->buf_end=0;
  }
  ppsi->pms_section_number=0;
  ppsi->pms_last_section_number=0;
  ppsi->pms_version_number=ptransport->psi_version_number;
} else {
  printf("Current Section already buffered\n");
  // Skip remaining section.
  ppsi->pms_skip_section=1;
  return 1;
}
} // end of if start of PMS section

if (ppsi->pms_skip_section==1) {
  return 1;
}

//Start buffering section payload into section_buf.
//Note: buf_end indicates the next available slot.
start=ppsi->pmt[pms_section_number]->buf_start;
end=ppsi->pmt[pms_section_number]->buf_end;

for (i=0;i<ptransport->payload_byte;i++) {
  ppsi->pmt[pms_section_number]->section_buf[end+i]=ptransport->data_byte[i];
}
ppsi->pmt[pms_section_number]->buf_end+=ptransport->payload_byte;

//if reassembling section complete, call decodePAS.
//section_length+3 bytes of overhead before and including
//the section_length field.
//< when stuffing occurs. More accurate method
//exists but doesn't matter this case.
if (ppsi->pms_section_length+3<ppsi->pmt[pms_section_number]->buf_end) {
  printf("section %d assemblage complete\n",pms_section_number);

114
return 2;
}

return 1;

} // end of reassemblePMS

// decodePMS

// function:
//     - Decodes a PMS, Program Association Table, section by
//     section.
// entry:
//     - pointer to PSI data structure
//     - pointer to PMT structure within the PSI data structure
// return:
//     - error
//     - section complete
//     - table complete

int decodePMS(
    STRUCT_PSI* ppsi,
    STRUCT_PMS** ppmt
) {
    int pointer_field;
    int bit_ptr=0;
    int pms_section_number;
    int garbage;
    int program_data;
    int loop_number;
    int temp;
    int i,j;
    int return_flag=1;
    int flag1=0;

    pms_section_number=ppsi->pms_section_number;
    pointer_field=readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8);
    if (pointer_field>0) {
        garbage=readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,pointer_field);
    } // Get to the start of the section
ppmt[pms_section_number]->table_id=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8);
ppmt[pms_section_number]->section_syntax_indicator=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,1);
if (readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,1)!=0) {
  printf("Error detected, the bit should be 0\n");
  return 0;
} //This bit must be '0'

ppmt[pms_section_number]->reserved_1=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,2); //2 bits
ppmt[pms_section_number]->section_length=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,12); //12 bits
ppmt[pms_section_number]->program_number=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,16); //16 bits
ppmt[pms_section_number]->reserved_2=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,2); //2 bits
ppmt[pms_section_number]->version_number=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,5); //5 bits
ppmt[pms_section_number]->current_next_indicator=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,1); //1 bit
ppmt[pms_section_number]->section_number=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8); //8 bits
ppmt[pms_section_number]->last_section_number=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8); //8 bits

if (ppmt[pms_section_number]->section_number!=0 ||
ppmt[pms_section_number]->last_section_number!=0) {
  printf("Section_number and Last_section_number should all be 0\n");
  return 0;
}

//check for continuity: section_number, transport_packet_continuity
//This has to be done using transport_packet_continuity count in
//decodePES/decodeTransport
if ((ppsi->pms_section_number+1)<ppsi->program_map_PID_count) {
  ppsi->pms_section_number++;
} else if ((ppsi->pms_section_number+1)==ppsi->program_map_PID_count) {
  printf("Table complete\n");
  return_flag=2;
} else {
  printf("problem matching section number in decodePMS");
}
ppmt[pms_section_number]->reserved_3=
    readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,3);
ppmt[pms_section_number]->PCR_PID=
    readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,13);
ppmt[pms_section_number]->reserved_4=
    readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,4);
ppmt[pms_section_number]->program_info_length=
    readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,12);

if (ppmt[pms_section_number]->program_info_length!=0) {
    ppmt[pms_section_number]->descriptor_1=(unsigned char*)
        malloc(ppmt[pms_section_number]->program_info_length);
    for (i=0;i<ppmt[pms_section_number]->program_info_length;i++) {
        ppmt[pms_section_number]->descriptor_1[i]=
            readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8);
    }
}
//9 is up to program_info_length field; 4 for CRC_32 field;
program_data=ppmt[pms_section_number]->
    section_length-9-4-ppmt[pms_section_number]->program_info_length;
loop_number=program_data/4;

// Since pms section number is always 0, only one section allocated in table.
ppmt[pms_section_number]->PID_array_size=loop_number;
ppmt[pms_section_number]->stream_type=(int*)malloc(loop_number);
ppmt[pms_section_number]->reserved_5=(int*)malloc(loop_number);
ppmt[pms_section_number]->elementary_PID=(int*)malloc(loop_number);
ppmt[pms_section_number]->reserved_6=(int*)malloc(loop_number);
ppmt[pms_section_number]->ES_info_length=(int*)malloc(loop_number);
ppmt[pms_section_number]->descriptor_2=(unsigned char**)malloc(loop_number);

//temp is the start bit_position of CRC_32, indexed from 0
temp=ppmt[pms_section_number]->section_length*8-32;
j=0;
flag1=0;
while(bit_ptr<temp) {
    ppmt[pms_section_number]->stream_type[j]=
        readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8);
    ppmt[pms_section_number]->reserved_5[j]=
        readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,3);
    ppmt[pms_section_number]->elementary_PID[j]=
        readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,13);
    ppmt[pms_section_number]->reserved_6[j]=
    }
readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,4);
ppmt[pms_section_number]->ES_info_length[j]=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,12);
ppmt[pms_section_number]->descriptor_2[j]=
  (unsigned char*)malloc(ppmt[pms_section_number]->ES_info_length[j]);
for (i=0;i<ppmt[pms_section_number]->ES_info_length[j];i++) {
  ppmt[pms_section_number]->descriptor_2[j]=
    readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,8);
}
j++;
flag1++;
}

ppsi->element_PID_count=j;
ppmt[pms_section_number]->CRC_32=
  readbits(ppmt[pms_section_number]->section_buf,&bit_ptr,32);
return return_flag;
}

/* Function: *//* -Reassembles a complete CAS, *//* Conditional Access Section. *//* entry: *//* -pointer to decoded transport packet *//* -pointer to PSI data structure *//* return: *//* -error *//* -assemble Section complete *//* -nothing assembled, duplicate */
int reassembleCAS(
  STRUCT_TRANSPORT_PACKET* ptransport,
  STRUCT_PSI* ppsi
)
{
  int pas_section_number;
  int cas_section_number;
  int pms_section_number;

  int i;
  int pointer_field;
  /* Code for reassembleCAS function goes here */
}
int bit_ptr = 0;
int garbage;
int start, end;

if (ptransport->PID == 0x01) { //CAS
    cas_section_number = ppsi->cas_section_number;
    //if start of a CAS section
    if (ptransport->payload_unit_start_indicator == 1) {
        ppsi->cas_skip_section = 0;
        ppsi->cas_section_length = ptransport->psi_section_length;
        //if haven't, allocate mem for section structure.
        //and initialize.
        if (ppsi->cas[cas_section_number] == 0) {
            ppsi->cas[cas_section_number] = (STRUCT_CAS*)malloc(
                sizeof(STRUCT_CAS));
            memset(ppsi->cas[cas_section_number], 0, sizeof(STRUCT_CAS));
            ppsi->cas[cas_section_number]->section_buf =
                (unsigned char*)malloc(MAXPSI_SECTION_LENGTH);
            memset(ppsi->cas[cas_section_number]->
                section_buf, 0, MAXPSI_SECTION_LENGTH);
        } else if ((ptransport->psi_version_number != ppsi->cas_version_number) &&
            (ppsi->cas_section_number == ppsi->cas_last_section_number)) {
            // Note: it's when new version is in but table not yet complete that
            // the following case is considered.
            if (ppsi->cas_section_number != ppsi->cas_last_section_number) {
                printf("current CAS table not complete\n");
                return 0;
            }
            for (i = 0; i < ppsi->cas_last_section_number; i++) {
                ppsi->cas[i]->buf_start = ppsi->cas[i]->buf_end = 0;
            }
            ppsi->cas_section_number = 0;
            ppsi->cas_last_section_number = 0;
            ppsi->cas_version_number = ptransport->psi_version_number;
        } else {
            printf("CAS Table already buffered\n");
            ppsi->cas_skip_section = 1;
            return 1;
        }
    } //end of if start of CAS section.

    if (ppsi->cas_skip_section == 1) {
        return 1;
    }
    //Start buffering section payload into section_buf.

}
start=ppsi->cat[cas_section_number]->buf_start;
end=ppsi->cat[cas_section_number]->buf_end;

for (i=0;i<ptransport->payload_byte;i++) {
  ppsi->cat[cas_section_number]->section_buf[end+i]=ptransport->data_byte[i];
}

ppsi->cat[cas_section_number]->buf_end+=ptransport->payload_byte;

//if reassembling section complete, call decodeCAS.
if (ppsi->cas_section_length+3<=ppsi->cat[cas_section_number]->buf_end) {
  printf("CAS section %d assemblage complete\n",cas_section_number);
  return 3;
}
} // end of if CAS

return 1;

} // end of reassembleCAS

int main(
  int argc,
  char *argv[]
) {

  int c;

  STRUCT_PES_PACKET *ppes;    // Individual instance of a PES packet.
  STRUCT_TRANSPORT_PACKET *ptransport;
  unsigned int bytes_read;
  char * file_name = "2xtl.mpg";
  FILE * transport_file, *PAS_file, *PMS_file;
  int sync_byte_location = 0;
  int byte_counter = 0;
  int sync_location_detect = 0;
  unsigned char present;
  unsigned char buf[TRANSPORT_PACKET_LENGTH];
  int flag_count=0;
  int null_count = 0;
  int pes_count = 0;
  int psi_count = 0;
  int temp = 0;
  int byte_read;
int i = 0;
int j=0;
int k =0;

int Transport_retum_flag = 0;
int assemble_PES_return_flag=0;
int decode_PES_return_flag=0;
int assemble_PAS_return_flag=0;
int assemble_CAS_return_flag=0;
int assemble_PMS_return_flag=0;
int assemble_NIS_return_flag=0;
int decode_PAS_return_flag=0;
int decode_CAS_return_flag=0;
int decode_PMS_return_flag=0;
int decode_NIS_return_flag=0;

if ((PAS_file=fopen("pas_program_list","at"))==NULL)goto bail_out;

onlyPID=-7;

while ((c=getopt(argc,argv,"p:v:t"))!= EOF) {
    switch (c) {
        case 'p': // Check for specified PID
            onlyPID=atoi(optarg); // Need to convert from hex
            fprintf(PAS_file,"nPID in decimal is \%d\n",onlyPID);
            break;
        case 'v':
            verbose++;
            break;
        case 't': // Check for PSI
            onlyPID=-7;
            default:
            onlyPID=-7;
            break;
    }
} // end of scanning for options

argv=&argv[optind];
argc = argc - optind;

if (badopt) {
    printf(helpmsg);
    return -1;
}
// Note: Malloc and Free are expensive computational-wise.
// Here I am increasing speed at the expense of storage space
// by allocating memory all at once at the beginning
// and free them all at once at the end instead of on a need basis.
PSIinfo = (STRUCT_PSI*) malloc(sizeof(STRUCT_PSI));

// Setting the individual table pointers to 0 is useful when
// determining whether a table is the very first one for a
// transport stream rather an updated version.
memset(PSIinfo, 0, sizeof(STRUCT_PSI));

// Equivalent to MAXPES_STREAM*8 since a pointer is 4 bytes
temp = MAXPES_STREAM * sizeof(STRUCT_PES_PACKET*);

// PESList is address of the pointer to the start of the array of
// pointers to PES_PACKET_STRUCT
memset(PESList, 0, temp);

ppes = (STRUCT_PES_PACKET*) malloc(sizeof(STRUCT_PES_PACKET));
memset(ppes, 0, sizeof(STRUCT_PES_PACKET));

ptransport = (STRUCT_TRANSPORT_PACKET*) malloc
(sizeof(STRUCT_TRANSPORT_PACKET));
memset(ptransport, 0, sizeof(STRUCT_TRANSPORT_PACKET));
ptransport->adapt_field = (STRUCT_ADAPTATION_FIELD*) malloc
(sizeof(STRUCT_ADAPTATION_FIELD));
memset(ptransport->adapt_field, 0, sizeof(STRUCT_ADAPTATION_FIELD));

// Maximum length < adaptation_field_length - 5; 1 byte for the field itself;
ptransport->adapt_field->private_data_type =
(unsigned char*) malloc(TRANSPORT_PACKET_LENGTH);
memset(ptransport->adapt_field->
private_data_type, 0, (TRANSPORT_PACKET_LENGTH));

ptransport->adapt_field->reserved_stuffing = (unsigned char*) malloc
(TRANSPORT_PACKET_LENGTH);
memset(ptransport->adapt_field->
reserved_stuffing, 0, (TRANSPORT_PACKET_LENGTH));

ptransport->adapt_field->stuffing_byte =
(unsigned char *)malloc(TRANSPORT_PACKET_LENGTH);

memset(ptransport->adap_field->
    stuffing_byte, 0, (TRANSPORT_PACKET_LENGTH));

ptransport->data_byte = (unsigned char *)malloc(TRANSPORT PACKET LENGTH);
memset(ptransport->data_byte, 0, (TRANSPORT_PACKET_LENGTH));

memset(buf, 0, TRANSPORT_PACKET_LENGTH);

// Open Transport packet file.
// Use of b is critical. Otherwise ascii rep of EOF
// In binary stream terminates the read process.
if ((transport_file = fopen(file_name, "rb")) == NULL) {
    printf("Unable to Access File\n");
    goto bai_out;
}

// Scan for start of a transport packet.
while (!sync_location_detect && present != (unsigned char)NULL) {
    // Read in one byte at a time.
    if ((bytes_read = fread(&present, 1, 1, transport_file)) != 1) {
        printf("error reading bytes");
        goto bail_out;
    }

    // Check if byte read is Transport packet sync.
    if (present == (unsigned char)0x47) {
        // If sync found, but cannot buffer the whole packet, return error.
        if ((bytes_read =
                fread(buf, 1, TRANSPORT_PACKET_LENGTH, transport_file)) != 188) {
            printf("error: cannot buffer file");
            goto bail_out;
        }

        // If sync found, and packet buffered, update relevant fields.
        if (buf[187] == (unsigned char)0x47) {
            sync_location_detect = 1;
            sync_byte_location = byte_counter;
        }
    }
}

byte_counter++;
} // end of while.

// Right now, byte_counter points to the byte right after 0x47.
if (fseek(transport_file,(long)(byte_counter-1),SEEK_SET) != 0) {
    printf("error resetting stream cursor");
goto bail_out;
}

// Loop to process the Transport_packet stream.
while(1) {
    flag_count++;
i++;

    // If no more data to buffer, then exit.
    if ((bytes_read = fread(buf, 1, TRANSPORT_PACKET_LENGTH, transport_file)) != 188) {
        fprintf(PAS_file, "current transport packet number is %d\n",flag_count);
        fprintf(PAS_file, "current null packet number is %d\n",null_count);
        fprintf(PAS_file, "current psi packet number is %d\n",psi_count);
        fprintf(PAS_file, "current pes packet number is %d\n",pes_count);
        if( ferror( transport_file ) )
            { perror( "Read error" );
            }
        printf("can't buffering data to 188 bytes\nEnd of File reached\n");
goto bail_out;
    }

    // Call to decodeTransport.
    Transport_return_flag=decodeTransport(buf,ptransport);

    switch (Transport_return_flag) {
    case 0:
        printf("decodeTransport Process Terminated\n");
goto bail_out;
    case 1:
        null_count++;
goto cmd_done;
    case 2:
        pes_count++;
        unbounded_length:
        assemble_PES_return_flag=reassemblePES(ptransport,PESlist);
goto return_PES;
    case 3:

psi_count++;
assemble_PAS_return_flag=reassemblePAS(ptransport, PSIinfo);
goto return_PAS;
case 4:
psi_count++;
case 5:
psi_count++;
assemble_PMS_return_flag=reassemblePMS(ptransport, PSIinfo);
goto return_PMS;
case 6:
psi_count++;
case 7:
psi_count++;
case 8:
psi_count++;
goto cmd_done;
default:
pes_count++;
goto cmd_done;
} // end of switch for transport_return

return_PES:

// Check assembly of PES packets
switch (assemble_PES_return_flag) {
case 0:
printf("PES packet assembly interrupted\n");
goto bail_out;
case 1:
break;
case 2:
decode_PES_return_flag=decodePES(PESlist, ptransport->PID);
switch (decode_PES_return_flag) {
case 0:
printf("decoding PES interrupted\n");
goto bail_out;
case 1:
break;
default:
break;
}
break;
case 3: // when video stream has unbounded length
decode_PES_return_flag=decodePES(PESlist, ptransport->PID);
switch (decode_PES_return_flag) {
case 0:
printf("decoding PES interrupted\n");
goto bail_out;
case 1:
    break;
default:
    break;
} // end of decode_PES_return_flag

goto unbounded_length; // new PES packet already in TS packet, just reassemble it.
    break;
default:
    printf("default from return_PES\n");
} // end of switch assemble_PES_return_flag

goto cmd_done;

return_PAS:
switch (assemble_PAS_return_flag) {
    case 0:
        printf("PAS assembly interrupted\n");
goto bail_out;
case 1:
        printf("successfully appended to section\n");
break;
case 2:
        printf("successfully assembled section\n");
de decode_PAS_return_flag=decodePAS(PSIinfo, PSIinfo->pat);
switch (decode_PAS_return_flag) {
    case 0:
        printf("decoding PAS interrupted\n");
goto bail_out;
case 1:
        printf("decoding PAS section complete\n");
break;
case 2:
        printf("decoding PAT complete\n");
    // At this point, want to update list of PMT_PID and NIT_PID
    PSIinfo->program_map_PID=(int*)malloc(PSIinfo->program_PID_count);
    PSIinfo->program_number=(int*)malloc(PSIinfo->program_number_count);
k=0;
    for (i=0; i<=PSIinfo->pas_section_number; i++) {
            for (j=0; j<PSIinfo->pat[i]->PID_array_size; j++) {
                PSIinfo->program_number[k]=PSIinfo->pat[i]->program_number[j];
                if (PSIinfo->pat[i]->program_map_PID[j]!=-1) {
                    PSIinfo->program_map_PID[k]=PSIinfo->pat[i]->program_map_PID[j];
                    PSIinfo->program_map_PID_count++;
                    PSIinfo->program_number_count++;
                }
            }
        }
    break;} // end of switch decode_PAS_return_flag
} // end of switch assemble_PAS_return_flag

126
fprintf(PAS_file, "program number %d: program_map_PID %d\n", 
PSIinfo->program_number[k], PSIinfo->program_map_PID[k]);
k++;
}
}
}
default:
    break;
}
break;
default:
    printf("default from return_PAS\n");
}
printf("return_PAS");
goto cmd_done;

return_PMS:
switch (assemble_PMS_return_flag) {
    case 0:
        printf("PMS assembly interrupted\n");
        goto bail_out;
    case 1:
        printf("successfully appended to section\n");
        break;
    case 2:
        printf("successfully assembled section\n");
        decode_PMS_return_flag=decodePMS(PSIinfo, PSIinfo->pmt);
switch (decode_PMS_return_flag) {
    case 0:
        printf("decoding PMS interrupted\n");
        goto bail_out;
    case 1:
        printf("decoding PMS section complete\n");
        break;
    case 2:
        printf("decoding PMT complete\n");

// At this point, want to update list of PMT_PID and NIT_PID
fprintf(PAS_file, "elementary PIDs for program number %d: \n", 
PSIinfo->pmt[0]->program_number);
for (i=0; i<PSIinfo->element_PID_count; i++) {
    fprintf(PAS_file, "%d\n", PSIinfo->pmt[0]->elementary_PID[i]);
}
default:
    break;
}
break;
default:
    printf("default from return_PMS\n");
}

printf("return_PMS");
goto cmd_done;

cmd_done:
    continue;
} // end of while(1)

// Call on reassemblePES to piece together the packets.
// Differentiate between PSI and PES as well as between
// single PID or multiple PID assembly

bail_out:

    free(ppes);
    free(ptransport->data_byte);
    free(ptransport->adapt_field->stuffing_byte);
    free(ptransport->adapt_field->reserved Stuffing);
    free(ptransport->adapt_field->private_data_type);
    free(ptransport->adapt_field);
    free(ptransport);
    fclose(PAS_file);
    fclose(PMS_file);
    return 0;
} // end of main

// Right now, decodes one PID at a time.
// need to enable that it is capable of decoding all PIDs
// at once.

/////////////////////////////////////////////////////////////////////////
// decodeTransport
//
// function:
//    Decode a Transport packet.
//
// entry: buffer of 188 bytes; 188 is the size of one transport packet.
// transport_packet
//
// return: error
int decodeTransport(unsigned char *buf,
  STRUCT_TRANSPORT_PACKET *ptransport)
{

  int total_payload_byte = 184;  // 188 - 4 bytes for control info
  int total_adpt_ext_byte = 0;
  int total_adpt_byte = 0;
  int bit_counter = 0;
  int i;

  FILE *out_file;

  int byte_read;
  int PSI_flag=0;   // 1 if transport packet contains PSI stuff
  int pointer_field;
  int buf_bit_counter=0;
  int return_flag = 9;  // default to 9; unspecified PES

  // Got 188 bytes of transport stream
  // Check for sync word
  if (buf[0] != (unsigned char)0x47) {
    printf("error: packets does not start with right sync\n");
    return 0;
  }

  // Start decoding transport packet header information.
  // Get ride of the sync byte
  bit_counter = 8;   //Get ride of the sync byte
  ptransport->transport_error_indicator = readbits(buf, &bit_counter, 1);
  ptransport->payload_unit_start_indicator = readbits(buf, &bit_counter, 1);
  ptransport->transport_priority = readbits(buf, &bit_counter, 1);
  ptransport->old_PID=ptransport->PID;
  ptransport->PID = readbits(buf, &bit_counter, 13);

  // 0x1FFF is reserved as PID for null packets.
  if (ptransport->PID==(int)0x1FFF) {
    return 1;
  }
// 0x0002-0x000F are reserved PIDs
if (ptransport->PID>=(int)0x0002 && ptransport->PID<=(int)0x000F) {
    printf("reserved bit combinations\n");
    return 0;
}

// User requested PSI.
// PSI: PAT, CAT, PMT, NIT
if (onlyPID==-7) {
    PSI_flag=1;
    // Payload of Transport Packet contains PSI sections.
    switch (ptransport->PID) {
        case 0x0000:
            return_flag=3; // PAS
            break;
        case 0x0001:
            return_flag=4; // CAS
            break;
        default:
            if (PSIinfo->program_map_PID_current &&
                element_in_list(ptransport->PID, PSIinfo->program_map_PID,
                                PSIinfo->program_map_PID_count)) {
                return_flag=5;
            } else { // else check for NIT and Privat
                return return_flag;
            }
    }
    // end of switch
    // User requested specific PID
} else if (ptransport->PID==onlyPID) {
    return_flag = 2;
} else {
    if (PSIinfo->program_map_PID_current &&
        element_in_list(ptransport->PID, PSIinfo->program_map_PID, 
                        PSIinfo->program_map_PID_count) ||
        ptransport->PID==0x0000 || ptransport->PID==0x0001) {
        return_flag=8;
    }
    return return_flag;
}

// Continue decoding Transport packet header information.
ptransport->transport_scrambling_control = readbits(buf, &bit_counter, 2);
ptransport->adaptation_field_control = readbits(buf, &bit_counter, 2);
ptransport->continuity_counter = readbits(buf, &bit_counter, 4);
// If adaptation_field_control = 10, then adaptation field only.
// If adaptation_field_control = 11, then adaptation field followed
// by payload.
if (ptransport->adaptation_field_control == (int)2 ||
    ptransport->adaptation_field_control == (int)3) {
    // Start decoding adaptation_field information.
    ptransport->adapt_field->adaptation_field_length = readbits(buf,&bit_counter,8);
    // If no information contained in adaptation_field, then skip it.
    if (ptransport->adapt_field->adaptation_field_length == 0)
        goto bailout;
    // If information in adaptation_field, then decode it.
    ptransport->adapt_field->discontinuity_indicator = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->random_access_indicator = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->elementary_stream_priority_indicator =
        readbits(buf,&bit_counter,1);
    ptransport->adapt_field->PCR_flag = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->OPCR_flag = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->splicing_point_flag = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->transport_private_data_flag = readbits(buf,&bit_counter,1);
    ptransport->adapt_field->adaptation_field_extension_flag =
        readbits(buf,&bit_counter,1);
    
    total_adpt_byte = total_adpt_byte + 1;

    // PCR_flag = 1 indicates Program_Clock_Reference is there.
    // PCR is useful when decoder is implemented with hardware.
    if (ptransport->adapt_field->PCR_flag == 1) {
        ptransport->adapt_field->program_clock_reference_base =
            readbits(buf,&bit_counter,33);
        ptransport->adapt_field->reserved_PCR = readbits(buf,&bit_counter,6);
        ptransport->adapt_field->program_clock_reference_extension =
            readbits(buf,&bit_counter,9);

        total_adpt_byte = total_adpt_byte + 6;
    }

    // OPCR_flag = 1 indicates Original_Progam_Clock_Reference is there.
    // OPCR is useful when insted of decoding a Transport stream,
    // programs are decoded from one and then reorganized into another,
    // different Transport stream. OPCR then refers to the PCR of the
    // first stream. The only case when OPCR = PCR is if the two streams
    // are identical.
    if (ptransport->adapt_field->OPCR_flag == 1) {

ptransport->adapt_field->original_program_clock_reference_base=
readbits(buf,&bit_counter,33);
ptransport->adapt_field->reserved_OPCR=
readbits(buf,&bit_counter,6);
ptransport->adapt_field->original_program_clock_reference_extension=
readbits(buf,&bit_counter,9);

total_adpt_byte=total_adpt_byte+6;
}

// splicing_point_flag = 1 when splicing occurs within Transport stream.
if (ptransport->adapt_field->splicing_point_flag == 1) {
    ptransport->adapt_field->splice_countdown=readbits(buf,&bit_counter,8);
    total_adpt_byte=total_adpt_byte+1;
}

// Check if there is private data.
if (ptransport->adapt_field->transport_private_data_flag == 1) {
    ptransport->adapt_field->transport_private_data_length=
    readbits(buf,&bit_counter,8);
    // Read in private-data. // memory allocated in main() for maximum length. // Tradeoff between memory and speed.
    for (i=0;i<ptransport->adapt_field->transport_private_data_length;i++) {
        ptransport->adapt_field->private_data_type[i]=readbits(buf,&bit_counter,8);
    }
    total_adpt_byte=
    total_adpt_byte+1+ptransport->adapt_field->transport_private_data_length;
}

// Adaptation_field_extension contains primarily splicing related information. // For practical purposes, there is no need to consider it at this point other // than recording it down.
if (ptransport->adapt_field->adaptation_field_extension_flag == 1) {
    ptransport->adapt_field->adaptation_field_extension_length=
    readbits(buf,&bit_counter,8);
    ptransport->adapt_field->ltw_flag=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->piecewise_rate_flag=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->seamless_splice_flag=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->reserved_adaptation=readbits(buf,&bit_counter,5);
    total_adpt_ext_byte = total_adpt_ext_byte+1;
}
if (ptransport->adapt_field->ltw_flag == 1) {
    ptransport->adapt_field->ltw_valid_flag=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->ltw_offset=readbits(buf,&bit_counter,15);
    total_adpt_ext_byte = total_adpt_ext_byte+2;
}

if (ptransport->adapt_field->piecewise_rate_flag == 1) {
    ptransport->adapt_field->reserved_piecewise_rate=readbits(buf,&bit_counter,2);
    ptransport->adapt_field->piecewise_rate=readbits(buf,&bit_counter,22);
    total_adpt_ext_byte = total_adpt_ext_byte+3;
}

if(ptransport->adapt_field->seamless_splice_flag == 1) {
    ptransport->adapt_field->splice_type=readbits(buf,&bit_counter,4);
    ptransport->adapt_field->DTS_next_AU32_30=readbits(buf,&bit_counter,3);
    ptransport->adapt_field->marker_bit1=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->DTS_next_AU29_15=readbits(buf,&bit_counter,15);
    ptransport->adapt_field->marker_bit2=readbits(buf,&bit_counter,1);
    ptransport->adapt_field->DTS_next_AU14_0=readbits(buf,&bit_counter,15);
    ptransport->adapt_field->marker_bit3=readbits(buf,&bit_counter,1);
    total_adpt_ext_byte = total_adpt_ext_byte+5;
}

total_adpt_byte=total_adpt_byte+total_adpt_ext_byte;

total_adpt_ext_byte =
    ptransport->adapt_field->adaptation_field_extension_length-total_adpt_ext_byte;

// Read in the stuffing bytes.
// Again, memory allocated in mainO.
for(i=0; i<total_adpt_ext_byte; i++) {
    ptransport->adapt_field->reserved_stuffing[i]=readbits(buf,&bit_counter,8);
}
}

} // end of if adaptation_field_extension_flag

total_adpt_byte=ptransport->adapt_field->adaptation_field_length-total_adpt_byte;

// Read in stuffing bytes.
// Memory allocated in mainO.
for(i=0;i<total_adpt_byte;i++) {
    ptransport->adapt_field->stuffing_byte[i]=readbits(buf,&bit_counter,8);
}
At this point, either no information in adaptation_field or finished reading it.
bailout:
   // 1 byte for the field adaptation_field_length.
   // total_payload_byte refers to the payload.
   total_payload_byte = 184 - 1 - ptransport->adapt_field->adaptation_field_length;
} // end of if adaptation_field_control

// Check if there is payload in the Transport packet.
// 01 indicates payload only; 11 indicates payload following adaptation_field.
if (ptransport->adaptation_field_control == (int)1 ||
    ptransport->adaptation_field_control == (int)3) {
    // Check for permanent error in Transport packet.
    if (ptransport->transport_error_indicator) {
        printf("bit error in transport packet\n");
        return 0;
    }

    // Read in payload. Memory allocated in main().
    for (i=0;i<total_payload_byte;i++) {
        ptransport->data_byte[i]=readbits(buf,&bit_counter,8);
    }
} // end of if adaptation_field_control -> payload exists.

// If a PES packet commences with this transport packet,
// make sure to note the length of it.
// Length_field for PES starts from the fourth byte and ends at
// the fifth byte of a PES packet.
// 256 = 2^8.
if (ptransport->payload_unit_start_indicator==1 && PSI_flag==0) {
    // If a video stream in payload, a length of 0 means unbounded.
    // Then detect end of packet by payload_start_indicator.
    // Use continuity check.
    // If end of file reached, just decode it.
    ptransport->old_pes_packet_length=ptransport->pes_packet_length;
    ptransport->pes_packet_length=
        ptransport->data_byte[4]*256+ptransport->data_byte[5];
}
if (ptransport->pes_packet_length==0) {
    ptransport->pes_packet_length=60000;
}

// If a Section commences with this transport packet, note its length.
if (ptransport->payload_unit_start_indicator==1 && PSI_flag==1) {
// pointer_field indicates the start of a PSI section.
buf_bit_counter=0;
pointer_field=readbits(ptransport->data_byte,&buf_bit_counter,8);

//buf_bit_counter+pointer+field points to the start of a section, i.e. table_id
//The section_length field starts 12 bits after that.
buf_bit_counter=buf_bit_counter+pointer_field+12;
ptransport->psi_section_length=
    readbits(ptransport->data_byte,&buf_bit_counter,12);
buf_bit_counter+=18;
ptransport->psi_version_number=
    readbits(ptransport->data_byte,&buf_bit_counter,5);
}

// Also, record down the total payload in bytes for this transport packet.
ptransport->payload_byte=total_payload_byte;

    return return_flag;
} // end of decodeTransport;

/////////////////////////////////////////////////////
// reassemblePES
//
// function:
//   assembles PES packets from segments carried by transport packets.
//
// entry:   transport_packet containing decoded information from decodeTransport
//   an array of pes_packets, indexed with their PIDs
//
// return:   error
//   assemble successful
//   assemble entire packet successful
//
// the fully assembled pes_packet is carried in the array.
/////////////////////////////////////////////////////

int // 0 is error; 1 is append successful; 2 is PES packet complete;
    reassemblePES(
STRUCT_TRANSPORT_PACKET* ptransport,
STRUCT_PES_PACKET** PESlist
    )
{

    FILE * out_file;
    int i;
    int start, end;
    int bytes_read;

    return 0;
}
// Check if start of PES packet.
if (ptransport->payload_unit_start_indicator==1) {
  if (ptransport->old_pes_packet_length==60000) {
    ptransport->old_pes_packet_length=0;
    // Records down the total pes packet payload in case of unspecified pes_length.
    PESlist[ptransport->PID]->total_payload_byte=
      PESlist[ptransport->PID]->packet_end;
    return 3;
  }
  // Allocate memory for STRUCT_PES_PACKET and buf if not done so already.
  if (PESlist[ptransport->PID]==0) {
    PESlist[ptransport->PID]=(STRUCT_PES_PACKET*)
      malloc(sizeof(STRUCT_PES_PACKET));
    memset(PESlist[ptransport->PID],0,sizeof(STRUCT_PES_PACKET));

    // 6 bytes: 3 for sync; 1 for id; 2 for length_field;
    // packet_buf will contain entire PES packet.
    PESlist[ptransport->PID]->packet_buf=(unsigned char*)
      malloc(ptransport->pes_packet_length+6);
    memset(PESlist[ptransport->PID]->
      packet_buf,0,ptransport->pes_packet_length+6);
  }

  // Initialization
  PESlist[ptransport->PID]->packet_start_indicator=
    ptransport->payload_unit_start_indicator;
  PESlist[ptransport->PID]->PES_packet_length=
    ptransport->pes_packet_length;
  PESlist[ptransport->PID]->packet_start=0;
  PESlist[ptransport->PID]->packet_end=0;
  PESlist[ptransport->PID]->new_packet_flag=1;

  // Allocate new buffer if existing one not large enough.
  if (PESlist[ptransport->PID]->PES_packet_length<ptransport->pes_packet_length) {
    printf("current buf not large enough, reallocate\n");
    printf("current size is %d, new size is %d\n",PESlist[ptransport->PID]->
      PES_packet_length,
    ptransport->pes_packet_length);
    PESlist[ptransport->PID]->PES_packet_length=ptransport->pes_packet_length;

    free(PESlist[ptransport->PID]->packet_buf);

    PESlist[ptransport->PID]->packet_buf=(unsigned char*)
      malloc(PESlist[ptransport->PID]->PES_packet_length+6);

    free(PESlist[ptransport->PID]->
      packet_buf,0,ptransport->pes_packet_length+6);
    }
memset(PESlist[ptransport->PID]->
packet_buf, 0, ptransport->pes_packet_length + 6);
}

// If not start of packet, and no memory allocated, then error
// or ignore packets till the start of PES packet received.
} else if (PESlist[ptransport->PID] == 0) {

printf("Start of PES Packet not received yet\n");
return 0;

// If start of PES packet expected but current packet not it.
// New_packet_flag stays 0 until start of PES packet.
// Then, it stays 1 until end of PES packet.
// Check for case when middle of packet loss.
} else if (PESlist[ptransport->PID]->new_packet_flag == 0) {

printf("New Start of PES packet expected\n");
return 0;

// Continuity Check if not first Transport packet of a PES packet.
// +16-1 when dealing with current_counter is 15 and incoming packet
// packet with counter 0.
// Check for possible duplication
// If duplication, then open file for write and reset pointer using fseek.
} else if (PESlist[ptransport->PID]->continuity_counter !=
(ptransport->continuity_counter + 16-1) % 16) {

printf("discontinuity\n");
return 0;
}

// Ready to buffer PES packet
// Note: packet_start and packet_end indexed on 0 being the first element
// packet_end points to the next available slot.
// Update continuity_counter;
PESlist[ptransport->PID]->continuity_counter = ptransport->continuity_counter;
start = PESlist[ptransport->PID]->packet_start;
end = PESlist[ptransport->PID]->packet_end;

// Buffer in the payload.
for (i = 0; i < ptransport->payload_byte; i++) {
PESlist[ptransport->PID]->packet_buf[end + i] = ptransport->data_byte[i];
}
// Output to file.
if ((out_file = fopen("pes_out_file_video", "ab")) == NULL) {
    printf("Unable to Access File\n");
    return 0;
}

bytes_read=fwrite(ptransport->data_byte,1,ptransport->payload_byte,out_file);
fclose(out_file);

// Update buffer entry and ending positions.
PESlist[ptransport->PID]->packet_end+=ptransport->payload_byte;

//return: 2 packet assembly complete
//Also set new_packet_flag=0, so looking for
//new start packet
//Also, initialize packet_end and packet_start;
//< when there are stuffing bytes at the end
//to complete a transport payload section.
if (PESlist[ptransport->PID]->packet_end==
PESlist[ptransport->PID]->PES_packet_length+6) {
    return 2;
}

return 1;
} // end of reassemblePES


// decodePES
//
// function:
//  decodes one PES packet at a time.
//
// entry: a PES_packet array
//  a PES_packet PID
//
// return: error
//  decode_complete
//
// output payload of a PES packet to raw_file

# 1 decode_complete
decodePES(
    STRUCT_PES_PACKET** PESlist,
    int PID
)
{
    FILE* out_raw_file;
    int total_payload_byte;
    int header_byte = 0; //for header info
    int header_extension_byte = 0; //for header_extension info
    int stuffing_byte_count = 0;
    int i;
    int byte_read;
    int bit_counter=0;

    //If not start of PES packet, skip header decoding.
    if (!PESlist[PID]->packet_start_indicator) {
        printf("No Start of Packet????\n");
        return 0;
    }

    //If start of PES packet, start decoding header info.
    if ((PESlist[PID]->packet_start_code_prefix=readbits(PESlist[PID]->packet_buf,&bit_counter,24))!=(int)0x000001) {
        printf("Not start of PES packet\n");
        return 0;
    }

    PESlist[PID]->stream_id=readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    PESlist[PID]->PES_packet_length=
        readbits(PESlist[PID]->packet_buf,&bit_counter,16);
    if (PESlist[PID]->PES_packet_length==0) {
        PESlist[PID]->PES_packet_length=60000;
        total_payload_byte=PESlist[PID]->total_payload_byte;
    } else {
        total_payload_byte=PESlist[PID]->PES_packet_length;
    }

    //If stream_id is program_stream_map, private_stream_2, ECM, EMM, program_stream_directory, DSMCC_stream or ITU-T Rec. H.222.1 type E stream, then data byte commences.
    if (PESlist[PID]->stream_id==(int)0xBC||PESlist[PID]->stream_id==(int)0xBF||
        PESlist[PID]->stream_id==(int)0xF0||PESlist[PID]->stream_id==(int)0xF1||
        PESlist[PID]->stream_id==(int)0xFF||PESlist[PID]->stream_id==(int)0xF2||
        PESlist[PID]->stream_id==(int)0xF8) {

    
}
goto bailout;

//If stream_id is padding_stream, then padding byte commences.
} else if (PESlist[PID]->stream_id==(int)0xBE) {

PESlist[PID]->padding_byte_indicator=1;
goto bailout;

//Start decoding additional header information.
} else {
    if (readbits(PESlist[PID]->packet_buf,&bit_counter,2)!=(int)2) {
        printf("should be binary sequence 10\n");
        return
    }

//Fill in the blanks with what to do with these stuff.
PESlist[PID]->PES_scrambling_control=
    readbits(PESlist[PID]->packet_buf,&bit_counter,2);
PESlist[PID]->PES_priority=readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->data_alignment_indicator=
    readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->copyright=readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->PTS_DTS_flags=readbits(PESlist[PID]->packet_buf,&bit_counter,2);
PESlist[PID]->ESCR_flag=readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->ES_rate_flag=readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->DSM_trick_mode_flag=
    readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->additional_copy_info_flag=
    readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->PES_CRC_flag=
    readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->PES_extension_flag=
    readbits(PESlist[PID]->packet_buf,&bit_counter,1);
PESlist[PID]->PES_header_data_length=
    readbits(PESlist[PID]->packet_buf,&bit_counter,8);

header_byte+=3;

//If PTS_DTS_flags == '10', then read in 5 bytes of additional info
if (PESlist[PID]->PTS_DTS_flags==(int)2) {
    for (i=0;i<5;i++) {
        PESlist[PID]->PTS[i]=
            readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    }
}
header_byte+=5;
}

//If PTS_DTS_flags == '11', then read in 10 bytes of additional info
//PTS_DTS contain Presentation/Decoding Time Stamps.
//They refer to times when aud/vid frames enter decoding buffers.
if (PESlist[PID]->PTS_DTS_flags==(int)3) {
    for (i=0;i<10;i++) {
        PESlist[PID]->PTS[i]=readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    }
    header_byte+=10;
}

//If ESCR_flag=='1', then read int 6 bytes of additional info
//ESCR contain Elementary_Stream_Clock_Reference
if (PESlist[PID]->ESCR_flag==(int)1) {
    for (i=0;i<6;i++) {
        PESlist[PID]->ESCR[i]=readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    }
    header_byte+=6;
}

//If ES_rate_flag=='1', then read in 3 bytes of additional info
if (PESlist[PID]->ES_rate_flag==(int)1) {
    for (i=0;i<3;i++) {
        PESlist[PID]->ES_rate[i]=readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    }
    header_byte+=3;
}

//If DSM_trick_mode_flag=='1', then read in 4 bytes
if (PESlist[PID]->DSM_trick_mode_flag==(int)1) {
    for (i=0;i<4;i++) {
        PESlist[PID]->DSM_trick_mode[i]=
            readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    }
    header_byte+=4;
}

//If additional_copy_info_flag == '1', then read in 1 byte
if (PESlist[PID]->additional_copy_info_flag==(int)1) {
    PESlist[PID]->additional_copy_info=
        readbits(PESlist[PID]->packet_buf,&bit_counter,8);
    header_byte+=1;
}
//If PESCRC_flag == '1', then read in 2 bytes
if (PESlist[PID]->PESCRC_flag==(int)1) {
    PESlist[PID]->previous_PES_packet_CRC =
        readbits(PESlist[PID]->packet_buf,&bit_counter,16);
    header_byte+=2;
}

//At this point, read in extension bytes and stuffing bytes following them.
//PES_header_data_length includes these extension bytes along with stuffing bytes.
i = readbits(PESlist[PID]->
        packet_buf,&bit_counter,PESlist[PID]->PES_header_data_length);

//Right now, the header bytes are assumed to be part of it.
total_payload_byte=total_payload_byte-PESlist[PID]->PES_header_data_length-3;
//3 header flags
}
bailout:

//Start buffering in raw_data.
//Consider the possibility that header info spills into the second packet
//immediately after the transport packet containing the start of a PES stream.
if (PESlist[PID]->padding_byte_indicator) {
    printf("stuffing byte only for padding purposes, ignore\n");
    return 0;
}

if((out_raw_file=fopen("raw_file_video","ab"))==NULL) {
    printf("error opening raw_file\n");
    return 0;
}
byte_read=fwrite(PESlist[PID]->packet_buf+6+3+PESlist[PID]->
    PES_header_data_length,1,total_payload_byte,out_raw_file);

fclose(out_raw_file);
return 1;
} //end of decodePES.

//readbits
//
//function:
//    Converts number of bits into integer.
//
// entry: buffer to read from
// bit_position to starting reading
// bits to read: maximum is 32 or (long int)

// return: integral conversion of the binary bits read

int readbits(unsigned char * buf,
             int *bit_ptr,
             int bits_to_read)
{
    int byte_pos, bit_pos;
    div_t result = div((*bit_ptr), 8); // Calculate byte.bit location from *bit_ptr.
    unsigned long temp; // 4 bytes; unsigned makes sure shifting in 0's

    byte_pos = result.quot;
    bit_pos = result.rem;

    // If reading bits from one single byte.
    if (bit_pos<bits_to_read<=8) {
        temp = (unsigned long) buf[byte_pos];
        // Get rid of the first 3 unused bytes. Then shift in 0's to fill them.
        temp = temp<<(bit_pos+24)>(32-bits_to_read); // long is 4 bytes
    }

    // If reading bits from two bytes
    else if (bit_pos<bits_to_read<=16) {
        // Convert into integer from binary manually.
        temp = (unsigned long)(buf[byte_pos]*256+buf[byte_pos+1]);
        // Get rid of the first 2 bytes. Then shift in 0's.
        temp = temp<<(bit_pos+16)>(32-bits_to_read); // long is 4 bytes
    }

    // If reading bits from 3 bytes
    else if (bit_pos<bits_to_read<=24) {
        temp = (unsigned long)((buf[byte_pos]*256+
                                buf[byte_pos+1])*256+buf[byte_pos+2]);
        temp = temp<<(bit_pos+8)>(32-bits_to_read);
    }

    // If reading bits from 4 bytes.
    else if (bit_pos<bits_to_read<=32) {
        temp = (unsigned long)(((buf[byte_pos]*256+buf[byte_pos+1])*
                                256+buf[byte_pos+2])*256+buf[byte_pos+3]);
    }
temp = temp<<bit_pos>>(32-bits_to_read);
}

// Update bit_ptr position.
*bit_ptr=(*bit_ptr)+bits_to_read;

return (int)temp;
} // end of readbits

// element_in_list
//
// function:
// Checks if element in in a given list
//
// entry: element
// list
// list length
//
// return: element_found
// element_not_found

int // return: 1 element_found; 0 element_not_found
element_in_list(int element, int* list, int list_count)
{

int i=0;

while(i<list_count) {
    if (element==list[i]) {
        return 1;
    }
    i++;
}

return 0;
} // end of element_in_list
References


[12] Course Notes, Chapter 4, Steinmetz and Nahrstedt, Summer 1996.