DEPARTAMENTO DE

ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

MULTIPLEXAGEM E DESMULTIPLEXAGEM DE FLUXOS DE TRANSPORTE MPEG-2

André Manuel Coutinho Almeida de Bragança

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

Rua dos Bragas, 4099 Porto Codex – PORTUGAL

MULTIPLEXAGEM E DESMULTIPLEXAGEM DE FLUXOS DE TRANSPORTE MPEG-2

André Manuel Coutinho Almeida de Bragança

MULTIPLEXAGEM E DESMULTIPLEXAGEM DE FLUXOS DE TRANSPORTE MPEG-2

André Manuel Coutinho Almeida de Bragança

Licenciado em Engenharia Electrotécnica e de Computadores
pela Faculdade de Engenharia da Universidade do Porto

Tese submetida para satisfação parcial
dos requisitos do programa de Mestrado em
Engenharia Electrotécnica e de Computadores
perfil de Telecomunicações

Faculdade de Engenharia da Universidade do Porto

Departamento de Engenharia Electrotécnica
e de Computadores

681.3(043)
BRA/MUL

0451
BRA/MUL

Porto, Junho de 1995

Tese realizada sob supervisão do Professor Doutor

Artur Pimenta Alves

Professor Associado do Departamento de Engenharia Electrotécnica e de Computadores

Faculdade de Engenharia da Universidade do Porto

AGRADECIMENTOS

Ao Professor Doutor Artur Pimenta Alves, por ter tornado possível a realização desta tese, pelo apoio prestado e pela confiança depositada.

A todos os meus colegas do INESC, pelas sugestões, disponibilidade e interesse demonstrado.

Aos meus colegas do Banco Borges & Irmão pelo optimismo constante com que sempre me apoiaram.

À minha família, por tudo.

À Isabel pelo tempo, paciência, amizade e, sobretudo, pela soma das pequenas ajudas nas alturas certas.

Agradeço ainda ao INESC, pelas condições de trabalho oferecidas.

Finalmente, agradeço à JNICT, entidade financiadora deste mestrado.

Resumo

Foi estabelecida pelo grupo MPEG (*Moving Picture Experts Group*) uma norma que define a representação codificada de vídeo e áudio associado para armazenamento ou transmissão digital. A especificação MPEG Sistema aborda a combinação de um ou mais fluxos, juntamente com informação de sincronismo, num único fluxo multiplexado.
Este texto descreve o desenvolvimento de uma ferramenta em software para a simulação e teste de fluxos conformes com a norma MPEG Sistema (mais precisamente, fluxos de transporte MPEG-2).
É mencionada a motivação deste trabalho e o contexto em que se realiza. Após uma breve descrição do conteúdo e evolução da norma, passa-se à sua implementação prática, através da construção de uma estrutura de multiplexagem e desmultiplexagem. São discutidos os aspectos mais relevantes da norma e das opções tomadas.

ABSTRACT

The MPEG (Moving Picture Experts Group) established a standard that defines the coded representation of video and associated audio, for digital storage and transmission. The MPEG Systems specification addresses the problem of combining one ore more data streams with timing information to form a single multiplexed stream.
This text describes the development of a software tool for the simulation and testing of MPEG System bitstreams (in particular, MPEG-2 transport stream).
The motivation for this work and its context is explained. After a brief description of the standard contents and its evolution, one is taken to its practical implementation, through the building of a multiplexing and demultiplexing structure. Relevant aspects of the standard and options taken are discussed.

Índice

1. INTRODUÇÃO

1.1 Motivação

O interesse pelas novas tecnologias de informação e comunicação é generalizado. A comunidade científica, fabricantes, governantes e a imprensa têm vindo a estimular o surgimento de elevado número de teleserviços com requisitos diferentes e por vezes difíceis de prever, para satisfazer (e criar) as necessidades de uma população heterogénea constituída por clientes domésticos e pelo mundo empresarial.

A transição de ambientes analógicos para ambientes digitais e os avanços significativos em *software*, *hardware* e redes de comunicação, a par de um esforço de normalização a nível mundial, têm permitido conceber a combinação de dados, texto, gráficos, imagens fixas, vídeo e som em aplicações *multimédia* cada vez mais adaptadas à comunicação humana.

Muitas destas aplicações, apesar da publicidade em torno delas, encontram-se ainda em fase de desenvolvimento experimental. O fabricante, devido ao rápido evoluir dos acontecimentos, enfrenta um elevado número de decisões, desde o sistema de compressão, passando pelo sistema de transmissão, até ao terminal do utilizador. Verifica-se que as maiores dificuldades estão no funcionamento do sistema na sua totalidade, mesmo que as partes que o constituem se encontrem devidamente testadas.

A normalização, para além de aumentar a economia e a compatibilidade, assume particular relevância na contribuição para a boa articulação entre as partes do sistema.

O vídeo é um dos tipos de dados que tem maior interesse comercial, mas também apresenta maiores exigências em termos de largura de banda. Assim, o assunto da compressão de vídeo tem vindo a merecer grande atenção. A norma MPEG (Moving Picture Experts Group) surgiu para responder a essas necessidades.

Esta dissertação realiza-se no âmbito da norma MPEG ao nível de sistema, onde o centro das atenções se concentra na sincronização entre vídeo e áudio e o transporte dos vários tipos de dados necessários para as aplicações multimédia.

1.2 Objectivos

Pretende-se o estudo da norma MPEG ao nível de sistema, com vista ao desenvolvimento de simuladores e de *software* de teste de fluxos binários codificados. A análise da arquitectura do instrumento produzido permitirá uma melhor compreensão da abrangência da norma e das suas possibilidades em termos de aplicação prática.

Para o efeito, procedeu-se ao desenvolvimento de um sistema de multiplexagem e desmultiplexagem, possibilitando o sincronismo entre áudio e vídeo, para a validação de fluxos binários concordantes com a norma, em ambiente UNIX ou MS-DOS.

A investigação feita baseou-se principalmente no documento da norma, devido à presente escassez de outras publicações no que concerne os aspectos de sistema.

Quanto à estrutura da tese, é de referir:

- no **capítulo 2** é feita uma breve referência a aspectos gerais do contexto que motivou o desenvolvimento da norma.

- no **capítulo 3** a norma MPEG é sumariamente descrita em termos do seu conteúdo e evolução.

- nos **capítulos 4 e 5** surge o projecto propriamente dito: como em consequência da interpretação, organização e condensação da norma se procedeu à construção de *software* para a sua implementação parcial, descreve-se aqui o método adoptado na elaboração da referida ferramenta, bem como as suas características mais relevantes. Finalmente, apresenta-se o código desenvolvido para facilitar a visualização de tais características.

- no **capítulo 6** são feitas considerações finais e apontam-se possíveis evoluções do tema.

2. CONTEXTO

O surgimento de uma multiplicidade de novas aplicações audiovisuais multimédia é um facto, e a norma MPEG desempenha um papel importante no seu desenvolvimento; cite-se como exemplo a televisão de alta definição HDTV (The Grand Alliance, 1995; Fox, 1995; Ninomiya, 1995; Chiang, 1994).

Para se poder falar da norma MPEG é necessário referir o contexto que a motivou e onde se processou a sua evolução. Aborda-se o conceito de DSM (*Digital Storage Media*), revêem-se noções gerais de compressão de dados e faz-se o enquadramento de acordo com os organismos de normalização.

2.1 Meios de Armazenamento Digital

A capacidade dos Meios de Armazenamento Digital (DSM) tem vindo a aumentar rapidamente na última década. O conceito de DSM inclui não só os equipamentos de armazenamento convencionais mas também os canais de comunicações. A sua capacidade poderá, portanto, ser caracterizada pelo armazenamento de *bytes* (ou *bits*) e/ou pelo débito da transferência de dados (*bytes* ou *bits* por segundo).

Em 1985, surgiu o CD-ROM (*Compact Disc Read-Only Memory*) como o meio de armazenamento para distribuição de grandes quantidades de dados (cerca de 650 M*Bytes*), mais eficiente em termos de custo devido à sua popularidade. Em 1993, foram vendidos cerca de 7 milhões de leitores (Asthana, 1994). Relacionado com o CD para áudio, suporta o acesso directo a sectores individuais de dados, permitindo uma velocidade de transferência entre os 1,5 Mbit/s e 5 Mbit/s.

Existem muitos outros meios de armazenamento, começando pela simples diskette (1,5 Mbps), passando pelo Laser Disk e pelos discos Bernoulli, Floptical, WORM e Opto-magnéticos. Anunciam-se já novos desenvolvimentos, como o SD-DVD (*Super Density Digital Video Disc*), talvez disponível em 1996, com previsões de vendas na Europa de 3 milhões de equipamentos de leitura (Fonseca, 1995).

No que diz respeito às redes de comunicação, a variedade também é grande: Ethernet, Token Ring e FDDI são as mais conhecidas. Verifica-se que as ligações de fibra óptica são a maneira mais eficaz, em termos de custo, para transmitir rapidamente grandes volumes de dados digitais. Contudo, a pesquisa nesta área é muito intensa, sendo a evolução acelerada e difícil de prever.

A literatura sobre o assunto é extensa, podendo-se encontrar facilmente mais referências: Fox, 1991; Comerford, 1994; Cole, 1993.

2.2 Esquemas de compressão

A compressão é essencial para o uso de áudio, imagens e vídeo em ambientes digitais. Sem ela, um M*byte* de espaço é ocupado por cerca de seis segundos de áudio de qualidade CD, por uma única imagem de 640 x 480 pixeis com cor de 24 *bits* ou por um único quadro (1/30 s) de vídeo CIF. Tem havido muita pesquisa neste campo, existindo múltiplas implementações que utilizam *software*, *hardware* ou ambos para uma grande variedade de métodos de compressão (Fox, 1991).

A figura seguinte mostra uma classificação possível dos vários métodos:

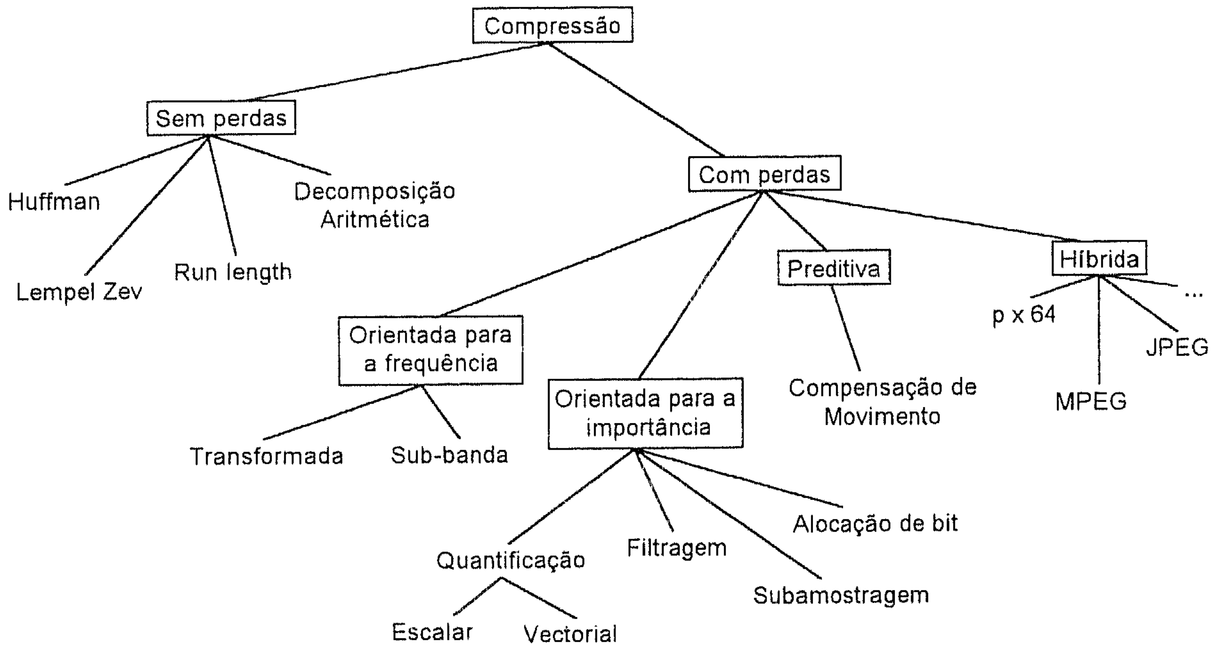

Figura 1. Esquemas de compressão (Fox, 1991)

Nos esquemas **sem perdas** (*Lossless*) a informação original pode ser perfeitamente recuperada. Não é adicionado ruído ao sinal (*noiseless*) e as redundâncias são eliminadas através de técnicas estatísticas ou de decomposição (*entropy coding*). Como exemplos temos a codificação de Huffman, que usa menos *bits* para os símbolos mais comuns, e a codificação *run-length*, que substitui sequências de símbolos iguais por pares símbolo-contagem. Estas técnicas proporcionam taxas de compressão não superiores a 3:1, sendo utilizadas apenas para aplicações especiais (*e.g.* imagens médicas).

A aproximação de compressão **com perdas** (*Lossy*) aproveita as não linearidades das capacidades analíticas do sistema visual e auditivo humano para proporcionar codificações com grandes taxas de compressão (de 10:1 a 50:1 em imagens fixas; de 50:1 a 200:1 em sequências vídeo) que, quando descodificadas, são subjectivamente similares ao original.

As técnicas **preditivas** (*Predictive*) fazem o cálculo de valores subsequentes pela observação de valores prévios e transmissão das diferenças (normalmente pequenas) entre os dados reais e calculados.

Um exemplo é a compensação de movimento, que se baseia no facto de imagens consecutivas numa sequência vídeo serem quase iguais ou terem um bloco de pixeis ligeiramente translaccionados. Esta técnica pode utilizar preditores *causais* (predição pura), em que se usam valores anteriores para o cálculo do novo valor, de implementação relativamente simples e com perdas de qualidade localizadas, ou preditores *não causais* (codificação interpolativa), em que se usam vizinhanças codificadas conjuntamente ou por blocos (usando transformadas), permitindo maior eficiência, embora com a desvantagem de as degradações se estenderem a toda a imagem. Espera-se que o desenvolvimento de processadores paralelos e redes neuronais facilite a análise de imagens e produção de vectores de movimento.

Dada a diferente sensibilidade do Homem às várias combinações de frequência espacial e temporal, utilizam-se técnicas de compressão **orientada para a frequência** (*Frequency-oriented*).

A codificação de *sub-banda* separa (utilizando filtros, por exemplo) as combinações de frequência, codificando com mais fidelidade aquelas mais importantes para a percepção humana.

A codificação com *transformadas* normalmente envolve frequências espaciais (*e.g.* em imagens isoladas). A aproximação mais comum aplica a transformada discreta do co-seno DCT (*Discrete Cosine Transform*), que é relacionada com a transformada rápida de Fourier.

Na compressão **orientada para a importância** (*Importance-oriented*) usam-se outras características, em vez da frequência. Um exemplo é o da *filtragem* das imagens, eliminando detalhes que não podem ser percebidos. Ou então, a codificação com mais *bits* de partes importantes da imagem, como as orlas (*edges*), em detrimento de regiões homogéneas e grandes. Pode-se também, fazer *subamostragem* (menos *bits* para a crominância).

A *quantificação escalar* mapeia os vários valores num número fixo de *bits*, enquanto que a *quantificação vectorial* usa vectores de valores bidimensionais, por exemplo, e mapeia-os num símbolo de código. São usados conjuntos de códigos das imagens, com os vectores mais importantes. Os vectores de dados são aproximados, minimizando o erro quadrático médio.

A **codificação híbrida** (*Hybrid coding*) combina várias aproximações. Por exemplo: DCT e ADPCM; codificação de sub-banda e DCT; ADPCM e quantificação vectorial.

Os sistemas e normas para compressão vídeo muitas vezes aplicam compensação de movimento para compressão temporal, codificação com transformadas para compressão espacial, e códigos de Huffman ou aritméticos para compressão estatística. A norma em estudo faz uso deste tipo de codificação.

2.3 Normalização de serviços Audiovisuais Multimédia

As relações entre as entidades que emitem recomendações e normas têm-se modificado à medida que as fronteiras entre os vários serviços se tornam vagas e se assiste à integração de vários tipos de dados. Verifica-se uma maior colaboração entre as várias organizações e a elaboração de textos comuns. Por outro lado, para além dos organismos oficiais, surgem consórcios cujas normas *de facto* têm uma forte influência na evolução das normas formais, resultando num processo mais orientado para o mercado.

Segundo o ITU-T[1], a normalização de serviços áudio visuais multimédia assenta nos seguintes pontos (Asatani, 1994):

- Definição do serviço.
- Método de codificação de informação áudio e vídeo.
- Tipo de serviços telemáticos.
- Definição de sinalização (controlo de terminais).
- Método de multiplexagem dos sinais.
- Modo de comunicação entre terminais.
- Avaliação do grau e da qualidade de serviço.

O estudo da normalização sobre teleconferência, conferência áudio-gráfica e serviços telemáticos tem sido levado a cabo pelos grupos de estudo SG (*Study Groups*) 1, 8, 12 e 15, tendo dado origem a séries de recomendações nas várias áreas:

- Série **F**: definições de serviço;
- Série **G**: codificação de áudio;
- Série **H**: modo de comunicação; função de controlo do meio; controlo de sinais; construção de terminais; codificação de vídeo; método de multiplexagem para vídeo, áudio e dados;
- Série **I**: protocolo de acesso;
- Série **P**: qualidade de comunicação;
- Série **T**: serviços de dados/telemáticos.

[1] *International Telecommunications Union - Telecommunication Standardization Sector*, originário, depois de reorganizado em 1993, do CCITT (*Comité Consultatif International de Telegraphie et Telephonie*).

A coordenação dos vários grupos é feita pelo JCG (*Joint Coordination Group*).

Nestas áreas, em colaboração com o ITU-T, e desenvolvendo trabalho de acordo com outros pontos de vista, surgem também o ITU-R[2] e o ISO/IEC JTC 1[3], este último organizado em *SC (Sub-Committees)* e WG (*Working Groups*). Na figura seguinte mostram-se os grupos cujas actividades se inter-relacionam:

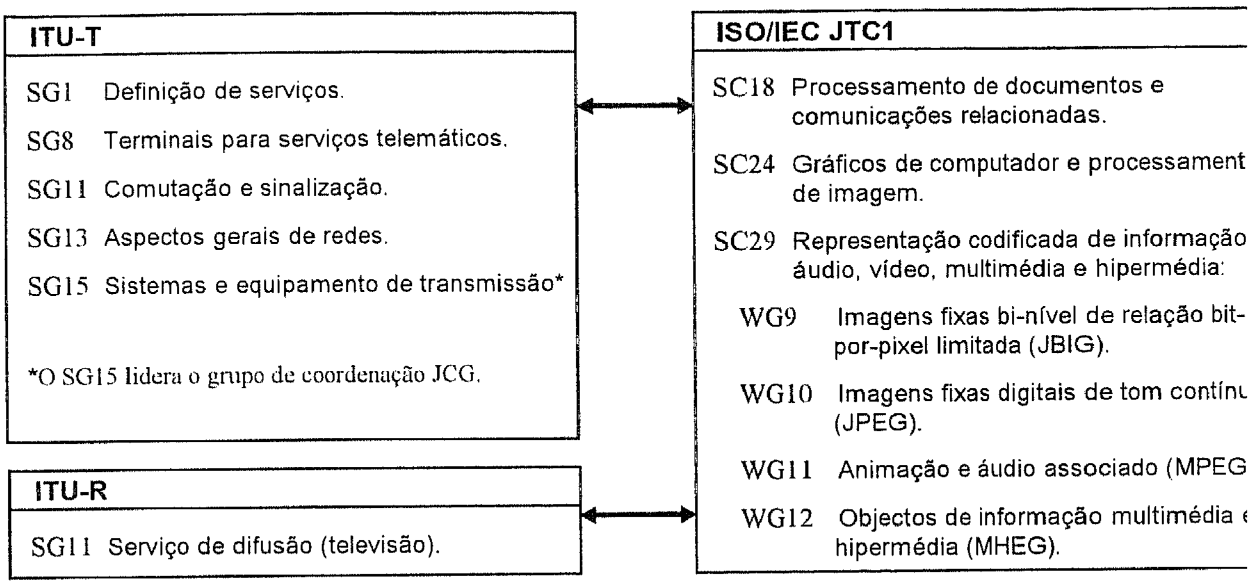
ITU-T
SG1 Definição de serviços.
SG8 Terminais para serviços telemáticos.
SG11 Comutação e sinalização.
SG13 Aspectos gerais de redes.
SG15 Sistemas e equipamento de transmissão*
*O SG15 lidera o grupo de coordenação JCG.
ITU-R
SG11 Serviço de difusão (televisão).
ISO/IEC JTC1
SC18 Processamento de documentos e comunicações relacionadas.
SC24 Gráficos de computador e processament de imagem.
SC29 Representação codificada de informação áudio, vídeo, multimédia e hipermédia:
WG9 Imagens fixas bi-nível de relação bit-por-pixel limitada (JBIG).
WG10 Imagens fixas digitais de tom contínu (JPEG).
WG11 Animação e áudio associado (MPEG
WG12 Objectos de informação multimédia hipermédia (MHEG).

Figura 2. Grupos relacionados com a normalização de serviços audiovisuais multimédia no ITU e ISO/IEC JTC1 (Asatani, 1994).

A normalização de técnicas de compressão de dados, principalmente a codificação de vídeo, tem sido um dos pontos de grande desenvolvimento e centro das atenções dada a sua importância.

Até muito recentemente os fabricantes de sistemas de videoconferência usavam algoritmos próprios e não compatíveis, impedindo a comunicação com equipamentos de outros fabricantes. A maioria das actividades de estandardização para a codificação de vídeo iniciaram-se com o trabalho do CCITT sobre videoconferência e videotelefonia (Warwick, 1993). Em fins de 1990, através da recomendação H.320, começaram a ser ratificadas uma série de normas de inter-operabilidade de videoconferência que tinham sido um ponto de

[2] *International Telecommunications Union - Radiocommunication Sector*, originário do CCIR (*International Consultative Committee on Broadcasting*).
[3] *International Standards Organization / International Electrotechnical Commission - Joint Technical Committee 1.*

partida para grupos com interesses noutros segmentos do mercado do vídeo digital. Estas normas, para a compatibilidade na compressão, transmissão, troca e apresentação de áudio e vídeo, incluem, entre outras, a recomendação H.261 sobre codificação de vídeo a p x 64 kbps.

Em face da convergência, cada vez maior, entre a indústria das telecomunicações, computadores e electrónica de consumo, através da partilha de tecnologia digital, a ISO fomentou um esforço para o desenvolvimento de uma norma para vídeo e áudio associado em DSM, conhecido como MPEG[4] (Le Gall, 1991). As actividades deste grupo (ISO/IEC JTC 1/ SC29/WG11 - ver Figura 2) começaram em 1988 e cobrem, para além da compressão de vídeo e áudio, os aspectos ligados à sua sincronização e multiplexagem dos fluxos de dados resultantes. Inicialmente, para débitos na ordem dos 1,5 Mbps, a norma MPEG, numa segunda fase dita MPEG-2, contempla débitos maiores, formatos de vídeo com qualidade parametrizável e transmissão em meios sujeitos a ruído. Encontra-se em planeamento uma fase MPEG-4 para aplicações de débito muito baixo (*e.g.* televigilância). A colaboração entre a ISO e o ITU é estreita, sendo os documentos submetidos a aprovação elaborados em conjunto.

Inicialmente, no mesmo grupo de trabalho da ISO, o JPEG[5] foi bastante importante para o desenvolvimento do MPEG, devido à sua actividade no estudo da compressão de imagens fixas (Le Gall, 1991). O MPEG faz uso de sequências de imagens, mas aproveitando a redundância entre elas.

Em adição à codificação, são necessárias normas para as camadas superiores do processo de desenvolvimento de aplicações multimédia. Neste contexto, têm surgido vários esforços organizados, tais como o MHEG[6] e o HyTime[7] (Fox, 91).

O ITU-T SG 15 considera três áreas de estandardização, relacionadas com desenvolvimentos futuros de serviços audiovisuais multimédia (Asatani, 94). A primeira área abrange a regulamentação destes serviços no âmbito da RDIS-BL[8], onde a codificação de vídeo, segundo a recomendação H.262 (MPEG-2), é uma das tecnologias centrais. A segunda área contempla a normalização dos serviços multimédia implementados em redes móveis ou na rede telefónica, estando prevista para finais de 1996 uma recomendação

[4] *Moving Picture Experts Group.*
[5] *Joint Photographic Experts Group.*
[6] *Multimedia and Hypermedia Information Coding Expert Group.*
[7] ISO/IEC 10744.
[8] Rede Digital com Integração de Serviços, de Banda Larga, cujas recomendações básicas foram concluídas em 1990 pelo CCITT SG 18 (hoje ITU-T SG 13). É considerada a plataforma chave para o futuro das comunicações multimédia e aplicações de alta velocidade (de Prycker, 1991).

revista e integrada sobre vídeotelefones. Na terceira área serão melhoradas recomendações existentes usadas na RDIS[9] (*e.g.* teleconferência multiponto e questões de segurança).

Fora destas organizações de normalização há uma variedade de actividades que poderão afectar todo o desenvolvimento das normas. Um dos exemplos é a *Internet*, onde decorrem muitas experiências e onde já existem alguns serviços disponíveis. Por exemplo, a WWW (*World Wide Web*) disponibiliza bases de dados com texto e elementos audiovisuais 24 horas por dia. Outro exemplo é o surgimento de grupos como o IETF[10], cujas discussões sobre várias especificações poderão vir a ter impacto na sua normalização.

Devido ao objectivo do presente trabalho, no capítulo seguinte é feita uma descrição do desenvolvimento e do conteúdo das normas MPEG.

[9] Rede Digital com Integração de Serviços, de banda estreita (Nunes, Casaca, 1992).
[10] *Internet Engineering Task Force.*

3. ASPECTOS DA NORMA MPEG

3.1 Fases

Em 1988, o esforço de estandardização do grupo MPEG pretendia obter resultados rapidamente, de modo a evitar o aparecimento de múltiplas normas *de facto* incompatíveis entre si. O objectivo era a definição de métodos eficientes de armazenamento e recuperação, em tempo real, de sinais de vídeo e áudio associado, usando um débito binário até 1,5 Mbps (um valor típico para CD). Com aplicações multimédia interactivas em vista, estipularam-se débitos na ordem dos 1,2 Mbps para vídeo e 250 Kbps para dois canais de áudio.

Para isso, seguiu-se uma metodologia onde, depois de identificados todos os objectivos sobre os quais deviam ser concentrados os esforços, as várias entidades que compunham o grupo entraram em competição entre si, apresentando propostas para análise e teste. Finalmente, foi feita a convergência, de uma forma cooperante, onde as ideias e técnicas mais prometedoras foram integradas numa única solução (Le Gall, 1991). Depois de ter sido apresentado em 1990, de acordo com o calendário estabelecido, o esboço da norma (*Committee Draft*), esta foi aprovada por unanimidade em Novembro de 1992 e publicada no documento ISO/IEC 11172.

O sucesso deste empreendimento foi tão grande que foi lançada uma segunda fase, apelidada de MPEG-2, estendendo os objectivos da fase anterior, dita MPEG-1, para contemplar débitos de vídeo de qualidade crescente (desde 3 Mbps até 15 Mbps, para televisão convencional; superiores, para HDTV[11]) e novas frequências de amostragem de áudio. Esta fase foi iniciada em 1991 e concluída em 1994, incluindo os objectivos entretanto programados para uma terceira fase (Ferreira, 1994). Espera-se a sua adopção pelo ITU e pela ISO já em 1995.

Existe agora a fase MPEG-4 que visa débitos reduzidos, na ordem das dezenas de Kbps.

[11] *High Definition TV.*

Os documentos MPEG são publicados em quatro partes:

- Parte 1 - **Sistemas**. Define uma estrutura multiplexada para combinação de dados áudio e vídeo e representação da informação temporal necessária para a sua apresentação sincronizada, em tempo real.

- Parte 2 - **Vídeo**. Especifica a representação codificada de vídeo e o seu processo de descodificação.

- Parte 3 - **Áudio**. Especifica a representação codificada de áudio e o processo de descodificação necessário para reconstruir o sinal.

- Parte 4 - **Testes de conformidade**. Procedimentos para determinação das características dos fluxos de *bits* codificados e do processo de descodificação, bem como para teste de conformidade com o estabelecido na norma.

Seguindo esta organização, os pontos seguintes procuram explicar de uma forma sumária o que está por trás de cada uma das três primeiras partes. Como o foco do trabalho são as especificações de sistema, o vídeo e o áudio são ilustrados recorrendo à fase MPEG-1, relevando os aspectos mais importantes.

3.2 Vídeo

3.2.1 Normas relacionadas

JPEG (Joint Photographic Experts Group)

A norma JPEG é um algoritmo geral para codificação de imagens fixas, desenvolvido para ajudar a captura de imagens individuais numa sequência vídeo.

Normalmente, é utilizada codificação sequencial usando quantificação escalar da transformada DCT e codificação de Huffman (ou aritmética) para compressão. O descodificador reverte o processo (o algoritmo é simétrico). As aplicações controlam a quantificação e as tabelas de Huffman (Ang, 1991).

Cada componente da imagem original (*e.g.* Y, U e V) é dividida em blocos não sobrepostos de 8x8 pixeis. Estes blocos são transformados usando uma DCT de 8 por 8, obtendo-se 64 coeficientes que representam o conteúdo em termos de frequência do bloco. O valor do coeficiente no canto superior esquerdo mede o termo **dc** (frequência nula). Os outros valores são termos **ac** que medem a energia do sinal com o aumento da frequência horizontal, da esquerda para a direita, e com o aumento da frequência vertical, de cima para baixo. Uma imagem 8x8 de valor constante, por exemplo, apenas tem o termo **dc** diferente de zero.

Em seguida, os coeficientes DCT são quantificados. O passo de quantificação varia com o componente (por exemplo, a luminância é mais importante para o olho humano que a crominância) e com a frequência (os coeficientes de maior frequência são subjectivamente menos importantes).

Após a quantificação, os coeficientes são reordenados num vector, através de uma leitura em zig-zag que os coloca aproximadamente em ordem crescente de frequência, e codificados sem perdas com o algoritmo de Huffman (codificação de comprimento variável). A codificação dos termos **dc** e **ac** é feita com parâmetros diferentes para aumentar a eficiência.

A norma inclui modos opcionais de compressão e descompressão. É suportada codificação progressiva, onde a imagem é codificada usando várias passagens, permitindo ao utilizador visualizar versões da imagem cada vez mais detalhadas. É também especificada codificação hierárquica, onde imagens de mais baixa resolução são acedidas antes de imagens de maior resolução. Uma extensão do algoritmo, utilizando um preditor e codificador de entropia, permite codificação sem perdas.

Recomendação CCITT H.261

Esta recomendação especifica um método de comunicação para videotelefonia. É também referida como p x 64, pois pode ser usada para produzir sequências vídeo comprimidas a débitos de p x 64 Kbps, sendo p um inteiro entre 1 e 32. Para p = 1, apenas é possível transmitir um sinal de baixa qualidade. Para p = 32, transmitem-se sinais vídeo de alta qualidade a 2 Mbps (Ang, 1991).

A organização e interpretação dos *bits* é especificada de modo a que *codecs* de fabricantes diferentes possam estabelecer uma sessão. Embora o codificador CCITT seja mais complexo que o JPEG, o descodificador é mais simples.

A codificação híbrida conjuga a transformada DCT (para blocos 8x8) e a codificação preditiva (com estimação de movimento envolvendo blocos de luminância 16x16), usando memória para comparação de macroblocos. A codificação espacial utiliza DCT seguida de quantificação escalar. Um filtro em anel remove ruído de alta frequência. Dependendo do preenchimento do *buffer* de saída, o tamanho de quantificação pode ser variado para modificar o débito. Um multiplexador providencia codificação estatística.

3.2.2 MPEG

3.2.2.1 Algoritmo ISO/IEC 11172-2

O algoritmo assegura características desejáveis para uma gama variada de aplicações: acesso aleatório; pesquisas rápidas para a frente e para trás; visualização para a frente e para trás; sincronização áudio e vídeo; robustez contra erros; atrasos limitados a 150 ms; editabilidade.

Tal como na recomendação H.261, o algoritmo proposto usa compressão *intra-frame* (imagem a imagem) e *inter-frame* (entre imagens). Por um lado, os requisitos de qualidade exigem o ganho em eficiência da compressão *inter-frame*; por outro lado, a acessibilidade ao fluxo comprimido é facilitada pela compressão *intra-frame*. A funcionalidade do diagrama de blocos do codificador H.261 é, em grande parte, aplicável. Contudo, os detalhes da quantificação e compensação de movimento são diferentes.

Assim, há uma grande sobreposição entre as capacidades da H.261 e do MPEG, podendo alguns dos componentes VLSI envolvidos ser usados em ambos.

Este algoritmo utiliza compensação de movimento em blocos 16x16 e codificação DCT de blocos 8x8, seguida de quantificação e codificação de entropia. A compensação de movimento envolve codificação preditiva e interpolativa. Como no JPEG, os quadros (*frames*) de referência são codificados, utilizando métodos baseados na DCT. Estas imagens comprimidas moderadamente servem de ponto de acesso aleatório e denominam-se imagens intra (I). Os quadros previstos (P) são baseados na imagem anterior (I ou P). Os quadros interpolados, imagens bidireccionais (B), oferecem a maior taxa de compressão, mas baseiam-se nas imagens passadas e futuras, não servindo como referência. Finalmente, a informação de movimento é codificada estatisticamente.

3.2.2.2 Estrutura

A estrutura da representação codificada de vídeo têm vários níveis. É constituída por um cabeçalho de sequência vídeo e pelas camadas da sequência. Estas são classificadas em: grupo de imagens GOP (*Group Of Pictures*), imagens (*Pictures*), fatias (*Slices*), macroblocos e blocos. As duas camadas inferiores incluem compensação de movimento e dados para a DCT.

A sintaxe do fluxo de vídeo reflecte a estrutura em camadas, sendo bastante fácil identificar vários pontos do fluxo devido ao uso de um prefixo único (0x000001) para cada delimitador (*start code*):

nome	valor do *start code* (hexadecimal)
picture_start_code	00
slice_start_code	01 até AF
reserved	B0
reserved	B1
user_data_start_code	B2
sequence_header_code	B3
sequence_error_code	B4
extension_start_code	B5
reserved	B6
sequence_end_code	B7
group_start_code	B8
system start codes	B9 até FF

Veja-se como exemplo um extracto do "filme" BIRDSHOW (fornecido com o descodificador de MPEG para Windows, versão 2.0, da Xing Technology Corp.), em hexadecimal:

```
000000    00 00 01 B3  0A 00 78 19  FF FF E0 84  00 00 01 B2
000010    00 00 0C E8  00 00 01 B8  0C 00 06 00  00 00 01 00
000020    00 4F FF FF  FF F1 00 00  01 B2 FF FF  00 00 01 01
000030    43 F8 F9 30  EE 26 CD 69  28 92 9C 2E  CD 7A E8 3A
 . . .
000BC0    7A 86 79 C8  30 1A 92 62  5A 72 49 6C  30 C6 75 44
000BD0    C0 00 00 01  00 00 8F FF  FF FF FC 00  00 01 B2 FF
000BE0    FF 00 00 01  01 43 F8 F9  30 EE 26 CD  69 28 92 9C
000BF0    2E CD 7A E8  3A F6 8A 24  05 E2 7D DD  C9 28 0F A9
 . . .
02C2D0    07 9C 83 01  A0 26 25 A7  24 96 C3 0C  67 34 4C 00
02C2E0    00 01 B7
```

A flexibilidade dos parâmetros das sequências MPEG proporciona uma gama larga de resoluções espaciais e temporais, permitindo o uso de vários débitos.

A compressão vídeo de resoluções de televisão (360 x 240 pixeis) origina um débito de cerca de 1.2 Mbps. Este pode ser usado, por exemplo, em CD-ROM e em canais T1 (1.544 Mbps).

Para garantir a inter-operabilidade de equipamento, usando MPEG, foi definido um subconjunto de parâmetros. Todos os descodificadores MPEG devem ser capazes de operar um "núcleo" de fluxo de *bits* com os seguintes limites superiores: resolução de 720 x 576 pixeis, 30 quadros por segundo, débito de 1.86 Mbps e um *buffer* de descodificador de 46 K*bytes*.

3.3 Áudio

O processo de codificação utiliza um esquema perceptual (compressão orientada para a importância) e é organizado em três camadas (*layers*) de complexidade e desempenho crescentes. Cada uma visa contextos de aplicação distintos, usando débitos binários diferentes (Ferreira, 1994).

O áudio com qualidade de CD requer dois canais de 706 Kbps. O débito, após compressão, é de 128 Kbps por canal (ou possivelmente de 64 Kbps), ou seja, uma compressão de 1.4 Mbps para 0.256 Mbps.

São suportadas taxas de amostragem de 32, 44.1 e 48 KHz com 16 *bits* por amostra; os atrasos serão inferiores a 80 ms; serão endereçadas unidades inferiores a 1/30 s.

As disposições da norma com valor normativo dizem respeito só à estrutura de trama e ao processo de descodificação.

3.4 Sistema

3.4.1 MPEG-1 (ISO/IEC 11172-1)

A especificação MPEG ao nível de sistema aborda a combinação de um ou mais fluxos (*bitstreams*) de áudio (ISO/IEC 11172-3), vídeo (ISO/IEC 11172-2) ou privados, com informação de sincronismo num único fluxo de *bits* multiplexado (ISO/IEC 11172-1) para armazenamento ou transmissão digital.

Assim, um fluxo ISO/IEC 11172-1 é constituído por duas camadas. A camada interior, de *compressão*, é constituída pela codificação dos dados. A camada exterior, do *sistema*, suporta as funções necessárias para o uso de um ou mais fluxos comprimidos de dados num sistema:

1. Sincronização da apresentação da informação descodificada;
2. Construção do fluxo multiplexado;
3. Gestão de *buffers* para codificação de dados, iniciação e acesso aleatório;
4. Identificação temporal.

A camada do sistema divide-se em duas subcamadas: uma para operações relativas à multiplexagem (funções 3 e 4) - *Pack Layer;* outra para operações específicas dos fluxos (funções 1 e 2) - *Packet Layer.*

3.4.1.1 Operações relativas à multiplexagem

A recuperação de dados, a partir do DSM (*Digital Storage Media*), o ajuste de relógios e a gestão de *buffers* estão intimamente relacionados. Se o débito do DSM é controlável, deve ser ajustado para não sobrecarregar (nem subcarregar) os *buffers* de descodificação. Se o débito do DSM não é controlável, os descodificadores dos fluxos elementares devem sincronizar-se com o DSM para que os *buffers* funcionem correctamente.

Estas tarefas são reguladas pelos cabeçalhos dos *packs*. Estes especificam a altura em que os dados os dados devem entrar no descodificador de sistema a partir do DSM, servindo o horário de chegada como referência para gestão de *buffers* e correcção de relógios.

O primeiro *pack* do fluxo ISO/IEC 11172 fornece os parâmetros necessários para informar o descodificador dos recursos necessários (por exemplo, o débito máximo de dados e o maior número de canais vídeo simultâneos).

3.4.1.2 Operações específicas dos fluxos de bits

Os fluxos ISO/IEC 11172 são formados pela multiplexagem de fluxos elementares. Estes, em adição aos fluxos áudio e vídeo, podem ser privados, reservados ou de preenchimento (*padding*). São suportados até 32 fluxos de áudio e 16 de vídeo. São previstos dois fluxos privados de dados: um completamente privado e outro obedecendo a uma sintaxe para o suporte de sincronização e gestão de *buffers*. Pode ser definido um número ilimitado de sub-fluxos. Os fluxos são temporalmente divididos em pacotes seriados, de comprimento fixo ou variável, que contêm dados codificados (de um único fluxo elementar).

Os cabeçalhos dos pacotes contêm um código de identificação do fluxo (*stream_id*) que torna possível a desmultiplexagem e reconstrução dos fluxos elementares a partir do fluxo ISO/IEC 11172.

Na camada dos pacotes existem etiquetas temporais (*time stamps*) que se referem à descodificação individual dos fluxos elementares. Para se obter a sincronização entre múltiplos fluxos, os codificadores guardam as etiquetas na altura da captura. Estas são transmitidas com os dados associados, e os descodificadores usam-nas para organizar a apresentação dos dados.

A camada dos pacotes está relacionada com a camada de compressão, através da ligação das etiquetas temporais, codificadas nos cabeçalhos, aos tempos de apresentação dos dados descodificados. No entanto, os pacotes não necessitam de começar nos *start codes* do fluxo comprimido.

3.4.1.3 STD

O STD (*System Target Decoder*) é um descodificador hipotético que permite definir exactamente os eventos de descodificação e os seus instantes de ocorrência, pois estabelece um modelo para o processo de descodificação durante a construção dos fluxos ISO/IEC 11172-1.

Como cada fluxo faz a parametrização do STD (*e.g.* tamanho dos *buffers*), o codificador deve certificar-se de que a apresentação do fluxo é feita correctamente no STD correspondente. O descodificador real deve compensar as suas diferenças em relação ao modelo STD.

A informação temporal transportada ao nível de sistema é a codificação de uma amostra do valor do relógio de sistema, cuja frequência é de 90 KHz.

Os dados de um pacote são entregues ao *buffer* do fluxo elementar. O cabeçalho do pacote pode ser usado para controlar o sistema.

No instante da descodificação de uma unidade de acesso, esta é retirada do *buffer* e descodificada para uma unidade de apresentação.

No caso de um fluxo de vídeo, as unidades de acesso podem não estar na ordem de apresentação. Assim, imagens-I ou imagens-P guardadas antes de imagens-B devem ser atrasadas no *buffer* de reordenação, até ser descodificada a próxima imagem-I ou imagem-P, antes de serem apresentadas.

O instante em que é apresentada uma unidade de acesso coincide com o de descodificação, se as unidades de apresentação não necessitarem de ser reordenadas.

3.4.1.4 Estrutura

Um fluxo ISO/IEC 11172-1 é constituído por um ou mais fluxos elementares multiplexados conjuntamente. Estes, por sua vez, são constituídos por unidades de acesso (representações codificadas das unidades de apresentação). A unidade de apresentação para um fluxo elementar de vídeo é uma imagem. A de um fluxo de áudio, corresponde às amostras de uma trama áudio.

Os dados dos fluxos elementares são guardados em pacotes. O cabeçalho do pacote começa com um *start code* de 4 *bytes*, que identifica o fluxo e pode conter etiquetas temporais referentes à primeira unidade de acesso. Os dados do pacote contêm um número variável de *bytes* contíguos do fluxo elementar.

Os pacotes são organizados em *packs*, cujo cabeçalho é usado para armazenar informação temporal e de débito.

O fluxo multiplexado começa com um cabeçalho de sistema que pode ser repetido opcionalmente. Este contém parâmetros de sistema definidos no fluxo.

A representação gráfica da sintaxe é feita no apêndice F da norma.

3.4.2 MPEG-2 (ISO/IEC 13818-1)

3.4.2.1 Tipos de fluxo

A fase MPEG-2 contempla dois tipos de multiplexagem.

Os fluxos de programa (PS - *Program Stream*) são similares aos definidos na fase MPEG-1 (documento ISO/IEC 11172-1). Resultam da combinação de pacotes dos fluxos elementares PES (*Packetized Elementary Stream*), com uma base temporal comum, num único fluxo de *bits*.

Cada pacote PES é constituído apenas por um fluxo elementar, podendo ser de comprimento variável, geralmente relativamente grande.

O uso deste tipo de fluxos é projectado para ambientes relativamente livres de ruído.

O segundo tipo de fluxo multiplexado, apelidado de transporte (TS - *Transport Stream*), está adaptado a ambientes onde há grandes probabilidades de erro, pois é constituído por pequenos pacotes de 188 *bytes*.

A conversão entre os dois tipos de fluxo está prevista na especificação da norma, utilizando--se pacotes PES, comuns às duas sintaxes.

3.4.2.2 Fluxo de Transporte

Um fluxo de transporte é constituído por programas. Estes, por sua vez, contêm fluxos elementares.

Um pacote de transporte (TP, *Transport Packet*) contém pacotes PES. Um pacote PES inclui unidades de acesso de um fluxo elementar.

Cada TP é identificado pelo PID (*Packet Identification*). Este é referenciado através das Tabelas de Informação Específica de Programa (PSI, *Program Specific Information*), ficando identificado o conteúdo do TP, pois a cada PID está associado apenas um tipo de informação.

A representação gráfica da sintaxe encontra-se no apêndice F da norma.

Tal como no MPEG-1, a sincronização entre os vários fluxos elementares é feita através de PTS (*Presentation Time Stamp*) e DTS (*Decoding Time Stamp*), geralmente em unidades de 90 KHz. Uma das principais funções do cabeçalho do pacote PES é o transporte destes selos temporais.

Para a temporização do fluxo de sistema usa-se: o SCR (*System Clock Reference*), no caso de PS, e o PCR (*Program Clock Reference*), para cada programa do TS. Estas referências temporais são especificadas em unidades de 27 MHz.

4. DESCODIFICADOR DE FLUXOS BINÁRIOS MPEG SISTEMA

4.1 Leitura da norma

A leitura e compreensão da norma MPEG revela-se bastante complexa. Ao nível de sistema, a literatura é escassa e, a norma em si, dado o seu carácter generalista, não se revela pródiga em exemplos concretos. Um único exemplo, de MPEG-1, foi encontrado no apêndice 1-A.5 do documento ISO/IEC 11172-1.

A parte vídeo, por outro lado, encontra-se bem explorada. É fácil encontrar documentação e, em termos de *software*, existem numerosos descodificadores e codificadores disponíveis, por exemplo, *via Internet*.

Aproveitando os ensinamentos de um pequeno programa[12] que efectua uma simples busca (*parser*) a fluxos de vídeo MPEG-1, para extracção de informação constante nos cabeçalhos dos pacotes, decidiu-se, como ponto de partida para este trabalho, implementar um *parser* de fluxos de sistema MPEG-1.

4.2 Parser de MPEG-1

O programa aqui implementado lê um ficheiro binário com um fluxo MPEG-1 sistema, gerando outro ficheiro. Este contém os *start_codes* encontrados durante o *parse*, o endereço destes e o número de *bytes* entre *start_codes* consecutivos (ver ponto 3.2.2.2). A informação contida nos cabeçalhos dos vários blocos, que constituem o fluxo analisado, é interpretada e visualizada. Os fluxos elementares são reconstituídos e colocados em ficheiros separados.

O fluxo utilizado para testar o programa foi obtido na *Internet*, sendo originário de um CD de teste. É constituído por uma sequência de 142 imagens com cenas de um filme de futebol americano e por uma sinusóide de 1 KHz como áudio.

[12] "InfoMPEG " de Dennis Lee (1993).

Mostra-se de seguida um extracto do resultado do *parser*.

```
Parsing de AVI (endereco, bytes seguintes, start_code, descricao)

000000     8  BA  pack_start_code
 -------------------------------------------------
| system_clock_reference = 12610 (0.140111 s)
| mux_rate = 3486 (174300 bytes/s)
 -------------------------------------------------

00000C    14  BB  system_header_start_code
 -------------------------------------------------
| header_length = 12 bytes
| rate_bound  = 3486 (debito maximo do mux e' de 174300 bytes/s)
| audio_bound = 1 stream(s)
| (n. maximo de fluxos audio multiplexados em simultaneo)
| fixed_flag = 1 (debito fixo)
| CSPS_flag  = 1 (fluxo obedece aos parametros restringidos CSPS)
| system_audio_lock_flag = 1
| (relacao constante da taxa de amostragem com relogio de sistema)
| system_video_lock_flag = 1
| (relacao constante da taxa de imagens com relogio de sistema)
| video_bound = 1 stream(s)
| (n. maximo de fluxos video multiplexados em simultaneo)
| reserved_byte = 255
| Fluxo(s) elementar(es) = 2
|
| Fluxo E0 : Video stream - number 0
|            STD_buffer_bound_scale    = 0
|            STD_buffer_size_bound     = 40
|            (Tamanho de buffer maximo = 5120 bytes)
|
| Fluxo C0 : Audio stream - number 0
|            STD_buffer_bound_scale    = 0
|            STD_buffer_size_bound     = 32
|            (Tamanho de buffer maximo = 4096 bytes)
 -------------------------------------------------

00001E    17  E0    Video stream - number 0
 -------------------------------------------------
| packet_length  = 2288 bytes
| stuffing_bytes = 3
| STD_buffer_scale   = 0
| STD_buffer_size    = 40
| (Tamanho de buffer = 5120 bytes)
| presentation_time_stamp = 40236 (0.447067 s)
| decoding_time_stamp     = 36486 (0.4054 s)
| packet_data    = 2273 bytes
 -------------------------------------------------

000033     8  B3      v(sequence_header_code)
00003F     4  B8      v(group_start_code)
000047     4  00          v(picture_start_code)
00004F    14  01              SLICE
...
```

Os fluxos elementares obtidos servem para testar a implementação do codificador descrito no capítulo 5.

Entretanto, foi criada uma estrutura base de *software* que serve como ponto de partida para um *parser* de fluxos MPEG-2.

Foram também aclarados os vários conceitos envolvidos e constatada a instabilidade da norma, com permanentes actualizações de pormenor.

A estrutura de dados desenvolvida tem em vista o acesso aos *bits* individuais dos *bytes* lidos, tendo sido pensada para permitir o seu uso quer no sistema operativo UNIX quer em MS-DOS.

4.3 Parser de MPEG-2

O programa lê um ficheiro MPEG-2 sistema na sua forma binária.

No caso de um PS (fluxo de programa), como este é semelhante ao MPEG-1, e não foram encontrados fluxos de teste, a leitura é feita usando (ver pág. 44 a 55) as rotinas do *parser* descrito no ponto anterior.

Para um TS (fluxo de transporte) é igualmente criado um ficheiro com a informação dos pacotes. É também lida toda a informação contida nos cabeçalhos dos vários blocos que constituem o fluxo analisado. Os fluxos elementares são reconstituídos e colocados em ficheiros separados.

4.3.1 sync_byte

A primeira questão levantada têm a ver com o *byte* de sincronismo, especificado na sintaxe de *transport_packet()* com o valor *sync_byte* = 0x47 (hexadecimal). Para diminuir a probabilidade de emulação do *byte*, o codificador deve evitar atribuir este valor a campos recorrentes, nomeadamente ao PID (anexo G da norma).

Cabe ao descodificador decidir se o sincronismo foi alcançado. No caso em estudo, como o meio é relativamente livre de ruído (armazenamento de ficheiros em disco), foram feitas as seguintes assumpções:

- pode haver lixo entre TPs (Pacotes de Transporte).
- nesses *bytes* não há emulação de sincronismo.

Nos casos em que não se possa assegurar as condições ideais, interessará que o descodificador verifique vários *bytes* de sincronismo consecutivos (ver pág. 56).

4.3.2 PSI

Na implementação do descodificador, um dos pontos que requereu maior atenção foi a leitura das tabelas de PSI (Informação Específica de Programa).

Neste caso, contrariamente ao que se passa com o transporte de pacotes PES, o campo *payload_unit_start* apenas indica se o primeiro *byte* depois do cabeçalho é ou não um apontador para início de uma secção da tabela.

Como os pacotes do fluxo de transporte (TPs) são pequenos relativamente ao tamanho máximo das secções, a informação pode vir repartida por vários TPs não consecutivos. Podem surgir seis situações diferentes na leitura de um TP com PSI:

1. Não começa nenhuma secção no TP.
2. Começa uma secção no início do TP, terminando noutro qualquer.
3. Começa uma secção no início do TP, terminando no mesmo pacote.
4. Acaba uma secção no TP, não se seguindo outra.
5. Acaba uma secção no TP, seguindo-se outra que acaba no mesmo TP.
6. Acaba uma secção no TP, seguindo-se outra que termina noutro qualquer.

A solução encontrada, usando *buffers* e apontadores para o estado actual e para a última informação processada, foi adoptada também, devido à sua robustez, no caso mais simples da leitura de pacotes PES.

O problema da actualização de uma tabela PSI gira à volta dos campos *version_number* e *current_next_indicator*. Quando o primeiro é incrementado a tabela transmitida é nova, sendo aplicável quando o segundo estiver activo.

A leitura da norma <u>parece</u> indicar que só se considera o *current_next_indicator* quando houve uma mudança de versão.

Na implementação foram assumidas duas situações independentes:

- Existência de dois *buffers*, um para a tabela actual e outro para ir recebendo a tabela nova. Quando a versão muda, se a tabela nova for igual à actual, é porque a versão mudou sem actualização da tabela.

- Utilização da tabela nova, quando o *current_next_indicator* fica activo.

Outra questão verificada foi a possibilidade de se extraír os fluxos elementares, mesmo sem as tabelas de PSI estarem constituídas (basta utilizar o PID respectivo).

4.3.3 adaptation_field()

Verificou-se que o uso principal do campo de adaptação *adaptation_field()*, ver pág. 62 a 69, para além do transporte de PCR, é o enchimento do TP em situações de fim do pacote PES. Se não existir *adaptation_field()*, estão disponíveis 184 *bytes*. Se *adaptation_field_length* = 0, temos um *byte* de enchimento (*stuffing_byte*) correspondente ao cabeçalho de adaptação. Se *adaptation_field_length* = 183, estão disponíveis 0 *bytes* e o TP não transporta nem PES nem PSI.

4.3.4 Exemplo de parse

Dado existirem vários fluxos de testes na *Internet*, o programa foi suficientemente testado, revelando-se bastante robusto. Mostra-se de seguida, a título de exemplo, um extracto do *parse* de um fluxo originário da Asia Matsushita Electric Ltd. (6 primeiros pacotes):

```
ams parse:
 ---------------------------------------------------
| PID = 0x0000 (Program Association Table)
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PROGRAM ASSOCIATION SECTION********************
| pointer_field = 0 bytes
| table_id = 0
| section_syntax_indicator = 1
| section_length = 17
| transport_stream_id = 21845
| version_number = 0
| current_next_indicator = 1
| section_number = 0
| last_section_number = 0
|
| program_number = 1
| program_map_PID = 0x00C8
|
| program_number = 2
| program_map_PID = 0x012C
|
| CRC_32 = 2.19049e+08
|
| 163 bytes left - 163 stuffing_bytes (0xFF) detected
 ------------------------------------------------- 1

 -------------------------------------------------
| PID = 0x00C8
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PROGRAM MAP SECTION*************************
| pointer_field = 0 bytes
| table_id                = 2
| section_syntax_indicator = 1
| section_length = 246
| program_number = 1
| version_number          = 0
| current_next_indicator = 1
| section_number          = 0
| last_section_number     = 0
| PCR_PID = 0x00C9
| program_info_length = 0
|
| stream_type = 2    -> ITU-T Rec.H262 | ISO/IEC 13818-2 Video
| elementary_PID = 0x00C9
| ES_info_length = 5
|
| descriptor_tag     = 2: video_stream_descriptor
| descriptor_length = 3
| multiple_frame_rate_flag   = 0
| frame_rate_code            = 5
| MPEG_2_flag                = 1
| constrained_parameter_flag = 0
| still_picture_flag         = 0
| profile_and_level_indication = 72
| chroma_format                = 1
| frame_rate_extension_flag    = 0
|
| stream_type = 3    -> ISO/IEC 11172 Audio
| elementary_PID = 0x00CA
| ES_info_length = 3
|
| descriptor_tag     = 3: audio_stream_descriptor
| descriptor_length = 1
| free_format_flag = 0
| ID               = 1
| layer            = 1
|
| stream_type = 3    -> ISO/IEC 11172 Audio
| elementary_PID = 0x00CB
| ES_info_length = 3
|
| descriptor_tag     = 3: audio_stream_descriptor
| descriptor_length = 1
| free_format_flag = 0
| ID               = 1
| layer            = 1
|
| stream_type = 6    -> ITU-T Rec.H222.0|ISO/IEC 13818-1 PES packets containing
private data
| elementary_PID = 0x00CC
| ES_info_length = 202
 ------------------------------------------------- 2

 -------------------------------------------------
| PID = 0x00C8
| transport_error_indicator = 0
| payload_unit_start_indicator  = 0
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 1 (counter of payloads with the same PID)
```

```
|PROGRAM MAP SECTION**************************
|
| descriptor_tag    = 5: registration_descriptor
| descriptor_length = 200
| (...)
|
| CRC_32 = 4.33939e+07
|
| 118 bytes left - 118 stuffing_bytes (0xFF) detected
 ------------------------------------------------- 3

 -------------------------------------------------
| PID = 0x012C
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PROGRAM MAP SECTION**************************
| pointer_field = 0 bytes
| table_id              = 2
| section_syntax_indicator = 1
| section_length = 236
| program_number = 2
| version_number          = 0
| current_next_indicator = 1
| section_number          = 0
| last_section_number     = 0
| PCR_PID = 0x012D
| program_info_length = 0
|
| stream_type = 1   -> ISO/IEC 11172 Video
| elementary_PID = 0x012D
| ES_info_length = 3
|
| descriptor_tag    = 2: video_stream_descriptor
| descriptor_length = 1
| multiple_frame_rate_flag   = 0
| frame_rate_code            = 4
| MPEG_2_flag                = 0
| constrained_parameter_flag = 1
| still_picture_flag         = 0
|
| stream_type = 3   -> ISO/IEC 11172 Audio
| elementary_PID = 0x012E
| ES_info_length = 3
|
| descriptor_tag    = 3: audio_stream_descriptor
| descriptor_length = 1
| free_format_flag = 0
| ID               = 1
| layer            = 1
|
| stream_type = 6   -> ITU-T Rec.H222.0|ISO/IEC 13818-1 PES packets containing
private data
| elementary_PID = 0x012F
| ES_info_length = 202
 ------------------------------------------------- 4

 -------------------------------------------------
| PID = 0x012C
| transport_error_indicator = 0
| payload_unit_start_indicator  = 0
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 1 (counter of payloads with the same PID)
|PROGRAM MAP SECTION**************************
|
| descriptor_tag    = 5: registration_descriptor
| descriptor_length = 200
| (...)
|
| CRC_32 = 4.13923e+09
|
| 128 bytes left - 128 stuffing_bytes (0xFF) detected
```

```
------------------------------------------------ 5

------------------------------------------------
| PID = 0x00C9
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 3 (adaptation_field followed by payload)
| continuity_counter = 0 (counter of payloads with the same PID)
|ADAPTATION FIELD*******************************
| adaptation_field_length = 7
| PCR_flag                      = 1
|
| PCR_base = 97
| PCR_extension = 214      (PCR = 0.0010857 s)
|
| 0 bytes left - 0 stuffing_bytes (0xFF) detected
|PES PACKET*************************************
| stream_id = 0xE0  ->  Video stream - number 0
| PES_packet_length = 0 bytes
| PES_priority            = 1
| original_or_copy        = 0 (copy)
| PTS_DTS_flags           = 3
| PES_header_data_length = 10
|
| PTS = 27730 (0.308111 s)
| DTS = 24730 (0.274778 s)
| (157 data_bytes)
------------------------------------------------ 6
...
```

Em termos de sintaxe falta implementar alguns descritores, a *private_section()* e a CAT (*Conditional Access Table*), bem como proceder à verificação do código de redundância cíclica CRC (já previsto através do uso de *buffers* na leitura da informação). Poderá também ser feita a recolha de algumas indicações estatísticas (*e.g.* número e tipo de pacotes; *overhead*) e a verificação das restrições temporais impostas pela norma (frequência de codificação de PCR e PTS).

4.4 Código fonte

4.4.1 README

```
MPEG2 system bitstreams parser: ISO/IEC 13818-1 (N0721 - June 94)
------------------------------------------------------------
Arguments: [-{switch}...] filename [filename ...] (accepts *)

  switch: 0, writes all values of all fields of the system level;
          1, skips the reserved fields (default option);
          2, skips reserved and most, first level, inactive flag fields;
          A, always store the payload of TPs even with incomplete PSI tables
          N, doesn't store the TPs payloads (overrides the -A switch)
          L, generates a log file with warnings/errors found during processing

Output:
 - ASCII parse file ({input filename}.{parser name}) holding the values encoded
   in the fields of the TP headers.
 - Optional log file ({input filename}.LOG).
 - Binary files ({input filename}{PID}.{stream_id}) with the recovered
   elementary streams.

NB: Program streams are, for the moment, being assumed as MPEG1
------------------------------------------------------------
parser v.18 - Andre'Braganca @1994 - INESC/Programa Ciencia
comments to: amb@newton.inescn.pt


Characteristics

o The program accepts as an input one (or more) binary disk file which holds a
  multiplexed bitstream encoded according to the Transport Stream syntax
  specified in ISO 13818-1 (version of June 1994). It generates an ASCII disk
  file with the information it retrieves from the fields of the headers of
  transport packets (TPs) as well as from the payload of TPs which carry the PSI
  (Program Specific Information) tables. Elementary bitstreams are recovered and
  stored in separate disk files, which may be used as inputs to the respective
  MPEG2 elementary decoders;

o The time to access the disk has been reduced by using buffers to store data
  from the disk files;

o This program has been implemented in Borland C for DOS. In order to ensure
  compatibility with UNIX (gcc) and because of the different representation of
  bytes in these two operating systems, unions with bit fields have been used to
  enclose the bytes read from the disk file. Depending on the actual environment
  being used, these bit fields may or not be inverted. Another solution to
  ensure portability, would be to use masks and shifts in order to access
  individual bits within the bytes. However this solution would make the source
  code very hard to read;

o The bitstream stored in the input disk file is firstly subjected to a
  pre-analysis to determine whether it is a Program or a Transport Stream. A
  Program Stream will be interpreted (for the moment) as an MPEG1 bitstream
  while a Transport Stream is decoded using the syntax and procedures described
  in ISO 13818-1;

o Buffers are used to hold the information conveyed in the TP payloads (PSI
  sections and PES packets). Information is interpretated as soon as possible,
  even if it is scattered across several non contiguous TPs. Buffering will
  (soon) allow verification of CRC.

o Linked lists are used to hold PSI. This information is used by the parser to
  identify the origin of the incoming TPs;

o Payload of duplicated TPs is not extracted (by verification of the
  continuity_counter in TP header);

o It is possible to run the program with the option of extracting and storing
  the payload of all TPs before having received the complete PSI tables (-A
  switch);
```

```
o There is also an option to generate a log file to document the occurrence of
  errors during the demultiplexing process.


Versions

v.17 - Distributed by anonymous ftp to newton.inescn.pt

v.18 - Source code translated to English (comments, variable names, etc).
```

4.4.2 Listagem

```
#define DOS /* Delete this if compiled in UNIX */

/*****************************************************************************

 MPEG2 system bitstreams parser: ISO/IEC 13818-1 (N0721 - June 94)

 version 18 - Andre' Braganca @ 8/11/94 - INESC/Programa Ciencia


 author: Andre' Braganca

 mail:   INESC - Instituto de Engenharia de Sistemas e Computadores
         Largo Mompilher 22
         Apartado 4433
         4007 PORTO CODEX
         PORTUGAL

 phone:  02 2087830
 fax:     02 2087829

 email:  amb@newton.inescn.pt
         amb@bart.inescn.pt

******************************************************************************


 CHARACTERISTICS:

 The program accepts as an input one (or more) binary disk file which holds a
 multiplexed bitstream encoded according to the Transport Stream syntax
 specified in ISO 13818-1 (version of June 1994). It generates an ASCII disk
 file with the information it retrieves from the fields of the headers of
 Transport stream Packets (TPs) as well as from the payload of TPs which carry
 the PSI (Program Specific Information) tables. Elementary bitstreams are
 recovered and stored in separate disk files, which may be used as inputs to
 the respective MPEG2 elementary decoders;

 Program streams are, for the moment, being assumed as MPEG1.

 The time to access the disk has been reduced by using buffers to store data
 from the disk files;

 This program has been implemented in Borland C for DOS. In order to ensure
 compatibility with UNIX (gcc) and because of the different representation of
 bytes in these two operating systems, unions with bit fields have been used to
 enclose the bytes read from the disk file. Depending on the actual environment
 being used, these bit fields may or not be inverted. Another solution to
 ensure portability, would be to use masks and shifts in order to access
 individual bits within the bytes. However this solution would make the source
 code very hard to read;

 The bitstream stored in the input disk file is firstly subjected to a
 pre-analysis to determine wether it is a Program or a Transport Stream. A
 Program Stream will be interpreted (for the moment) as an MPEG1 bitstream
 while a Transport Stream is decoded using the syntax and procedures described
 in ISO 13818-1;

 Buffers are used to hold the information conveyed in the TP payloads (PSI
 sections and PES packets). Information is interpretated as soon as possible,
 even if it is scattered across several non contiguous TPs. Buffering will
 (soon) allow verification of CRC.

 Linked lists are used to hold PSI. This information is used by the parser to
 identify the origin of the incoming TPs;

 Payload of duplicated TPs is not extracted (by verification of the
 continuity_counter in TP header);

 It is possible to run the program with the option of extracting and storing
 the payload of all TPs before having received the complete PSI tables (-A
 switch);

 There is also an option to generate a log file to document the occurrence of
 errors during the demultiplexing process.
```

```
MPEG2 PARSER DEVELOPMENT:

10.C - It is based on 7.C, the last development of the MPEG1 parser.
 Program streams are interpreted as ISO 11172-1. Development will concentrate
on Transport streams.
 Adaptation field.

11.C - Payload analysis:
Program_Association_section, Conditional_Access_section, Program_Map_section.
 Use of buffers for PSI sections because TPs are small and information could
be scattered across several packets.
 Use of linked lists to keep PSI. The main goal is to have a list of PIDs used
by the parser to know what kind of packets it's dealling with.
 This lists have just one pointer. The last element has a pointer value of
NULL. The first element is pointed by a global variable (if NULL the PSI table
is inactive). An empty table is a list with just one element (sentinel).
Insertion is simple, at the begining of the list. Lists are not ordered
because searches will probably be made in a small universe.

12.C - Simple PES packets analysis (doesn't work).

13.C - Bufferized reading of PES packets.
Elementary Streams (ES) extraction (with problems).

14.C - Solved extraction problem by avoiding to put bytes in the buffer when
reading groups of bytes (ES estraction or padding).
 Ignored adaptation_field_length = 0 to read Audio from MATRA bitstream.

15.C - Standard actualization (N0601 to N0721).
 Testing with more bitstreams (small corrections).
 Option of extracting and storing the TP payloads without waitting for PSI.
To of this I created a PMT type table where all PIDs, whose TPs may contain
PES, are inserted. This table acts as a flag in the payload interpretation: if
active is used instead of the PM_table (see TS_TRANSPORT_PACKET()).
 Revision of Program_Use() and the notation for output files.
 Detected another problem in extraction: one byte lost when checking stuffing
bytes.
 Lack of memory (DOS) when testing MATSUSHITA bitstream with unconditional
extraction of PES. The extra table uses much memory.
 Detected the non utilization of shifts when accessing bits in masked bytes.

16.C - By checking the continuity_counter avoided the PES extraction in
duplicate TPs.
 Use of switches as arguments in the command line: unconditional extraction of
PES; detail level in the parsing file; not extract PES; log file generation.
 Substituted some fprintf(stderr,...) by perror().

17.C - Translated the outputs to English due to a possible utilization of this
program in the COUGAR project.
 Detail level in which, in the first level of flags, the parsing file only
shows active flags (most cases).
 Promotion of the TP counter from local to global in order to appear in the
log file.

18.C - Translation of the source code from Portuguese to English in order to
make it avaliable to COUGAR.


TO DO:    Private_section
          CRC field in Hexadecimal
          CRC check
          CAT
          remaining descriptors
          show statistics in log file
          verify file I/O buffers efficiency
          check MPEG1 adress counter
          avoid global variables by using the static keyword

*************************************************************************/

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
```

```
#ifdef DOS
  #include <dir.h>
  #include <dos.h>
  #define MAX_FILES 100
#endif

#define PES_BUFFER_SIZE 470 /*Maximum buffer size (+1) for the PES_packets
                             header (in PMT). Could include data_bytes. */

/* /// Compilation problem in UNIX ///// */

#ifndef SEEK_SET
  #define SEEK_SET 0
#endif
#ifndef SEEK_CUR
  #define SEEK_CUR 1
#endif

/* /// Type definitions //////////////// */

typedef enum { NO, YES } boolean;
typedef struct PA_table *PAT_ptr;
typedef struct PM_table *PMT_ptr;
typedef struct CA_table *CAT_ptr;

/* /// Global variables //////////////// */

boolean extract, /* Do/don't extract PES. See Extracts(), RemoveStream(),
                    FindSwitch(). */
        log, /* Creation of a log file with errors found in the analysed stream
               Will also have some statistics (e.g. overhead).
               See FindSwitch(), Main(), Warn(). */
        unconditional;/* Type of PES extraction:
                          1 - all PIDs
                          0 - after having the PSI tables.
                        See TRANSPORT_STREAM(), FindSwitch().
                        In case 1 the new table may require much memory and
                        make the parsing to abort.*/

unsigned char detail;/* Detail level in the parsing file.
                       Initialized in FindSwitch. */

unsigned long n_TPs; /* TP counter. Initialized in PROGRAM_STREAM() (due to
                       Warn()).
                       Initialized and used in TRANSPORT_STREAM(). Used in
                       Warn(); */

char no_extension[80];/*Defined in NameOut(). Used in RemoveStream(),
                       Extracts().*/

struct buffer
{
  /* 1024 is the maximum size of a PSI section */
  unsigned char buffer[1024]; /*must be unsigned !*/
  int p; /* buffer pointer */
  int descriptor_start; /* buffer pointer; start of most recent descriptor */
  int bytes_left; /* Remaining bytes to read in order to complete the section*/
}
  PA_section, /*Used in TS_PROGRAM_ASSOCIATION_SECTION() */
  CA_section, /*Used in TS_CA_SECTION() */
  PM_section; /*Used in TS_PROGRAM_MAP_SECTION() */

/* TO DO: make it a static local variable !*/

struct
{
  char buffer[4096]; /* Maximum size of a private section */
  int p; /* buffer pointer */
  int bytes_left; /* Remaining bytes to complete the section */
}
  private_section; /*Used in PRIVATE_SECTION() */


struct PA_table
{
  int program_number;
  int PID;
  PAT_ptr next;
};

struct PM_table
{
  /*int stream_type; Initialized in TS_PROGRAM_MAP_SECTION(), NOT USED!*/
  int elementary_PID; /*Initialized in TS_PROGRAM_MAP_SECTION()*/
         char     continuity_counter;/*Used     in     TS_TRANSPORT_PACKET();     Init.in
TS_PROGRAM_MAP_SECTION()*/
        boolean     old_stream_removed;     /*Activated     in     PES_PACKET();     Init.in
TS_PROGRAM_MAP_SECTION()*/
  unsigned char buffer[PES_BUFFER_SIZE];/*PES_packet header buffer */
  int p, info_start, info_end; /* buffer pointers */
      int    field;    /*Which    part    of    PES_packet    is    being    read*//*Init.   in
TS_PROGRAM_MAP_SECTION()*/
  unsigned flag[7]; /*PES_packet flags buffer (helps deciding field)*/
  int packet_length; /* If null then length is unknown */
  int bytes_left; /* Remaining bytes to complete PES_packet*/
  int header_data_length; /* To confirm stuffing_bytes */
  PMT_ptr next;
};

struct CA_table
{
  int CA_PID;
  CAT_ptr next;
};

/* Pointers to start of PSI table linked lists (current and new version).
   Initialized in TRANSPORT_STREAM(), updated in TS_..._SECTION().
   Used in TS_TRANSPORT_PACKET. */
PAT_ptr PAT,
        new_PAT;
PMT_ptr PMT,
        new_PMT,
        ALL;/*Used in PES extraction without PSI tables*/
CAT_ptr CAT,
        new_CAT;

int PAT_version; /* version_number of new_PAT */
int PMT_version; /* version_number of new_PMT */
int CAT_version; /* version_number of new_CAT */
/* Init. in TRANSPORT_STREAM(), updated in TS_..._SECTION() */

/* /// Function prototypes ///////////// */

void     Program_Use                       (void);
void     Expands_meta_ch                   (int *, char ***);
char    *Obtains_path                      (char *);
void     NameOut                           (char *, char *, char *);
void     FindSwitch                        (int *, char ***);
boolean  PROGRAM_STREAM                    (FILE *, FILE *);
void     STARTCODE_prefix                  (FILE *);
void     PACK                              (FILE *, FILE *);
void     SYSTEM_HEADER                     (FILE *, FILE *);
void     PACKET                            (int, FILE *, FILE *);
void     RemoveStream                      (int, int);
void     Extracts                           (FILE *, int, int, int);
boolean  TRANSPORT_STREAM                  (FILE *, FILE *);
void     TS_TRANSPORT_PACKET               (FILE *, FILE *);
PAT_ptr  PA_PID                            (PAT_ptr, int, int *);
PMT_ptr  Elementary_PID                    (PMT_ptr, int);
PMT_ptr  ALL_PID                           (PMT_ptr, int);
void     TS_ADAPTATION_FIELD               (int, int *, FILE *, FILE *);
void     TS_PROGRAM_ASSOCIATION_SECTION    (int, int *, FILE *, FILE *);
void     TS_CA_SECTION                     (int, int *, FILE *, FILE *);
void     TS_PROGRAM_MAP_SECTION            (int, int *, FILE *, FILE *);
void     DESCRIPTOR                                 (struct buffer *, FILE *);
void     VIDEO_STREAM_DESCRIPTOR                    (struct buffer *, FILE *);
void     AUDIO_STREAM_DESCRIPTOR                    (struct buffer *, FILE *);
void     SYSTEM_CLOCK_DESCRIPTOR                    (struct buffer *, FILE *);
void     MULTIPLEX_BUFFER_UTILIZATION_DESCRIPTOR    (struct buffer *, FILE *);
void     MAXIMUM_BITRATE_DESCRIPTOR                 (struct buffer *, FILE *);
void     _DESCRIPTOR /*DUMMY*/                    (struct buffer *, FILE *);
void     PES_PACKET                        (PMT_ptr, int, int *, FILE *, FILE *);
void     Warn                              (FILE *, char *);
```

```
/* ////////////////////////////////////// */


void
Program_Use()
/************************************
 ISO 13818-1 streams parser.
 ************************************/
{
  fprintf(stderr,"---------------------------------------------------------------\n");
   fprintf(stderr,"MPEG2  system  bitstreams  parser:  ISO/IEC  13818-1  (N0721  -  June
94)\n\n");
  fprintf(stderr,"Arguments: [-(switch)...] filename [filename ...] (accepts *)\n");
   fprintf(stderr,"     switch:  0,  writes  all  values  of  all  fields  of  the  system
level;\n");
  fprintf(stderr,"              1, skips the reserved fields (default option);\n");
  fprintf(stderr,"                2, skips reserved and most, first level, inactive flag
fields;\n");
  fprintf(stderr,"               A, always store the payload of TPs even with incomplete
PSI tables\n");
   fprintf(stderr,"                 N, doesn't store the TPs payloads (overrides the A
switch)\n");
  fprintf(stderr,"              L, generates a log file with warnings/errors found during
processing\n");
  fprintf(stderr,"\nOutputs:\n");
   fprintf(stderr," - ASCII parse file ((input filename).(parser name)) holding the
values encoded\n");
  fprintf(stderr,"   in the fields of the TP headers.\n");
  fprintf(stderr," - Optional log file ((input filename).LOG).\n");
    fprintf(stderr,"  -  Binary  files  ((input  filename)(PID).(stream_id))  with  the
recovered\n");
  fprintf(stderr,"   elementary streams.\n\n");
  fprintf(stderr,"NB: Program streams are, for the moment, being assumed as MPEG1\n");
  fprintf(stderr,"--------------------------------------------------------------\n");
  fprintf(stderr,"parser v.18 - Andre'Braganca @1994 - INESC/Programa Ciencia\n");
  fprintf(stderr,"send comments to: amb@newton.inescn.pt\n");
  exit(0);
}


#ifdef DOS

void
Expands_meta_ch(int *argc, char **argv[])
/************************************
 Expands the metacharacters in the arguments specified in the command line
 (e.g. "*.mpg"). In case of UNIX this is made at shell level.
 Routine adapted from "InfoMPEG @1993 Dennis Lee".
 ************************************/
{
  boolean exists_file, expands;
  struct ffblk file_info;
  char file_name[80], path[80];
  register int i, j, new_argc=0;
  char **argv_buffer = (char **)malloc(MAX_FILES * sizeof(char *));


  for(i=0; i < *argc; i++)
  {
    expands = NO; /* There's no wildcard '*' until you find one */

    for(j = strlen((*argv)[i])-1; j >= 0; j--)
    {
      if( (*argv)[i][j] == '*' )
      {
        expands = YES;
        break;
      }
    }

    if(expands)
    {
      strcpy(file_name, (*argv)[i]);
      strcpy(path, Obtains_path((*argv)[i]));

      exists_file = !findfirst(file_name, &file_info, FA_RDONLY);
      while (exists_file)
```

```
        {
          argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
          strcpy(argv_buffer[new_argc], path);
          strcat(argv_buffer[new_argc++], file_info.ff_name);
          exists_file = !findnext(&file_info);
        }
      }
      else
      {
        argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
        strcpy(argv_buffer[new_argc++], (*argv)[i]);
      }
    }

  *argc = new_argc;
  *argv = argv_buffer;
}


char *
Obtains_path (char *name)
/*************************************
 Eliminates the file name, leaving its path only.
 Routine adapted from "InfoMPEG @1993 Dennis Lee".
 *************************************/
{
  register int i;


  for(i=strlen(name)-1; name[i]!='\\' && name[i]!='/' && name[i]!=':'; i--)
    name[i] = 0;

  return name;
}

#endif


void
NameOut (char *output, char *input, char *executable_name)
/*************************************
 Obtains the output filename by adding/modifying an extension to the input.
 The extension is made from the first 3 (or less) letters of the program name
 without the path.
 *************************************/
{
    register int i, j;
    char extension[80];
    /*char no_extension[80]; GLOBAL VAR. !*/


    strcpy(extension, executable_name);
    for(i = strlen(extension) - 1;
        i>=0 && extension[i]!='\\' && extension[i]!='/' && extension[i]!=':';
        i--) ;
    extension[4] = '\0';
    extension[0] = '.';
    for(j=1; j<4; j++)
    {
      extension[j] = executable_name[i+j];
      if(extension[j]=='.')
      {
        extension[j]='\0';
        break;
      }
    }


    strcpy(output, input);

    for(i=strlen(output)-1; i>=0; i--)
    {
      if(output[i]=='.')
      {
        output[i]='\0';
        break;
      }
    }

    /*Global variable initialization.*/
    strcpy(no_extension, output);

    strcat(output, extension);
}


void
FindSwitch(int *argc, char **argv[])
/***************************************
 Finds (and removes), in the command line arguments, instructions to define the
 program behaviour.
 **************************************/
{
  /* boolean extract, unconditional, log;
     unsigned char detail; GLOBAL VAR. !*/
  register int i,
               new_argc=0;
  char **argv_buffer = (char **)malloc(*argc * sizeof(char *));

  /* Switches initialization (defaults) */
  extract = YES;
  unconditional = NO;
  log = NO;
  detail = 1;

  /* Search */
  for(i=0; i < *argc; i++)
  {
    if( (*argv)[i][0] == '-') /* switch indicator */
    {
      switch( toupper( (*argv)[i][1] ) )
      {
       case 'N': /* Not extract PES */
                 extract = NO;
                 break;
       case 'A': /* Unconditional PES extraction (without PSI) */
                 unconditional = YES;
                 break;
       case 'L': /* Creates log file (errors and statistics) */
                 log = YES;
                 break;
       case '0': /* Parsing file with maximum detail */
                 detail = 0;
                 break;
       case '1': /* Parsing file without 'reserved' fields */
                 detail = 1;
                 break;
       case '2': /* Parsing file without inactive flag fields */
                 detail = 2;
                 break;
       default : fprintf(stderr,"\nInvalid switch !");
                 Program_Use();
                 break;
      }
    }
    else
    {
      argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
      strcpy(argv_buffer[new_argc++], (*argv)[i]);
    }
  }

  *argc = new_argc;
  *argv = argv_buffer;
}


main(int argc, char *argv[])
/***************************************
 Organizes the parsing of the input files.
 The file is read first as a Program Stream. If not valid it's reanalysed as a
 Transport Stream.
 **************************************/
{
  /* boolean log;
     char no_extension[80]; GLOBAL VAR. !*/
  FILE *in, *out;
  char str[80], s[80];
  int file;


  FindSwitch(&argc, &argv); /* Initializes global variables according to the
                               command line.*/

#ifdef DOS
  Expands_meta_ch(&argc, &argv);
#endif

  if(argc < 2) Program_Use();

  for(file=1; file < argc; file++)
  {
    if((in=fopen(argv[file],"rb")) == NULL)
    {
      sprintf(str,"\nrb - can't open %s", argv[file]);
      perror(str);
      continue;
    }

    NameOut(str, argv[file], argv[0]);
    if((out=fopen(str,"wt")) == NULL)
    {
      sprintf(s,"\nwt - can't open %s", str);
      perror(s);
      fclose(in);
      continue;
    }

    /*Buffers, using an indirect call to malloc, for the input and output
      streams in order to minimize disk access.*/
    if (setvbuf(in, NULL, _IOFBF, 15360) != 0)
      fprintf(stderr,"\n(failed to set up buffer for input file)");
    if (setvbuf(out, NULL, _IOFBF, 15360) != 0)
      fprintf(stderr,"\n(failed to set up buffer for output file)");


    fprintf(stdout,"\nProcessing %s to %s\n", argv[file], str);
    fprintf(out,"%s parse:",argv[file]);

    if( log ) /* file with errors and stream statistics */
    {
      sprintf(s,"%s.LOG", no_extension);
      if(remove(s))
      {
        sprintf(str,"(can't remove %s (log file))", s);
        perror(str);
      }
    }

    if( !PROGRAM_STREAM(in, out) )
    {
      rewind(in);
      if( !TRANSPORT_STREAM(in, out) )
        fprintf(stderr,"\n%s isn't a valid ISO13818-1 bitstream !\n",argv[file]);
    }


    fclose(in);
    fclose(out);
  }

  return 0;
}


/*///////////////////////////////////////////////////////////////////////////
  PROGRAM_STREAM ROUTINES
///////////////////////////////////////////////////////////////////////////*/


boolean
```

```
PROGRAM_STREAM (FILE *in, FILE *out)
/************************************
 For the moment only reads MPEG1 streams !
 Returns YES or NO depending on a valid parsing of the stream.
 ***********************************/
{
  /* unsigned long n_TPs; GLOBAL VAR. !*/
  int byte;
  boolean finished;
  unsigned long address,
                next_address;


  n_TPs = 1; /* variable initialized just because of Warn() */

  STARTCODE_prefix(in);
  address = ftell(in);
  byte = getc(in);
  if(byte != 0xBA) /* pack_start_code */
  {
    return NO; /* Invalid stream */
  }

  fprintf(out,"(address, following bytes, start_code, description)\n");

  finished = NO;
  while(!finished)
  {
    /* The start_code address is the prefix start */
    fprintf(out,"\n%06X", address-3);

    STARTCODE_prefix(in);
    next_address = fteil(in);

    /* Number of following bytes (not including start_codes) and the STARTCODE */
    fprintf(out,"%51d  %02X", next_address-4 -address, byte);

    switch(byte)
    {
      case 0x00: /* */
                 fprintf(out,"           v(picture_start_code)");
                 break;
      case 0xB0: /* reserved */
      case 0xB1:
      case 0xB6:
                 fprintf(out,"  RESERVED");
                 break;
      case 0xB2: /* */
                 fprintf(out,"  (user_data_start_code)");
                 break;
      case 0xB3: /* */
                 fprintf(out,"       v(sequence_header_code)");
                 break;
      case 0xB4: /* */
                 fprintf(out,"       v(sequence_error_code)");
                 break;
      case 0xB5: /* */
                 fprintf(out,"       v(extension_start_code)");
                 break;
      case 0xB7: /* */
                 fprintf(out,"       v(sequence_end_code)");
                 break;
      case 0xB8: /* */
                 fprintf(out,"       v(group_start_code)");
                 break;

      case 0xBA: /* start of MPEG system stream */
                 fprintf(out,"  pack_start_code");
                 fseek(in, address + 1, SEEK_SET);
                 PACK(in, out);
                 fseek(in, next_address, SEEK_SET);
                 break;
      case 0xBB: /* start of MPEG system header */
                 fprintf(out,"  system_header_start_code");
                 fseek(in, address + 1, SEEK_SET);
                 SYSTEM_HEADER(in, out);
                 fseek(in, next_address, SEEK_SET);
                break;
        case 0xBC: /* stream_id */
                fprintf(out,"    reserved_stream");
                fseek(in, address + 1, SEEK_SET);
                PACKET(byte, in, out);
                fseek(in, next_address, SEEK_SET);
                break;
        case 0xBD: /* stream_id */
                fprintf(out,"    private_stream_1");
                fseek(in, address + 1, SEEK_SET);
                PACKET(byte, in, out);
                fseek(in, next_address, SEEK_SET);
                break;
        case 0xBE: /* stream_id */
                fprintf(out,"    padding_stream");
                fseek(in, address + 1, SEEK_SET);
                PACKET(byte, in, out);
                fseek(in, next_address, SEEK_SET);
                break;
        case 0xBF: /* stream_id */
                fprintf(out,"    private_stream_2");
                fseek(in, address + 1, SEEK_SET);
                PACKET(byte, in, out);
                fseek(in, next_address, SEEK_SET);
                break;
        case 0xB9: /* end of MPEG system stream */
                fprintf(out,"  iso_11172_end_code");
                finished = YES;
                break;
        default  :
                if(byte >0x00 && byte < 0xB0) /* packet slices */
                  fprintf(out,"                    SLICE");


                if(byte>=0xC0 && byte<=0xDF)  /* stream_id */
                {
                  fprintf(out,"    Audio stream - number ");
                  fprintf(out,"%d", byte & 31); /* mask 00011111B */
                  fseek(in, address + 1, SEEK_SET);
                  PACKET(byte, in, out);
                  fseek(in, next_address, SEEK_SET);
                }
                if(byte>=0xE0 && byte<=0xEF)  /* stream_id */
                {
                  fprintf(out,"    Video stream - number ");
                  fprintf(out,"%d", byte & 15); /* mask 00001111B */
                  fseek(in, address + 1, SEEK_SET);
                  PACKET(byte, in, out);
                  fseek(in, next_address, SEEK_SET);
                }
                if(byte>=0xF0 && byte<=0xFF)  /* stream_id */
                {
                  fprintf(out,"    reserved Data stream - number ");
                  fprintf(out,"%d", byte & 15); /* mask 00001111B */
                  fseek(in, address + 1, SEEK_SET);
                  PACKET(byte, in, out);
                  fseek(in, next_address, SEEK_SET);
                }
      }

      if(!finished)
      {
        if(feof(in))
        {
         Warn(out,"premature EOF");
         finished = YES;
        }
        else
        {
         address = next_address;
         byte = getc(in);
        }
      }
   }

   return YES; /* parsing done */
}
```

```
void
STARTCODE_prefix (FILE *stream)
/*************************************
 Inspects the stream sequentialy until a start_code prefix is found (0x000001).
 Another routine will use the byte, left in the stream, which specifies the
 start_code type.
 *************************************/
{
   unsigned long counter=0;

   while(!feof(stream))
   {
     switch(getc(stream))
     {
       case 0x00: counter++;
                  break;
       case 0x01: if (counter >= 2) return;
       default  : counter=0;
     }
   }
}


void
PACK (FILE *in, FILE *out)
/*************************************
 pack_start_code = 4 bytes (already read).

 marker bits = 9 bits               \
 system_clock_reference = 33 bits    > 8 bytes.
 mux_rate = 22 bits                 /

 *************************************/
{
  struct bit_fields
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned marker_bit_1                : 1;
    unsigned system_clock_reference_32_30 : 3;
    unsigned no_name                     : 4;

    unsigned system_clock_reference_29_22 : 8;

    unsigned marker_bit_2                : 1;
    unsigned system_clock_reference_21_15 : 7;

    unsigned system_clock_reference_14_7  : 8;

    unsigned marker_bit_3                : 1;
    unsigned system_clock_reference_6_0   : 7;

    unsigned mux_rate_21_15              : 7;
    unsigned marker_bit_4                : 1;

    unsigned mux_rate_14_7               : 8;

    unsigned marker_bit_5                : 1;
    unsigned mux_rate_6_0                : 7;

#else
    unsigned no_name                     : 4;
    unsigned system_clock_reference_32_30 : 3;
    unsigned marker_bit_1                : 1;

    unsigned system_clock_reference_29_22 : 8;

    unsigned system_clock_reference_21_15 : 7;
    unsigned marker_bit_2                : 1;

    unsigned system_clock_reference_14_7  : 8;

    unsigned system_clock_reference_6_0   : 7;
    unsigned marker_bit_3                   :  1;

    unsigned marker_bit_4                   :  1;
    unsigned mux_rate_21_15                 :  7;

    unsigned mux_rate_14_7                  :  8;

    unsigned mux_rate_6_0                   :  7;
    unsigned marker_bit_5                   :  1;
#endif
  };

  register int i;
  double sys_clk_ref;
  unsigned long int mux_rate;
  union
  {
    struct bit_fields bits;
    char byte[8];
  } SCR;


  fprintf(out,"\n -------------------------------------------------\n");

  /*Read 8 bytes of SCR.*/
  for(i=0; i<8; i++)  SCR.byte[i]=(char)getc(in);

  if(  SCR.bits.no_name       != 0x02 ||
      !SCR.bits.marker_bit_1          ||
      !SCR.bits.marker_bit_2          ||
      !SCR.bits.marker_bit_3          ||
      !SCR.bits.marker_bit_4          ||
      !SCR.bits.marker_bit_5             )
  {
    Warn(out,"Invalid markers in SCR");
    return;
  }

  sys_clk_ref = SCR.bits.system_clock_reference_32_30 * 1073741824 /* 2^30 */
               +SCR.bits.system_clock_reference_29_22 * 4194304    /* 2^22 */
               +SCR.bits.system_clock_reference_21_15 * 32768      /* 2^15 */
               +SCR.bits.system_clock_reference_14_7  * 128        /* 2^7 */
               +SCR.bits.system_clock_reference_6_0 ;
      fprintf(out,"|   system_clock_reference   =   %g   (%g   s)\n",   sys_clk_ref,
sys_clk_ref/90000);

  mux_rate = SCR.bits.mux_rate_21_15 * 32768
            +SCR.bits.mux_rate_14_7  * 128
            +SCR.bits.mux_rate_6_0;
  fprintf(out,"| mux_rate = %lu (%lu bytes/s)", mux_rate, mux_rate*50);
  fprintf(out,"\n -------------------------------------------------\n");
}


void
SYSTEM_HEADER (FILE *in, FILE *out)
/*************************************
 system_header_start_code = 4 bytes (already read).

 header_length = 2 bytes

 marker_bits = 3 bits \
 flags = 4 bits        \
 rate_bound = 22 bits   > 5 bytes.
 audio_bound = 6 bits  /
 video_bound = 5 bits /

 reserved_byte = 1 byte

 stream_id = 1 byte               \
 no_name = 2 bits                  \ 3 bytes * n.streams
 STD_buffer_bound_scale = 1 bit    /
 STD_buffer_size_bound = 13 bits /

 *************************************/
{
  struct bit_fields
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).*/
    unsigned header_length_15_8  :  8;

    unsigned header_length_7_0   :  8;

    unsigned rate_bound_21_15    :  7;
    unsigned marker_bit_1        :  1;

    unsigned rate_bound_14_7     :  8;

    unsigned marker_bit_2        :  1;
    unsigned rate_bound_6_0      :  7;

    unsigned CSPS_flag           :  1;
    unsigned fixed_flag          :  1;
    unsigned audio_bound         :  6;


    unsigned video_bound         :  5;
    unsigned marker_bit_3        :  1;
    unsigned sys_video_lock_flag :  1;
    unsigned sys_audio_lock_flag :  1;

    unsigned reserved_byte       :  8;

#else
    unsigned header_length_15_8  :  8;

    unsigned header_length_7_0   :  8;

    unsigned marker_bit_1        :  1;
    unsigned rate_bound_21_15    :  7;

    unsigned rate_bound_14_7     :  8;

    unsigned rate_bound_6_0      :  7;
    unsigned marker_bit_2        :  1;

    unsigned audio_bound         :  6;
    unsigned fixed_flag          :  1;
    unsigned CSPS_flag           :  1;

    unsigned sys_audio_lock_flag :  1;
    unsigned sys_video_lock_flag :  1;
    unsigned marker_bit_3        :  1;
    unsigned video_bound         :  5;

    unsigned reserved_byte       :  8;

#endif
  };

  union
  {
    struct bit_fields bits;
    char byte[8];
  } lead;

  struct stream_fields
  {
    unsigned stream_id               : 8;

#ifdef DOS
    unsigned STD_buf_size_bound_12_8  : 5;
    unsigned STD_buf_bound_scale      : 1;
    unsigned no_name                  : 2;

#else
    unsigned no_name                  : 2;
    unsigned STD_buf_bound_scale      : 1;
    unsigned STD_buf_size_bound_12_8  : 5;

#endif
```

```
    unsigned STD_buf_size_bound_7_0    : 8;
  };

  union
  {
    struct stream_fields bits;
    char byte[3];
  } stream;

  register int i, header_length, streams, stream_id;
  unsigned long int rate_bound;


  fprintf(out,"\n -------------------------------------------------\n");

  /*Read first 8 bytes of System header (fixed).*/
  for(i=0; i<8; i++)  lead.byte[i]=(char)getc(in);

  if( !lead.bits.marker_bit_1 ||
      !lead.bits.marker_bit_2 ||
      !lead.bits.marker_bit_3     )
  {
    Warn(out,"Invalid markers in System Header");
    return;
  }


  header_length = lead.bits.header_length_15_8 * 256
                 +lead.bits.header_length_7_0;
  fprintf(out,"| header_length = %d bytes\n", header_length);


  rate_bound = lead.bits.rate_bound_21_15 * 32768
              +lead.bits.rate_bound_14_7  * 128
              +lead.bits.rate_bound_6_0;
  fprintf(out,"| rate_bound  = %lu", rate_bound);
  fprintf(out," (maximum mux rate is %lu bytes/s)\n", rate_bound*50);

  fprintf(out,"| audio_bound = %d stream(s)\n", lead.bits.audio_bound);
  fprintf(out,"| (maximum number of multiplexed audio streams)\n");

  if( lead.bits.fixed_flag )
    fprintf(out,"| fixed_flag = 1 (fixed bitrate)\n");
  else
    fprintf(out,"| fixed_flag = 0 (variable bitrate)\n");

  if(detail < 2 || lead.bits.CSPS_flag)
  {
    fprintf(out,"| CSPS_flag  = %d", lead.bits.CSPS_flag);
    if( lead.bits.CSPS_flag )
      fprintf(out," (meets the CSPS constraints)\n");
    else
      fprintf(out," (stream not limited to the CSPS constraints)\n");
  }


  if(detail < 2 || lead.bits.sys_audio_lock_flag)
  {
    fprintf(out,"| system_audio_lock_flag = %d\n",lead.bits.sys_audio_lock_flag);
    if( lead.bits.sys_audio_lock_flag )
       fprintf(out,"| (constant relationship between the sampling rate and the system
clock)\n");
    else
      fprintf(out,"| (independent system clock)\n");
  }


  if(detail < 2 || lead.bits.sys_video_lock_flag)
  {
    fprintf(out,"| system_video_lock_flag = %d\n",lead.bits.sys_video_lock_flag);
    if( lead.bits.sys_video_lock_flag )
       fprintf(out,"| (constant relationship between the sampling rate and the system
clock)\n");
    else
      fprintf(out,"| (independent system clock)\n");
  }
```

```
fprintf(out,"| video_bound = %d stream(s)\n", lead.bits.video_bound);
fprintf(out,"| (maximum number of multiplexed video streams)\n");

if(detail < 1)
  fprintf(out,"| reserved_byte = %d\n", lead.bits.reserved_byte);


streams = header_length - 6;
if(streams % 3)
{
  Warn(out,"Elementary streams of not fit header length");
  return;
}
streams /= 3;
fprintf(out,"| Elementary stream(s) = %d\n", streams);

while(streams)
{
  /*Read 3 bytes of streams.*/
  for(i=0; i<3; i++)  stream.byte[i]=(char)getc(in);

  stream_id = stream.bits.stream_id;
  RemoveStream(0, stream_id);
  fprintf(out,"|\n| Stream %X : ", stream_id);
  switch( stream_id )
  {
    case 0xB8: /* stream_id */
               fprintf(out,"refers to all audio streams");
               break;
    case 0xB9: /* stream_id */
               fprintf(out,"refers to all video streams");
               break;
    case 0xBC: /* stream_id */
               fprintf(out,"reserved_stream");
               break;
    case 0xBD: /* stream_id */
               fprintf(out,"private_stream_1");
               break;
    case 0xBE: /* stream_id */
               fprintf(out,"padding_stream");
               break;
    case 0xBF: /* stream_id */
               fprintf(out,"private_stream_2");
               break;
    default   :
               if(stream_id>=0xC0 && stream_id<=0xDF)
               {
                 fprintf(out,"Audio stream - number ");
                 fprintf(out,"%d", stream_id & 31);
                 /* mask 00011111B */
               }
               if(stream_id>=0xE0 && stream_id<=0xEF)
               {
                 fprintf(out,"Video stream - number ");
                 fprintf(out,"%d", stream_id & 15);
                 /* mask 00001111B */
               }
               if(stream_id>=0xF0 && stream_id<=0xFF)
               {
                 fprintf(out,"reserved Data stream - number ");
                 fprintf(out,"%d", stream_id & 15);
                 /* mask 00001111B */
               }
  }
  if( stream.bits.no_name != 3 )  Warn(out,"Invalid marker");
  else
  {
            fprintf(out,"\n|                     STD_buffer_bound_scale           =
%d",stream.bits.STD_buf_bound_scale);
            fprintf(out,"\n|                     STD_buffer_size_bound            =
%d",stream.bits.STD_buf_size_bound_12_8*256+stream.bits.STD_buf_size_bound_7_0);
      fprintf(out,"\n|             (Maximum buffer size = ");
      if(stream.bits.STD_buf_bound_scale)
        fprintf(out,"%ld
bytes)\n",(long)(stream.bits.STD_buf_size_bound_12_8*256+stream.bits.STD_buf_size_boun
d_7_0)*1024);
```

```
      else
        fprintf(out,"%ld
bytes)\n",(long)(stream.bits.STD_buf_size_bound_12_8*256+stream.bits.STD_buf_size_boun
d_7_0)*128);
      }
    streams--;
  }
  fprintf(out," -----------------------------------------------\n");
}


void
PACKET (int stream_id, FILE *in, FILE *out)
/*************************************
 packet_start_code = 4 bytes (already read).

 packet_length = 2 bytes

 stuffing_byte (0xFF) = max 16 bytes

 especificacao buffer = 2 bytes

 especificacao time stamps = 1 or 5 or 10 bytes

 *************************************/
{
  struct buffer_fields
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).*/
    unsigned STD_buf_size_12_8  : 5;
    unsigned STD_buf_scale      : 1;
    unsigned no_name            : 2;

#else
    unsigned no_name            : 2;
    unsigned STD_buf_scale      : 1;
    unsigned STD_buf_size_12_8  : 5;

#endif

    unsigned STD_buf_size_7_0   : 8;
  };

  union
  {
    struct buffer_fields bits;
    char byte[2];
  } buffer;

  struct time_stamps_fields
  {
#ifdef DOS
    unsigned marker_bit_1 : 1;
    unsigned PTS_32_30    : 3;
    unsigned no_name_1    : 4;

    unsigned PTS_29_22    : 8;

    unsigned marker_bit_2 : 1;
    unsigned PTS_21_15    : 7;

    unsigned PTS_14_7     : 8;

    unsigned marker_bit_3 : 1;
    unsigned PTS_6_0      : 7;

    unsigned marker_bit_4 : 1;
    unsigned DTS_32_30    : 3;
    unsigned no_name_2    : 4;

    unsigned DTS_29_22    : 8;

    unsigned marker_bit_5 : 1;
    unsigned DTS_21_15    : 7;

    unsigned DTS_14_7     : 8;
    unsigned marker_bit_6 : 1;
    unsigned DTS_6_0      : 7;

#else
    unsigned no_name_1    : 4;
    unsigned PTS_32_30    : 3;
    unsigned marker_bit_1 : 1;

    unsigned PTS_29_22    : 8;

    unsigned PTS_21_15    : 7;
    unsigned marker_bit_2 : 1;

    unsigned PTS_14_7     : 8;

    unsigned PTS_6_0      : 7;
    unsigned marker_bit_3 : 1;

    unsigned no_name_2    : 4;
    unsigned DTS_32_30    : 3;
    unsigned marker_bit_4 : 1;

    unsigned DTS_29_22    : 8;

    unsigned DTS_21_15    : 7;
    unsigned marker_bit_5 : 1;

    unsigned DTS_14_7     : 8;

    unsigned DTS_6_0      : 7;
    unsigned marker_bit_6 : 1;

#endif
  };

  union
  {
    struct time_stamps_fields bits;
    char byte[10];
  } time;

  register int i, byte, packet_length;
  double time_stamp;



  fprintf(out,"\n --------------------------------------------------\n");

  packet_length = getc(in)*256 + getc(in); /*2 bytes*/
  fprintf(out,"| packet_length  = %d bytes\n", packet_length);

  if(stream_id != 0xBF) /*private_stream_2*/
  {
    /*stuffing_byte*/
    i=0;
    do
    {
      i++;
      if(i > 17)
      {
        Warn(out,"Too much stuffing_bytes (>16)");
        return;
      }
      byte = getc(in);
    } while(byte == 0xFF);
    fprintf(out,"| stuffing_bytes = %d\n", i-1);
    packet_length -= i-1;


    buffer.byte[0] = (char)byte;
    if(buffer.bits.no_name == 1)
    {
      buffer.byte[1] = (char)getc(in);
      fprintf(out,"| STD_buffer_scale   = %d\n",buffer.bits.STD_buf_scale);
                                  fprintf(out,"|     STD_buffer_size                    =
%d\n",buffer.bits.STD_buf_size_12_8*256+buffer.bits.STD_buf_size_7_0);
```

```
        fprintf(out,"| (Buffer size = ");
        if(buffer.bits.STD_buf_scale)
          fprintf(out,"%ld
bytes)\n",(long)(buffer.bits.STD_buf_size_12_8*256+buffer.bits.STD_buf_size_7_0)*1024)
;
        else
          fprintf(out,"%ld
bytes)\n",(long)(buffer.bits.STD_buf_size_12_8*256+buffer.bits.STD_buf_size_7_0)*128);
        packet_length -= 2;
        byte = getc(in);
      }

      if(byte == 0x0F)
      {
        fprintf(out,"| No time stamps\n");
        packet_length--;
      }
      else
      {
        time.byte[0] = byte;
        if(time.bits.no_name_1!=2 && time.bits.no_name_1!=3)
        {
          Warn(out,"Invalid marker(s) in packet header (time stamps)");
          return;
        }

        for(i=1; i<5; i++) time.byte[i]=(char)getc(in);
        if( !time.bits.marker_bit_1 ||
            !time.bits.marker_bit_2 ||
            !time.bits.marker_bit_3     )
        {
          Warn(out,"Invalid marker(s) in packet header (PTS)");
          return;
        }
        time_stamp = time.bits.PTS_32_30 * 1073741824 /* 2^30 */
                    +time.bits.PTS_29_22 * 4194304    /* 2^22 */
                    +time.bits.PTS_21_15 * 32768      /* 2^15 */
                    +time.bits.PTS_14_7  * 128        /* 2^7  */
                    +time.bits.PTS_6_0 ;
                        fprintf(out,"|     presentation_time_stamp     =     %g     (%g
s)\n",time_stamp,time_stamp/90000);
        packet_length -= 5;

        if(time.bits.no_name_1 == 3)
        {
          for(i=5; i<10; i++) time.byte[i]=(char)getc(in);
          if( !time.bits.marker_bit_4 ||
              !time.bits.marker_bit_5 ||
              !time.bits.marker_bit_6 ||
              !time.bits.no_name_2        )
          {
            Warn(out,"Invalid marker(s) in packet header (DTS)");
            return;
          }
          time_stamp = time.bits.DTS_32_30 * 1073741824 /* 2^30 */
                      +time.bits.DTS_29_22 * 4194304    /* 2^22 */
                      +time.bits.DTS_21_15 * 32768      /* 2^15 */
                      +time.bits.DTS_14_7  * 128        /* 2^7  */
                      +time.bits.DTS_6_0 ;
          fprintf(out,"|      decoding_time_stamp                =     %g     (%g
s)\n",time_stamp,time_stamp/90000);
          packet_length -= 5;
        }
      }
      fprintf(out,"| packet_data     = %d bytes\n", packet_length);
    }

    /* Elementary streams extraction */
    if(stream_id != 0xBE) Extracts(in, 0, stream_id, packet_length);

    fprintf(out," ---------------------------------------------------\n");
}


void
RemoveStream (int PID, int stream_id)
/***********************************
```

```
 Used in 'SYSTEM_HEADER()' and 'PES_PACKET()'.
 Removes the stream file in order to prevent 'Extracts()' from adding streams
 to existing trash.
 ************************************/
{
  /* boolean extract;
     char no_extension[80]; GLOBAL VAR. !*/
  char s[80], str[80];

  if( extract )
  {
    sprintf(s,"%.4s%04X.%X", no_extension, PID, stream_id);
    if(remove(s))
    {
      sprintf(str,"(can't remove %s (elementary stream))", s);
      perror(str);
    }
  }
}


void
Extracts (FILE *in, int PID, int stream_id, int length)
/************************************
 Adds packet_data_bytes to a file which name is composed of the input filename
 and the PID, and which extension is the stream_id in hexadecimal.
 The number of bytes written (length) should be the indicated in the
 packet_header, not the number of bytes until next packet.
 ************************************/
{
  /* boolean extract;
     char no_extension[80]; GLOBAL VAR. !*/
  char str[80], s[80];
  FILE *stream;

  if( !extract ) /* Ignores */
  {
    fseek(in, length, SEEK_CUR);
    return;
  }

  sprintf(s,"%.4s%04X.%X", no_extension, PID, stream_id);

  if((stream=fopen(s,"ab")) == NULL)
  {
    sprintf(str,"\nab - can't open %s (elementary stream)", s);
    perror(str);
    return;
  }

  while(length--)  putc(getc(in),stream);

  fclose(stream);
}


/*/////////////////////////////////////////////////////////////////////////////
  TRANSPORT_STREAM ROUTINES
/////////////////////////////////////////////////////////////////////////////*/


boolean
TRANSPORT_STREAM (FILE *in, FILE *out)
/************************************
 In the Transport Packets (TPs) identification I assume that there can be trash
 between packets and without synchronism byte emulation.
 Returns YES or NO depending on a valid stream parsing.
 Counts the number of processed packets.
 Initializes the PSI (Program Specific Information) linked list pointers
 (global variables). An empty table is a one element list.
 ************************************/
{
  /* PAT_ptr PAT, new_PAT; PMT_ptr PMT, new_PMT; CAT_ptr CAT, new_CAT;
     int PAT_version, PMT_version, CAT_version;
     boolean unconditional;
     unsigned long n_TPs;     GLOBAL VAR.! */
```

```
boolean aux; PAT_ptr pat_aux; PMT_ptr pmt_aux; CAT_ptr cat_aux;/*Auxiliary*/

while(getc(in) != 0x47)  if(feof(in)) return NO; /* Luck ? */


/* Initialization of PSI tables (empty).*/
if( (new_PAT=(PAT_ptr)malloc(sizeof(struct PA_table))) == NULL )
 {puts("TRANSPORT_STREAM(): malloc error"); exit(1);}
new_PAT->next=NULL;
if( (new_PMT=(PMT_ptr)malloc(sizeof(struct PM_table))) == NULL )
 {puts("TRANSPORT_STREAM(): malloc error"); exit(1);}
new_PMT->next=NULL;
if( (new_CAT=(CAT_ptr)malloc(sizeof(struct CA_table))) == NULL )
 {puts("TRANSPORT_STREAM(): malloc error"); exit(1);}
new_CAT->next=NULL;
PAT = new_PAT; /* The current table is new */
PMT = new_PMT;
CAT = new_CAT;
PAT_version = 0;
PMT_version = 0;
CAT_version = 0;

/* Initialization of a PMT type table used to extract PES before the PSI
tables are complete */
if( unconditional )
{
  if( (ALL=(PMT_ptr)malloc(sizeof(struct PM_table))) == NULL )
  {puts("TRANSPORT_STREAM(): malloc error"); exit(1);}
  ALL->next=NULL;
}
else ALL = NULL;


TS_TRANSPORT_PACKET(in, out);
n_TPs = 1;
fprintf(out,"1\n");
printf("TS_packets:\n1\t\a");

while( !feof(in) ) if(getc(in) == 0x47)
                   {
                     TS_TRANSPORT_PACKET(in, out);
                     n_TPs++;
                     fprintf(out,"%ld\n", n_TPs);
                     printf("%ld\t", n_TPs);
                   }
printf("\a");


/* Memory release */

if(ALL != NULL)
{
  while(ALL->next != NULL)
  {
    pmt_aux = ALL->next;
    free(ALL);
    ALL = pmt_aux;
  }
  free(ALL);
}


if(PAT == new_PAT) aux = NO;
else aux = YES;

if(PAT != NULL)
{
  while(PAT->next != NULL)
  {
    pat_aux = PAT->next;
    free(PAT);
    PAT = pat_aux;
  }
  free(PAT);
}
```

```
    if(aux && new_PAT != NULL)
    {
      while(new_PAT->next != NULL)
      {
        pat_aux = new_PAT->next;
        free(new_PAT);
        new_PAT = pat_aux;
      }
      free(new_PAT);
    }


    if(PMT == new_PMT) aux = NO;
    else aux = YES;

    if(PMT != NULL)
    {
      while(PMT->next != NULL)
      {
        pmt_aux = PMT->next;
        free(PMT);
        PMT = pmt_aux;
      }
      free(PMT);
    }

    if(aux && new_PMT != NULL)
    {
      while(new_PMT->next != NULL)
      {
        pmt_aux = new_PMT->next;
        free(new_PMT);
        new_PMT = pmt_aux;
      }
      free(new_PMT);
    }


    if(CAT == new_CAT) aux = NO;
    else aux = YES;

    if(CAT != NULL)
    {
      while(CAT->next != NULL)
      {
        cat_aux = CAT->next;
        free(CAT);
        CAT = cat_aux;
      }
      free(CAT);
    }

    if(aux && new_CAT != NULL)
    {
      while(new_CAT->next != NULL)
      {
        cat_aux = new_CAT->next;
        free(new_CAT);
        new_CAT = cat_aux;
      }
      free(new_CAT);
    }


    return YES;
}


void
TS_TRANSPORT_PACKET (FILE *in, FILE *out)
/****************************************
 sync_byte = 1 byte (already read).

 transport_error_indicator = 1 bit       \
 payload_unit_start_indicator = 1 bit     \
 transport_priority = 1 bit                \
 PID = 13 bits                              > 3 bytes.
 transport_scrambling_control = 2 bits   /
 adaptation_field_control = 2 bits      /
 continuity_counter = 4 bits           /

 adaptation_field and/or data_bytes = until 184 bytes.

 ************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned PID_12_8                     :  5;
    unsigned transport_priority           :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_error_indicator    :  1;

    unsigned PID_7_0                    :  8;

    unsigned continuity_counter           :  4;
    unsigned adaptation_field_control     :  2;
    unsigned transport_scrambling_control :  2;

#else
    unsigned transport_error_indicator    :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_priority           :  1;
    unsigned PID_12_8                     :  5;

    unsigned PID_7_0                    :  8;

    unsigned transport_scrambling_control :  2;
    unsigned adaptation_field_control     :  2;
    unsigned continuity_counter           :  4;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;

  int PID;

  /* PAT_ptr PAT; PMT_ptr PMT, ALL; GLOBAL VAR.! */

  register int i;
  char str[80];
  int bytes_read;
  int PM_flag;
  PAT_ptr PA_ptr;
  PMT_ptr PES_ptr;


  fprintf(out,"\n ------------------------------------------------\n");

  /*Read 3 bytes of TP header.*/
  for(i=0; i<3; i++)  header.byte[i]=(char)getc(in);
  if( feof(in) )
  {
    Warn(out,"Premature EOF in TP header");
    return;
  }
  bytes_read = 4;

  /* Packet Identifier */
  PID = header.bits.PID_12_8 * 256
       +header.bits.PID_7_0;
  fprintf(out,"| PID = 0x%04X ", PID);
  if(PID == 0) fprintf(out,"(Program Association Table)");
  if(PID == 1) fprintf(out,"(Conditional Access Table)");
  if(PID == 0x1FFF)
  {
    fprintf(out,"(Null Packet");
    if( header.bits.payload_unit_start_indicator ||
        header.bits.transport_scrambling_control ||
        !header.bits.adaptation_field_control
      )
      fprintf(out," with invalid header");
    fprintf(out,")");
  }

  /* transport_error_indicator */
                          fprintf(out,"\n|            transport_error_indicator             =
%d",header.bits.transport_error_indicator);
  if( header.bits.transport_error_indicator )
    fprintf(out," (at least one uncorrectable bit error)");

  /* payload_unit_start_indicator */
                   fprintf(out,"\n|          payload_unit_start_indicator             =
%d\n",header.bits.payload_unit_start_indicator);

  /* transport_priority */
       fprintf(out,"|    transport_priority                                =
%d\n",header.bits.transport_priority);

  /* transport_scrambling_control */
                    fprintf(out,"|          transport_scrambling_control             =
%d",header.bits.transport_scrambling_control);
  if( header.bits.transport_scrambling_control )
    fprintf(out," (scrambled - user defined)\n");
  else
    fprintf(out,"\n");

  /* adaptation_field_control */
          fprintf(out,"|     adaptation_field_control                  =     %d
",header.bits.adaptation_field_control);
  switch( header.bits.adaptation_field_control )
  {
    case 0: i = 188 - bytes_read;
            fprintf(out,"(Reserved for future use: %d bytes ignored)\n",i);
            fseek(in, i, SEEK_CUR);
            fprintf(out," ------------------------------------------------ ");
            return;
    case 1: fprintf(out,"(payload only)\n");
            break;
    case 2: fprintf(out,"(adaptation_field only)\n");
            break;
    case 3: fprintf(out,"(adaptation_field followed by payload)\n");
            break;
  }

  /* continuity_counter */
/*
SEE STANDARD !!!
*/
  fprintf(out,"| continuity_counter = %d ",header.bits.continuity_counter);
  fprintf(out,"(counter of payloads with the same PID)\n");

  if(PID == 0x1FFF)
  {
    i = 188 - bytes_read;
    fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
    bytes_read = 188;
    fprintf(out," ------------------------------------------------ ");
    return;
  }

  /* adaptation_field */
  i = header.bits.adaptation_field_control;
  if( i == 2  ||  i == 3 )  TS_ADAPTATION_FIELD(i, &bytes_read, in, out);

  /* payload */
  if( header.bits.adaptation_field_control == 1 ||
      header.bits.adaptation_field_control == 3    )
  {
    if(bytes_read >= 188) Warn(out,"No space for payload");
    else
    {
      i = header.bits.payload_unit_start_indicator;
```

```
        if(PID == 0)  TS_PROGRAM_ASSOCIATION_SECTION(i, &bytes_read, in, out);
        else
        if(PID == 1)  TS_CA_SECTION(i, &bytes_read, in, out);
        else
        {
          PA_ptr = PA_PID(PAT, PID, &PM_flag);

          if(PA_ptr != NULL)
          {
            if(!PM_flag)  fprintf(out,"|\n| (Network Information Table)\n");
            else
            TS_PROGRAM_MAP_SECTION(i, &bytes_read, in, out);
          }
          else
          {
/* if( CA_PID(PID) ) CA() else ... */

            /*If there's a table with all PIDs then PES extraction is
              independent of PMT */
            if(ALL != NULL)
            {
              PES_ptr = ALL_PID(ALL, PID);

              if(PES_ptr == NULL)
              {
                /*Linked list new element.*/
                if( (PES_ptr=(PMT_ptr)malloc(sizeof(struct PM_table))) == NULL )
                {puts("TS_TRANSPORT_PACKET(): malloc error"); exit(1);}

                /*Insert new element in table. At the list start, just after the
                  head */
                PES_ptr->next = ALL->next;
                ALL->next = PES_ptr;

                PES_ptr->elementary_PID = PID;
                PES_ptr->old_stream_removed = NO;
                PES_ptr->field = 0; /*PES_PACKET() ignored if there's no unit_start*/
              }
              else /* duplicate packets */
              if(PES_ptr->continuity_counter == header.bits.continuity_counter)
              {
                i = 188 - bytes_read;
                sprintf(str,"Duplicate packets: payload ignored (%d bytes)",i);
                Warn(out, str);
                fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
                bytes_read = 188;
                fprintf(out," ------------------------------------------------ ");
                return;
              }

              PES_ptr->continuity_counter = header.bits.continuity_counter;
              PES_PACKET(PES_ptr,i,&bytes_read,in,out);
            }
            else
            {
              PES_ptr = Elementary_PID(PMT, PID);

              if( PES_ptr != NULL &&
                  PES_ptr->continuity_counter != header.bits.continuity_counter )
              {
                PES_ptr->continuity_counter = header.bits.continuity_counter;
                PES_PACKET(PES_ptr, i, &bytes_read, in, out);
              }
              else
              {
                i = 188 - bytes_read;
                fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
                bytes_read = 188;

                if(PES_ptr == NULL)
                  sprintf(str,"Unknown PID: payload ignored (%d bytes)",i);
                else
                  sprintf(str,"Duplicate packets: payload ignored (%d bytes)",i);

                Warn(out, str);
              }
            }
          }
        }
      }
    }

    fprintf(out," ------------------------------------------------- ");
  }


  PAT_ptr
  PA_PID (PAT_ptr table, int PID, int *PM_flag)
  /*************************************
   Searches PA_table (GLOBAL) looking for the PID given as an argument.
   If found it's relative (except for program_number = 0, Network) to packets
   with PM_table (the flag is activated).
   Returns a pointer to the found table entry (or NULL).
   The first element of the linked isn't part of the table information.
   *************************************/
  {
    PAT_ptr ptr;


    *PM_flag = 0;
    if(table != NULL) /*active table*/
    {
      ptr = table->next;
      while(ptr != NULL)
      {
        if(ptr->PID == PID)
        {
          if(ptr->program_number != 0) *PM_flag = 1;
          return ptr;
        }
        ptr = ptr->next;
      }
    }

    return NULL;
  }


  PMT_ptr
  Elementary_PID (PMT_ptr table, int PID)
  /*************************************
   Searches PM_table (GLOBAL) looking for the PID given as an argument.
   If found it's relative to packets with elementary streams.
   Returns a pointer to the found table entry (or NULL).
   The first element of the linked isn't part of the table information.
   *************************************/
  {
    PMT_ptr ptr;

    if(table != NULL) /*active table*/
    {
      ptr = table->next;
      while(ptr != NULL)
      {
        if(ptr->elementary_PID == PID) return ptr;
        ptr = ptr->next;
      }
    }

    return NULL;
  }


  PMT_ptr
  ALL_PID (PMT_ptr table, int PID)
  /*************************************
   Searches ALL table (GLOBAL) looking for the PID given as an argument.
   If found it could be relative to packets with elementary streams.
   Returns a pointer to the found table entry (or NULL).
   The first element of the linked isn't part of the table information.
   *************************************/
  {
    PMT_ptr ptr;
  if(table == NULL)
  { puts("error!"); exit(3); }

  ptr = table->next;
  while(ptr != NULL)
  {
    if(ptr->elementary_PID == PID) return ptr;
    ptr = ptr->next;
  }

  return NULL;
}


void
TS_ADAPTATION_FIELD (int control, int *bytes_read, FILE *in, FILE *out)
/***************************************
 adaptation_field_length = 1 byte

 descontinuity_indicator = 1 bit              \
 random_access_indicator = 1 bit               \
 elementary_stream_priority_indicator = 1 bit   \
 PCR_flag = 1 bit                                > 1 byte.
 OPCR_flag = 1 bit                              /
 splicing_point_flag = 1 bit                   /
 transport_private_data_flag = 1 bit          /
 adaptation_field_extension_flag = 1 bit     /

 program_clock_reference_base = 33 bits      \
 reserved = 6 bits                            > 6 optional bytes.
 program_clock_reference_extension = 9 bits /

 original_program_clock_reference_base = 33 bits      \
 reserved = 6 bits                                     > 6 optional bytes.
 original_program_clock_reference_extension = 9 bits /

 splice_countdown = 1 optional byte.

 transport_private_data_length = 1 optional byte.
 private_data_byte = n optional bytes.

 adaptation_field_extension_length = 8 bits \
 ltw_flag = 1 bit                            \
 piecewise_rate_flag = 1 bit                  > 2 optional bytes.
 seamless_splice_flag = 1 bit                /
 reserved = 5 bits                          /

 stuffing_byte = n optional bytes.

 ***************************************/
{
  struct flag_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned adaptation_field_extension_flag      :  1;
    unsigned transport_private_data_flag          :  1;
    unsigned splicing_point_flag                  :  1;
    unsigned OPCR_flag                            :  1;
    unsigned PCR_flag                             :  1;
    unsigned elementary_stream_priority_indicator :  1;
    unsigned random_access_indicator              :  1;
    unsigned descontinuity_indicator              :  1;

#else
    unsigned descontinuity_indicator              :  1;
    unsigned random_access_indicator              :  1;
    unsigned elementary_stream_priority_indicator :  1;
    unsigned PCR_flag                             :  1;
    unsigned OPCR_flag                            :  1;
    unsigned splicing_point_flag                  :  1;
    unsigned transport_private_data_flag          :  1;
    unsigned adaptation_field_extension_flag      :  1;
#endif
  };

  union
  {
    struct flag_bits bits;
    char byte;
  } flags;

  struct PCR_bits /* Used for PCR and OPCR */
  {
    unsigned PCR_base_32_25     :  8;

    unsigned PCR_base_24_17     :  8;

    unsigned PCR_base_16_9      :  8;

    unsigned PCR_base_8_1       :  8;

#ifdef DOS
    unsigned PCR_extension_8    :  1;
    unsigned reserved           :  6;
    unsigned PCR_base_0         :  1;
#else
    unsigned PCR_base_0         :  1;
    unsigned reserved           :  6;
    unsigned PCR_extension_8    :  1;
#endif

    unsigned PCR_extension_7_0 :  8;
  };

  union
  {
    struct PCR_bits bits;
    char byte[6];
  } PCR;

  struct extension_flag_bits
  {
#ifdef DOS
    unsigned reserved               :  5;
    unsigned seamless_splice_flag   :  1;
    unsigned piecewise_rate_flag    :  1;
    unsigned ltw_flag               :  1;

#else
    unsigned ltw_flag               :  1;
    unsigned piecewise_rate_flag    :  1;
    unsigned seamless_splice_flag   :  1;
    unsigned reserved               :  5;

#endif
  };

  union
  {
    struct extension_flag_bits bits;
    char byte;
  } extension;

  struct ltw_bits
  {
#ifdef DOS
    unsigned ltw_offset_14_8 :  7;
    unsigned ltw_valid_flag  :  1;

#else
    unsigned ltw_valid_flag  :  1;
    unsigned ltw_offset_14_8 :  7;

#endif

    unsigned ltw_offset_7_0  :  8;
  };

  union
  {
    struct ltw_bits bits;
    char byte[2];
  } ltw;

  struct piecewise_rate_bits
  {
#ifdef DOS
    unsigned piecewise_rate_21_16 :  5;
    unsigned reserved             :  2;

#else
    unsigned reserved             :  2;
    unsigned piecewise_rate_21_16 :  6;

#endif

    unsigned piecewise_rate_15_8  :  8;
    unsigned piecewise_rate_7_0   :  8;
  };

  union
  {
    struct piecewise_rate_bits bits;
    char byte[2];
  } piecewise_rate;

  struct seamless_splice_bits
  {
#ifdef DOS
    unsigned marker_bit_1       :  1;
    unsigned DTS_next_au_32_30  :  3;
    unsigned splice_type        :  4;

    unsigned DTS_next_au_29_22  :  8;

    unsigned marker_bit_2       :  1;
    unsigned DTS_next_au_21_15  :  7;

    unsigned DTS_next_au_14_8   :  8;

    unsigned marker_bit_3       :  1;
    unsigned DTS_next_au_7_0    :  7;

#else
    unsigned splice_type        :  4;
    unsigned DTS_next_au_32_30  :  3;
    unsigned marker_bit_1       :  1;

    unsigned DTS_next_au_29_22  :  8;

    unsigned DTS_next_au_21_15  :  7;
    unsigned marker_bit_2       :  1;

    unsigned DTS_next_au_14_8   :  8;

    unsigned DTS_next_au_7_0    :  7;
    unsigned marker_bit_3       :  1;

#endif
  };

  union
  {
    struct seamless_splice_bits bits;
    char byte[5];
  } splice;

  int     adaptation_field_length;
  double PCR_base, PCR_extension,
        OPCR_base, OPCR_extension;
  int     splice_countdown,
        transport_private_data_length,
        adaptation_field_extension_length,
        stuffing_bytes;
  char str[80];
  register int i,
              n_bytes;/*counter of bytes in the adaptation_field*/


  fprintf(out,"|ADAPTATION FIELD*****************************\n");

  /*Read adaptation_field_length.*/
  adaptation_field_length = getc(in);
  if( feof(in) )
  {
    Warn(out,"Premature EOF in adaptation_field_length");
    exit(1);
  }
  n_bytes = 1;
  fprintf(out,"| adaptation_field_length = %d\n",adaptation_field_length);
  if(control == 3 && (adaptation_field_length > 183 || adaptation_field_length < 0) ||
     control == 2 && adaptation_field_length != 183 )
  {
    i = 188 - *bytes_read -n_bytes;
    sprintf(str,"Invalid length: %d bytes ignored",i);
    Warn(out, str);
    fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
    *bytes_read = 188;
    return;
  }
  if(!adaptation_field_length) /* single stuffing_byte*/
  {
    *bytes_read += n_bytes;
    return;
/* To be able to read MATRA bitstream one has to go back ! */
/* fseek(in, -1, SEEK_CUR);  return; */
  }

  /*Read flags byte.*/
  flags.byte = (char)getc(in);
  if( feof(in) )
  {
    Warn(out,"Premature EOF in flags");
    exit(1);
  }
  n_bytes++;

  /* descontinuity_indicator */
  i = flags.bits.descontinuity_indicator;
/*
SEE STANDARD !!!
*/
  if(detail < 2 || i)
  {
    fprintf(out,"| descontinuity_indicator = %d ", i);
     if(i) fprintf(out,"(new time base (PCR) in this or next packet(s) with the same
PID)");
    fprintf(out,"\n");
  }

  /* random_access_indicator */
  i = flags.bits.random_access_indicator;
/*
SEE STANDARD !!!
*/
  if(detail < 2 || i)
    fprintf(out,"| random_access_indicator = %d\n", i);

  /* elementary_stream_priority_indicator */
  i = flags.bits.elementary_stream_priority_indicator;
  if(detail < 2 || i)
    fprintf(out,"| ES_priority_indicator    = %d\n", i);

  /* PCR_flag */
  i = flags.bits.PCR_flag;
  if(detail < 2 || i)
    fprintf(out,"| PCR_flag                              = %d\n", i);

  /* OPCR_flag */
  i = flags.bits.OPCR_flag;
```

```
if(detail < 2 || i)
  fprintf(out,"| OPCR_flag                        = %d\n", i);

/* splicing_point_flag */
i = flags.bits.splicing_point_flag;
if(detail < 2 || i)
  fprintf(out,"| splicing_point_flag              = %d\n", i);

/* transport_private_data_flag */
i = flags.bits.transport_private_data_flag;
if(detail < 2 || i)
  fprintf(out,"| transport_private_data_flag      = %d\n", i);

/* adaptation_field_extension_flag */
i = flags.bits.adaptation_field_extension_flag;
if(detail < 2 || i)
  fprintf(out,"| adaptation_field_extension_flag = %d\n", i);

/* Program Clock Reference */
if( flags.bits.PCR_flag )
{
  /*Read 6 bytes of PCR.*/
  for(i=0; i<6; i++)  PCR.byte[i]=(char)getc(in);
  if( feof(in) )
  {
    Warn(out,"Premature EOF in PCR");
    exit(1);
  }
  n_bytes += 6;

  PCR_base = (double)PCR.bits.PCR_base_32_25 * 33554432
           +(double)PCR.bits.PCR_base_24_17 * 131072
           +(double)PCR.bits.PCR_base_16_9  * 512
           +(double)PCR.bits.PCR_base_8_1   * 2
           +(double)PCR.bits.PCR_base_0;
  fprintf(out,"|\n| PCR_base = %g\n", PCR_base);

  if(detail < 1)
    fprintf(out,"| reserved = %d\n", PCR.bits.reserved);

  PCR_extension = (double)PCR.bits.PCR_extension_8 * 256
               +(double)PCR.bits.PCR_extension_7_0;
  fprintf(out,"| PCR_extension = %g      ", PCR_extension);

  fprintf(out,"(PCR = %g s)\n", (PCR_base*300+PCR_extension)/27000000);
}

/* Original Program Clock Reference */
if( flags.bits.OPCR_flag )
{
  /*Read 6 bytes of OPCR.*/
  for(i=0; i<6; i++)  PCR.byte[i]=(char)getc(in);
  if( feof(in) )
  {
    Warn(out,"Premature EOF in OPCR");
    exit(1);
  }
  n_bytes += 6;

  OPCR_base = (double)PCR.bits.PCR_base_32_25 * 33554432
            +(double)PCR.bits.PCR_base_24_17 * 131072
            +(double)PCR.bits.PCR_base_16_9  * 512
            +(double)PCR.bits.PCR_base_8_1   * 2
            +(double)PCR.bits.PCR_base_0;
  fprintf(out,"|\n| OPCR_base = %g\n", OPCR_base);

  if(detail < 1)
    fprintf(out,"| reserved = %d\n", PCR.bits.reserved);

  OPCR_extension = (double)PCR.bits.PCR_extension_8 * 256
                 +(double)PCR.bits.PCR_extension_7_0;
  fprintf(out,"| OPCR_extension = %g      ", OPCR_extension);

  fprintf(out,"(OPCR = %g s)\n", (OPCR_base*300+OPCR_extension)/27000000);
}

/*Read splice_countdown.*/
```

```
/*
SEE STANDARD !!!
*/
  if( flags.bits.splicing_point_flag )
  {
    splice_countdown = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in splice_countdown");
      exit(1);
    }
    n_bytes++;

    fprintf(out,"|\n| splice_countdown = %d\n", splice_countdown);
  }

  /*Read private data.*/
  if( flags.bits.transport_private_data_flag )
  {
    /*Read length*/
    transport_private_data_length = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in transport_private_data_length");
      exit(1);
    }
    n_bytes++;

                              fprintf(out,"|\n|        transport_private_data_length        =
%d\n",transport_private_data_length);
    if((transport_private_data_length + n_bytes -1) > adaptation_field_length)
    {
      i = 188 - *bytes_read -n_bytes;
      sprintf(str,"Invalid length: %d bytes ignored",i);
      Warn(out, str);
      fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
      *bytes_read = 188;
      return;
    }

    /*Read data*/
    fprintf(out,"| (Private Data)\n");
    fseek(in, transport_private_data_length, SEEK_CUR);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in private data");
      exit(1);
    }
    n_bytes += transport_private_data_length;
  }

  /*Read adaptation_field extension.*/
/*
SEE STANDARD !!!
*/
  if( flags.bits.adaptation_field_extension_flag )
  {
    /*Read length*/
    adaptation_field_extension_length = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in adaptation_field_extension_length");
      exit(1);
    }
    n_bytes++;

                            fprintf(out,"|\n|       adaptation_field_extension_length       =
%d\n",adaptation_field_extension_length);
    if((adaptation_field_extension_length + n_bytes -1) > adaptation_field_length)
    {
      i = 188 - *bytes_read -n_bytes;
      sprintf(str,"Invalid length: %d bytes ignored",i);
      Warn(out, str);
      fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
      *bytes_read = 188;
      return;
    }
    if(adaptation_field_extension_length > 0)
    {
/*
SEE STANDARD !!!
*/
      /*Read extension flags.*/
      extension.byte = getc(in);
      if( feof(in) )
      {
        Warn(out,"Premature EOF in extension flags");
        exit(1);
      }
      n_bytes++;
      adaptation_field_extension_length--;

      /* ltw_flag */
      fprintf(out,"|\n| ltw_flag              = %d\n",extension.bits.ltw_flag);

      /* piecewise_rate_flag */
      fprintf(out,"| piecewise_rate_flag  = %d\n",extension.bits.piecewise_rate_flag);

      /* seamless_splice_flag*/
                                          fprintf(out,"|        seamless_splice_flag        =
%d\n",extension.bits.seamless_splice_flag);

      if(detail < 1)
        fprintf(out,"| reserved = %d\n", extension.bits.reserved);

      /* Legal Time Window */
      if(extension.bits.ltw_flag)
      {
        /*Read 2 bytes of ltw*/
        ltw.byte[0] = getc(in);
        ltw.byte[1] = getc(in);
        if( feof(in) )
        {
          Warn(out,"Premature EOF in ltw");
          exit(1);
        }
        n_bytes += 2;
        adaptation_field_extension_length -= 2;

        /* ltw_valid_flag*/
        fprintf(out,"|\n| ltw_valid_flag = %d\n",ltw.bits.ltw_valid_flag);

        /* ltw_offset*/
        fprintf(out,"| ltw_offset = %d\n",ltw.bits.ltw_offset_14_8 *256
                                        +ltw.bits.ltw_offset_7_0);
      }

      /* Piecewise Rate */
      if(extension.bits.piecewise_rate_flag)
      {
        /*Read 3 bytes of piecewise_rate*/
        for(i=0; i<3; i++)  piecewise_rate.byte[i]=(char)getc(in);
        if( feof(in) )
        {
          Warn(out,"Premature EOF in piecewise_rate");
          exit(1);
        }
        n_bytes += 3;
        adaptation_field_extension_length -= 3;

        if(detail < 1)
          fprintf(out,"|\n| reserved = %d\n",piecewise_rate.bits.reserved);

        /* piecewise_rate_offset*/
        fprintf(out,"| piecewise_rate = %d\n",
                    piecewise_rate.bits.piecewise_rate_21_16 *65536
                   +piecewise_rate.bits.piecewise_rate_15_8  *256
                   +piecewise_rate.bits.piecewise_rate_7_0);
      }

      /* Seamless Splice */
      if(extension.bits.seamless_splice_flag)
      {
        /*Read 5 bytes.*/
        for(i=0; i<5; i++)  splice.byte[i]=(char)getc(in);
        if( feof(in) )
        {
          Warn(out,"Premature EOF prematuro in Seamless Splice");
          exit(1);
        }
        n_bytes += 5;
        adaptation_field_extension_length -= 5;

        if( !splice.bits.marker_bit_1             ||
            !splice.bits.marker_bit_2             ||
            !splice.bits.marker_bit_3               )
        {
          i = 188 - *bytes_read -n_bytes;
          sprintf(str,"Splice: invalid marker(s) - %d bytes ignored",i);
          Warn(out, str);
          fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
          *bytes_read = 188;
          return;
        }

        /* splice_type */
        fprintf(out,"|\n| splice_type = %d\n",splice.bits.splice_type);

        /* DTS_next_au */
        fprintf(out,"| STS_next_au = %g\n",
                     (double)splice.bits.DTS_next_au_32_30 *1073741824
                    +(double)splice.bits.DTS_next_au_29_22 *4194304
                    +(double)splice.bits.DTS_next_au_21_15 *32768
                    +(double)splice.bits.DTS_next_au_14_8  *256
                    +(double)splice.bits.DTS_next_au_7_0);
      }

      if(adaptation_field_extension_length < 0)
      {
       i = 188 - *bytes_read -n_bytes;
       sprintf(str,"adaptation_field_extension   length   doesn't   fit:   %d   bytes
ignored",i);
       Warn(out, str);
       fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
       *bytes_read = 188;
       return;
      }

      /*Read reserved bytes.*/
      fprintf(out,"| (%d reserved bytes)\n",adaptation_field_extension_length);
      fseek(in, adaptation_field_extension_length, SEEK_CUR);
      if( feof(in) )
      {
       Warn(out,"Premature EOF in reserved bytes");
       exit(1);
      }
      n_bytes += adaptation_field_extension_length;
    }
  }

  /*Stuffing bytes*/
  fprintf(out,"|\n| %d bytes left - ",adaptation_field_length +1 - n_bytes);
  for(stuffing_bytes=0, i=0; i < adaptation_field_length +1 - n_bytes; i++)
    if(getc(in) == 0xFF) stuffing_bytes++;
  fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

  *bytes_read += (adaptation_field_length +1);
}


void
TS_PROGRAM_ASSOCIATION_SECTION(int unit_start,int *bytes_read,FILE *in,FILE *out)
/************************************
 pointer_field = 1 optional byte depending on 'unit_start'.

 table_id = 1 byte

 section_syntax_indicator = 1 bit \
 no_name = 1 bit                  \
 reserved1 = 2 bits               \
 section_length = 12 bits                > 5 bytes.
 transport_stream_id = 16 bits          /
 reserved2 = 2 bits                    /
 version_number = 5 bits              /
 current_next_indicator = 1 bit     /

 section_number = 1 byte;

 last_section_number = 1 byte;

 program_number = 16 bits                \
 reserved = 3 bits                        > N * 4 bytes.
 network_or_program_map_PID = 13 bits /

 CRC_32 = 4 bytes;

 Uses a buffer for the section, defined in a global struct. This way one can
 reconstruct the section from packet to packet and do the final CRC.
 descriptor_start is not used in this struct.

 ***********************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned section_length_11_8      :  4;
    unsigned reserved1                :  2;
    unsigned no_name                  :  1;
    unsigned section_syntax_indicator :  1;

    unsigned section_length_7_0       :  8;

    unsigned transport_stream_id_15_8 :  8;

    unsigned transport_stream_id_7_0  :  8;

    unsigned current_next_indicator   :  1;
    unsigned version_number           :  5;
    unsigned reserved2                :  2;

#else
    unsigned section_syntax_indicator :  1;
    unsigned no_name                  :  1;
    unsigned reserved1                :  2;
    unsigned section_length_11_8      :  4;

    unsigned section_length_7_0       :  8;

    unsigned transport_stream_id_15_8 :  8;

    unsigned transport_stream_id_7_0  :  8;

    unsigned reserved2                :  2;
    unsigned version_number           :  5;
    unsigned current_next_indicator   :  1;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[5];
  } header;

  struct table_bits
  {
    unsigned program_number_15_8            :  8;

    unsigned program_number_7_0             :  8;

#ifdef DOS
    unsigned network_or_program_map_PID_12_8 :  5;
    unsigned reserved                        :  3;
```

```
#else
    unsigned reserved                         :  3;
    unsigned network_or_program_map_PID_12_8 :  5;
#endif

    unsigned network_or_program_map_PID_7_0  :  8;
  };

  union
  {
    struct table_bits bits;
    char byte[6];
  } table;

  int    pointer_field,
         table_id,
         section_number,
         last_section_number,
         stuffing_bytes;
  double CRC_32;

  /* struct {} PA_section;
     PAT_ptr new_PAT, PAT;
     int PAT_version;       GLOBAL VAR.! */

  char str[80];
  register int i,
               n_bytes = 0;/* counter of PA_section bytes */
  int j, k;     /*Auxiliary*/
  PAT_ptr ptr; /*Auxiliary: free of PAT and new_PAT; insertion of new elements
                in table*/


  fprintf(out,"|PROGRAM ASSOCIATION SECTION*******************\n");

  if( unit_start )
  {
    /*Read pointer_field.*/
    pointer_field = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in pointer_field");
      exit(1);
    }
    n_bytes = 1;
    fprintf(out,"| pointer_field = %d bytes\n", pointer_field);
    if(pointer_field + *bytes_read + 1 > 188)
    {
      i = 188 - *bytes_read -n_bytes;
      sprintf(str,"Invalid length: %d bytes ignored",i);
      Warn(out, str);
      fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
      *bytes_read = 188;
      return;
    }

    if(pointer_field == 0) PA_section.p = 0; /*Byte 1: PA_section.p = 1*/
    /*! Maybe should verify if previous section was completely read.*/
  }

  while(*bytes_read + n_bytes < 188) /*Verifies end of packet.*/
  {
    /*Read a section byte to a buffer.*/
    PA_section.buffer[PA_section.p+1] = (char)getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in program_association_section.");
      exit(1);
    }
    n_bytes++;
    PA_section.p++;
    PA_section.bytes_left--;

    if(PA_section.p == 8) /*Header is in buffer.*/
    {
      /* table_id */
      table_id = PA_section.buffer[1];
      fprintf(out,"| table_id = %d\n", table_id);

      /*Read header.*/
      for(i=0; i<5; i++)  header.byte[i] = PA_section.buffer[i+2];

      /* section_syntax_indicator */
                                            fprintf(out,"|        section_syntax_indicator       =
%d\n",header.bits.section_syntax_indicator);

      if(  table_id != 0                          ||
          !header.bits.section_syntax_indicator ||
           header.bits.no_name                     )
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid PA_section header: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }

      if(detail > 1)
        fprintf(out,"| reserved = %d\n", header.bits.reserved1);

      /*Read section length.*/
      /*Initialization of 'PA_section.bytes_left'.*/
      PA_section.bytes_left = header.bits.section_length_11_8 * 256
                             +header.bits.section_length_7_0;
      fprintf(out,"| section_length = %d\n", PA_section.bytes_left);
      if(PA_section.bytes_left > 1021) /*See buffer size*/
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid length: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }
      PA_section.bytes_left -= 5;

      /* transport_stream_id */ /*JUST ONE TS ASSUMED !*/
      fprintf(out,"| transport_stream_id = %d\n",
                                  header.bits.transport_stream_id_15_8 * 256
                                 +header.bits.transport_stream_id_7_0       );

      if(detail > 1)
        fprintf(out,"| reserved = %d\n", header.bits.reserved2);

      /* version_number */
      fprintf(out,"| version_number = %d\n", header.bits.version_number);
      i = (int)header.bits.version_number - PAT_version;
      if(i != 0) /* Table version management */
      {
        if(i != 1 && i != -31)
        {
          i = 188 - *bytes_read -n_bytes;
          sprintf(str,"version_number not in sequence: %d bytes ignored",i);
          Warn(out, str);
          fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
          *bytes_read = 188;
          return;
        }

        if(PAT != new_PAT)
        {
          Warn(out,"Previous version not used");
          /*free new_PAT*/
          ptr = new_PAT;
          while(ptr != NULL)
          {
            ptr = ptr->next;
            free(new_PAT);
            new_PAT = ptr;
          }
        }

        if( (new_PAT=(PAT_ptr)malloc(sizeof(struct PA_table))) == NULL )
```

```
          {puts("TS_PROGRAM_ASSOCIATION_SECTION(): malloc error"); exit(1);}
        new_PAT->next=NULL;

        PAT_version = header.bits.version_number;
      }

      /* current_next_indicator */
                                       fprintf(out,"|        current_next_indicator        =
%d\n",header.bits.current_next_indicator);
      if(header.bits.current_next_indicator) /* Current table */
      {
        if(PAT != new_PAT)
        {
          /*free PAT*/
          ptr = PAT;
          while(ptr != NULL)
          {
            ptr = ptr->next;
            free(PAT);
            PAT = ptr;
          }

          PAT = new_PAT; /* New table is the current one */
        }
      }
      else if(PAT == new_PAT)  PAT = NULL; /* Desactivate current table */

      /* section_number */
      section_number = PA_section.buffer[7];
      fprintf(out,"| section_number = %d\n", section_number);

      /* last_section_number */
      last_section_number = PA_section.buffer[8];
      fprintf(out,"| last_section_number = %d\n", last_section_number);

    }
    else /*Verifies if table element (4 bytes) is in buffer.*/
    if(PA_section.p > 8 && !(PA_section.p % 4))
    {
      if(PA_section.bytes_left) /*Read one element.*/
      {
        for(i=0; i<4; i++)  table.byte[i]=PA_section.buffer[PA_section.p-3+i];

        /* program_number */
        i=table.bits.program_number_15_8 *256 + table.bits.program_number_7_0;
        fprintf(out,"|\n| program_number = %d\n", i);

        if(detail > 1)
          fprintf(out,"| reserved = %d\n", table.bits.reserved);

        /* network_PID or program_map_PID */
        if(i)
          fprintf(out,"| program_map_PID = ");
        else
          fprintf(out,"| network_PID = ");
        j = table.bits.network_or_program_map_PID_12_8 * 256
          +table.bits.network_or_program_map_PID_7_0;
        fprintf(out,"0x%04X\n", j);

        /* Verifies if PID is in table */
        ptr = PA_PID(new_PAT, j, &k);

        if(ptr == NULL)
        {
          /*New linked list element*/
          if( (ptr=(PAT_ptr)malloc(sizeof(struct PA_table))) == NULL )
            {puts("TS_PROGRAM_ASSOCIATION_SECTION(): malloc error"); exit(1);}

          /*Insert new element in table. At the list start, just after the head.*/
          ptr->next = new_PAT->next;
          new_PAT->next = ptr;
        }
        ptr->program_number = i;
        ptr->PID = j;
      }
      else /*Section END.*/
      {
        /* Read CRC_32.*/
        CRC_32 = (double)PA_section.buffer[PA_section.p-3] * 16777216 /* 2^24 */
               +(double)PA_section.buffer[PA_section.p-2] * 65536    /* 2^16 */
               +(double)PA_section.buffer[PA_section.p-1] * 256      /* 2^8  */
               +(double)PA_section.buffer[PA_section.p];
        fprintf(out,"|\n| CRC_32 = %g\n", CRC_32);

        /*! Here one would VERIFY THE CRC over the buffer*/

        /*Stuffing bytes*/
        if((n_bytes - 1) < (pointer_field +1))
        {
          /*One section ends and another starts.*/

          fprintf(out,"|\n| %d bytes left - ",pointer_field +1 - n_bytes);
          for(stuffing_bytes=0, i=0; i < pointer_field +1 - n_bytes; i++)
            if(getc(in) == 0xFF) stuffing_bytes++;
          fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

          n_bytes = pointer_field + 1;
          PA_section.p = 0; /*Byte 1: PA_section.p = 1*/
        }
        else  /*The section ended and rest of the packet is garbage.*/
        {
          fprintf(out,"|\n| %d bytes left - ",188 - n_bytes - *bytes_read);
          for(stuffing_bytes=0, i=0; i < 188 - n_bytes - *bytes_read; i++)
            if(getc(in) == 0xFF) stuffing_bytes++;
          fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

          n_bytes = 188 - *bytes_read;
        }
      }
    }
  }

  *bytes_read = 188;
}


void
TS_CA_SECTION(int unit_start,int *bytes_read,FILE *in,FILE *out)
/***********************************
 NOT TESTED !!!          (adapted from PA_section)

 pointer_field = 1 optional byte depending on 'unit_start'.

 table_id = 1 byte

 section_syntax_indicator = 1 bit \
 no_name = 1 bit                   \
 reserved1 = 2 bits                 \
 section_length = 12 bits            > 5 bytes.
 reserved2 = 18 bits                /
 version_number = 5 bits           /
 current_next_indicator = 1 bit   /

 section_number = 1 byte;

 last_section_number = 1 byte;

 N decritores of variable length;

 CRC_32 = 4 bytes;

 Uses a buffer for the section, defined in a global struct. This way one can
 reconstruct the section from packet to packet and do the final CRC.
 It's assumed that in a section there's at least 1 descriptor.

 Linked list with CA_table still not implemented !

 ************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned section_length_11_8      :  4;
    unsigned reserved1                :  2;
    unsigned no_name                  :  1;
    unsigned section_syntax_indicator :  1;

    unsigned section_length_7_0       :  8;

    unsigned reserved2                :  8;

    unsigned reserved3                :  8;

    unsigned current_next_indicator   :  1;
    unsigned version_number           :  5;
    unsigned reserved4                :  2;

#else
    unsigned section_syntax_indicator :  1;
    unsigned no_name                  :  1;
    unsigned reserved1                :  2;
    unsigned section_length_11_8      :  4;

    unsigned section_length_7_0       :  8;

    unsigned reserved2                :  8;

    unsigned reserved3                :  8;

    unsigned reserved4                :  2;
    unsigned version_number           :  5;
    unsigned current_next_indicator   :  1;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[5];
  } header;


  int     pointer_field,
          table_id,
          section_number,
          last_section_number,
          stuffing_bytes;
  double CRC_32;

  /* struct {} CA_section; GLOBAL VAR.! */


  char str[80];
  register int i,
               n_bytes = 0; /* counter of CA_section bytes */
  static int   needed_bytes;/*Needed bytes to read descriptor info from buffer.*/


  fprintf(out,"|CONDITIONAL ACCESS SECTION*******************\n");

  if( unit_start )
  {
    /*Read pointer_field.*/
    pointer_field = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in pointer_field");
      exit(1);
    }
    n_bytes = 1;
    fprintf(out,"| pointer_field = %d bytes\n", pointer_field);
    if(pointer_field + *bytes_read + 1 > 188)
    {
      i = 188 - *bytes_read -n_bytes;
      sprintf(str,"Invalid length: %d bytes ignored",i);
      Warn(out, str);
      fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
      *bytes_read = 188;
      return;
    }

    if(pointer_field == 0) CA_section.p = 0; /*Byte 1: CA_section.p = 1*/
    /*! Maybe should verify if previous section was completely read.*/
  }

  while(*bytes_read + n_bytes < 188) /*Verifies end of packet.*/
  {
    /*Read a section byte to a buffer.*/
    CA_section.buffer[CA_section.p+1] = (char)getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in conditional_access_section");
      exit(1);
    }
    n_bytes++;
    CA_section.p++;
    CA_section.bytes_left--;
    needed_bytes--;

    /*Section and first descriptor header is in buffer.*/
    if(CA_section.p == 10)
    {
      /* table_id */
      table_id = CA_section.buffer[1];
      fprintf(out,"| table_id = %d\n", table_id);

      /*Read section header.*/
      for(i=0; i<5; i++)  header.byte[i] = CA_section.buffer[i+2];

      /* section_syntax_indicator */
                                       fprintf(out,"|      section_syntax_indicator      =
%d\n",header.bits.section_syntax_indicator);

      if(  table_id != 1                 ||
          !header.bits.section_syntax_indicator ||
           header.bits.no_name               )
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid CA_section header: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }

      if(detail < 1)
        fprintf(out,"| reserved = %d\n", header.bits.reserved1);

      /*Read section length.*/
      /*Initialization of 'CA_section.bytes_left'.*/
      CA_section.bytes_left = header.bits.section_length_11_8 * 256
                            +header.bits.section_length_7_0;
      fprintf(out,"| section_length = %d\n", CA_section.bytes_left);
      if(CA_section.bytes_left > 1021) /*See buffer size*/
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid length: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }
      CA_section.bytes_left -= 7;

      if(detail < 1)
        fprintf(out,"| reserved = %d %d %d\n", header.bits.reserved2,
                                            header.bits.reserved3,
                                            header.bits.reserved4 );

      /* version_number */
      fprintf(out,"| version_number = %d\n", header.bits.version_number);

      /* current_next_indicator */
```

```
                                    fprintf(out,"|            current_next_indicator        =
%d\n",header.bits.current_next_indicator);

        /* section_number */
        section_number = CA_section.buffer[7];
        fprintf(out,"| section_number = %d\n", section_number);

        /* last_section_number */
        last_section_number = CA_section.buffer[8];
        fprintf(out,"| last_section_number = %d\n", last_section_number);


        /*First descriptor header.*/

        /*Needed bytes in buffer are the descriptor size and, if there is more,
          next's header (giving the descriptor size).*/
        if(CA_section.buffer[10] < (CA_section.bytes_left - 4))
          /* Size of descriptor plus next's header */
          needed_bytes = CA_section.buffer[10] + 2;
        else
          /* Size of descriptor plus CRC */
          needed_bytes = CA_section.buffer[10] + 4;

        if(needed_bytes > CA_section.bytes_left)
        {
          fprintf(out,"|\n| descriptor_tag = %d\n", CA_section.buffer[9]);
          fprintf(out,"| descriptor_length = %d\n", CA_section.buffer[10]);
          i = 188 - *bytes_read -n_bytes;
          sprintf(str,"Invalid descriptor length: %d bytes ignored",i);
          Warn(out, str);
          fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
          *bytes_read = 188;
          return;
        }

        /*Start of last descriptor kept in buffer.*/
        CA_section.descriptor_start = 9;
    }
        else /*Checks if there's enough info in buffer to read descriptor and what
follows.*/
      if(CA_section.p > 8 && !needed_bytes)
      {
        /*Read descriptor.*/
        DESCRIPTOR(&CA_section, out);

        if(CA_section.bytes_left) /*There's more descriptors.*/
        {
          i = CA_section.p;

          /*Next descriptor header.*/

          /*Needed bytes in buffer are the descriptor size and, if there is more,
            next's header (giving the descriptor size).*/
          if(CA_section.buffer[i] < (CA_section.bytes_left - 4))
            /* Size of descriptor plus next's header */
            needed_bytes = CA_section.buffer[i] + 2;
          else
            /* Size of descriptor plus CRC */
            needed_bytes = CA_section.buffer[i] + 4;

          if(needed_bytes > CA_section.bytes_left)
          {
            fprintf(out,"|\n| descriptor_tag = %d\n", CA_section.buffer[i-1]);
            fprintf(out,"| descriptor_length = %d\n", CA_section.buffer[i]);
             sprintf(str,"Invalid descriptor length: %d bytes ignored",188-*bytes_read-
n_bytes);
            Warn(out, str);
            fseek(in, 188 - *bytes_read -n_bytes, SEEK_CUR); /*Ignores rest of packet*/
            *bytes_read = 188;
            return;
          }

          /*Start of last descriptor kept in buffer.*/
          CA_section.descriptor_start = i-1;
        }
        else /*Section END.*/
        {
```

```
        /* Read CRC_32.*/
        CRC_32 = (double)CA_section.buffer[CA_section.p-3] * 16777216 /* 2^24 */
               +(double)CA_section.buffer[CA_section.p-2] * 65536    /* 2^16 */
               +(double)CA_section.buffer[CA_section.p-1] * 256      /* 2^8  */
               +(double)CA_section.buffer[CA_section.p];
        fprintf(out,"|\n| CRC_32 = %g\n", CRC_32);

        /*! Here one would VERIFY THE CRC over the buffer*/

        /*Stuffing bytes*/
        if((n_bytes - 1) < (pointer_field +1))
        {
          /*One section ends and another one starts.*/

          fprintf(out,"|\n| %d bytes left - ",pointer_field +1 - n_bytes);
          for(stuffing_bytes=0, i=0; i < pointer_field +1 - n_bytes; i++)
            if(getc(in) == 0xFF) stuffing_bytes++;
          fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

          n_bytes = pointer_field + 1;
          CA_section.p = 0; /*Byte 1: CA_section.p = 1*/
        }
        else  /*The section ended and rest of the packet is garbage.*/
        {
          fprintf(out,"|\n| %d bytes left - ",188 - n_bytes - *bytes_read);
          for(stuffing_bytes=0, i=0; i < 188 - n_bytes - *bytes_read; i++)
            if(getc(in) == 0xFF) stuffing_bytes++;
          fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

          n_bytes = 188 - *bytes_read;
        }
      }
    }
  }

  *bytes_read = 188;
}


void
TS_PROGRAM_MAP_SECTION(int unit_start,int *bytes_read,FILE *in,FILE *out)
/*****************************************
 pointer_field = 1 optional byte depending on 'unit_start'.

 table_id = 1 byte

 section_syntax_indicator = 1 bit \
 no_name = 1 bit                   \
 reserved1 = 2 bits                 \
 section_length = 12 bits            \
 program_number = 16 bits             \
 reserved2 = 2 bits                    \
 version_number = 5 bits                > 11 bytes.
 current_next_indicator = 1 bit        /
 section_number = 8 bits              /
 last_section_number = 8 bits        /
 reserved3 = 3 bits                 /
 PCR_PID = 13 bits                 /
 reserved4 = 4 bits               /
 program_info_length = 12 bits   /

 N descriptors of varying length;

 stream_type = 8 bits      \
 reserved = 3 bits          \
 elementary_PID = 13 bits    > N * 5 bytes.
 reserved = 4 bits          /
 ES_info_length = 12 bits  /

 N descriptors of varying length;

 CRC_32 = 4 bytes;

 Uses a buffer for the section, defined in a global struct. This way one can
 reconstruct the section from packet to packet and do the final CRC.

 Depending on table_id private data can be transported.
 TO DO ! (see table 2-35, pg55 - Stream Type Assignments: 0x05)
 ****************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned section_length_11_8      :  4;
    unsigned reserved1                :  2;
    unsigned no_name                  :  1;
    unsigned section_syntax_indicator :  1;

    unsigned section_length_7_0       :  8;

    unsigned program_number_15_8      :  8;

    unsigned program_number_7_0       :  8;

    unsigned current_next_indicator   :  1;
    unsigned version_number           :  5;
    unsigned reserved2                :  2;

    unsigned section_number           :  8;

    unsigned last_section_number      :  8;

    unsigned PCR_PID_12_8             :  5;
    unsigned reserved3                :  3;

    unsigned PCR_PID_7_0              :  8;

    unsigned program_info_length_11_8 :  4;
    unsigned reserved4                :  4;

    unsigned program_info_length_7_0  :  8;

#else
    unsigned section_syntax_indicator :  1;
    unsigned no_name                  :  1;
    unsigned reserved1                :  2;
    unsigned section_length_11_8      :  4;

    unsigned section_length_7_0       :  8;

    unsigned program_number_15_8      :  8;

    unsigned program_number_7_0       :  8;

    unsigned reserved2                :  2;
    unsigned version_number           :  5;
    unsigned current_next_indicator   :  1;

    unsigned section_number           :  8;

    unsigned last_section_number      :  8;

    unsigned reserved3                :  3;
    unsigned PCR_PID_12_8             :  5;

    unsigned PCR_PID_7_0              :  8;

    unsigned reserved4                :  4;
    unsigned program_info_length_11_8 :  4;

    unsigned program_info_length_7_0  :  8;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[11];
  } header;
  struct ES_bits
  {
    unsigned  stream_type         :  8;

#ifdef DOS
    unsigned elementary_PID_12_8 :  5;
    unsigned reserved1           :  3;

    unsigned elementary_PID_7_0  :  8;

    unsigned ES_info_length_11_8 :  4;
    unsigned reserved2           :  4;
#else
    unsigned reserved1           :  3;
    unsigned elementary_PID_12_8 :  5;

    unsigned elementary_PID_7_0  :  8;

    unsigned reserved2           :  4;
    unsigned ES_info_length_11_8 :  4;
#endif

    unsigned ES_info_length_7_0  :  8;
  };

  union
  {
    struct ES_bits bits;
    char byte[5];
  } ES;

  int         pointer_field,
              table_id,
              stuffing_bytes;
  static int  field,      /* control of aplicable syntax */
              program_info_length,
              ES_info_length;
  double      CRC_32;

  /* struct {} PM_section;
     PMT_ptr new_PMT, PMT;
     int PMT_version;        GLOBAL VAR.! */

  char str[80];
  register int i,
               n_bytes = 0; /* counter of PM_section bytes */
  static int   descriptor_needed_bytes,/*Needed bytes to read descriptor info
                                         from buffer.*/
               info_start; /*Buffer pointer.*/
  PMT_ptr      ptr;  /*Auxiliary: free of PMT and new_PMT; insertion of new elements
                in table*/


  fprintf(out,"|PROGRAM MAP SECTION****************************\n");

  if( unit_start )
  {
    /*Read pointer_field.*/
    pointer_field = getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in pointer_field");
      exit(1);
    }
    n_bytes = 1;
    fprintf(out,"| pointer_field = %d bytes\n", pointer_field);
    if(pointer_field + *bytes_read + 1 > 188)
    {
      i = 188 - *bytes_read -n_bytes;
      sprintf(str,"Invalid length: %d bytes ignored",i);
      Warn(out, str);
      fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
      *bytes_read = 188;
      return;
    }

    if(pointer_field == 0) PM_section.p = 0; /*Byte 1: PM_section.p = 1*/
    /*! Maybe should verify if previous section was completely read.*/
  }

  while(*bytes_read + n_bytes < 188) /*Verifies end of packet.*/
  {
    /*Read a section byte to a buffer.*/
    PM_section.buffer[PM_section.p+1] = (char)getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in program_map_section");
      exit(1);
    }
    n_bytes++;
    PM_section.p++;
    PM_section.bytes_left--;
    descriptor_needed_bytes--;

    if(PM_section.p == 1) /* table_id is in buffer */
    {
      /* table_id */
      table_id = PM_section.buffer[1];
      fprintf(out,"| table_id                    = %d", table_id);
      if(table_id != 2)
      {
        if(table_id == 0xFF)
          fprintf(out," -> forbidden\n");
        if(table_id >= 0x40 && table_id <= 0xFE)
          fprintf(out," -> User private\n");     /* TS_PRIVATE_SECTION(); !!! */
        if(table_id >= 0x03 && table_id <= 0x3F)
          fprintf(out," -> ISO/IEC 13818 reserved\n");

        i = 188 - *bytes_read -n_bytes;
        fprintf(out,"| (%d bytes ignored)\n",i);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }
    }
    else
    if(PM_section.p == 12) /*Header is in buffer.*/
    {
      /*Read header.*/
      for(i=0; i<11; i++)  header.byte[i] = PM_section.buffer[i+2];

      /* section_syntax_indicator */
                                      fprintf(out,"\n|      section_syntax_indicator      =
%d\n",header.bits.section_syntax_indicator);

      if(  !header.bits.section_syntax_indicator ||
           header.bits.no_name                       )
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid PM_section header: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }

      if(detail > 1)
        fprintf(out,"| reserved = %d\n", header.bits.reserved1);

      /*Read section length.*/
      /*Initialization of 'PM_section.bytes_left'.*/
      PM_section.bytes_left = header.bits.section_length_11_8 * 256
                            +header.bits.section_length_7_0;
      fprintf(out,"| section_length = %d\n", PM_section.bytes_left);
      if(PM_section.bytes_left > 1021) /*See buffer size*/
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid length: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }
      PM_section.bytes_left -= 9;
```

```
/* program_number */
fprintf(out,"| program_number = %d\n",
                              header.bits.program_number_15_8 * 256
                             +header.bits.program_number_7_0       );

if(detail < 1)
  fprintf(out,"| reserved = %d\n", header.bits.reserved2);

/* version_number */
fprintf(out,"| version_number          = %d\n", header.bits.version_number);
i = (int)header.bits.version_number - PMT_version;
if(i != 0) /* Table version management */
{
  if(i != 1 && i != -31)
  {
    i = 188 - *bytes_read -n_bytes;
    sprintf(str,"version_number not in sequence: %d bytes ignored",i);
    Warn(out, str);
    fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
    *bytes_read = 188;
    return;
  }

  if(PMT != new_PMT)
  {
    Warn(out,"Previous version not used");
    /*free new_PMT*/
    ptr = new_PMT;
    while(ptr != NULL)
    {
      ptr = ptr->next;
      free(new_PMT);
      new_PMT = ptr;
    }
  }

  if( (new_PMT=(PMT_ptr)malloc(sizeof(struct PM_table))) == NULL )
    {puts("TS_PROGRAM_MAP_SECTION(): malloc error"); exit(1);}
  new_PMT->next=NULL;

  PMT_version = header.bits.version_number;
}

/* current_next_indicator */
                                   fprintf(out,"|      current_next_indicator       =
%d\n",header.bits.current_next_indicator);
if(header.bits.current_next_indicator) /* Current table */
{
  if(PMT != new_PMT)
  {
    /*free PMT*/
    ptr = PMT;
    while(ptr != NULL)
    {
      ptr = ptr->next;
      free(PMT);
      PMT = ptr;
    }

    PMT = new_PMT; /* New table is the current one */
  }
}
else if(PMT == new_PMT)  PMT = NULL; /* Desactivate current table */

/* section_number */
fprintf(out,"| section_number          = %d\n", header.bits.section_number);
if(header.bits.section_number)
  Warn(out,"Invalid section_number");

/* last_section_number */
fprintf(out,"| last_section_number     = %d\n",header.bits.last_section_number);
if(header.bits.last_section_number)
  Warn(out,"Invalid last_section_number");

if(detail < 1)
  fprintf(out,"| reserved = %d\n", header.bits.reserved3);
```

```
      /* PCR_PID */
      fprintf(out,"| PCR_PID = 0x%04X\n", header.bits.PCR_PID_12_8 * 256
                                         +header.bits.PCR_PID_7_0      );

      if(detail > 1)
        fprintf(out,"| reserved = %d\n", header.bits.reserved4);

      /* program_info_length */
      program_info_length = header.bits.program_info_length_11_8 * 256
                           +header.bits.program_info_length_7_0;
      fprintf(out,"| program_info_length = %d\n", program_info_length);
/*IMPROVE ROBUSTNESS !*/
      if(program_info_length+4 > PM_section.bytes_left)
      {
        i = 188 - *bytes_read -n_bytes;
        sprintf(str,"Invalid length: %d bytes ignored",i);
        Warn(out, str);
        fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
        *bytes_read = 188;
        return;
      }

      if(program_info_length)  field = 1; /* Program descriptors */
      else
      {
        field = 2; /* ES info. */
        info_start = PM_section.p;
      }
    }
    else
    if(PM_section.p > 12) /* flag is defined */
    {
      switch(field)
      {
       case 1: /* Program descriptors */

        if(PM_section.p == 14) /*First descriptor header.*/
        {
          program_info_length -= 2;

          /*Needed bytes in buffer are the descriptor size and, if there is more,
            next's header (giving the descriptor size).*/
          descriptor_needed_bytes = PM_section.buffer[14];
          if(PM_section.buffer[14] < program_info_length)
            /* Size of descriptor plus next's header */
            descriptor_needed_bytes += 2;

          if(descriptor_needed_bytes > program_info_length)
          {
            fprintf(out,"|\n| descriptor_tag = %d\n", PM_section.buffer[13]);
            fprintf(out,"| descriptor_length = %d\n", PM_section.buffer[14]);
            i = 188 - *bytes_read -n_bytes;
            sprintf(str,"Invalid descriptor length: %d bytes ignored",i);
            Warn(out, str);
            fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
            *bytes_read = 188;
            return;
          }

          /*Start of last descriptor kept in buffer.*/
          PM_section.descriptor_start = 13;
        }
        else
        if(PM_section.p > 14 && !descriptor_needed_bytes)
        {
          program_info_length-= PM_section.buffer[PM_section.descriptor_start+1];

          /*Read descriptor.*/
          DESCRIPTOR(&PM_section, out);

          if(program_info_length) /*There's more descriptors.*/
          {
            program_info_length -= 2;

            i = PM_section.p;
          /*Next descriptor header.*/

          /*Needed bytes in buffer are the descriptor size and, if there is more,
            next's header (giving the descriptor size).*/
          descriptor_needed_bytes = PM_section.buffer[i];
          if(PM_section.buffer[i] < program_info_length)
            /* Size of descriptor plus next's header */
            descriptor_needed_bytes += 2;

          if(descriptor_needed_bytes > program_info_length)
          {
            fprintf(out,"|\n| descriptor_tag = %d\n", PM_section.buffer[i-1]);
            fprintf(out,"| descriptor_length = %d\n", PM_section.buffer[i]);
            i = 188 - *bytes_read -n_bytes;
            sprintf(str,"Invalid descriptor length: %d bytes ignored",i);
            Warn(out, str);
            fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
            *bytes_read = 188;
            return;
          }

          /*Start of last descriptor kept in buffer.*/
          PM_section.descriptor_start = i-1;
        }
        else
        {
          field = 2; /* ES info. */
          info_start = PM_section.p;
        }
      }
      break;

     case 2: /* ES info. */

      if(PM_section.bytes_left < 4)  field = 4; /* CRC and end */
      else
      if(PM_section.p == (info_start + 5)) /* ES header in buffer */
      {
        /*Read ES header.*/
        for(i=0; i<5; i++)  ES.byte[i]=PM_section.buffer[PM_section.p-4+i];

        /* stream_type */
        fprintf(out,"|\n| stream_type = %d   -> ", ES.bits.stream_type);
        switch(ES.bits.stream_type)
        {
         case 0x00:
                   fprintf(out,"ITU-T | ISO/IEC Reserved\n");
                   break;
         case 0x01:
                   fprintf(out,"ISO/IEC 11172 Video\n");
                   break;
         case 0x02:
                   fprintf(out,"ITU-T Rec.H262 | ISO/IEC 13818-2 Video\n");
                   break;
         case 0x03:
                   fprintf(out,"ISO/IEC 11172 Audio\n");
                   break;
         case 0x04:
                   fprintf(out,"ISO/IEC 13818-3 Audio\n");
                   break;
         case 0x05:
                   fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 private_section\n");
                   break;
         case 0x06:
                        fprintf(out,"ITU-T  Rec.H222.0|ISO/IEC  13818-1  PES  packets
containing private data\n");
                   break;
         case 0x07:
                   fprintf(out,"ISO/IEC 13522 MHEG\n");
                   break;
         case 0x08:
                   fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 DSM CC\n");
                   break;
         case 0x09:
                        fprintf(out,"ITU-T   Rec.H222.0|ISO/IEC   13818-1/11172-1
Auxiliary\n");
                   break;
            default   :
                      if(ES.bits.stream_type>=0x0A && ES.bits.stream_type<=0x7F)
                        fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 Reserved\n");
                      else
                      if(ES.bits.stream_type>=0x80 && ES.bits.stream_type<=0xFF)
                        fprintf(out,"User Private\n");
          }

          if(detail > 1)
            fprintf(out,"| reserved = %d\n", ES.bits.reserved1);

          /* elementary_PID */
          i = ES.bits.elementary_PID_12_8 * 256
             +ES.bits.elementary_PID_7_0;
          fprintf(out,"| elementary_PID = 0x%04X\n", i);

          /* Verifies if PID is in table */
          ptr = Elementary_PID(new_PMT, i);

          if(ptr == NULL)
          {
            /*New linked list element*/
            if( (ptr=(PMT_ptr)malloc(sizeof(struct PM_table))) == NULL )
              {puts("TS_PROGRAM_MAP_SECTION(): malloc error"); exit(1);}

            /*Insert new element in table. At the list start, just after the head.*/
            ptr->next = new_PMT->next;
            new_PMT->next = ptr;

            ptr->elementary_PID = i;
                        ptr->continuity_counter    =    -1;/*Impossible    value.    See
TS_TRANSPORT_PACKET()*/
            ptr->old_stream_removed = NO;
            ptr->field = 0; /*PES_PACKET() ignored if there's no unit_start*/
          }

          /*ptr->stream_type = ES.bits.stream_type; NOT USED!*/

          if(detail > 1)
            fprintf(out,"| reserved = %d\n", ES.bits.reserved2);

          /* ES_info_length */
          ES_info_length = ES.bits.ES_info_length_11_8 * 256
                          +ES.bits.ES_info_length_7_0;
          fprintf(out,"| ES_info_length = %d\n", ES_info_length);
/*IMPROVE ROBUSTNESS !*/
          if(ES_info_length+4 > PM_section.bytes_left)
          {
            i = 188 - *bytes_read -n_bytes;
            sprintf(str,"Invalid length: %d bytes ignored",i);
            Warn(out, str);
            fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
            *bytes_read = 188;
            return;
          }

          if(ES_info_length)  field = 3; /* ES descriptors */

          info_start = PM_section.p;
        }
        break;

        case 3: /* ES descriptors */

        if(PM_section.p == info_start+2) /*First descriptor header.*/
        {
          ES_info_length -= 2;

          /*Needed bytes in buffer are the descriptor size and, if there is more,
            next's header (giving the descriptor size).*/
          descriptor_needed_bytes = PM_section.buffer[PM_section.p];
          if(PM_section.buffer[PM_section.p] < ES_info_length)
            /* Size of descriptor plus next's header */
            descriptor_needed_bytes += 2;

          if(descriptor_needed_bytes > ES_info_length)
          {
                fprintf(out,"|\n| descriptor_tag = %d\n", PM_section.buffer[13]);
                fprintf(out,"| descriptor_length = %d\n", PM_section.buffer[14]);
                i = 188 - *bytes_read -n_bytes;
                sprintf(str,"Invalid descriptor length: %d bytes ignored",i);
                Warn(out, str);
                fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
                *bytes_read = 188;
                return;
              }

              /*Start of last descriptor kept in buffer.*/
              PM_section.descriptor_start = PM_section.p - 1;
            }
            else
            if(PM_section.p>info_start+2 && !descriptor_needed_bytes)
            {
              ES_info_length-= PM_section.buffer[PM_section.descriptor_start+1];

              /*Read descriptor.*/
              DESCRIPTOR(&PM_section, out);

              if(ES_info_length) /*There's more descriptors.*/
              {
                ES_info_length -= 2;

                i = PM_section.p;

                /*Next descriptor header.*/

                /*Needed bytes in buffer are the descriptor size and, if there is more,
                  next's header (giving the descriptor size).*/
                descriptor_needed_bytes = PM_section.buffer[i];
                if(PM_section.buffer[i] < ES_info_length)
                  /* Size of descriptor plus next's header */
                  descriptor_needed_bytes += 2;

                if(descriptor_needed_bytes > ES_info_length)
                {
                  fprintf(out,"|\n| descriptor_tag = %d\n", PM_section.buffer[i-1]);
                  fprintf(out,"| descriptor_length = %d\n", PM_section.buffer[i]);
                     sprintf(str,"Invalid descriptor length: %d bytes ignored",188 -
*bytes_read -n_bytes);
                  Warn(out, str);
                     fseek(in, 188 - *bytes_read -n_bytes, SEEK_CUR); /*Ignores rest of
packet*/
                  *bytes_read = 188;
                  return;
                }

                /*Start of last descriptor kept in buffer.*/
                PM_section.descriptor_start = i-1;
              }
              else
              {
                field = 2; /* ES info. (verifies again if there's more ES)*/
                info_start = PM_section.p;
              }
            }
            break;

            case 4: /* CRC and section END */

            if(PM_section.p == info_start+4) /* CRC */
            {
              if( PM_section.bytes_left )
              {
                i = 188 - *bytes_read -n_bytes;
                sprintf(str,"Invalid CRC length: %d bytes ignored",i);
                Warn(out, str);
                fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
                *bytes_read = 188;
                return;
              }

              /* Read CRC_32.*/
              CRC_32 = (double)PM_section.buffer[PM_section.p-3] * 16777216 /* 2^24 */
                      +(double)PM_section.buffer[PM_section.p-2] * 65536    /* 2^16 */
              +(double)PM_section.buffer[PM_section.p-1] * 256      /* 2^8  */
              +(double)PM_section.buffer[PM_section.p];
          fprintf(out,"|\n| CRC_32 = %g\n", CRC_32);

          /*! Here one would VERIFY THE CRC over the buffer*/

          /*Stuffing bytes*/
          if((n_bytes - 1) < (pointer_field +1))
          {
            /*One section ends and another one starts.*/

            fprintf(out,"|\n| %d bytes left - ",pointer_field +1 - n_bytes);
            for(stuffing_bytes=0, i=0; i < pointer_field +1 - n_bytes; i++)
              if(getc(in) == 0xFF) stuffing_bytes++;
            fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

            n_bytes = pointer_field + 1;
            PM_section.p = 0; /*Byte 1: PM_section.p = 1*/
          }
          else  /*The section ended and rest of the packet is garbage.*/
          {
            fprintf(out,"|\n| %d bytes left - ",188 - n_bytes - *bytes_read);
            for(stuffing_bytes=0, i=0; i < 188 - n_bytes - *bytes_read; i++)
              if(getc(in) == 0xFF) stuffing_bytes++;
            fprintf(out,"%d stuffing_bytes (0xFF) detected\n", stuffing_bytes);

            n_bytes = 188 - *bytes_read;
          }
        }
        break;
      }
    }
  }

  *bytes_read = 188;
}


void
DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/***************************************
 With the descriptor_tag, indexed by stream_buffer->descriptor_start, present
 in the buffer (see global struct) which contains the descriptor that will be
 parsed, this routine does the routing to the appropriate sub-routine.
 ***************************************/
{
  register int i;

  i = stream_buffer->buffer[stream_buffer->descriptor_start];
  fprintf(out,"|\n| descriptor_tag     = %d: ",i);
  switch(i)
  {
    case  2: /* video_stream_descriptor */
             fprintf(out,"video_stream_descriptor\n");
             VIDEO_STREAM_DESCRIPTOR(stream_buffer, out);
             break;
    case  3: /* audio_stream_descriptor */
             fprintf(out,"audio_stream_descriptor\n");
             AUDIO_STREAM_DESCRIPTOR(stream_buffer, out);
             break;
    case  4: /* hierarchy_descriptor */
             fprintf(out,"hierarchy_descriptor\n");
             _DESCRIPTOR(stream_buffer, out);
             break;
    case  5: /* registration_descriptor */
             fprintf(out,"registration_descriptor\n");
             _DESCRIPTOR(stream_buffer, out);
             break;
    case  6: /* data_stream_alignment_descriptor */
             fprintf(out,"data_stream_alignment_descriptor\n");
             _DESCRIPTOR(stream_buffer, out);
             break;
    case  7: /* target_background_grid_descriptor */
             fprintf(out,"target_background_grid_descriptor\n");
             _DESCRIPTOR(stream_buffer, out);
             break;
    case  8: /* video_window_descriptor */
```

```
                fprintf(out,"video_window_descriptor\n");
                _DESCRIPTOR(stream_buffer, out);
                break;
      case  9: /* CA_descriptor */
                fprintf(out,"CA_descriptor\n");
                _DESCRIPTOR(stream_buffer, out);
                break;
      case 10: /* ISO_639_language_descriptor */
                fprintf(out,"ISO_639_language_descriptor\n");
                _DESCRIPTOR(stream_buffer, out);
                break;
      case 11: /* system_clock_descriptor */
                fprintf(out,"system_clock_descriptor\n");
                SYSTEM_CLOCK_DESCRIPTOR(stream_buffer, out);
                break;
      case 12: /* multiplex_buffer_utilization_descriptor */
                fprintf(out,"multiplex_buffer_utilization_descriptor\n");
                MULTIPLEX_BUFFER_UTILIZATION_DESCRIPTOR(stream_buffer, out);
                break;
      case 13: /* copyright_descriptor */
                fprintf(out,"copyright_descriptor\n");
                _DESCRIPTOR(stream_buffer, out);
                break;
      case 14: /* maximum_bitrate_descriptor */
                fprintf(out,"maximum_bitrate_descriptor\n");
                MAXIMUM_BITRATE_DESCRIPTOR(stream_buffer, out);
                break;
      case 15: /* private_data_indicator_descriptor */
                fprintf(out,"private_data_indicator_descriptor\n");
                _DESCRIPTOR(stream_buffer, out);
                break;
      default:
                if(i>=16 && i<=63 || !i || i==1) fprintf(out,"Reserved\n");
                else
                if(i>=64 && i<=255) fprintf(out,"User Private\n");
                else
                fprintf(out,"ERROR!\n");
   }
}


void
VIDEO_STREAM_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 Reads the descriptor info contained in the buffer (global struct).
 *************************************/
{
  register int j, i;

  /* descriptor_length */
  j = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n", j);
  if(j !=1 && j != 3)
  {
    Warn(out,"Invalid descriptor length");
    return;
  }

  /* Read flags */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 2];

  /* multiple_frame_rate_flag */
  fprintf(out,"| multiple_frame_rate_flag    = %d\n",(i&128) >> 7);/*mask: 1000 0000*/

  /* frame_rate_code */
  fprintf(out,"| frame_rate_code             = %d\n",(i&120) >> 3);/*mask: 0111 1000*/

  /* MPEG_2_flag */
  fprintf(out,"| MPEG_2_flag                 = %d\n",(i&4) >> 2);/*mask: 0000 0100*/

  /* constrained_parameter_flag */
  fprintf(out,"| constrained_parameter_flag = %d\n",(i&2) >> 1);/*mask: 0000 0010*/

  /* still_picture_flag */
  fprintf(out,"| still_picture_flag          = %d\n", i&1);/*mask: 0000 0001*/

  if(i&4) /* MPEG_2_flag */
  {
    if(j != 3)
    {
      Warn(out, "MPEG_2_flag: invalid descriptor length");
      return;
    }

    /* Read 1 byte */
    i = stream_buffer->buffer[stream_buffer->descriptor_start + 3];

    /* profile_and_level_indication */
    fprintf(out,"| profile_and_level_indication = %d\n", i);

    /* Read 3rd byte */
    i = stream_buffer->buffer[stream_buffer->descriptor_start + 4];

    /* chroma_format */
     fprintf(out,"| chroma_format                          = %d\n",(i&192) >> 6);/*mask: 1100
0000*/

    /* frame_rate_extension_flag */
      fprintf(out,"| frame_rate_extension_flag      = %d\n",(i&32) >> 5);/*mask: 0010
0000*/

    if(detail < 1)
      fprintf(out,"| reserved = %d\n", i&31);/*mask: 0001 1111*/
  }
}


void
AUDIO_STREAM_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 Reads the descriptor info contained in the buffer (global struct).
 *************************************/
{
  register int i;

  /* descriptor_length */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n", i);
  if(i != 1)
  {
    Warn(out,"Invalid descriptor length");
    return;
  }

  /* Read byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 2];

  /* free_format_flag */
  fprintf(out,"| free_format_flag = %d\n",(i&128) >> 7);/*mask: 1000 0000*/

  /* ID */
  fprintf(out,"| ID                = %d\n",(i&64) >> 6);/*mask: 0100 0000*/

  /* layer */
  fprintf(out,"| layer             = %d\n",(i&48) >> 4);/*mask: 0011 0000*/

  if(detail < 1)
    fprintf(out,"| reserved = %d\n", i&15);/*mask: 0000 1111*/
}


void
SYSTEM_CLOCK_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 Reads the descriptor info contained in the buffer (global struct).
 *************************************/
{
  register int i;

  /* descriptor_length */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n", i);
  if(i != 2)
  {
    Warn(out,"Invalid descriptor length");
    return;
  }

  /* Read 1 byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 2];

  /* external_clock_reference_indicator */
  fprintf(out,"| external_clock_reference_indicator = %d\n",(i&128) >> 7);/*mask: 1000
0000*/

  if(detail < 1)
    fprintf(out,"| reserved = %d\n",(i&64) >> 6);/*mask: 0100 0000*/

  /* clock_accuracy_integer */
  fprintf(out,"| clock_accuracy_integer  = %d\n", i&63);/*mask: 0011 1111*/

  /* Read 1 byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 3];

  /* clock_accuracy_exponent */
  fprintf(out,"| clock_accuracy_exponent = %d\n",(i&224) >> 5);/*mask: 1110 0000*/

  if(detail < 1)
    fprintf(out,"| reserved = %d\n", i&31);/*mask: 0001 1111*/
}


void
MULTIPLEX_BUFFER_UTILIZATION_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 Reads the descriptor info contained in the buffer (global struct).
 *************************************/
{
  register int i;

  /* descriptor_length */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n", i);
  if(i != 3)
  {
    Warn(out, "Invalid descriptor length");
    return;
  }

  /* Read 1 byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 2];

  /* mdv_valid_flag */
  fprintf(out,"| mdv_valid_flag            = %d\n",(i&128) >> 7);/*mask: 1000 0000*/

  /* multiplex_delay_variation, read 1 more byte */
  i &= 127;     /*mask: 0111 1111*/
  i =  i * 256 /* 2^8 */
    + stream_buffer->buffer[stream_buffer->descriptor_start + 3];
  fprintf(out,"| multiplex_delay_variation = %d\n", i);

  /* Read 1 byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 4];

  /* multiplex_strategy */
  fprintf(out,"| multiplex_strategy        = %d\n",(i&224) >> 5);/*mask: 1110 0000*/

  if(detail < 1)
    fprintf(out,"| reserved = %d\n", i&31);/*mask: 0001 1111*/
}


void
MAXIMUM_BITRATE_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 Reads the descriptor info contained in the buffer (global struct).
 *************************************/
{
  register int i;
  double rate;
  /* descriptor_length */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n", i);
  if(i != 3)
  {
    Warn(out, "Invalid descriptor length");
    return;
  }

  /* Read 1 byte */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 2];

  if(detail > 1)
    fprintf(out,"| reserved = %d\n",(i&192) >> 6);/*mask: 1100 0000*/

  /* maximum_bitrate, read 2 more bytes */
  i &= 63; /*mask: 0011 1111*/
  rate = (double)i * 65536 /* 2^16 */
      + (double)stream_buffer->buffer[stream_buffer->descriptor_start + 3] *256 /* 2^8
*/
     + (double)stream_buffer->buffer[stream_buffer->descriptor_start + 4];
  fprintf(out,"| maximum_bitrate = %g (%g bytes/s)\n", rate, rate*50);
}


void
_DESCRIPTOR(struct buffer *stream_buffer, FILE *out)
/*************************************
 DUMMY
*************************************/
{
  register int i;

  /* descriptor_length */
  i = stream_buffer->buffer[stream_buffer->descriptor_start + 1];
  fprintf(out,"| descriptor_length = %d\n| (...)\n", i);
}


void
PES_PACKET (PMT_ptr PES, int unit_start, int *bytes_read, FILE *in, FILE *out)
/*************************************
 packet_start_code_prefix = 3 bytes

 stream_id = 1 byte

 PES_packet_length = 2 bytes

 PES header = 3 optional bytes

 PTS and DTS = 0 or 5 or 10 bytes

 ESCR = 6 optional bytes

 ES_rate = 3 optional bytes

 DSM_trick_mode = 1 optional byte

 adicional_copy_info = 1 optional byte

 PES_CRC = 2 optional bytes

 PES_extension flags = 1 optional byte

 PES_private_data = 16 optional bytes

 pack_header field = until 256 bytes

 program_packet_sequence_counter = 2 optional bytes

 P_STD_buffer = 2 optional bytes

 PES_extension_flag_2 = until 128 reserved bytes (optional)

 stuffing_byte (0xFF) = max. 32 bytes
```

```
  Due to the limited length of TPs, a PES_packet buffer (kept in PM_table) is
 used to save the PES_packet header so that it can be interpreted as soon as
 possible (and not only after having received all the information). In the
 future the data_bytes will also be saved in order to do the PES_CRC.
  Even though all flags are included the routine was made only for Transport
 streams.
  All marker_bits must be 1.

 *************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned original_or_copy           : 1;
    unsigned copyright                  : 1;
    unsigned data_alignment_indicator   : 1;
    unsigned PES_priority               : 1;
    unsigned PES_scrambling_control     : 2;
    unsigned noname                     : 2;

    unsigned PES_extension_flag         : 1;
    unsigned PES_CRC_flag               : 1;
    unsigned additional_copy_info_flag : 1;
    unsigned DSM_trick_mode_flag        : 1;
    unsigned ES_rate_flag               : 1;
    unsigned ESCR_flag                  : 1;
    unsigned PTS_DTS_flags              : 2;

#else
    unsigned noname                     : 2;
    unsigned PES_scrambling_control     : 2;
    unsigned PES_priority               : 1;
    unsigned data_alignment_indicator   : 1;
    unsigned copyright                  : 1;
    unsigned original_or_copy           : 1;

    unsigned PTS_DTS_flags              : 2;
    unsigned ESCR_flag                  : 1;
    unsigned ES_rate_flag               : 1;
    unsigned DSM_trick_mode_flag        : 1;
    unsigned additional_copy_info_flag : 1;
    unsigned PES_CRC_flag               : 1;
    unsigned PES_extension_flag         : 1;

#endif
    unsigned PES_header_data_length     : 8;
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;


  struct PTS_DTS_bits
  {
#ifdef DOS
    unsigned marker_bit_1 : 1;
    unsigned PTS_32_30    : 3;
    unsigned noname_1     : 4;

    unsigned PTS_29_22    : 8;

    unsigned marker_bit_2 : 1;
    unsigned PTS_21_15    : 7;

    unsigned PTS_14_7     : 8;

    unsigned marker_bit_3 : 1;
    unsigned PTS_6_0      : 7;

    unsigned marker_bit_4 : 1;
    unsigned DTS_32_30    : 3;
    unsigned noname_2      : 4;

    unsigned DTS_29_22     : 8;

    unsigned marker_bit_5 : 1;
    unsigned DTS_21_15     : 7;

    unsigned DTS_14_7      : 8;

    unsigned marker_bit_6 : 1;
    unsigned DTS_6_0       : 7;

#else
    unsigned noname_1      : 4;
    unsigned PTS_32_30     : 3;
    unsigned marker_bit_1 : 1;

    unsigned PTS_29_22     : 8;

    unsigned PTS_21_15     : 7;
    unsigned marker_bit_2 : 1;

    unsigned PTS_14_7      : 8;

    unsigned PTS_6_0       : 7;
    unsigned marker_bit_3 : 1;

    unsigned noname_2      : 4;
    unsigned DTS_32_30     : 3;
    unsigned marker_bit_4 : 1;

    unsigned DTS_29_22     : 8;

    unsigned DTS_21_15     : 7;
    unsigned marker_bit_5 : 1;

    unsigned DTS_14_7      : 8;

    unsigned DTS_6_0       : 7;
    unsigned marker_bit_6 : 1;

#endif
  };

  union
  {
    struct PTS_DTS_bits bits;
    char byte[10];
  } stamp;


  struct ESCR_bits
  {
#ifdef DOS
    unsigned ESCR_29_28              : 2;
    unsigned marker_bit_1            : 1;
    unsigned ESCR_32_30              : 3;
    unsigned reserved                : 2;

    unsigned ESCR_27_20              : 8;

    unsigned ESCR_14_13              : 2;
    unsigned marker_bit_2            : 1;
    unsigned ESCR_19_15              : 5;

    unsigned ESCR_12_5               : 8;

    unsigned ESCR_extension_8_7  : 2;
    unsigned marker_bit_3            : 1;
    unsigned ESCR_4_0                : 5;

    unsigned marker_bit_4            : 1;
    unsigned ESCR_extension_6_0  : 7;

#else
    unsigned reserved                : 2;
    unsigned ESCR_32_30              : 3;
    unsigned marker_bit_1         : 1;
    unsigned ESCR_29_28           : 2;

    unsigned ESCR_27_20           : 8;

    unsigned ESCR_19_15           : 5;
    unsigned marker_bit_2         : 1;
    unsigned ESCR_14_13           : 2;

    unsigned ESCR_12_5            : 8;

    unsigned ESCR_4_0             : 5;
    unsigned marker_bit_3         : 1;
    unsigned ESCR_extension_8_7   : 2;

    unsigned ESCR_extension_6_0   : 7;
    unsigned marker_bit_4         : 1;

#endif
  };

  union
  {
    struct ESCR_bits bits;
    char byte[6];
  } ESCR;


  struct ES_rate_bits
  {
#ifdef DOS
    unsigned ES_rate_21_15   : 7;
    unsigned marker_bit_1    : 1;

    unsigned ES_rate_14_7    : 8;

    unsigned marker_bit_2    : 1;
    unsigned ES_rate_6_0     : 7;

#else
    unsigned marker_bit_1    : 1;
    unsigned ES_rate_21_15   : 7;

    unsigned ES_rate_14_7    : 8;

    unsigned ES_rate_6_0     : 7;
    unsigned marker_bit_2    : 1;

#endif
  };

  union
  {
    struct ES_rate_bits bits;
    char byte[3];
  } ES_rate;


  struct DSM_trick_bits
  {
#ifdef DOS
    unsigned bits_3               : 2;
    unsigned bits_2               : 1;
    unsigned bits_1               : 2;
    unsigned trick_mode_control   : 3;

#else
    unsigned trick_mode_control   : 3;
    unsigned bits_1               : 2;
    unsigned bits_2               : 1;
    unsigned bits_3               : 2;

#endif
  };

  union
  {
    struct DSM_trick_bits bits;
    char byte;
  } DSM_trick;


  struct flag_bits
  {
#ifdef DOS
    unsigned PES_extension_flag_2                       :  1;
    unsigned reserved                                   :  3;
    unsigned P_STD_buffer_flag                          :  1;
    unsigned program_packet_sequence_counter_flag :  1;
    unsigned pack_header_field_flag                     :  1;
    unsigned PES_private_data_flag                      :  1;

#else
    unsigned PES_private_data_flag                      :  1;
    unsigned pack_header_field_flag                     :  1;
    unsigned program_packet_sequence_counter_flag :  1;
    unsigned P_STD_buffer_flag                          :  1;
    unsigned reserved                                   :  3;
    unsigned PES_extension_flag_2                       :  1;

#endif
  };

  union
  {
    struct flag_bits bits;
    char byte;
  } flags;


  struct program_packet_bits
  {
#ifdef DOS
    unsigned program_packet_sequence_counter : 7;
    unsigned marker_bit_1                    : 1;

    unsigned original_stuff_length           : 6;
    unsigned MPEG1_MPEG2_identifier          : 1;
    unsigned marker_bit_2                    : 1;

#else
    unsigned marker_bit_1                    : 1;
    unsigned program_packet_sequence_counter : 7;

    unsigned marker_bit_2                    : 1;
    unsigned MPEG1_MPEG2_identifier          : 1;
    unsigned original_stuff_length           : 6;

#endif
  };

  union
  {
    struct program_packet_bits bits;
    char byte[2];
  } program_packet;


  struct buffer_bits
  {
#ifdef DOS
    unsigned P_STD_buf_size_12_8   : 5;
    unsigned P_STD_buf_scale       : 1;
    unsigned noname                : 2;

#else
    unsigned noname                : 2;
    unsigned P_STD_buf_scale       : 1;
    unsigned P_STD_buf_size_12_8   : 5;

#endif

    unsigned P_STD_buf_size_7_0    : 8;
  };
```

```
union
{
  struct buffer_bits bits;
  char byte[2];
} buffer;


int    stream_id;
double time,
       rate;


char str[80];
register int i,
             n_bytes = 0;/* bytes counter */

fprintf(out,"|PES PACKET*********************************\n");

if(unit_start)
{
  PES->field = 2; /* do the header */
  PES->p = 0;
  PES->info_start = 1;
  PES->info_end = 6;
}

while(*bytes_read + n_bytes < 188) /*Verifies end of packet.*/
{
  if( !PES->field ) /* Ignores PES_packet (eg. padding) */
  {
    i = 188 - *bytes_read -n_bytes;
    fprintf(out,"| (rest of PES_packet ignored: %d bytes)\n",i);
    fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
    *bytes_read = 188;
    return;
  }
  else
  if( PES->field == 1) /* PES_packet_data_bytes extraction*/
  {
    i = 188 - *bytes_read -n_bytes;/* Bytes left in TS_packet*/

    if(PES->packet_length && i > PES->bytes_left)
    {
      i = PES->bytes_left;
      PES->field = 0; /* Ignores the rest */

      Warn(out,"Standard violation: can't have stuffing after data_bytes");
      /* Because if packet_length = 0 then length is unknown. One reads until
         the end of TS_PACKET. Stuffing is made in ADAPTATION_FIELD.*/
    }

             Extracts(in,  PES->elementary_PID,  PES->buffer[4],  i);/*stream_id=PES-
>buffer[4]*/
    fprintf(out,"| (%d data_bytes)\n",i);
    n_bytes += i;
    PES->bytes_left -= i;
  }
  else
  {
    /*Read PES_packet byte to the corresponding buffer.*/
    PES->buffer[PES->p+1] = (char)getc(in);
    if( feof(in) )
    {
      Warn(out,"Premature EOF in PES_packet");
      exit(1);
    }
    n_bytes++;
    PES->p++;
    PES->bytes_left--;
    PES->header_data_length--;
  }


  if( PES->field == 20) /* Stuffing_bytes */
  {
```

```
  if( PES->buffer[PES->p] != 0xFF )
  {
   sprintf(str,"Invalid stuffing_byte = 0x%02X",PES->buffer[PES->p]);
   Warn(out, str);
  }

  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
}
else
if(PES->p == PES->info_end) /*Enough info to interpret.*/
{
  switch(PES->field)
  {
   case 2: /* Non optional header */

   /* packet_start_code_prefix */
   if(PES->buffer[1] || PES->buffer[2] || PES->buffer[3]!=1)
   {
     Warn(out, "Invalid packet_start_code_prefix");
     PES->field = 0; /* Ignores rest of PES_packet */
     break;
   }

   /* stream_id */
   stream_id = PES->buffer[4];
   fprintf(out,"| stream_id = 0x%02X  ->  ", stream_id);
   switch(stream_id)
   {
    case 0xBC: /* program stream map */
               fprintf(out,"program stream map\n");
               break;
    case 0xBD: /* private_stream_1 */
               fprintf(out,"private_stream_1\n");
               break;
    case 0xBE: /* padding_stream */
               fprintf(out,"padding_stream\n");
               break;
    case 0xBF: /* private_stream_2 */
               fprintf(out,"private_stream_2\n");
               break;
    case 0xF0: /* ECM */
               fprintf(out,"ECM\n");
               break;
    case 0xF1: /* EMM */
               fprintf(out,"EMM\n");
               break;
    case 0xF2: /* DSM CC */
               fprintf(out,"DSM CC\n");
               break;
    case 0xF3: /* MHEG (ISO 13522) */
               fprintf(out,"MHEG (ISO 13522)\n");
               break;
    case 0xFF: /* program_stream_directory */
               fprintf(out,"program_stream_directory\n");
               break;
    default   :
               if(stream_id>=0xC0 && stream_id<=0xDF) /* MPEG1 or MPEG2 */
               {
                 fprintf(out,"Audio stream - number ");
                 fprintf(out,"%d\n", stream_id & 31);/*mask 00011111B*/
               }
               else
               if(stream_id>=0xE0 && stream_id<=0xEF) /* MPEG1 or MPEG2 */
               {
                 fprintf(out,"Video stream - number ");
                 fprintf(out,"%d\n", stream_id & 15);/*mask 00001111B*/
               }
               else
               if(stream_id>=0xF4 && stream_id<=0xFE) /* reserved data stream */
               {
                 fprintf(out,"reserved Data stream - number ");
                 fprintf(out,"%d", stream_id & 15);  /*mask 00001111B*/
               }
               else
               {
                 i = 188 - *bytes_read -n_bytes;
                 fprintf(out,"\n");
```

```
                sprintf(str,"Invalid stream_id: %d bytes ignored",i);
                Warn(out, str);
                PES->field = 0; /* Ignores rest of PES_packet */
                fseek(in, i, SEEK_CUR); /*Ignores rest of packet*/
                *bytes_read = 188;
                return;
              }
    }

    if(PES->old_stream_removed == NO) /*Initialized in PM_section*/
    {
      RemoveStream(PES->elementary_PID, stream_id);
      PES->old_stream_removed = YES;
      /*Implies removal every time that there's a new table and the same PID !*/
    }

    /* PES_packet_length */
    PES->packet_length = PES->buffer[5]*256 + PES->buffer[6];
    PES->bytes_left = PES->packet_length;
    fprintf(out,"| PES_packet_length = %d bytes\n", PES->packet_length);
    if(!PES->packet_length && (stream_id<0xE0 || stream_id>0xEF)) /* && !TS */
    {
      Warn(out, "Invalid length (not video)");
      PES->field = 0; /* Ignores rest of PES_packet */
      break;
    }

    if(stream_id == 0xBE) /* padding_stream */
    {
      PES->field = 0; /* Ignores padding_bytes */
    }
    else
    if( stream_id == 0xBC ||  /* program_stream_map */
        stream_id == 0xBF ||  /* private_stream_2 */
        stream_id == 0xF0 ||  /* ECM_stream */
        stream_id == 0xF1 ||  /* EMM_stream */
        stream_id == 0xFF    )/* program_stream_directory */
    {
      PES->field = 1; /* Extracts PES_packet_data_byte */
    }
    else /* optional PES header */
    {
      PES->field = 3;
      PES->info_start = PES->p + 1;
      PES->info_end = PES->p + 3;
    }
    break;


      case 3: /* Start of optional header (flags) */

    /*Read 3 bytes of header.*/
    for(i=0; i<3; i++)  header.byte[i]=PES->buffer[PES->info_start+i];

    if( header.bits.noname != 2)
    {
      Warn(out, "Invalid PES_packet header marker");
      PES->field = 0; /* Ignores rest of PES_packet */
      break;
    }

    /* PES_scrambling_control */
    i = header.bits.PES_scrambling_control;
    if(detail < 2 || i)
    {
      fprintf(out,"| PES_scrambling_control    = %d", i);
      if( i )
        fprintf(out," (scrambled - user defined)\n");
      else
        fprintf(out,"\n");
    }

    /* PES_priority */
    i = header.bits.PES_priority;
    if(detail < 2 || i)
      fprintf(out,"| PES_priority               = %d\n", i);
```

```
/* data_alignment_indicator */
i = header.bits.data_alignment_indicator;
if(detail < 2 || i)
  fprintf(out,"| data_alignment_indicator  = %d\n", i);

/* copyright */
i = header.bits.copyright;
if(detail < 2 || i)
  fprintf(out,"| copyright                 = %d\n", i);

/* original_or_copy */
fprintf(out,"| original_or_copy        = %d",header.bits.original_or_copy);
if( header.bits.original_or_copy )
fprintf(out," (original)\n");
else
  fprintf(out," (copy)\n");

/* PTS_DTS_flags */
i = header.bits.PTS_DTS_flags;
PES->flag[0] = i;
if(detail < 2 || i)
{
  fprintf(out,"| PTS_DTS_flags             = %d\n", i);
  if(i == 1)
  {
    Warn(out, "Invalid value");
    PES->field = 0; /* Ignores rest of PES_packet */
    break;
  }
}

/* ESCR_flag */
i = header.bits.ESCR_flag;
PES->flag[1] = i;
if(detail < 2 || i)
  fprintf(out,"| ESCR_flag                 = %d\n", i);

/* ES_rate_flag */
i = header.bits.ES_rate_flag;
PES->flag[2] = i;
if(detail < 2 || i)
  fprintf(out,"| ES_rate_flag              = %d\n", i);

/* DSM_trick_mode_flag */
i = header.bits.DSM_trick_mode_flag;
PES->flag[3] = i;
if(detail < 2 || i)
  fprintf(out,"| DSM_trick_mode_flag       = %d\n", i);

/* additional_copy_info_flag */
i = header.bits.additional_copy_info_flag;
PES->flag[4] = i;
if(detail < 2 || i)
  fprintf(out,"| additional_copy_info_flag = %d\n", i);

/* PES_CRC_flag */
i = header.bits.PES_CRC_flag;
PES->flag[5] = i;
if(detail < 2 || i)
  fprintf(out,"| PES_CRC_flag              = %d\n", i);

/* PES_extension_flag */
i = header.bits.PES_extension_flag;
PES->flag[6] = i;
if(detail < 2 || i)
  fprintf(out,"| PES_extension_flag        = %d\n", i);

/* PES_header_data_length */
PES->header_data_length = header.bits.PES_header_data_length;
fprintf(out,"| PES_header_data_length = %d\n",PES->header_data_length);
if(PES->packet_length && PES->header_data_length > PES->bytes_left)
{
  Warn(out, "Invalid length");
  PES->field = 0; /* Ignores rest of PES_packet */
  break;
}
```

```
/* Which is the first active flag. */
for(i=0; !PES->flag[i] && i<7; i++);
switch(i)
{
 case  0: /*PTS_DTS_flags*/
          PES->field = 4;
          PES->info_end = PES->p + 5; /* DTS decision afterwards */
          break;
 case  1: /*ESCR_flag*/
          PES->field = 6;
          PES->info_end = PES->p + 6;
          break;
 case  2: /*ES_rate_flag*/
          PES->field = 7;
          PES->info_end = PES->p + 3;
          break;
 case  3: /*DSM_trick_mode_flag*/
          PES->field = 8;
          PES->info_end = PES->p + 1;
          break;
 case  4: /*additional_copy_info_flag*/
          PES->field = 9;
          PES->info_end = PES->p + 1;
          break;
 case  5: /*PES_CRC_flag*/
          PES->field = 10;
          PES->info_end = PES->p + 2;
          break;
 case  6: /*PES_extension_flag*/
          PES->field = 11;
          PES->info_end = PES->p + 1;
          break;
 default: /*All flags are zero */
          if( !PES->header_data_length )  PES->field = 1; /*extraction*/
          else
          if(PES->header_data_length > 32)
          {
            Warn(out, "Too much stuffing_bytes (>32)");
            PES->field = 0; /*Ignores rest of PES_packet*/
          }
          else PES->field = 20; /*stuffing bytes*/
}
PES->info_start = PES->p + 1;
break;


case 4: /* PTS */

/*Read 5 bytes of PTS.*/
for(i=0; i<5; i++)  stamp.byte[i]=PES->buffer[PES->info_start+i];

fprintf(out,"|\n");

if(  stamp.bits.noname_1 != PES->flag[0] )
  Warn(out, "PTS: invalid inicial marker");
if( /* stamp.bits.noname_1 != PES->flag[0] ||*/
    !stamp.bits.marker_bit_1             ||
    !stamp.bits.marker_bit_2             ||
    !stamp.bits.marker_bit_3               )
{
  Warn(out, "PTS: invalid marker(s)");
  PES->field = 0; /* Ignores rest of PES_packet */
  break;
}

time = (double)stamp.bits.PTS_32_30 * 1073741824 /* 2^30 */
      +(double)stamp.bits.PTS_29_22 * 4194304    /* 2^22 */
      +(double)stamp.bits.PTS_21_15 * 32768      /* 2^15 */
      +(double)stamp.bits.PTS_14_7  * 128        /* 2^7  */
      +(double)stamp.bits.PTS_6_0 ;
fprintf(out,"| PTS = %g (%g s)\n", time, time/90000);

if(PES->flag[0] == 3) /* DTS */
{
  PES->field = 5;
  PES->info_end = PES->p + 5;
```

```
}
else
{
  /* Which is the next active flag */
  for(i=1; !PES->flag[i] && i<7; i++);
  switch(i)
  {
   case  1: /*ESCR_flag*/
            PES->field = 6;
            PES->info_end = PES->p + 6;
            break;
   case  2: /*ES_rate_flag*/
            PES->field = 7;
            PES->info_end = PES->p + 3;
            break;
   case  3: /*DSM_trick_mode_flag*/
            PES->field = 8;
            PES->info_end = PES->p + 1;
            break;
   case  4: /*additional_copy_info_flag*/
            PES->field = 9;
            PES->info_end = PES->p + 1;
            break;
   case  5: /*PES_CRC_flag*/
            PES->field = 10;
            PES->info_end = PES->p + 2;
            break;
   case  6: /*PES_extension_flag*/
            PES->field = 11;
            PES->info_end = PES->p + 1;
            break;
   default: /*All flags are zero */
            if( !PES->header_data_length )  PES->field = 1; /*extraction*/
            else
            if(PES->header_data_length > 32)
            {
              Warn(out, "Too much stuffing_bytes (>32)");
              PES->field = 0; /*Ignores rest of PES_packet*/
            }
            else PES->field = 20; /*stuffing bytes*/
  }
}
PES->info_start = PES->p + 1;
break;


case 5: /* DTS */

/*Read 5 bytes of DTS.*/
for(i=0; i<5; i++)  stamp.byte[i+5]=PES->buffer[PES->info_start+i];

if(  stamp.bits.noname_2 != 1 ||
    !stamp.bits.marker_bit_4  ||
    !stamp.bits.marker_bit_5  ||
    !stamp.bits.marker_bit_6      )
{
  Warn(out, "DTS: invalid marker(s)");
  PES->field = 0; /* Ignores rest of PES_packet */
  break;
}

time = (double)stamp.bits.DTS_32_30 * 1073741824 /* 2^30 */
      +(double)stamp.bits.DTS_29_22 * 4194304    /* 2^22 */
      +(double)stamp.bits.DTS_21_15 * 32768      /* 2^15 */
      +(double)stamp.bits.DTS_14_7  * 128        /* 2^7  */
      +(double)stamp.bits.DTS_6_0 ;
fprintf(out,"| DTS = %g (%g s)\n", time, time/90000);

/* Which is the next active flag */
for(i=1; !PES->flag[i] && i<7; i++);
switch(i)
{
 case  1: /*ESCR_flag*/
          PES->field = 6;
          PES->info_end = PES->p + 6;
          break;
 case  2: /*ES_rate_flag*/
```

```
            PES->field = 7;
            PES->info_end = PES->p + 3;
            break;
    case  3: /*DSM_trick_mode_flag*/
            PES->field = 8;
            PES->info_end = PES->p + 1;
            break;
    case  4: /*additional_copy_info_flag*/
            PES->field = 9;
            PES->info_end = PES->p + 1;
            break;
    case  5: /*PES_CRC_flag*/
            PES->field = 10;
            PES->info_end = PES->p + 2;
            break;
    case  6: /*PES_extension_flag*/
            PES->field = 11;
            PES->info_end = PES->p + 1;
            break;
    default: /*All flags are zero */
            if( !PES->header_data_length )  PES->field = 1; /*extraction*/
            else
            if(PES->header_data_length > 32)
            {
              Warn(out, "Too much stuffing_bytes (>32)");
              PES->field = 0; /*Ignores rest of PES_packet*/
            }
            else PES->field = 20; /*stuffing bytes*/
  }
  PES->info_start = PES->p + 1;
  break;


case 6: /* ESCR */

 /*Read 6 bytes of ESCR.*/
 for(i=0; i<6; i++)  ESCR.byte[i]=PES->buffer[PES->info_start+i];

 fprintf(out,"|\n");

 if( !ESCR.bits.marker_bit_1  ||
     !ESCR.bits.marker_bit_2  ||
     !ESCR.bits.marker_bit_3  ||
     !ESCR.bits.marker_bit_4      )
 {
   Warn(out, "ESCR: invalid marker(s)");
   PES->field = 0; /* Ignores rest of PES_packet */
   break;
 }

 if(detail < 1)
   fprintf(out,"| reserved = %d\n", ESCR.bits.reserved);

 time = (double)ESCR.bits.ESCR_32_30 * 1073741824 /* 2^30 */
       +(double)ESCR.bits.ESCR_29_28 * 268435456  /* 2^28 */
       +(double)ESCR.bits.ESCR_27_20 * 1048576    /* 2^20 */
       +(double)ESCR.bits.ESCR_19_15 * 32768      /* 2^15 */
       +(double)ESCR.bits.ESCR_14_13 * 8192       /* 2^13 */
       +(double)ESCR.bits.ESCR_12_5  * 32         /* 2^5  */
       +(double)ESCR.bits.ESCR_4_0 ;
 fprintf(out,"| ESCR_base = %g\n", time);

 rate = (double)ESCR.bits.ESCR_extension_8_7 * 128
       +(double)ESCR.bits.ESCR_extension_6_0;
 fprintf(out,"| ESCR_extension = %g     ", rate);

 fprintf(out,"(ESCR = %g s)\n", (time*300+rate)/27000000);

 /* Which is the next active flag */
 for(i=2; !PES->flag[i] && i<7; i++);
 switch(i)
 {
  case  2: /*ES_rate_flag*/
           PES->field = 7;
           PES->info_end = PES->p + 3;
           break;
  case  3: /*DSM_trick_mode_flag*/
```

```
             PES->field = 8;
             PES->info_end = PES->p + 1;
             break;
   case   4: /*additional_copy_info_flag*/
             PES->field = 9;
             PES->info_end = PES->p + 1;
             break;
   case   5: /*PES_CRC_flag*/
             PES->field = 10;
             PES->info_end = PES->p + 2;
             break;
   case   6: /*PES_extension_flag*/
             PES->field = 11;
             PES->info_end = PES->p + 1;
             break;
   default:  /*All flags are zero */
             if( !PES->header_data_length )  PES->field = 1; /*extraction*/
             else
             if(PES->header_data_length > 32)
             {
               Warn(out, "Too much stuffing_bytes (>32)");
               PES->field = 0; /*Ignores rest of PES_packet*/
             }
             else PES->field = 20; /*stuffing bytes*/
  }
  PES->info_start = PES->p + 1;
  break;


case 7: /* ES_rate */

  /*Read 3 bytes of ES_rate.*/
  for(i=0; i<3; i++)  ES_rate.byte[i]=PES->buffer[PES->info_start+i];

  fprintf(out,"|\n");

  if( !ES_rate.bits.marker_bit_1  ||
      !ES_rate.bits.marker_bit_2      )
  {
    Warn(out, "ES_rate: invalid marker(s)");
    PES->field = 0; /* Ignores rest of PES_packet */
    break;
  }

  rate = (double)ES_rate.bits.ES_rate_21_15 * 32768
        +(double)ES_rate.bits.ES_rate_14_7  * 128
        +(double)ES_rate.bits.ES_rate_6_0;
  fprintf(out,"| ES_rate = %g (%g bytes/s)\n", rate, rate*50);
  if( !rate )
    Warn(out, "Invalid value");

  /* Which is the next active flag */
  for(i=3; !PES->flag[i] && i<7; i++);
  switch(i)
  {
   case   3: /*DSM_trick_mode_flag*/
             PES->field = 8;
             PES->info_end = PES->p + 1;
             break;
   case   4: /*additional_copy_info_flag*/
             PES->field = 9;
             PES->info_end = PES->p + 1;
             break;
   case   5: /*PES_CRC_flag*/
             PES->field = 10;
             PES->info_end = PES->p + 2;
             break;
   case   6: /*PES_extension_flag*/
             PES->field = 11;
             PES->info_end = PES->p + 1;
             break;
   default:  /*All flags are zero */
             if( !PES->header_data_length )  PES->field = 1; /*extraction*/
             else
             if(PES->header_data_length > 32)
             {
               Warn(out, "Too much stuffing_bytes (>32)");
```

```
                PES->field = 0; /*Ignores rest of PES_packet*/
              }
              else PES->field = 20; /*stuffing bytes*/
        }
        PES->info_start = PES->p + 1;
        break;


        case 8: /* DSM_trick_mode */

        /*Read byte of DSM_trick_mode.*/
        DSM_trick.byte = PES->buffer[PES->info_start];

        fprintf(out,"|\n|           trick_mode_control          =          %d            ->
",DSM_trick.bits.trick_mode_control);
        switch( DSM_trick.bits.trick_mode_control )
        {
           case  0: /* Fast Forward */
                fprintf(out,"Fast Forward\n");
                fprintf(out,"| field_id = %d\n",DSM_trick.bits.bits_1);
                fprintf(out,"| intra_slice_refresh = %d\n",DSM_trick.bits.bits_2);
                fprintf(out,"| frequency_truncation = %d\n",DSM_trick.bits.bits_3);
                break;
           case  1: /* Slow Motion */
                fprintf(out,"Slow Motion\n");
                i = DSM_trick.bits.bits_1 * 8
                   +DSM_trick.bits.bits_2 * 4
                   +DSM_trick.bits.bits_3;
                fprintf(out,"| field_rep_cntrl = %d\n", i);
                break;
           case  2: /* Freeze Frame */
                fprintf(out,"Freeze Frame\n");
                fprintf(out,"| field_id = %d\n",DSM_trick.bits.bits_1);
                i = DSM_trick.bits.bits_2 * 4
                   +DSM_trick.bits.bits_3;
                if(detail > 1)
                  fprintf(out,"| reserved = %d\n", i);
                break;
           case  3: /* Fast Reverse */
                fprintf(out,"Fast Reverse\n");
                fprintf(out,"| field_id = %d\n",DSM_trick.bits.bits_1);
                fprintf(out,"| intra_slice_refresh = %d\n",DSM_trick.bits.bits_2);
                fprintf(out,"| frequency_truncation = %d\n",DSM_trick.bits.bits_3);
                break;
           case  4: /* Slow Reverse */
                fprintf(out,"Slow Reverse\n");
                i = DSM_trick.bits.bits_1 * 8
                   +DSM_trick.bits.bits_2 * 4
                   +DSM_trick.bits.bits_3;
                fprintf(out,"| field_rep_cntrl = %d\n", i);
                break;
           default: /* reserved */
                fprintf(out,"reserved\n");
        }

        /* Which is the next active flag */
        for(i=4; !PES->flag[i] && i<7; i++);
        switch(i)
        {
          case  4: /*additional_copy_info_flag*/
                PES->field = 9;
                PES->info_end = PES->p + 1;
                break;
          case  5: /*PES_CRC_flag*/
                PES->field = 10;
                PES->info_end = PES->p + 2;
                break;
          case  6: /*PES_extension_flag*/
                PES->field = 11;
                PES->info_end = PES->p + 1;
                break;
          default: /*All flags are zero */
                if( !PES->header_data_length )  PES->field = 1; /*extraction*/
                else
                if(PES->header_data_length > 32)
                {
                  Warn(out, "Too much stuffing_bytes (>32)");
                PES->field = 0; /*Ignores rest of PES_packet*/
              }
              else PES->field = 20; /*stuffing bytes*/
    }
    PES->info_start = PES->p + 1;
    break;


   case 9: /* additional_copy_info */

    /*Read byte of additional_copy_info.*/
    i = PES->buffer[PES->info_start];

    fprintf(out,"|\n");

    /* marker_bit */
    if( !(i & 128) ) /* mask 1000 0000 */
    {
      Warn(out, "Additional_copy_info: invalid marker");
      PES->field = 0; /* Ignores rest of PES_packet */
      break;
    }

    fprintf(out,"| additional_copy_info_flag = %d\n",i & 127);
    /* mask 0111 1111 */


    /* Which is the next active flag */
    for(i=5; !PES->flag[i] && i<7; i++);
    switch(i)
    {
     case  5: /*PES_CRC_flag*/
              PES->field = 10;
              PES->info_end = PES->p + 2;
              break;
     case  6: /*PES_extension_flag*/
              PES->field = 11;
              PES->info_end = PES->p + 1;
              break;
     default: /*All flags are zero */
              if( !PES->header_data_length )  PES->field = 1; /*extraction*/
              else
              if(PES->header_data_length > 32)
              {
                Warn(out, "Too much stuffing_bytes (>32)");
                PES->field = 0; /*Ignores rest of PES_packet*/
              }
              else PES->field = 20; /*stuffing bytes*/
    }
    PES->info_start = PES->p + 1;
    break;


   case 10: /* PES_CRC */

    /*Read previous_PES_packet_CRC.*/
    i = PES->buffer[PES->info_start] *256 + PES->buffer[PES->info_start +1];

    fprintf(out,"|\n| previous_PES_packet_CRC = %d\n",i);

    /*! Here one would VERIFY THE CRC */

    /* If next flag is active */
    if( PES->flag[6] )
    {
      PES->field = 11;
      PES->info_end = PES->p + 1;
    }
    else
    {
      if( !PES->header_data_length )  PES->field = 1; /*extraction*/
      else
      if(PES->header_data_length > 32)
      {
        Warn(out, "Too much stuffing_bytes (>32)");
        PES->field = 0; /*Ignores rest of PES_packet*/
      }
```

```
  else PES->field = 20; /*stuffing bytes*/
}
PES->info_start = PES->p + 1;
break;


case 11: /* PES_extension */

/*Read flags byte.*/
flags.byte = PES->buffer[PES->info_start];

/* PES_private_data_flag */
PES->flag[0] = flags.bits.PES_private_data_flag;
fprintf(out,"|\n| PES_private_data_flag    = %d\n", PES->flag[0]);

/* pack_header_field_flag */
PES->flag[1] = flags.bits.pack_header_field_flag;
fprintf(out,"| pack_header_field_flag   = %d\n", PES->flag[1]);

/* program_packet_sequence_counter_flag */
PES->flag[2] = flags.bits.program_packet_sequence_counter_flag;
fprintf(out,"| program_packet_sequence_counter_flag = %d\n",PES->flag[2]);

/* P_STD_buffer_flag */
PES->flag[3] = flags.bits.P_STD_buffer_flag;
fprintf(out,"| P_STD_buffer_flag        = %d\n", PES->flag[3]);

if(detail > 1)
  fprintf(out,"| reserved = %d\n",flags.bits.reserved);

/* PES_extension_flag_2*/
PES->flag[4] = flags.bits.PES_extension_flag_2;
fprintf(out,"| PES_extension_flag_2     = %d\n", PES->flag[4]);

/* Which is the first active flag */
for(i=0; !PES->flag[i] && i<5; i++);
switch(i)
{
 case  0: /*PES_private_data_flag*/
          PES->field = 12;
          PES->info_end = PES->p + 16; /* DTS decision afterwards */
          break;
 case  1: /*pack_header_field_flag*/
          PES->field = 13;
          PES->info_end = PES->p + 1; /* pack_field-length */
          break;
 case  2: /*program_packet_sequence_counter_flag*/
          PES->field = 15;
          PES->info_end = PES->p + 2;
          break;
 case  3: /*P_STD_buffer_flag*/
          PES->field = 16;
          PES->info_end = PES->p + 2;
          break;
 case  4: /*PES_extension_flag_2*/
          PES->field = 17;
          PES->info_end = PES->p + 1; /* PES_extension_field_length */
          break;
 default: /*All flags are zero */
          if( !PES->header_data_length )  PES->field = 1; /*extraction*/
          else
          if(PES->header_data_length > 32)
          {
            Warn(out, "Too much stuffing_bytes (>32)");
            PES->field = 0; /*Ignores rest of PES_packet*/
          }
          else PES->field = 20; /*stuffing bytes*/
}
PES->info_start = PES->p + 1;
break;


case 12: /* PES_private_data */

/* Here one would read the PES_private_data ! */

/* Which is the next active flag */
```

```
        for(i=1; !PES->flag[i] && i<5; i++);
        switch(i)
        {
         case  1: /*pack_header_field_flag*/
                  PES->field = 13;
                  PES->info_end = PES->p + 1; /* pack_field-length */
                  break;
         case  2: /*program_packet_sequence_counter_flag*/
                  PES->field = 15;
                  PES->info_end = PES->p + 2;
                  break;
         case  3: /*P_STD_buffer_flag*/
                  PES->field = 16;
                  PES->info_end = PES->p + 2;
                  break;
         case  4: /*PES_extension_flag_2*/
                  PES->field = 17;
                  PES->info_end = PES->p + 1; /* PES_extension_field_length */
                  break;
         default: /*All flags are zero */
                  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
                  else
                  if(PES->header_data_length > 32)
                  {
                    Warn(out, "Too much stuffing_bytes (>32)");
                    PES->field = 0; /*Ignores rest of PES_packet*/
                  }
                  else PES->field = 20; /*stuffing bytes*/
        }
        PES->info_start = PES->p + 1;
        break;


       case 13: /* pack_field_length */

        /* Read pack_field_length */
        i = PES->buffer[PES->info_start];
        fprintf(out,"|\n| pack_field_length = %d\n", i);
/* VALIDATE WITH header data length !!! */

        if(i)
        {
          PES->field = 14;
          PES->info_end = PES->p + i;
        }
        else
        {
          /* Which is the next active flag */
          for(i=2; !PES->flag[i] && i<5; i++);
          switch(i)
          {
           case  2: /*program_packet_sequence_counter_flag*/
                    PES->field = 15;
                    PES->info_end = PES->p + 2;
                    break;
           case  3: /*P_STD_buffer_flag*/
                    PES->field = 16;
                    PES->info_end = PES->p + 2;
                    break;
           case  4: /*PES_extension_flag_2*/
                    PES->field = 17;
                    PES->info_end = PES->p + 1; /* PES_extension_field_length */
                    break;
           default: /*All flags are zero */
                    if( !PES->header_data_length )  PES->field = 1; /*extraction*/
                    else
                    if(PES->header_data_length > 32)
                    {
                      Warn(out, "Too much stuffing_bytes (>32)");
                      PES->field = 0; /*Ignores rest of PES_packet*/
                    }
                    else PES->field = 20; /*stuffing bytes*/
          }

        }
        PES->info_start = PES->p + 1;
        break;
        case 14: /* pack_header_field */

/* FOR THE MOMENT pack_header() IS IGNORED !!! */
/* Identify if TS or PS. Identify if MPEG1 or MPEG2. */

        /* Which is the next active flag */
        for(i=2; !PES->flag[i] && i<5; i++);
        switch(i)
        {
         case  2: /*program_packet_sequence_counter_flag*/
                  PES->field = 15;
                  PES->info_end = PES->p + 2;
                  break;
         case  3: /*P_STD_buffer_flag*/
                  PES->field = 16;
                  PES->info_end = PES->p + 2;
                  break;
         case  4: /*PES_extension_flag_2*/
                  PES->field = 17;
                  PES->info_end = PES->p + 1; /* PES_extension_field_length */
                  break;
         default: /*All flags are zero */
                  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
                  else
                  if(PES->header_data_length > 32)
                  {
                    Warn(out, "Too much stuffing_bytes (>32)");
                    PES->field = 0; /*Ignores rest of PES_packet*/
                  }
                  else PES->field = 20; /*stuffing bytes*/
        }
        PES->info_start = PES->p + 1;
        break;


        case 15: /* program_packet_sequence_counter */

        /*Read 2 bytes of program_packet_sequence_counter.*/
        for(i=0; i<2; i++)  program_packet.byte[i]=PES->buffer[PES->info_start+i];

        if( !program_packet.bits.marker_bit_1 ||
            !program_packet.bits.marker_bit_2     )
        {
          Warn(out,"Program_packet_sequence_counter: invalid marker(s)");
          PES->field = 0; /* Ignores rest of PES_packet */
          break;
        }

        fprintf(out,"|\n| program_packet_sequence_counter = %d\n",
                         program_packet.bits.program_packet_sequence_counter);

        fprintf(out,"| MPEG1_MPEG2_identifier = %d\n",
                       program_packet.bits.MPEG1_MPEG2_identifier);

        fprintf(out,"| original_stuff_length = %d\n",
                       program_packet.bits.original_stuff_length);

        /* Which is the next active flag */
        for(i=3; !PES->flag[i] && i<5; i++);
        switch(i)
        {
         case  3: /*P_STD_buffer_flag*/
                  PES->field = 16;
                  PES->info_end = PES->p + 2;
                  break;
         case  4: /*PES_extension_flag_2*/
                  PES->field = 17;
                  PES->info_end = PES->p + 1; /* PES_extension_field_length */
                  break;
         default: /*All flags are zero */
                  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
                  else
                  if(PES->header_data_length > 32)
                  {
                    Warn(out, "Too much stuffing_bytes (>32)");
```

```
           PES->field = 0; /*Ignores rest of PES_packet*/
         }
         else PES->field = 20; /*stuffing bytes*/
}
PES->info_start = PES->p + 1;
break;


case 16: /* P_STD_buffer */

/* Read 2 bytes of P_STD buffer */
for(i=0; i<2; i++)  buffer.byte[i]=PES->buffer[PES->info_start+i];

if( buffer.bits.noname != 1 )
{
  Warn(out, "P_STD_buffer: invalid marker");
  PES->field = 0; /* Ignores rest of PES_packet */
  break;
}

fprintf(out,"|\n| P_STD_buffer_scale    = %d\n",buffer.bits.P_STD_buf_scale);
i = buffer.bits.P_STD_buf_size_12_8 *256
   +buffer.bits.P_STD_buf_size_7_0;
fprintf(out,"| P_STD_buffer_size     = %d (Tamanho de buffer = ",i);
if(buffer.bits.P_STD_buf_scale)
  fprintf(out,"%ld bytes)\n",(long)i *1024);
else
  fprintf(out,"%ld bytes)\n",(long)i *128);

/* If next flag is active */
if( PES->flag[4] )
{
  PES->field = 17;
  PES->info_end = PES->p + 1; /* PES_extension_field_length */
}
else
{
  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
  else
  if(PES->header_data_length > 32)
  {
    Warn(out, "Too much stuffing_bytes (>32)");
    PES->field = 0; /*Ignores rest of PES_packet*/
  }
  else PES->field = 20; /*stuffing bytes*/
}
PES->info_start = PES->p + 1;
break;


case 17: /* PES_extension_field_length */

/*Read byte of PES_extension_field_length.*/
i = getc(in);

/* marker_bit */
if( !(i & 128) ) /* mask 1000 0000 */
{
  Warn(out,"PES_extension_field_2: invalid marker");
  PES->field = 0; /* Ignores rest of PES_packet */
  break;
}

i &= 127; /* mask 0111 1111 */
fprintf(out,"|\n| PES_extension_field_length = %d\n", i);

if(i)
{
  PES->field = 18;
  PES->info_end = PES->p + i; /*Ignores PES_extension_field*/
}
else
{
  if( !PES->header_data_length )  PES->field = 1; /*extraction*/
  else
  if(PES->header_data_length > 32)
  {
```

```
              Warn(out, "Too much stuffing_bytes (>32)");
              PES->field = 0; /*Ignores rest of PES_packet*/
            }
            else PES->field = 20; /*stuffing bytes*/
          }
          PES->info_start = PES->p + 1;
          break;

        case 18: /* PES_extension_field */

          /* Here one would read the PES_extension_field ! */

          if( !PES->header_data_length )  PES->field = 1; /*extraction*/
          else
          if(PES->header_data_length > 32)
          {
            Warn(out, "Too much stuffing_bytes (>32)");
            PES->field = 0; /*Ignores rest of PES_packet*/
          }
          else PES->field = 20; /*stuffing bytes*/
          break;
        }
      }
    }

  *bytes_read = 188;
}


void
Warn(FILE *out, char *str)
/************************************
 Prints a warning in the parsing and in a log file.
 ************************************/
{
  /* boolean log;
        unsigned long n_TPs;
        char no_extension[80]; GLOBAL VAR. !*/
  char errorstr[80], s[80];
  FILE *logfile;

  if( log )
  {
    sprintf(s,"%s.LOG", no_extension);

    if((logfile=fopen(s,"at")) == NULL)
    {
        sprintf(errorstr,"\nat - can't open %s (log file)",s);
        perror(errorstr);
        return;
    }
    sprintf(s,"%ld\t! %s\n", n_TPs + 1, str);
    fprintf(logfile, s);

    fclose(logfile);
  }

  fprintf(out,"! ");
  fprintf(out, str);
  fprintf(out,"\n");
}
```

5. CODIFICADOR DE FLUXOS BINÁRIOS DE TRANSPORTE MPEG-2 SISTEMA

5.1 Introdução

A estrutura básica do codificador segue a lógica do que foi implementado nos descodificadores através de uma operação inversa. Os diversos campos de informação são mapeados nos *bytes* enviados.

No entanto, levantam-se questões mais complexas, pois é necessário simular toda a etiquetagem temporal e estabelecer uma estratégia de multiplexagem, assegurar que todos os casos de ajustes no empacotamento sejam considerados e que os valores sejam correctamente codificados.

Nos pontos seguintes, descreve-se o percurso seguido e relevam-se os pontos mais importantes, justificando as decisões tomadas.

5.2 Entrada de dados

A primeira questão que se pôs foi como introduzir toda a quantidade de dados de uma forma flexível.

Decidiu-se utilizar o formato do ficheiro com o resultado do *parse* feito pelo descodificador MPEG-2 (ver ponto 4.3.4), pois este é facilmente alterável com um editor de texto. Desta maneira, testes de desmultiplexagem e multiplexagem consecutivos para comparação ficariam também facilitados.

O programa é corrido tendo como argumento estes ficheiros de gestão de multiplexagem. Dados extra são introduzidos manualmente durante a prototipagem. Posteriormente, serão incorporados no ficheiro de gestão.

Uma vez que este trabalho se centra sobre a parte de sistema da norma, são utilizados os fluxos elementares MPEG-1 referidos no ponto 4.2..

5.3 Etiquetas temporais

A etiqueta temporal de referência PCR (*Program Clock Reference*) é medida em ciclos do relógio de sistema, de frequência 27 MHz. Como sabemos a quantidade de *bytes* que vamos produzir, é essencial determinar o débito do fluxo de transporte para determinar o PCR.

As etiquetas temporais PTS e DTS (*Presentation* e *Decoding Time Stamps*) de um fluxo elementar são medidas em ciclos do valor base (1/300) do relógio de sistema: 90 KHz.

O estudo de um protótipo de um codificador de MPEG-1 Sistema[13] permitiu estabelecer um método para obter etiquetas temporais a partir das unidades de acesso de fluxos vídeo e áudio.

Uma unidade de acesso é a representação codificada de uma unidade de apresentação (imagens de vídeo ou tramas de áudio).

Se a imagem é a primeira de um GOP (*Group of Pictures*), então a unidade de acesso contém o cabeçalho de informação referente a esse GOP. O mesmo se passa para uma sequência de imagens.

A unidade de apresentação para áudio é o conjunto de amostras de uma trama de áudio.

Como não se espera, à partida, que uma multiplexagem em *software* seja feita em tempo real, a solução de pré-processar os fluxos elementares torna-se muito atraente, pois permite ler e confirmar a informação necessária sobre o fluxo.

Uma multiplexagem feita à medida que se lê os fluxos elementares, implica uma gestão complexa de *buffers* e não se justifica para os objectivos deste trabalho.

5.3.1 Vídeo

A primeira imagem, em termos de descodificação, de um GOP é do tipo I. A última, em termos de apresentação, é do tipo I ou P. Os *start_codes* que identificam as imagens estão sempre alinhados ao *byte*.

[13] "mplex" de Christof Moar (1994).

Na sintaxe de vídeo para o *sequence_header()* obtém-se o campo *picture_rate*, que fornece informação sobre o número de imagens por segundo e a partir do qual são feitas todas as assumpções sobre o fluxo.

O cálculo do tempo de descodificação é imediato:

$$ciclos\_por\_imagem = \frac{RELOGIO\_BASE}{imagens\_por\_segundo}$$

$$DTS = ordem\_de\_descodificacao \times ciclos\_por\_imagem$$

A *ordem_de_descodificação* é a sequência em que as unidades de acesso são lidas.

O campo *temporal_reference*, presente na unidade de acesso, informa-nos da ordem de apresentação da imagem no GOP. Como a ordem de descodificação não pode ser maior do que a de apresentação, e devido ao reordenamento das imagens, temos:

$$PTS = (temporal\_reference + 1 - ordem\_de\_descodificacao\_no\_GOP + ordem\_de\_descodificacao)$$
$$\times\ ciclos\_por\_imagem$$

É também fornecido o débito do fluxo através do campo *bit_rate*. Pode-se, no entanto, a partir do *picture_rate*, calcular um valor mais correcto:

$$debito = \frac{tamanho\_do\_fluxo}{numero\_de\_imagens} \times imagens\_por\_segundo$$

A rotina implementada extrai também informação adicional (pág. 148 a 153). Mostra-se a seguir um exemplo com o fluxo obtido pelo parser de MPEG-1 (ver ponto 4.2):

```
Scanning video stream w/PID = 0x901 (2305) and stream_id = 0xE0 (224).

Picture headers:     142
File length    = 863689
Stream length = 862655

sequence_start_codes   :        1
sequence_end_codes     :        1
GOPs                   :       13
Pictures               :      142
             I         :       12 -> avg. size =  10922 bytes
             P         :       36 -> avg. size =   9458 bytes
             B         :       94 -> avg. size =   4160 bytes
             D         :        0 -> avg. size =      0 bytes

Horizontal size:     352
Vertical size  :     240     Aspect ratio :  1.0000 (VGA etc)

Picture rate   :   24.000 frames/sec
Bit rate       :    12678 bytes/sec (  35888 bits/sec)
Computed rate  :    15178 bytes/sec (  55888 bits/sec) [based on picture rate]
Vbv buffer size:    40960 bytes
CSPF           :        1
```

5.3.2 Áudio

As tramas áudio apenas necessitam de PTS (que coincide com o DTS) e assumem-se alinhadas ao *byte*.

O número de amostras por segundo é obtido através do campo *sampling_frequency* e a quantidade de amostras, através do *layer* (camada). Desta maneira, o PTS avalia-se directamente a partir da ordem de descodificação:

$$PTS = \frac{ordem\_de\_descodificacao \times amostras \times RELOGIO\_BASE}{amostras\_por\_segundo} + PTS\_minimo\_de\_video$$

Adiciona-se o menor tempo de apresentação de vídeo (*PTS_minimo_de_video*) para que o áudio não seja apresentado antes do vídeo, quando a ordem de descodificação for igual a zero.

O débito do fluxo resulta do cruzamento do *layer* com o *bitrate_index.*

Mostra-se a seguir o resultado da rotina implementada (pág. 154 a 157), para o exemplo de áudio:

```
Scanning audio stream w/PID = 0x902 (2306) and stream_id = 0xC0 (192).

Frame headers:     230
File length    = 96222
Stream length = 96131

Syncwords       :      230
Frames          :        9 with size     417 bytes
                :      221 with size     418 bytes

Layer           :        2
CRC checksums   :        no  (protection_bit = 1)

Bit rate        :     16384 bytes/sec (128 kbit/sec)
Frequency       :      44.1 kHz

Mode            :         3 single channel
Mode extension  :         0
Copyright bit   :         0 no copyright
Original/Copy   :         0 copy
Emphasis        :         0 none
```

5.4 Débito do fluxo de transporte

O STD (*Standard Target Decoder*) descodifica apenas um programa de cada vez. As indicações temporais constantes na norma são relativas à base de tempo desse programa.

A altura em que um *byte* entra no STD pode ser determinada pela contagem dos *bytes* - no fluxo de transporte completo - entre PCRs sucessivos do programa em causa.

Os fluxos de transporte podem conter vários programas com bases temporais independentes. Conjuntos separados de PCRs (indicados pelos *PCR_PID*) são necessários para cada programa independente e, portanto, os PCRs não podem ter a mesma localização (condição necessária para se poder variar o débito de transporte).

Não é possível que o débito do fluxo de transporte seja variável quando contém vários programas com bases temporais independentes. É possível ter fluxos de transporte de débito constante com vários programas de débito variável.

A discussão das questões relacionadas com débitos variáveis é feita no apêndice D da norma. Para efeitos do trabalho em curso, assume-se que o débitos de transporte (*debito_TS*) e dos programas são constantes:

$$debito\_TS = \sum_{p=1}^{numero\_programas} debito\_programa_p + debito\_PSI + debito\_extra$$

Basicamente, o débito de transporte é constituído pelos débitos dos programas que o compõem. De acordo com o apêndice C da norma, é necessário, para além do débito inerente às tabelas de PSI, considerar um débito extra para elementos respeitantes à transmissão do fluxo, que se poderá também usar para acertos, devido ao esquema de multiplexagem.

5.4.1 Informação Específica de Programa

Uma tabela de PSI divide-se em secções com comprimento máximo de 1024 *bytes* (excepto no caso de tabelas privadas, que podem ir até aos 4096 *bytes*).

A tabela de associação PAT (de PID = 0) estabelece uma relação entre os programas e os PIDs dos pacotes que transportam a sua definição.

É a tabela de mapeamento de programas PMT que define os elementos de um programa. Cada programa só pode ser definido numa secção.

Na implementação, sem prejuízo do raciocínio geral, assume-se que cada secção cabe num pacote de transporte e que cada pacote transporta apenas uma secção. No caso da PAT, isto limita o número máximo de programas. No caso da PMT, implica que é necessário um TP por programa, facilitando o manuseamento mas aumentando a largura de banda necessária. Deste modo, obtém-se o débito através do número de pacotes necessários e a frequência de ocorrência da informação:

$$debito\_PSI = (1 + numero\_programas) \times 188 \times frequencia\_de\_PSI$$

Para se poder comparar com o relógio, em ciclos, basta calcular:

$$entrega\_de\_PSI = \frac{RELOGIO\_SISTEMA}{frequencia\_de\_PSI}$$

5.4.2 Programa

O débito de um programa é constituído pela largura de banda necessária ao envio de cada um dos fluxos elementares que o compõem. Para o seu cálculo tem que se entrar em linha de conta com os cabeçalhos necessários à transmissão da informação (*overhead*):

$$debito\_programa = \sum_{n=1}^{numero\_ES} \left(debito\_ES_n \times overhead_n\right)$$

O débito dos fluxos elementares (*debito_ES*) foi visto no ponto 5.3. O *overhead* para cada fluxo elementar deve-se aos cabeçalhos dos pacotes de PES e de transporte.

Cabeçalho de Transporte

O cabeçalho de transporte (pág. 161 a 164) compreende 4 *bytes* fixos e o campo de adaptação, usado principalmente para o PCR (8 *bytes*) e para *stuffing* (de 1 a 184 *bytes*).

É obrigatório incluir PCR nos TPs, de PID = PCR_PID, que transportam o início de um pacote PES que contenha unidades de acesso.

Um ponto de acesso é indicado por *random_access_indicator* = 1 e por *payload_unit_start* = 1. Nesta implementação, o primeiro só é activado se o segundo também o for.

Pode haver, em TPs intermédios (PID = PCR_PID), PCRs extra para verificar a restrição da frequência de codificação de PCR (ponto 2.7 da norma). A restrição diz respeito ao fluxo de transporte completo (com *padding*, PSI e outros programas), devendo ser, por isso, verificada na altura da multiplexagem.

O intervalo máximo entre PCRs de 0,1s permite projectar e construir PLLs (*Phase Locked Loops*) estáveis. Contudo, em muitas aplicações, a PLL permite aumentar este intervalo.

Normalmente, se não houver restrições activas, apenas o TP que transporta o início do pacote PES em causa é que inclui PCR. O último TP com o pacote PES apenas incluirá o *stuffing* necessário para o ajuste de comprimento.

Conhecido o débito de transporte consegue-se prever quantos TPs é que podem existir entre os PCRs de um programa:

$$TP\_entre\_PCRs = INT\left(\frac{0{,}1 \times debito\_TS}{188}\right)$$

O *overhead* TP_{oh} é calculado em relação ao pacote PES (unidade de transporte de dados):

$$capacidade\_TP = 188 - cabeçalho\_fixo - tamanho\_PCR$$

$$stuffing = capacidade\_TP - (tamanho\_pacote\_PES \ \% \ capacidade\_TP)$$

$$TPs\_por\_pacote\_PES = INT\left(\frac{tamanho\_pacote\_PES}{capacidade\_TP}\right) + 1$$

$$TP_{oh} = (cabeçalho\_fixo + tamanho\_PCR) \times TPs\_por\_pacote + stuffing$$

Se o PID for diferente do PCR_PID, então: *tamanho_PCR* = 0 *bytes* e *capacidade_TP* = 184 *bytes*.

Na implementação em causa, se PID = PCR_PID, assume-se o pior caso, ou seja, aquele em que existe PCR em todos os TPs. Esta simplificação pode ser facilmente absorvida pelo *debito_extra*. Se *stuffing* for maior ou igual ao *tamanho_PCR* multiplicado pelos *TPs_por_pacotes_PES*, então, para o mesmo *tamanho_pacote_PES*, o *overhead* é igual ao caso anterior. Ou seja: o impacto de se considerar o pior caso não é muito significativo para valores de *tamanho_pacote_PES* normais (na ordem dos 2 *Kbytes*). Fazendo os cálculos com este valor, obtém-se um TP_{oh} = 208 *bytes* por pacote PES.

Cabeçalho de pacote PES

O cabeçalho compreende 9 *bytes* fixos e uma quantidade variável de *bytes*, dependendo das *flags* activas. Nesta implementação consideram-se apenas as etiquetas temporais (5 *bytes* para cada uma).

Os *bytes* de *stuffing* previstos no cabeçalho podem ser usados para providenciar um cabeçalho de tamanho constante.

A existência do início de uma unidade de acesso no pacote implica um PTS. Se for diferente do DTS, este último também é incluído.

Poder-se-ia calcular um valor médio do *overhead* PES_{oh} . No caso de vídeo, utilizar-se-ia o tamanho médio das imagens (média ponderada dos tamanhos médios de cada tipo de imagem) e assumir-se-ia que seria necessário PTS e DTS nas imagens I e P.

Neste caso, como se trata de um valor médio, teríamos que confiar no *debito_extra* para absorver os picos de 19 *bytes*.

Como na implementação em causa (pág. 173 a 184) é conveniente saber o valor exacto da capacidade do pacote PES, para acertos e adaptações dos pacotes às unidades de acesso, providencia-se *stuffing* no cabeçalho, de modo a que este tenha um valor fixo (PES_{oh} = 19 *bytes* para vídeo; PES_{oh} = 14 *bytes* para áudio).

Overhead

A maneira mais simples de se calcular este valor é a seguinte:

$$overhead = \frac{capacidade\_do\_pacote + cabeçalhos}{capacidade\_do\_pacote}$$

De onde facilmente se chega a:

$$overhead = \frac{tamanho\_pacote\_PES + TP_{oh}}{tamanho\_pacote\_PES + PES_{oh}}$$

5.5 Multiplexagem

Depois de calculado o somatório dos débitos dos programas e tabelas PSI, o débito extra permite obter um débito total de transporte capaz de suportar as restrições impostas pelo esquema de multiplexagem.

O controlo sobre o débito extra permite evitar o cálculo iterativo entre *débito_TS* e estratégia de multiplexagem, definindo logo à partida o valor sobre o qual se vão realizar todos os cálculos.

A estratégia de multiplexagem implementada simula os *buffers* existentes no descodificador para cada fluxo elementar. No caso de sobrecarga (*overflow*) são gerados TPs nulos. Para evitar a subcarga (*underflow*) é usado um atraso temporal inicial (representativo do tempo que os dados demoram a chegar e a encher os *buffers*) e são estimados os tempos de chegada de cada unidade de acesso.

A sobrecarga implica perda de dados, o que poderá ter consequências mais ou menos gravosas. A subcarga implica que o descodificador terá de utilizar técnicas para disfarçar a falta de dados. Para variações mínimas, dependendo da aplicação, o descodificador poderá, por exemplo, alterar ligeiramente o valor de relógio. Será preferível então ser mais rigoroso com as sobrecargas do que com as subcargas.

O valor do PCR corrente é facilmente calculável:

$$PCR\_corrente = \frac{bytes\_enviados + posicao\_do\_PCR}{debito\_TS} \times RELOGIO\_SISTEMA$$

Da mesma maneira pode calcular-se o tempo da entrega de uma unidade de acesso:

$$PCR\_da\_proxima\_AU = \frac{bytes\_enviados + tamanho\_TP + bytes\_para\_envio\_da\_AU}{debito\_TS}$$

$$\times RELOGIO\_SISTEMA$$

Neste caso considerou-se que seria enviado um TP diferente e depois, consecutivamente, a unidade de acesso. É possível definir outros critérios, dependendo da aplicação.

Na multiplexagem usa-se o mesmo relógio (PCR) para todos os programas. Isto não invalida a possibilidade de usar bases temporais diferentes, nomeadamente através do atraso temporal inicial (pág. 138).

O enchimento dos *buffers* é feito com pacotes de transporte, descontando no comprimento das unidades de acesso através dos pacotes PES (pág. 183).

São feitas as verificações relativas ao *random_access_indicator*, aos PTS e DTS e também em relação à frequência de codificação dos PCRs.

O esquema de multiplexagem usado (pág. 136 a 140) ordena, num vector, os fluxos elementares em relação ao seu débito, transmitindo um pacote se:

- existe espaço no *buffer* correspondente;
- o fluxo não acabou;
- os outros são entregues a tempo.

Caso isto não aconteça para nenhum dos fluxos, verifica-se se:

- existe espaço no *buffer* correspondente;
- o fluxo não acabou;
- o fluxo poderá chegar atrasado.

Por último, se não existir espaço nos *buffers*, então é enviado um pacote nulo.

Anteriormente, verificou-se a necessidade de transmitir PSI, considerando a PAT como referência, fazendo-o somente se não houvesse fluxos em via de chegarem atrasados.

As experiências realizadas demonstraram que este algoritmo não tinha bom desempenho, tendendo a enviar um fluxo consecutivamente, estabilizando em situações em que todos os fluxos chegavam atrasados.

A solução deste problema estará no ordenamento dinâmico do vector de fluxos elementares, em função do tempo de entrega da unidade de acesso corrente.

5.6 Código fonte

5.6.1 README

```
/**************************************************************************
 README.V25 - 95/06/30                                  Andre M.C.A. Braganca
***************************************************************************/

MPEG-2 System Transport Stream Multiplexer: ISO/IEC 13818-1 (NO721 - June 94)
-----------------------------------------------------------------------
Arguments: [-{switch}...] filename [filename ...] (accepts *)

  switch: L, generates log file with valid commands found in manager file.
          B, displays simulated decoder buffers fullness.

Input:

 - MUX manager files as arguments.

 - Elementary streams referenced in manager file.

Output:

 - Binary MPEG2 Transport Stream file ({input filename}.{mux name}).

 - Optional log file ({input filename}.LOG).

-----------------------------------------------------------------------
mux v.25 - Andre'Braganca @1995 - INESC/Programa Ciencia
send comments to: amb@bart.inescn.pt


CHARACTERISTICS


o The program accepts as inputs binary files holding elementary streams (ES)
  and manager files defining the multiplexing strategy and providing PSI
  (Program Specific Information) and PES (Packetized Elementary Stream)
  information.
  Different manager files give origin to different multiplexed bitstreams,
  encoded with the Transport Stream syntax specified in ISO 13818-1 (June 1994
  version);


o The input elementary bitstream files are identified in the manager file by
  the, decimal or hexadecimal, PID and stream_id ({PID}.{stream_id});


o This files are previously scaned to gather access units information and to
  calculate program rates. TS rate is found using the program rates, the PSI
  frequency and an extra overhead rate.


o Bitstream length fits the sum of the access units length and may be smaller
  than file length.


o When an elementary bitstream file ends, the PES_packet_length is adjusted
  accordingly. So stuffing is made correctly in the adaptation_field.


o Linked lists are used to hold PSI and PES information so that the Transport
  Packets (TPs) can be easily generated according to the multiplexing strategy.


o The multiplexing strategy uses simulated decoder buffers. Overflow is avoided
  by sending padding packets. Uses decoding times to try to avoid underflow.


o This program has been implemented in Borland C for DOS. In order to ensure
  compatibility with UNIX (gcc) and because of the different representation of
```

bytes in these two operating systems, unions with bit fields have been used to enclose the bytes written to the disk file. Depending on the actual environment being used, these bit fields may or not be inverted. Another solution to ensure portability, would be to use masks and shifts in order to access individual bits within the bytes. However this solution would make the source code very hard to read;

MANAGER FILE

o The manager file is an ASCII file with the same format of the file generated by the MPEG2 parser (of the same author). Information is gruped under a PID (Packet Identifier).

o Data is provided in lines with the format (enclosed in ""):
"| <variable> = <value>".

o So, the program first searches lines which carrie the PID, "| PID = <value>", in order to read information regarding the PAT, PMT or each ES.

o Data lines not starting with '|' followed by ' ' are discarded.

o Data lines without '=' separating <variable> and <value> are discarded.

o Data lines with <variable> unknown are discarded.

o Data lines for a given <variable> may be repeated. Only the last one is used.

o Lines starting with ' ' followed by '-' end a group of information for a given PID.

o Empty lines ('|' followed by any number of spaces) end a repetitive group of information.

o Repetitive groups of information are:
- program info in PAT, started by "| program_number = <value>".
- stream info in PMT, started by "| stream_type = <value>".

o PID and stream_id values identify elementary stream files. Can be decimal or hexadecimal.

o In the file, PAT should came first ("| PID = 0") and be followed by PMT sections. FOR THE TIME BEING a TP carries just one section, which fits just one TP. So there are as much PMT Transport Packets as there are programs.

Example of Manager File (same format as the parser output, irrelevant lines are discarded):

```
 ---------------------------------------------------
| PID = 0x0000 (Program Association Table)
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PROGRAM ASSOCIATION SECTION*****************
| pointer_field = 0 bytes
| table_id = 0
| section_syntax_indicator = 1
| section_length = 13
| transport_stream_id = 2048
| version_number = 0
| current_next_indicator = 1
| section_number = 0
| last_section_number = 0
|
| program_number = 1
| program_map_PID = 0x0100
|
| CRC_32 = 1.02515e+09
|
| 167 bytes left - 167 stuffing_bytes (0xFF) detected
 ------------------------------------------------- 1

 -------------------------------------------------
| PID = 0x0100
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PROGRAM MAP SECTION**************************
| pointer_field = 0 bytes
| table_id                 = 2
| section_syntax_indicator = 1
| section_length = 23
| program_number = 1
| version_number         = 0
| current_next_indicator = 1
| section_number         = 0
| last_section_number    = 0
| PCR_PID = 0x0901
| program_info_length = 0
|
| stream_type = 2   -> ITU-T Rec.H262 | ISO/IEC 13818-2 Video
| elementary_PID = 0x0901
| ES_info_length = 0
|
| stream_type = 3   -> ISO/IEC 11172 Audio
| elementary_PID = 0x0902
| ES_info_length = 0
|
| CRC_32 = 1.12372e+08
|
| 157 bytes left - 157 stuffing_bytes (0xFF) detected
 ------------------------------------------------- 2

 -------------------------------------------------
| PID = 0x0901
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 3 (adaptation_field followed by payload)
| continuity_counter = 0 (counter of payloads with the same PID)
|ADAPTATION FIELD******************************
| adaptation_field_length = 7
| descontinuity_indicator = 0
| random_access_indicator = 1
| ES_priority_indicator   = 0
| PCR_flag                        = 1
| OPCR_flag                       = 0
| splicing_point_flag             = 0
| transport_private_data_flag     = 0
| adaptation_field_extension_flag = 0
|
| PCR_base = 0
| PCR_extension = 0      (PCR = 0 s)
|
| 0 bytes left - 0 stuffing_bytes (0xFF) detected
|PES PACKET**********************************
| stream_id = 0xE0  -> Video stream - number 0
| PES_packet_length = 6033 bytes
| PES_scrambling_control    = 0
| PES_priority              = 0
| data_alignment_indicator  = 1
| copyright                 = 0
| original_or_copy          = 0 (copy)
| PTS_DTS_flags             = 3
| ESCR_flag                 = 0
| ES_rate_flag              = 0
| DSM_trick_mode_flag       = 0
| additional_copy_info_flag = 0
| PES_CRC_flag              = 0
| PES_extension_flag        = 0
| PES_header_data_length = 10
|
| PTS = 14356 (0.159511 s)
| DTS = 4096 (0.0455111 s)
| (157 data_bytes)
 ------------------------------------------------- 3

 -------------------------------------------------
| PID = 0x0901
| transport_error_indicator = 0
| payload_unit_start_indicator  = 0
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 1 (counter of payloads with the same PID)
|PES PACKET***********************************
| (184 data_bytes)
 ------------------------------------------------- 4


 -------------------------------------------------
| PID = 0x0902
| transport_error_indicator = 0
| payload_unit_start_indicator  = 1
| transport_priority            = 0
| transport_scrambling_control  = 0
| adaptation_field_control      = 1 (payload only)
| continuity_counter = 0 (counter of payloads with the same PID)
|PES PACKET***********************************
| stream_id = 0xC0  ->  Audio stream - number 0
| PES_packet_length = 6018 bytes
| PES_scrambling_control    = 0
| PES_priority              = 0
| data_alignment_indicator  = 1
| copyright                 = 0
| original_or_copy          = 0 (copy)
| PTS_DTS_flags             = 2
| ESCR_flag                 = 0
| ES_rate_flag              = 0
| DSM_trick_mode_flag       = 0
| additional_copy_info_flag = 0
| PES_CRC_flag              = 0
| PES_extension_flag        = 0
| PES_header_data_length = 5
|
! PTS: invalid inicial marker
| PTS = 14356 (0.159511 s)
| (170 data_bytes)
 ------------------------------------------------- 382


Data read by the multiplexer (for the moment):


| PID = 0x0000
| transport_priority = 0
| transport_scrambling_control = 0
| transport_stream_id = 2048
| program_number = 1
| program_map_PID = 0x0100
|
-
| PID = 0x0100
| transport_priority = 0
| transport_scrambling_control = 0
| PCR_PID = 0x0901
| stream_type = 2
| elementary_PID = 0x0901
|
| stream_type = 3
| elementary_PID = 0x0902
|
 -
| PID = 0x0901
| stream_id = 0xE0
| PES_packet_length = 6033
| PES_scrambling_control = 0
| PES_priority = 0
| copyright = 0
| original_or_copy = 0
 -
| PID = 0x0901
 -
| PID = 0x0902
| stream_id = 0xC0
| PES_packet_length = 6018
| PES_scrambling_control = 0
| PES_priority = 0
| copyright = 0
| original_or_copy = 0
 -
```

5.6.2 Listagem

```
#define DOS /* Delete this if compiled in UNIX */

/*************************************************************************

 MPEG2 system bitstreams Multiplexer: ISO/IEC 13818-1 (N0721 - June 94)

 version 25 - Andre' Braganca @ 09/07/95 - INESC/Programa Ciencia


 author: Andre' Braganca

 mail:    Grupo CDAV
          INESC - Instituto de Engenharia de Sistemas e Computadores
          Largo Mompilher 22
          Apartado 4433
          4007 PORTO CODEX
          PORTUGAL

 phone:         02 2087830
 fax:     02 2087829

 email:   amb@newton.inescn.pt
          amb@bart.inescn.pt

 *************************************************************************


 The program accepts as inputs binary disk files holding elementary streams and
 a management file in order to produce a multiplexed bitstream encoded
 according to the Transport Stream syntax specified in ISO 13818-1 (version of
 June 1994).

 MPEG2 PARSER DEVELOPMENT:

 20.C - It is based on 17.C, the first public development of the MPEG2 parser,
 and its data structure, which closely follows the standard.
  This prototype generates consecutive TPs with video PES. The flags and
 variable values (e.g. PCR) are defined in the code.

 21.C - Insertion of a PAT and a PMT packet before the other TPs: incipient TPs
 manager.
  Made the distinction between program and PSI TPs.

 22.C - Data structures with the necessary PES and PSI information for the
 multiplexing control. Reads the MANAGER file to initialize this info. This
 file has a format similar to the parser output.
  Translation of source code from Portuguese to English.
  PES packet length is adusted to the real ES length, so that last packet
 stuffing is correct.

 23.C - Cleaned up File_Size() function and other details.
  Improved the MANAGER file interpretation robustness.


 24.C - Parses video and audio streams for information and access units (au).
  Changed masks from decimal to hexadecimal.
  Modified data structure in order to multiplex more than one program. Assumed
 that sections of PSI tables fit into one TP and that one TP carries just one
 section (for PMT this means that there's a TP for each program description).
  Deleted Autoria().
  Calculates TS_rate.

 25.C - Multiplexing scheme: uses decoder buffer simulation to find when to
 transmit a TP. PSI information is inserted only if the ES aren't behind
 schedule.
  Implemented Warn() function so that multiplex restrictions violations can be
 registered.
  Time stamping.
  Buffer management.
  Numbered the access units.


 TO DO:
        Descriptors.
```

```
*****************************************************************************/

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
#include "math.h"

#ifdef DOS
  #include <dir.h>
  #include <dos.h>
  #define MAX_FILES 9 /* manager files */
#endif

/* /// Compilation problem in UNIX ///// */

#ifndef SEEK_SET
  #define SEEK_SET 0
#endif
#ifndef SEEK_CUR
  #define SEEK_CUR 1
#endif
#ifndef SEEK_END
  #define SEEK_END 2
#endif

/* /// Defines ////////////////////////// */

#define MAX_PMT_SECTIONS  5
#define MAX_ES            20

#define TP_HEADER          4
#define ADAP_FIELD_W_PCR   8
#define PES_OH_VIDEO      19 /* Worst case of overhead in video PES packets */
#define PES_OH_AUDIO      14 /* Worst case of overhead in audio PES packets */
#define PCR_OFFSET        11 /* Byte w/last bit of PCR_base */


#define SYSTEM_CLK   27000000.0 /* Hz */
#define BASE_CLK        90000.0 /* System Clock / 300 */


#define SEQUENCE_HEADER 0xB3
#define GROUP_START     0xB8
#define PICTURE_START   0x00
#define SEQUENCE_END    0xB7

#define AUDIO_SYNCWORD  0xFFF


/* /// Type definitions //////////////// */

typedef enum { NO, YES }      boolean;
typedef struct PAT_program  *PAT_program_ptr;
typedef struct PMT_stream   *PMT_stream_ptr;
typedef struct PMT_section  *PMT_section_ptr;
typedef struct buffer_queue *buffer_ptr;

/* /// Global variables //////////////// */

char no_extension[80];/*Defined in NameOut(). Used in RemoveStream(),
                        Extracts().*/

boolean logfile, /* Creation of a log file with errors during multiplexing.
                    See FindSwitch(), Main(), Warn(). */
        buffer_status; /* Indication of simulated decoder buffer fullness.
                          See FindSwitch(), Main(), clean_buffers(). */

unsigned long n_TPs; /* TP counter. Initialized in TRANSPORT_STREAM().*/


struct
{
  /* TP info. */
  char transport_priority;
  char transport_scrambling_control;
  char reserved;

  /* PAT info. */
  int  transport_stream_id;
  PAT_program_ptr program;/* Pointer to program info.*/

  /* Extra info. about the Transport Stream */
  int  n_progs;  /* Number of programs.*/
  int  n_streams;/* Total number of elementary streams.*/
  unsigned long TS_rate;

} PAT;


struct PAT_program
{
  int  number;
  char reserved;
  int  PID;
  PAT_program_ptr next;
};


struct PMT_section
{
  /* TP info. */
  char transport_priority;
  char transport_scrambling_control;
  char reserved;

  /* Program info. */
  int  PID;
  int  program_number;
  int  PCR_PID;
  double  last_PCR; /* used to check PCR frequency */
  double  startup_delay;/* Added to PTS and DTS (to avoid buffer underflow) */
  PMT_stream_ptr stream;/* Pointer to ES info.*/

  /* Extra info. about the program */
  int  n_streams;
  unsigned long prog_rate;

} PMT[MAX_PMT_SECTIONS];


struct access_unit
{
  long         number;
  long         length;
  unsigned int picture_coding_type; /* Not used for audio */
  double       DTS; /* For audio, DTS = PTS */
  double       PTS;
};


struct PMT_stream
{
  PMT_stream_ptr  next;
  PMT_section_ptr PMT;

  /* PMT info. */
  int  type;
  char reserved;
  int  elementary_PID;

  /* TP info. */
  char continuity_counter;

  /* PES info. */
  int  id;
  int  PES_packet_length;
  char PES_scrambling_control;
  char PES_priority;
  char copyright;
  char original_or_copy;

  /* Extra info. about the stream */
  int  PES_bytes_left; /* Used for comparisons with PES_packet_length */
  FILE *file; /* Pointer to bitstream file. If NULL file hasn't been found.*/
  long size;  /* File size.*/
  long offset;/* Stream size.*/
  int PES_payload;
  int TP_payload;
  unsigned long rate;

  /* Simulation of decoder buffer, used in the multiplexing scheme */
  buffer_ptr buffer;
  long buffer_size;
  long buffer_space;

  /* Video or audio Access Units information.*/
  FILE *info_file;
  struct access_unit au;
  long au_bytes_left;
  boolean au_start,
          au_on_time,
          underflow;
};


struct buffer_queue
{
  long          number;
  unsigned int  size;
  long          au_length;
  double        DTS;
  buffer_ptr    next;
};


PMT_stream_ptr ES[MAX_ES]; /* Sorted vector, used in the multiplexing scheme.*/


/* /// Data structures for ES info. //// */

/* Video */
static double picture_rates [9] = { 0., 24000./1001., 24., 25., 30000./1001.,
        30., 50., 60000./1001., 60. };

static double ratio [16] = { 0., 1., 0.6735, 0.7031, 0.7615, 0.8055,
        0.8437, 0.8935, 0.9157, 0.9815, 1.0255, 1.0695, 1.0950, 1.1575,
        1.2015, 0.};

/* Audio */
static unsigned int bitrate_index [3][16] =
    {{0,32,64,96,128,160,192,224,256,288,320,352,384,416,448,0},
     {0,32,48,56,64,80,96,112,128,160,192,224,256,320,384,0},
     {0,32,40,48,56,64,80,96,112,128,160,192,224,256,320,0}};

static double frequency [4] = {44.1, 48, 32, 0};
static unsigned int slots [4] = {12, 144, 0, 0};
static unsigned int samples [4] = {384, 1152, 0, 0};

static char mode [4][15] =
    { "stereo", "joint stereo", "dual channel", "single channel" };
static char copyright [2][20] =
    { "no copyright","copyright protected" };
static char original [2][10] =
    { "copy","original" };
static char emphasis [4][20] =
    { "none", "50/15 microseconds", "reserved", "CCITT J.17" };


/* /// Function prototypes ///////////// */

void    Program_Use                    (void);
void    Expands_meta_ch                (int *, char ***);
char   *Obtains_path                   (char *);
void    NameOut                        (char *, char *, char *);
void    FindSwitch                     (int *, char ***);
void    Warn                           (char *);
void    TRANSPORT_STREAM               (FILE *, FILE *);
void    Defaults                       (void);
void    Read_manager                   (FILE *);
```

```
int      Find_PID                         (FILE *, int *);
PAT_program_ptr Look_up_PAT               (int);
void     Reads_PAT                        (FILE *, int *);
void     Reads_PMT                        (int, FILE *, int *);
PMT_stream_ptr  Look_up_PMT               (int);
void     Reads_ES                         (PMT_stream_ptr, FILE *, int *);
long     File_Size                        (FILE *);
void     Get_video_info                   (PMT_stream_ptr, double *);
void     Get_audio_info                   (PMT_stream_ptr, double);
void     STARTCODE_prefix                 (FILE *);
void     Create_ES_pointers               (void);
int      Compare_ES                       (const void *, const void*);
void     TP_PAT                           (FILE *);
void     TP_PMT_section                   (int, FILE *);
void     TP_PES                           (PMT_stream_ptr, FILE *, double);
void      TS_ADAPTATION_FIELD                 (PMT_stream_ptr, int , int *, FILE *, int,
boolean, double);
void     TS_PROGRAM_ASSOCIATION_SECTION   (int *, FILE *);
void     TS_PROGRAM_MAP_SECTION           (int, int *, FILE *);
void      PES_PACKET                            (PMT_stream_ptr, int *, FILE *, boolean,
double, double);
void     TP_NULL                          (FILE *);
void     buffer                           (PMT_stream_ptr, long);
void     clean_buffers                    (double);
void     buffer_fullness                  (PMT_stream_ptr);

/* //////////////////////////////////////// */


void
Program_Use()
/*************************************
 ISO 13818-1 streams multiplexer.
**************************************/
{
  fprintf(stderr,"------------------------------------------------------------\n");
   fprintf(stderr,"MPEG-2 System Transport MUltipleXer: ISO/IEC 13818-1 (NO721 - June
94)\n\n");
  fprintf(stderr,"Arguments: [-(switch)...] filename [filename ...] (accepts *)\n");
  fprintf(stderr,"  switch: L, generates a log file with warnings/errors found during
processing\n");
  fprintf(stderr,"          B, displays simulated decoder buffers fullness\n");
  fprintf(stderr,"\nInputs: mux manager files\n");
  fprintf(stderr,"\nOutputs:\n");
    fprintf(stderr," - Binary MPEG2 Transport Stream file ((input filename).(mux
name)).\n");
  fprintf(stderr," - Optional log file ((input filename).LOG).\n");
  fprintf(stderr,"------------------------------------------------------------\n");
  fprintf(stderr,"mux v.25 - Andre'Braganca @1995 - INESC/Programa Ciencia\n");
  fprintf(stderr,"send comments to: amb@bart.inescn.pt\n");

  exit(0);
}



#ifdef DOS

void
Expands_meta_ch(int *argc, char **argv[])
/*************************************
 Expands the metacharacters in the arguments specified in the command line
 (e.g. "*.mpg"). In case of UNIX this is made at shell level.
 Routine adapted from "InfoMPEG @1993 Dennis Lee".
**************************************/
{
  boolean exists_file, expands;
  struct ffblk file_info;
  char file_name[80], path[80];
  register int i, j, new_argc=0;
  char **argv_buffer = (char **)malloc(MAX_FILES * sizeof(char *));


  for(i=0; i < *argc; i++)
  {
    expands = NO; /* There's no wildcard '*' until you find one */
    for(j = strlen((*argv)[i])-1; j >= 0; j--)
    {
      if( (*argv)[i][j] == '*' )
      {
        expands = YES;
        break;
      }
    }

    if(expands)
    {
      strcpy(file_name, (*argv)[i]);
      strcpy(path, Obtains_path((*argv)[i]));

      exists_file = !findfirst(file_name, &file_info, FA_RDONLY);
      while (exists_file)
      {
        argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
        strcpy(argv_buffer[new_argc], path);
        strcat(argv_buffer[new_argc++], file_info.ff_name);
        exists_file = !findnext(&file_info);
      }
    }
    else
    {
      argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
      strcpy(argv_buffer[new_argc++], (*argv)[i]);
    }
  }

  *argc = new_argc;
  *argv = argv_buffer;
}


char *
Obtains_path (char *name)
/*************************************
 Eliminates the file name, leaving its path only.
 Routine adapted from "InfoMPEG @1993 Dennis Lee".
*************************************/
{
  register int i;


  for(i=strlen(name)-1; name[i]!='\\' && name[i]!='/' && name[i]!=':'; i--)
    name[i] = 0;

  return name;
}

#endif


void
NameOut (char *output, char *input, char *executable_name)
/*************************************
 Obtains the output filename by adding/modifying an extension to the input.
 The extension is made from the first 3 (or less) letters of the program name
 without the path.
*************************************/
{
    register int i, j;
    char extension[80];
    /*char no_extension[80]; GLOBAL VAR. !*/


    strcpy(extension, executable_name);
    for(i = strlen(extension) - 1;
        i>=0 && extension[i]!='\\' && extension[i]!='/' && extension[i]!=':';
        i--) ;
    extension[4] = '\0';
    extension[0] = '.';
    for(j=1; j<4; j++)
    {
      extension[j] = executable_name[i+j];
      if(extension[j]=='.')
```

```
        {
          extension[j]='\0';
          break;
        }
      }

      strcpy(output, input);

      for(i=strlen(output)-1; i>=0; i--)
      {
        if(output[i]=='.')
        {
          output[i]='\0';
          break;
        }
      }

      /*Global variable initialization.*/
      strcpy(no_extension, output);

      strcat(output, extension);
}


void
FindSwitch(int *argc, char **argv[])
/************************************************
 Finds (and removes), in the command line arguments, instructions to define the
 program behaviour.
*************************************************/
{
  register int i,
               new_argc=0;
  char **argv_buffer = (char **)malloc(*argc * sizeof(char *));

  /* Switches initialization (defaults) */
  logfile = NO;
  buffer_status = NO;

  /* Search */
  for(i=0; i < *argc; i++)
  {
    if( (*argv)[i][0] == '-') /* switch indicator */
    {
      switch( toupper( (*argv)[i][1] ) )
      {
       case 'L': /* Creates log file (errors and statistics) */
                 logfile = YES;
                 break;
       case 'B': /* Displays simulated decoder buffers fullness */
                 buffer_status = YES;
                 break;
       default : fprintf(stderr,"\nInvalid switch !");
                 Program_Use();
                 break;
      }
    }
    else
    {
      argv_buffer[new_argc] = (char *)malloc(80 * sizeof(char));
      strcpy(argv_buffer[new_argc++], (*argv)[i]);
    }
  }

  *argc = new_argc;
  *argv = argv_buffer;
}


void
Warn(char *str)
/***********************************************
 Prints a warning to stderr and in a log file.
************************************************/
{
  /* boolean log;
```

```
        unsigned long n_TPs;
        char no_extension[80]; GLOBAL VAR. !*/
  char errorstr[80], s[80];
  FILE *file;

  if( logfile )
  {
    sprintf(s,"%s.LOG", no_extension);

    if((file=fopen(s,"at")) == NULL)
    {
        sprintf(errorstr,"\nat - can't open %s (log file)",s);
        perror(errorstr);
        return;
    }
    sprintf(s,"%ld\t! %s\n", n_TPs + 1, str);
    fprintf(file, s);

    fclose(file);
  }

  fprintf(stderr,"\n! ");
  fprintf(stderr, str);
  fprintf(stderr,"\n");
}


main(int argc, char *argv[])
/****************************************
 Multiplexes the binary input files according to ISO 13818-1.
 This files are specified in a manager file which controls the multiplexing
 scheme.
 The manager file is given in the command line. Multiple files may be used to
 generate more than one multiplexed bitstream.
****************************************/
{
  /* boolean log;
     char no_extension[80]; GLOBAL VAR. !*/
  FILE *manager, *out;
  char str[80], s[80];
  int file;


  FindSwitch(&argc, &argv); /* Initializes global variables according to the
                               command line.*/

#ifdef DOS
  Expands_meta_ch(&argc, &argv);
#endif

  if(argc < 2) Program_Use();

  for(file=1; file < argc; file++)
  {
    if((manager=fopen(argv[file],"rt")) == NULL)
    {
      sprintf(str,"\nrt - can't open manager file %s", argv[file]);
      perror(str);
      continue;
    }

    NameOut(str, argv[file], argv[0]);
    if((out=fopen(str,"wb")) == NULL)
    {
      sprintf(s,"\nwb - can't open mux file %s", str);
      perror(s);
      fclose(manager);
      continue;
    }

    /*Buffers, using an indirect call to malloc, for the input and output
      streams in order to minimize disk access.*/
    if (setvbuf(manager, NULL, _IOFBF, 15360) != 0)
      fprintf(stderr,"\n(failed to set up buffer for input file)");
    if (setvbuf(out, NULL, _IOFBF, 15360) != 0)
      fprintf(stderr,"\n(failed to set up buffer for output file)");
```

```
          fprintf(stdout,"\n***   Using   [%s]   to   generate   multiplex   file   [%s]
***\n",argv[file],str);

    if( logfile ) /* file with multiplexing statistics and errors */
    {
      sprintf(s,"%s.LOG", no_extension);
      if(remove(s))
      {
        sprintf(str,"(can't remove %s (log file))", s);
        perror(str);
      }
    }

    TRANSPORT_STREAM(manager, out);


    fclose(manager);
    fclose(out);
  }

  return 0;
}


void
TRANSPORT_STREAM (FILE *manager, FILE *out)
/*****************************************
 Multiplexing management.
*****************************************/
{
  /* unsigned long n_TPs; struct PMT; GLOBAL VAR. ! */
  PMT_stream_ptr stream, ptr;
  double min_video_PTS,
         current_PCR,
         au_delivery,
         next_PSI_delivery;
  unsigned long freq_PSI,
                PSI_rate,
                extra_rate;
  register int i, j;
  int a,
      p,
      TP_payload,
      TP_per_PES_packet,
      adap_field_stuffing,
      TP_oh,
      PSI_flag;
  float overhead;
  unsigned int PES_au_bytes,
               au_bytes;
  boolean done,
          other_au_on_time;
  char str[80];


  /* Initializations */
  Defaults();
  Read_manager(manager);

  /* For each program, scans video and audio streams for access units and
     'struct PMT_stream' information. First video, in order to find the
     smallest video PTS, which is added to all audio PTS. This way, decoded
     audio isn't presented before video. This streams, for testing purposes
     are ISO11172 compliant.
     This is not standard procedure and can be modified. */
  for(i = 0; i < PAT.n_progs; i++)
  {
    printf("\n*** Program %d ***\n", i+1);

    stream = PMT[i].stream;
    while(stream != NULL)
    {
      if(stream->info_file != NULL) /*Is video or audio (defined in manager file)*/
      {
        if(stream->id >= 0xE0 && stream->id <= 0xEF) /* video */
        {
```

```
        Get_video_info(stream, &min_video_PTS);

        /* Contribution to program rate */
        a = stream->PES_packet_length;
        if(stream->elementary_PID == PMT[i].PCR_PID)
          p = ADAP_FIELD_W_PCR;
        else
          p = 0;
        TP_payload = 188 - TP_HEADER - p;
        TP_per_PES_packet = (int)ceil((double)a / (double)TP_payload);
        adap_field_stuffing = TP_payload - (a % TP_payload);

        stream->TP_payload  = TP_payload;
        stream->PES_payload = a - PES_OH_VIDEO;

        TP_oh = (TP_HEADER + p)*TP_per_PES_packet + adap_field_stuffing;

        overhead = (float)(a + TP_oh) / (float)(a - PES_OH_VIDEO);

        PMT[i].prog_rate += stream->rate * overhead;
      }
    }
    else /* Other types of data */
    {
      fprintf(stderr,"\nError scaning bitstream (PID = %d; stream_id = %d).\n",
                        stream->elementary_PID, stream->id);
      fprintf(stderr,"- For the moment, only video and audio is supported !\n");
      fprintf(stderr,"- Edit manager file and delete related info.");
      exit(1);
    }

    stream = stream->next;
  }

  stream = PMT[i].stream;
  while(stream != NULL)
  {
    if(   stream->info_file != NULL  /* Is video or audio */
       && stream->id >= 0xC0 && stream->id <= 0xDF) /* audio */
    {
      Get_audio_info(stream, min_video_PTS);

      /* Contribution to program rate */
      a = stream->PES_packet_length;
      if(stream->elementary_PID == PMT[i].PCR_PID)
        p = ADAP_FIELD_W_PCR;
      else
        p = 0;
      TP_payload = 188 - TP_HEADER - p;
      TP_per_PES_packet = (int)ceil((double)a / (double)TP_payload);
      adap_field_stuffing = TP_payload - (a % TP_payload);

      stream->TP_payload  = TP_payload;
      stream->PES_payload = a - PES_OH_AUDIO;

      TP_oh = (TP_HEADER + p)*TP_per_PES_packet + adap_field_stuffing;

      overhead = (float)(a + TP_oh) / (float)(a - PES_OH_AUDIO);

      PMT[i].prog_rate += stream->rate * overhead;
    }
    stream = stream->next;
  }

  /* Contribution to TS rate */
  PAT.TS_rate += PMT[i].prog_rate;

  /* Startup delay (added to PTS and DTS)*/
  printf("\n Startup delay added to PTS and DTS (90 khz cycles): ");
  scanf("%lg", &(PMT[i].startup_delay));
}


/* TS_rate */
printf("\n\n*** TS ***\n\n Frequency of PSI (1/sec): ");
scanf("%U", &freq_PSI);
PSI_rate = (1 + PAT.n_progs)* 188 * freq_PSI;
```

```
printf(" PSI rate = %lu bytes/s  (1 TP w/PAT and 1 TP for each PMT section)",
        PSI_rate);

PAT.TS_rate += PSI_rate;

printf("\n\n Computed TS_rate = %lu bytes/s", PAT.TS_rate);
printf("\n\n Specify an extra rate for NULL TPs, CA, FEC, ... (bytes/s):");
scanf("%U", &extra_rate);

PAT.TS_rate += extra_rate;
printf("\n Target TS_rate = %lu bytes/s\n\n *** Any Key ***\n", PAT.TS_rate);
scanf("%*c");
scanf("%*c");


/* Vector w/ sorted stream pointers. Highest rate first. Sets PAT.n_streams */
Create_ES_pointers();


/* Multiplexing scheme */

TP_PAT(out);
n_TPs = 1;
printf("TS_packets:\nl\t\a");
next_PSI_delivery = SYSTEM_CLK / freq_PSI;

/* This scheme sends one program definition per TP. */
for(i = 0; i < PAT.n_progs; i++)
{
  TP_PMT_section(i, out);
  n_TPs++;
  printf("%ld\t", n_TPs);
}
PSI_flag = 0;

/* Reads in first access unit information */
done = YES;
for(i=0; i < PAT.n_streams; i++)
{
  fread(&(ES[i]->au), sizeof(struct access_unit), 1, ES[i]->info_file);
  ES[i]->au.PTS += ES[i]->PMT->startup_delay;
  ES[i]->au.DTS += ES[i]->PMT->startup_delay;
  ES[i]->au_bytes_left = ES[i]->au.length;
  ES[i]->au_start = YES;
  if(ES[i]->offset) done = NO;
}


/* Multiplexing Scheme */
while(done == NO)
{
  /* One may use the same clock for all programs */

  current_PCR = (188*n_TPs + PCR_OFFSET) * SYSTEM_CLK / PAT.TS_rate;
  /* used for comparison with next_PSI_delivery and in TP_PES() */

  clean_buffers(current_PCR); /* If DTS < PCR (already decoded) */

  other_au_on_time = YES;
  for(i=0; i < PAT.n_streams; i++)
  {
    if(ES[i]->offset == 0)
    {
      ES[i]->au_on_time = YES;
      continue;
    }

    /* Minimum bytes necessary to transmit the au (or what's left). */

    PES_au_bytes = floor(ES[i]->au_bytes_left / ES[i]->PES_payload)
                   * ES[i]->PES_packet_length
                   + (ES[i]->au_bytes_left % ES[i]->PES_payload)
                   + (ES[i]->PES_packet_length - ES[i]->PES_payload);

    au_bytes = floor(PES_au_bytes / ES[i]->TP_payload) * 188
               + (PES_au_bytes % ES[i]->TP_payload)
               + (188 - ES[i]->TP_payload);
    /* Checks if this au CAN be delivered on time, even with a different TP
       in between. */
    au_delivery = (188*n_TPs + 188 + au_bytes) * BASE_CLK / PAT.TS_rate;

    if(au_delivery <= ES[i]->au.DTS || ES[i]->offset == 0)
      ES[i]->au_on_time = YES;
    else
    {
      ES[i]->au_on_time = NO;
      other_au_on_time  = NO;
    }
  }

  /* PSI multiplex (PAT followed by PMT) */
  if(PSI_flag == 0  &&  next_PSI_delivery <= current_PCR)  PSI_flag = 1;
  if(other_au_on_time  &&  PSI_flag) /*Timely PSI insertion depends on ES.*/
  {
    if(PSI_flag == 1) /* PAT */
    {
      TP_PAT(out);
      n_TPs++;
      printf("%ld\t", n_TPs);

      PSI_flag++;

      next_PSI_delivery += SYSTEM_CLK / freq_PSI;

      if(next_PSI_delivery <= current_PCR)
      {
        j = 0;
        while(next_PSI_delivery <= current_PCR)
        {
          next_PSI_delivery += SYSTEM_CLK / freq_PSI;
          j++;
        }
        sprintf(str,"Missed %d PSI deliverie(s)", j);
        Warn(str);
      }
    }
    else
    {
      TP_PMT_section(PSI_flag - 2, out);
      n_TPs++;
      printf("%ld\t", n_TPs);

      PSI_flag++;
      if(PSI_flag == PAT.n_progs + 2)  PSI_flag = 0;
    }

    done = YES;
  }

  /* Normal multiplex */
  for(i=0; i < PAT.n_streams; i++)
  {
    if(done == YES) break;
    if(ES[i]->offset == 0)  continue;

    other_au_on_time = YES;
         for(j = 0; j < PAT.n_streams; j++)
    {
      if(j == i) continue;
         if(ES[j]->au_on_time == NO)
      {
        other_au_on_time = NO;
        break;
      }
    }

    if( ES[i]->buffer_space >= 184 /* ES[i]->TP_payload, max. value */
       &&
        other_au_on_time == YES )
    {
      TP_PES(ES[i], out, current_PCR);
      n_TPs++;
      printf("%ld\t", n_TPs);
```

```
        done = YES;
      }
    }

    /* Multiplex w/timing violation */
    /*Could be for(i = PAT.n_streams - 1; i >= 0; i--) Lowest rate first.*/
    for(i=0; i < PAT.n_streams; i++)
    {
      if(done == YES) break;

      if( ES[i]->buffer_space >= 184 /* ES[i]->TP_payload, max. value */
         &&
          ES[i]->offset
         &&
          ES[i]->au_on_time == NO )
      {
        if(ES[i]->underflow == NO)
        {
          sprintf(str,"PID = 0x%04X, stream_id = 0x%02X: AU %ld may be late (delivery
time)",
                     ES[i]->elementary_PID, ES[i]->id, ES[i]->au.number);
          Warn(str);
        }

        TP_PES(ES[i], out, current_PCR);
        n_TPs++;
        printf("%ld\t", n_TPs);
        done = YES;
      }
    }

    /* Padding multiplex */
    if(done == NO)
    {
      Warn("Padding");
      TP_NULL(out);
      n_TPs++;
      printf("%ld\t", n_TPs);
      done = YES;
    }

    for(i=0; i < PAT.n_streams; i++)
    {
      if(ES[i]->offset)
      {
        done = NO;
        break;
      }
    }
  }
  printf("\a");
}


void
Defaults ()
/**************************************
 **************************************/
{
  register int i;

  PAT.transport_priority           = 0;
  PAT.transport_scrambling_control = 0;
  PAT.reserved                     = 0;
  PAT.transport_stream_id          = 0;
  PAT.program                      = NULL;
  PAT.n_progs                      = 0;
  PAT.n_streams                    = 0;
  PAT.TS_rate                      = 0;


  for(i=0; i < MAX_PMT_SECTIONS; i++)
  {
    PMT[i].transport_priority           = 0;
    PMT[i].transport_scrambling_control = 0;
    PMT[i].reserved                     = 0;
    /*  PMT[i].PID                          = 10;  see Read_manager() */
    /*  PMT[i].program_number                = 1;    see Read_manager() */
    PMT[i].PCR_PID                        = 0x1FFF;
    PMT[i].last_PCR                       = SYSTEM_CLK; /* High value */
    PMT[i].startup_delay                  = 0;
    PMT[i].stream                         = NULL;
    PMT[i].n_streams                      = 0;
    PMT[i].prog_rate                      = 0;
  }
}


void
Read_manager (FILE *manager)
/****************************************
 Reads a file with PSI definition and TPs management info.
 ****************************************/
{
  /* struct PAT; GLOBAL VAR. */
  int lines = 0, /* lines read */
      PID;
  PAT_program_ptr program;
  PMT_stream_ptr  stream;
  register int i;

  PID = Find_PID(manager, &lines);

  if(PID == 0) Reads_PAT(manager, &lines);
  else
  {
    fprintf(stderr,"\nError in manager file, line %d.\n", lines);
    fprintf(stderr,"- PAT info not at begining !\n");
    fprintf(stderr,"> Edit file and put info related to PID = 0 first.");
    exit(1);
  }

  /* PMT */
  for(i = 0; i < PAT.n_progs; i++)
  {
    PID = Find_PID(manager, &lines);
    if(PID < 0)
    {
      fprintf(stderr,"\nError in manager file, line %d.\n", lines);
           fprintf(stderr,"- PMT section %d not defined (%d programs)!\n", i+1,
PAT.n_progs);
      fprintf(stderr,"> Edit file and insert PMT related info.");
      exit(1);
    }

    /* Because it's more simple and it's decided that a TP carries just one
       program definition, one assumes, for the time being, that PMT section
       PIDs are all diferent. */
    program = Look_up_PAT(PID);

    if(program != NULL && program->number == 0)
    {
      fprintf(stderr,"\nError in manager file, line %d.\n", lines);
      fprintf(stderr,"- Network Information Table not supported !\n");
      fprintf(stderr,"> Edit file and delete related info (program_number = 0).");
      exit(1);
    }

    PMT[i].PID = PID;
    PMT[i].program_number = program->number; /* See Reads_PMT() */

    if(program != NULL) Reads_PMT(i, manager, &lines);
    else
    {
      fprintf(stderr,"\nError in manager file, line %d.\n", lines);
      fprintf(stderr,"- PMT section %d not after PAT (PID not found in PAT)!\n", i+1);
      fprintf(stderr,"> Edit file and put info related to PMT after PAT info.");
      exit(1);
    }
  }

  while(!feof(manager))
  {
    PID = Find_PID(manager, &lines);
    if(PID < 0) break;

    stream = Look_up_PMT(PID);

    if(stream != NULL) Reads_ES(stream, manager, &lines);
    else
    {
      fprintf(stderr,"\nWarning in manager file, line %d.\n", lines);
      fprintf(stderr,"- PID 0x%04X (%d) not found in PMT !\n", PID, PID);
       fprintf(stderr,"> Edit file and delete related info. in order to suppress this
warning.\n");
    }
  }

  /*Verify if all bitstream files are open.*/
  for(i = 0; i < PAT.n_progs; i++)
  {
    stream = PMT[i].stream;
    while(stream != NULL)
    {
      if(stream->file == NULL)
      {
        PID = stream->elementary_PID;
        fprintf(stderr,"\nError in manager file:\n- Data for PID ");
        fprintf(stderr,"0x%04X (%d) hasn't been found !\n", PID, PID);
        fprintf(stderr,"> Edit file and add related info.");
        exit(1);
      }
      stream = stream->next;
    }
  }
}


int
Find_PID(FILE *manager, int *lines)
/*************************************
 Looks for "| PID = xpto" in file. Returns xpto when successful, -1 otherwise.
*************************************/
{
  int i;
  char str[255], value[16];

  while( fgets(str, 255, manager) != NULL )
  {
    (*lines)++;

    if( !strncmp("| PID =", str, 7) )
    {
      sscanf(str,"%*c %*s %*c %s", value);
      if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
        sscanf(value,"%x", &i);
      else
        sscanf(value,"%d", &i);
      return i;
    }
  }

  return -1; /* Impossible value of PID */
}


PAT_program_ptr
Look_up_PAT (int PID)
/*************************************
 Searches PAT (GLOBAL VAR.) looking for the PID given as an argument.
 Returns a pointer to the found table entry (or NULL).
*************************************/
{
  /* PAT_program_ptr PAT.program; GLOBAL VAR. ! */
  PAT_program_ptr program;


  program = PAT.program;

  while(program != NULL)
  {
    if(program->PID == PID)  return program;

    program = program->next;
  }

  return NULL;
}


void
Reads_PAT(FILE *manager, int *lines)
/************************************
 ************************************/
{
  /* struct PAT; GLOBAL VAR. */
  boolean end = NO;
  int i;
  char str[255], s[80], value[16], c;
  PAT_program_ptr program;

  while( fgets(str, 255, manager) != NULL  &&  !end )
  {
    (*lines)++;

    /* Manager file syntax */
    if(str[0] == ' ' && str[1] == '-') break;
    if(str[0] != '|' || str[1] != ' ') continue;

    sscanf(str,"%*c %s %c %s", s, &c, value);
    if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
      sscanf(value,"%x", &i);
    else
      sscanf(value,"%d", &i);

    /* Manager file syntax */
    if(c != '=') continue;

    if( !strcmp("transport_priority", s) )
      PAT.transport_priority = (char)i;
    else
    if( !strcmp("transport_scrambling_control", s) )
      PAT.transport_scrambling_control = (char)i;
    else
    if( !strcmp("reserved", s) )
      PAT.reserved = (char)i;
    else
    if( !strcmp("transport_stream_id", s) )
      PAT.transport_stream_id = i;
    else
    if( !strcmp("program_number", s) )
    {
      if(PAT.program == NULL)
      {
           if((PAT.program  =  (PAT_program_ptr)malloc(sizeof(struct  PAT_program)))  ==
NULL)
        { Warn("Reads_PAT(): malloc error"); exit(1); }

        program = PAT.program;
      }
      else
      {
         if((program->next  =  (PAT_program_ptr)malloc(sizeof(struct  PAT_program)))  ==
NULL)
        { Warn("Reads_PAT(): malloc error"); exit(1); }

        program = program->next;
      }

      /* defaults */
      program->number = i;
      program->reserved = 0;
      program->PID = -1; /* invalid value */
      program->next = NULL;

      while( fgets(str, 255, manager) != NULL )
      {
        (*lines)++;

        /* Manager file syntax */
        if(str[0] == ' ' && str[1] == '-') { end = YES; break; }
        if(str[0] != '|') continue;
        if(str[1] == '\n') break;
        if(str[1] != ' ') continue;

        s[0] = '\0';
        sscanf(str,"%*c %s %c %s", s, &c, value);
        if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
          sscanf(value,"%x", &i);
        else
          sscanf(value,"%d", &i);

        /* Manager file syntax */
        if(s[0] == '\0') break;
        if(c != '=') continue;

        if( !strcmp("reserved", s) )
          program->reserved = (char)i;
        else
        if( !strcmp("program_map_PID", s) && program->number ||
            !strcmp("network_PID", s) && !(program->number)
          )
          program->PID = i;
      }

      if( program->PID < 0)
      {
        fprintf(stderr,"\nError in manager file, line %d.\n", (*lines)-1);
        if(program->number)
        {
          fprintf(stderr,"- PAT info missing a program_map_PID !\n");
          fprintf(stderr,"> Edit file and insert program_map_PID after");
          fprintf(stderr," program_number = %d.",program->number);
        }
        else
        {
          fprintf(stderr,"- PAT info missing network_PID !\n");
            fprintf(stderr,"> Edit file and insert network_PID after program_number =
0.");
        }
        exit(1);
      }
    }
  }

  if(PAT.program == NULL)
  {
    fprintf(stderr,"\nError in manager file, before line %d.\n", *lines);
    fprintf(stderr,"- No programs described in PAT !\n");
    fprintf(stderr,"> Edit file and define program_number.");
    exit(1);
  }
  else
  {
    program = PAT.program;
    while(program != NULL)
    {
      (PAT.n_progs)++;
      program = program->next;
    }
    if(PAT.n_progs > MAX_PMT_SECTIONS)
    {
      fprintf(stderr,"\nError in manager file, before line %d.\n", *lines);
      fprintf(stderr,"- There are more programs than allowed !\n");
      fprintf(stderr,"> Compile source code w/higher value of MAX_PMT_SECTIONS.");
      exit(1);
    }
  }
}

void
Reads_PMT(int section, FILE *manager, int *lines)
/****************************************
```

```
 For the moment there aren't any descriptors !
 ***************************************/
{
  boolean end = NO;
  int i;
  char str[255], s[80], value[16], c;
  PMT_stream_ptr stream;

  while( fgets(str, 255, manager) != NULL  &&  !end )
  {
    (*lines)++;

    /* Manager file syntax */
    if(str[0] == ' ' && str[1] == '-') break;
    if(str[0] != '|' || str[1] != ' ') continue;

    sscanf(str,"%*c %s %c %s", s, &c, value);
    if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
      sscanf(value,"%x", &i);
    else
      sscanf(value,"%d", &i);

    /* Manager file syntax */
    if(c != '=') continue;

    if( !strcmp("transport_priority", s) )
      PMT[section].transport_priority = (char)i;
    else
    if( !strcmp("transport_scrambling_control", s) )
      PMT[section].transport_scrambling_control = (char)i;
    else
/* Uses the value defined in the PAT (see Read_manager())
    if( !strcmp("program_number", s) )
      PMT[section].program_number = i;
    else
*/
    if( !strcmp("reserved", s) )
      PMT[section].reserved = (char)i;
    else
    if( !strcmp("PCR_PID", s) )
      PMT[section].PCR_PID = i;
    else
    if( !strcmp("stream_type", s) )
    {
      if(PMT[section].stream == NULL)
      {
          if((PMT[section].stream = (PMT_stream_ptr)malloc(sizeof(struct PMT_stream)))
== NULL)
        { Warn("Reads_PMT(): malloc error"); exit(1); }

        stream = PMT[section].stream;
      }
      else
      {
        if((stream->next = (PMT_stream_ptr)malloc(sizeof(struct PMT_stream))) == NULL)
        { Warn("Reads_PMT(): malloc error"); exit(1); }

        stream = stream->next;
      }

      /* defaults */
      stream->next                   = NULL;
      stream->PMT                    = &PMT[section];
      stream->type                   = i;
      stream->elementary_PID         = -1; /* invalid value */
      stream->reserved               = 0;
      stream->continuity_counter     = -1;  /* invalid value */
      stream->id                     = -1; /* invalid value */
      stream->PES_packet_length      = 2048;
      stream->PES_scrambling_control = 0;
      stream->PES_priority           = 0;
      stream->copyright              = 0;
      stream->original_or_copy       = 0;
      stream->PES_bytes_left         = 2048;
      stream->file                   = NULL;/*Has to check if bitstream exists.*/
      stream->size                   = 0;
      stream->offset                 = 0;
```

```
        stream->PES_payload            = 0;
        stream->TP_payload             = 0;
        stream->rate                   = 0;
        stream->buffer                 = NULL;
        stream->buffer_size            = 0;
        stream->buffer_space           = 0;
        stream->info_file              = NULL;
        stream->au.number              = 0;
        stream->au.length              = 0;
        stream->au.picture_coding_type = 0;
        stream->au.DTS                 = 0;
        stream->au.PTS                 = 0;
        stream->au_start               = NO;
        stream->au_on_time             = NO;
        stream->underflow              = NO;

        while( fgets(str, 255, manager) != NULL )
        {
          (*lines)++;

          /* Manager file syntax */
          if(str[0] == ' ' && str[1] == '-') { end = YES; break; }
          if(str[0] != '|') continue;
          if(str[1] == '\n') break;
          if(str[1] != ' ') continue;

          s[0] = '\0';
          sscanf(str,"%*c %s %c %d", s, &c, &i);
          sscanf(str,"%*c %s %c %s", s, &c, value);
          if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
            sscanf(value,"%x", &i);
          else
            sscanf(value,"%d", &i);

          /* Manager file syntax */
          if(s[0] == '\0') break;
          if(c != '=') continue;

          if( !strcmp("reserved", s) )
            stream->reserved = (char)i;
          else
          if( !strcmp("elementary_PID", s) )
            stream->elementary_PID = i;
        }

        if( stream->elementary_PID < 0)
        {
          fprintf(stderr,"\nError in manager file, line %d.\n", (*lines)-1);
          fprintf(stderr,"- PMT section %d missing an elementary_PID !\n", section);
          fprintf(stderr,"> Edit file and insert elementary_PID after stream_type.");
          exit(1);
        }
      }
    }

    if(PMT[section].stream == NULL)
    {
      fprintf(stderr,"\nError in manager file, before line %d.\n", *lines);
      fprintf(stderr,"- No streams described in PMT !\n");
      fprintf(stderr,"> Edit file and define stream_type.");
      exit(1);
    }
    else
    {
      stream = PMT[section].stream;
      while(stream != NULL)
      {
        (PMT[section].n_streams)++;
        stream = stream->next;
      }
    }
  }


  PMT_stream_ptr
  Look_up_PMT (int PID)
```

```
/*************************************
 Searches all PMT sections looking for the PID given as an argument.
 Returns a pointer to the found table entry (or NULL).
**************************************/
{
  /* PMT_stream_ptr PMT[section].stream; struct PAT; GLOBAL VAR. ! */
  PMT_stream_ptr stream;
  register int i;


  for(i = 0; i < PAT.n_progs; i++)
  {
    stream = PMT[i].stream;

    while(stream != NULL)
    {
      if(stream->elementary_PID == PID)  return stream;

      stream = stream->next;
    }
  }

  return NULL;
}


void
Reads_ES(PMT_stream_ptr stream, FILE *manager, int *lines)
/*************************************
 *************************************/
{
  int i;
  char str[255], s[80], value[16], c;

  while( fgets(str, 255, manager) != NULL )
  {
    (*lines)++;

    /* Manager file syntax */
    if(str[0] == ' ' && str[1] == '-') break;
    if(str[0] != '|' || str[1] != ' ') continue;

    sscanf(str,"%*c %s %c %s", s, &c, value);
    if(value[0] == '0' && (value[1] == 'x' || value[1] == 'X'))
      sscanf(value,"%x", &i);
    else
      sscanf(value,"%d", &i);

    /* Manager file syntax */
    if(c != '=') continue;

    if( !strcmp("stream_id", s) )
    {
      if(i < 0xBC)
      {
        fprintf(stderr,"\nError in manager file, line %d.\n", *lines);
        fprintf(stderr,"- Invalid stream_id value (< 0xBC) !\n");
        fprintf(stderr,"> Edit file and insert correct value.");
        exit(1);
      }

      stream->id = i;

      sprintf(s,"%X.%X", stream->elementary_PID, i); /* hexadecimal */

      if((stream->file = fopen(s,"rb")) == NULL)
      {
        sprintf(value,"%d.%d", stream->elementary_PID, i); /* decimal */

        if((stream->file = fopen(value,"rb")) == NULL)
        {
          sprintf(str,"\nError in manager file, line %d.\n- Can't open bitstream %s or
%s",*lines,s,value);
          perror(str);
               fprintf(stderr,"> Check manager file for PID and stream_id or check
bitstream filename.");
          exit(1);
        }
      }

      stream->size = File_Size(stream->file);

      if(i < 0xC0 || i > 0xDF && i < 0xE0 || i > 0xEF)/*Not video nor audio.*/
        stream->info_file = NULL;
      else
        if((stream->info_file = tmpfile()) == NULL)
        {
           sprintf(str,"\nError in manager file, line %d.\n- Can't open temporary file
for bitstream info");
          perror(str);
          fprintf(stderr,"> Try to check disk space.");
          exit(1);
        }
    }
    else
    if( !strcmp("PES_packet_length", s) )
    {
      stream->PES_packet_length = i;
      stream->PES_bytes_left = i;
    }
    else
    if( !strcmp("PES_scrambling_control", s) )
      stream->PES_scrambling_control = (char)i;
    else
    if( !strcmp("PES_priority", s) )
      stream->PES_priority = (char)i;
    else
    if( !strcmp("copyright", s) )
      stream->copyright = (char)i;
    else
    if( !strcmp("original_or_copy", s) )
      stream->original_or_copy = (char)i;
  }

  if(stream->id < 0)
  {
    i = stream->elementary_PID;
    fprintf(stderr,"\nError in manager file, before line %d.\n", *lines);
    fprintf(stderr,"- stream_id not defined for PID 0x%04X (%d) !\n",i,i);
    fprintf(stderr,"> Edit file and insert stream_id (0xBC to 0xFF).");
    exit(1);
  }
}


long
File_Size (FILE *file)
/***************************************
 Gets the bitstream size.
***************************************/
{
  long offset, size;

  offset = ftell(file);
  fseek(file, 0, SEEK_END);
  size = ftell(file);
  fseek(file, offset, SEEK_SET);

  return size;
}


void
Get_video_info(PMT_stream_ptr stream, double *min_video_PTS)
/***************************************
 Routine adapted from "mplex @ Christof Moar".
***************************************/
{
  struct sequence_header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned horizontal_size_11_4       :  8;

    unsigned vertical_size_11_8         :  4;
    unsigned horizontal_size_3_0        :  4;

    unsigned vertical_size_7_0           :  8;

    unsigned picture_rate               :  4;
    unsigned pel_aspect_ratio           :  4;

    unsigned bit_rate_17_10             :  8;

    unsigned bit_rate_9_2               :  8;

    unsigned vbv_buffer_size_9_5        :  5;
    unsigned marker_bit                 :  1;
    unsigned bit_rate_1_0               :  2;

    unsigned etc                        :  2;
    unsigned constrained_parameter_flag :  1;
    unsigned vbv_buffer_size_4_0        :  5;

#else
    unsigned horizontal_size_11_4       :  8;

    unsigned horizontal_size_3_0        :  4;
    unsigned vertical_size_11_8         :  4;

    unsigned vertical_size_7_0           :  8;

    unsigned pel_aspect_ratio           :  4;
    unsigned picture_rate               :  4;

    unsigned bit_rate_17_10             :  8;

    unsigned bit_rate_9_2               :  8;

    unsigned bit_rate_1_0               :  2;
    unsigned marker_bit                 :  1;
    unsigned vbv_buffer_size_9_5        :  5;

    unsigned vbv_buffer_size_4_0        :  5;
    unsigned constrained_parameter_flag :  1;
    unsigned etc                        :  2;

#endif
  };

  union
  {
    struct sequence_header_bits bits;
    char byte[8];
  } sequence_header;


  struct picture_header_bits
  {
#ifdef DOS
    unsigned temporal_reference_9_2 :  8;

    unsigned etc                    :  3;
    unsigned picture_coding_type    :  3;
    unsigned temporal_reference_1_0 :  2;

#else
    unsigned temporal_reference_9_2 :  8;

    unsigned temporal_reference_1_0 :  2;
    unsigned picture_coding_type    :  3;
    unsigned etc                    :  3;

#endif
  };

  union
  {
    struct picture_header_bits bits;
    char byte[2];
  } picture_header;



  register int i,
               byte;

  FILE *in = stream->file,
       *out= stream->info_file;

  double         pictures_per_second;
  double         cycles_per_picture;
  long           stream_length  = 0;
  long           stream_offset  = 0;
  unsigned long  decoding_order = 0;
  unsigned long  GOP_decoding_order;
  unsigned int   progress          ;
  long au_number = 0;

  struct
  {
    long          offset          ;
    unsigned int  n_sequence_headers;
    unsigned int  n_sequence_ends ;
    unsigned int  n_pictures          ;
    unsigned int  n_groups           ;
    unsigned int  num_frames[4]  ;
    unsigned long avg_frames[4]  ;
    unsigned int  picture_rate          ;
    unsigned int  bit_rate             ;
    unsigned int  computed_bit_rate;
  } info;

  struct access_unit au;

  info.n_sequence_headers = 1;
  info.n_sequence_ends    = 0;
  info.n_pictures          = 0;
  info.n_groups           = 0;
  for(i=0;i<4;i++)
  {
    info.num_frames[i] = 0;
    info.avg_frames[i] = 0;
  }
  au.picture_coding_type = 0; /*invalid value*/



  printf("\nScanning video stream w/PID = 0x%X (%d) and stream_id = 0x%X (%d).\n\n",
            stream->elementary_PID,stream->elementary_PID,stream->id,stream->id);

  STARTCODE_prefix(in);
  byte = getc(in);
  if(byte != SEQUENCE_HEADER)
  {
    fprintf(stderr,"\nError in stream w/PID = 0x%X (%d) and stream_id = 0x%X (%d).\n",
                      stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
    fprintf(stderr,"- Doesn't start with video sequence_header_code.\n");
    fprintf(stderr,"> Edit manager file and set stream_id to a non video value.");
    exit(1);
  }

  /* read sequence header */
  for(i=0; i<8; i++)  sequence_header.byte[i]=(char)getc(in);
  if( feof(in) )
  {
     fprintf(stderr,"\nError in video stream w/PID = 0x%X (%d) and stream_id = 0x%X
(%d).\n",
                      stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
    fprintf(stderr,"- Premature EOF in sequence header.\n");
    fprintf(stderr,"> Replace file by a non corrupted bitstream.");
    exit(1);
  }
  if( ! sequence_header.bits.marker_bit )
  {
     fprintf(stderr,"\nWarning in video stream w/PID = 0x%X (%d) and stream_id = 0x%X
(%d).\n",
                       stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
    fprintf(stderr,"- Invalid marker bit in sequence header.\n");
    fprintf(stderr,"> Replace file by a valid bitstream.\n\n");
  }

  i = sequence_header.bits.picture_rate;
  info.picture_rate = i;
  if(i>0 && i<9)
  {
    pictures_per_second = picture_rates[i];
    cycles_per_picture = BASE_CLK/pictures_per_second;
  }
  else
    cycles_per_picture = 0; /* invalid picture_rate; PTS and DTS = 0 */

  *min_video_PTS = 999999999999;

  STARTCODE_prefix(in);
  byte = getc(in);

  while( !feof(in) )
  {
    switch(byte)
    {
     case SEQUENCE_HEADER:
          info.n_sequence_headers++;
          break;

     case GROUP_START:
          info.n_groups++;
          GOP_decoding_order = 0;
          break;

     case PICTURE_START:
          /* skip access unit number 0 */
          if(au.picture_coding_type != 0)
          {
            stream_length = ftell( in ) - 4;/*minus the start_code*/
            au.length = stream_length - stream_offset;
            stream_offset = stream_length;

            au.number = (++au_number);
            fwrite (&au, sizeof(struct access_unit), 1, out);

            info.avg_frames[au.picture_coding_type-1]+=au.length;
          }

          /* read picture header */
          for(i=0; i<2; i++)  picture_header.byte[i]=(char)getc(in);
          if( feof(in) )
          {
            fprintf(stderr,"\nError in video stream w/PID = 0x%X (%d) and stream_id =
0x%X (%d).\n",
                       stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
            fprintf(stderr,"- Premature EOF in picture header.\n");
            fprintf(stderr,"> Replace file by a non corrupted bitstream.");
            exit(1);
          }

          au.picture_coding_type = picture_header.bits.picture_coding_type;

          /* Display order in GOP */
          i = (picture_header.bits.temporal_reference_9_2 << 2)
             + picture_header.bits.temporal_reference_1_0;

          au.DTS = decoding_order * cycles_per_picture;
          au.PTS = (i+1 - GOP_decoding_order + decoding_order) * cycles_per_picture;

          /* The smallest PTS of all video AUs is to be added to audio PTS */
          *min_video_PTS = (au.PTS < *min_video_PTS ? au.PTS : *min_video_PTS);
```

```
            decoding_order++;
            GOP_decoding_order++;

            if((au.picture_coding_type>0) && (au.picture_coding_type<5))
              info.num_frames[au.picture_coding_type-1]++;

            progress = (ftell(in)*100)/stream->size;
            info.n_pictures++;
            printf("Picture headers:%8d (%2d%%)\r",info.n_pictures,progress);

            break;

        case SEQUENCE_END:
            stream_length = ftell (in);
            au.length = stream_length - stream_offset;
            stream_offset = stream_length;

            au.number = (++au_number);
            fwrite(&au, sizeof (struct access_unit), 1, out);

            info.avg_frames[au.picture_coding_type-1]+=au.length;
            info.n_sequence_ends++;

            break;
      }

      STARTCODE_prefix(in);
      byte = getc(in);
    }

    printf("File length   = %ld              \nStream length = %ld\n\n",
                                              stream->size, stream_offset);

    info.offset = stream_offset;
    stream->offset = stream_offset;

    for(i=0; i<4; i++)
      if(info.num_frames[i] != 0) info.avg_frames[i] /= info.num_frames[i];

    /* bit_rate in units of 400 bits/sec rounded upwards,
       to obtain bytes/sec just multiply by 50.
       If equal to 0x3FFFF identifies variable bit rate operation. */
    info.bit_rate = (sequence_header.bits.bit_rate_17_10 << 10)
                   +(sequence_header.bits.bit_rate_9_2   << 2)
                   + sequence_header.bits.bit_rate_1_0;

    if(cycles_per_picture > 0)
    {
      i = (double)(info.offset)/(double)(info.n_pictures);/*avg_picture_size*/
      /* bytes/sec * 1/50 */
      info.computed_bit_rate = ceil(i * pictures_per_second / 1250) * 25;
    }
    else info.computed_bit_rate = 0;


    /* Display video info */

    printf("sequence_start_codes\t:%8u\n", info.n_sequence_headers);
    printf("sequence_end_codes  \t:%8u\n", info.n_sequence_ends);
    printf("GOPs                \t:%8u\n", info.n_groups);
    printf("Pictures            \t:%8u\n", info.n_pictures);
    printf("\t\tI\t:%8u -> avg. size = %6u bytes\n",
                                  info.num_frames[0], info.avg_frames[0]);
    printf("\t\tP\t:%8u -> avg. size = %6u bytes\n",
                                  info.num_frames[1], info.avg_frames[1]);
    printf("\t\tB\t:%8u -> avg. size = %6u bytes\n",
                                  info.num_frames[2], info.avg_frames[2]);
    printf("\t\tD\t:%8u -> avg. size = %6u bytes\n\n",
                                  info.num_frames[3], info.avg_frames[3]);

    printf("Horizontal size:%8u\n",(sequence_header.bits.horizontal_size_11_4 <<4)
                            + sequence_header.bits.horizontal_size_3_0);
    printf("Vertical size  :%8u",(sequence_header.bits.vertical_size_11_8 <<8)
                            + sequence_header.bits.vertical_size_7_0);

    i = sequence_header.bits.pel_aspect_ratio;
```

```
 printf("     Aspect ratio :  %1.4f ", ratio[i]);
 switch( i )
 {
  case  0: printf ("(forbidden)\n"); break;
  case  1: printf ("(VGA etc)\n"); break;
  case  3: printf ("(16:9, 625 line)\n"); break;
  case  6: printf ("(16:9, 525 line)\n"); break;
  case  8: printf ("(CCIR601, 625 line)\n"); break;
  case 12: printf ("(CCIR601, 525 line)\n"); break;
  case 15: printf ("(reserved)\n"); break;
  default: printf ("\n");
 }
 printf("\n");

 if(info.picture_rate == 0)
   printf("Picture rate    : invalid\n");
 else
 if(info.picture_rate < 9)
   printf("Picture rate    :    %2.3f frames/sec\n", pictures_per_second);
 else
 printf("Picture rate    : %x reserved\n",info.picture_rate);

 if(info.bit_rate == 0x3FFF)  printf("Bit rate        : variable\n");
 else
 printf("Bit rate          : %8u bytes/sec (%7u bits/sec)\n",
                                    info.bit_rate*50, info.bit_rate*400);
 printf("Computed rate   : %8u bytes/sec (%7u bits/sec)   [based on picture rate]\n",
                   info.computed_bit_rate*50, info.computed_bit_rate*400);
 printf("Vbv buffer size: %8u bytes\n",
                    ( (sequence_header.bits.vbv_buffer_size_9_5 <<5)
                    + sequence_header.bits.vbv_buffer_size_4_0) * 2048 );
 printf("CSPF              : %8u\n",sequence_header.bits.constrained_parameter_flag);


 rewind(in);
 rewind(out);


 /* Initialize 'struct PMT_stream' info. */
 if(info.bit_rate > info.computed_bit_rate)
   stream->rate = info.bit_rate * 50;
 else
   stream->rate = info.computed_bit_rate * 50;

 printf("\n STD video buffer size (KB): ");/*Will be implemented in manager file!*/
 scanf("%D", &(stream->buffer_size));
 stream->buffer_size *= 1024;
 stream->buffer_space = stream->buffer_size;
}


void
STARTCODE_prefix (FILE *stream)
/*************************************
 Inspects the stream sequentialy until a start_code prefix is found (0x000001).
 Another routine will use the byte, left in the stream, which specifies the
 start_code type.
*************************************/
{
  unsigned long counter=0;

  while(!feof(stream))
  {
    switch(getc(stream))
    {
      case 0x00: counter++;
                 break;
      case 0x01: if (counter >= 2) return;
      default  : counter=0;
    }
  }
}


void
Get_audio_info(PMT_stream_ptr stream, double min_video_PTS)
/*************************************
```

```
 Routine adapted from "mplex @ Christof Moar".
 **************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/

    unsigned syncword_11_4        :  8;

    unsigned protection_bit       :  1;
    unsigned layer                :  2;
    unsigned ID                   :  1;
    unsigned syncword_3_0         :  4;

    unsigned private_bit          :  1;
    unsigned padding_bit          :  1;
    unsigned sampling_frequency   :  2;
    unsigned bitrate_index        :  4;

    unsigned emphasis             :  2;
    unsigned original_copy        :  1;
    unsigned copyright            :  1;
    unsigned mode_extension       :  2;
    unsigned mode                 :  2;

#else
    unsigned syncword_11_4        :  8;

    unsigned syncword_3_0         :  4;
    unsigned ID                   :  1;
    unsigned layer                :  2;
    unsigned protection_bit       :  1;

    unsigned bitrate_index        :  4;
    unsigned sampling_frequency   :  2;
    unsigned padding_bit          :  1;
    unsigned private_bit          :  1;

    unsigned mode                 :  2;
    unsigned mode_extension       :  2;
    unsigned copyright            :  1;
    unsigned original_copy        :  1;
    unsigned emphasis             :  2;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[4];
  } header;


  register int i;

  FILE *in = stream->file,
       *out= stream->info_file;

  double         samples_per_second  ;
  long           stream_offset      = 0;
  unsigned long  decoding_order     = 0;
  unsigned int   frame_size          ;
  unsigned int   progress            ;
  unsigned int   skip                ;
  int            byte                ;
  long           au_number          = 0;

  struct
  {
    long          offset         ;
    unsigned int  n_syncwords    ;
    unsigned int  layer          ;
    unsigned int protection_bit ;
    unsigned int bit_rate       ;
    unsigned int num_frames [2] ;
    unsigned int size_frames[2] ;
  } info;

  struct access_unit au;


  info.num_frames[0] = 0;
  info.num_frames[1] = 0;

  printf("\nScanning audio stream w/PID = 0x%X (%d) and stream_id = 0x%X (%d).\n\n",
      stream->elementary_PID,stream->elementary_PID,stream->id,stream->id);

  /* Find syncword */
  byte = getc(in);
  while(!feof(in))
  {
    header.byte[0] = byte;
    header.byte[1] = getc(in);
    i = (header.bits.syncword_11_4 << 4) + header.bits.syncword_3_0;

    if(i == AUDIO_SYNCWORD)  break;
    byte = getc(in);
  }


  if( feof(in) )
  {
    fprintf(stderr,"\nError in stream w/PID = 0x%X (%d) and stream_id = 0x%X (%d).\n",
         stream->elementary_PID,stream->elementary_PID,stream->id,stream->id);
    fprintf(stderr,"- Doesn't contain the audio syncword 0x%X.\n", AUDIO_SYNCWORD);
    fprintf(stderr,"> Edit manager file and set stream_id to a non audio value.");
    exit(1);
  }

  /* Read remainig 2 header bytes */
  for(i = 2; i < 4; i++) header.byte[i] =(char)getc(in);

  if( feof(in) )
  {
     fprintf(stderr,"\nError in audio stream w/PID = 0x%X (%d) and stream_id = 0x%X
(%d).\n",
         stream->elementary_PID,stream->elementary_PID,stream->id,stream->id);
    fprintf(stderr,"- Premature EOF in header.\n");
    fprintf(stderr,"> Replace file by a non corrupted bitstream.");
    exit(1);
  }

  info.protection_bit = header.bits.protection_bit;
  info.n_syncwords = 1;

  if( ! info.protection_bit )
  {
    fprintf(stderr,"\nWarning in audio stream w/PID = 0x%X (%d) and stream_id = 0x%X
(%d).\n",
         stream->elementary_PID,stream->elementary_PID,stream->id,stream->id);
    fprintf(stderr,"- Invalid protection bit in header.\n");
    fprintf(stderr,"> Replace file by a valid bitstream.\n\n");
  }

  info.layer = 3 - header.bits.layer;
  info.bit_rate = bitrate_index[info.layer][header.bits.bitrate_index];
  /* kbit/s */

  samples_per_second = (double)frequency[header.bits.sampling_frequency];

  frame_size =  info.bit_rate / samples_per_second * slots[info.layer];

  info.size_frames[0] = frame_size;
  info.size_frames[1] = frame_size + 1;

  au.length = info.size_frames[header.bits.padding_bit];

  au.PTS = decoding_order * samples[info.layer] /samples_per_second * 90.
        + min_video_PTS;
  au.DTS = au.PTS;

  decoding_order++;

  au.number = (++au_number);
  fwrite(&au, sizeof (struct access_unit), 1, out);

  info.num_frames[header.bits.padding_bit]++;

  do
  {
    skip=au.length-4;

    if (skip & 0x1) getc(in);
    if (skip & 0x2) { getc(in); getc(in); }
    skip=skip>>2;

    for (i=0;i<skip;i++)
    {
      getc(in); getc(in); getc(in); getc(in);
    }

    stream_offset = ftell (in);

    header.byte[0] = getc(in);
    i = (header.bits.syncword_11_4 << 4) + header.bits.syncword_3_0;
    if(i == AUDIO_SYNCWORD)
    {
      if( ! info.protection_bit )
      {
          fprintf(stderr,"\nWarning in audio stream w/PID = 0x%X (%d) and stream_id =
0x%X (%d).\n",
                             stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
        fprintf(stderr,"- Invalid protection bit in header.\n");
        fprintf(stderr,"> Replace file by a valid bitstream.\n\n");
      }

      /* Read remainig 3 header bytes */
      for(i = 1; i < 3; i++) header.byte[i] =(char)getc(in);

      if( feof(in) )
      {
        fprintf(stderr,"\nError in audio stream w/PID = 0x%X (%d) and stream_id = 0x%X
(%d).\n",
                             stream->elementary_PID,stream->elementary_PID,stream->id,stream-
>id);
        fprintf(stderr,"- Premature EOF in header.\n");
        fprintf(stderr,"> Replace file by a non corrupted bitstream.");
        exit(1);
      }

      progress = (ftell(in)*100)/stream->size;
      info.n_syncwords++;
      printf("Frame headers:%8d (%2d%%)\r", info.n_syncwords, progress);

      au.length = info.size_frames[header.bits.padding_bit];

      au.PTS = decoding_order * samples[info.layer] /samples_per_second * 90.
             + min_video_PTS;
      au.DTS = au.PTS;

      decoding_order++;

      au.number = (++au_number);
      fwrite(&au, sizeof (struct access_unit), 1, out);

      info.num_frames[header.bits.padding_bit]++;

      getc(in);
    }
    else break;
  } while ( !feof(in) );

  printf("File length   = %ld                    \nStream length = %ld\n\n",
                                                   stream->size, stream_offset);
  info.offset = stream_offset;
  stream->offset = stream_offset;

  printf("Syncwords\t:%8u\n", info.n_syncwords);
  printf("Frames    \t:%8u  with size %6u bytes\n",
                                     info.num_frames[0], info.size_frames[0]);
  printf("          \t:%8u  with size %6u bytes\n\n",
                                     info.num_frames[1], info.size_frames[1]);
  printf("Layer     \t:%8u\n", info.layer + 1);

  if(info.protection_bit == 0) printf ("CRC checksums  :       yes");
  else  printf ("CRC checksums\t:       no");
  printf("  (protection_bit = %d)\n\n", info.protection_bit);

  if(header.bits.bitrate_index == 0)
    printf ("Bit rate\t:      free\n");
  else if(header.bits.bitrate_index == 0xF)
    printf ("Bit rate\t: reserved\n");
  else
    printf ("Bit rate\t: %8u bytes/sec (%3u kbit/sec)\n",
                                     info.bit_rate*128, info.bit_rate);

  if(header.bits.sampling_frequency == 3)
    printf("Frequency\t: reserved\n");
  else
    printf("Frequency\t:      %2.1f kHz\n", samples_per_second);

  i = header.bits.mode;
  printf("\nMode\t\t: %8u (%s)\n", i, mode[i]);
  printf("Mode extension\t: %8u\n", header.bits.mode_extension);
  i = header.bits.copyright;
  printf("Copyright bit\t: %8u (%s)\n", i, copyright[i]);
  i = header.bits.original_copy;
  printf("Original/Copy\t: %8u (%s)\n", i, original[i]);
  i = header.bits.emphasis;
  printf("Emphasis\t: %8u (%s)\n", i, emphasis[i]);


  rewind(in);
  rewind(out);


  /* Initialize 'struct PMT_stream' info. */
  /* For the moment, one assumes that the rate isn't variable.*/
  stream->rate = info.bit_rate * 128;

  printf("\n STD audio buffer size (KB): ");/*Will be implemented in manager file!*/
  scanf("%D", &(stream->buffer_size));
  stream->buffer_size *= 1024;
  stream->buffer_space = stream->buffer_size;
}


void
Create_ES_pointers()
/*************************************
 Finds total number of elementary streams.
 Defines a sorted vector by the stream rate (depends on function Compare_ES()).
 *************************************/
{
  /* struct PAT; PMT_stream_ptr ES; GLOBAL VAR. */
  PMT_stream_ptr stream;
  register int i, j = 0;

  /* Defines a vector w/pointers to all ES.*/
  for(i = 0; i < PAT.n_progs; i++)
  {
    PAT.n_streams += PMT[i].n_streams;

    if(PAT.n_streams > MAX_ES)
    {
      fprintf(stderr,"\nError in manager file.\n");
      fprintf(stderr,"- There are more streams than allowed (%d)!\n", MAX_ES);
      fprintf(stderr,"> Compile source code w/higher value of MAX_ES.");
      exit(1);
    }
    stream = PMT[i].stream;

    while(stream != NULL)
    {
      ES[j++] = stream;
      stream = stream -> next;
    }

  }

  /* Sorts the vector using the stream rate */
  qsort((void *)ES, j, sizeof(ES[0]), Compare_ES);
}


int
Compare_ES(const void *arg1, const void *arg2)
/*************************************
 The qsort() function (in Create_ES_pointers()) will sort the stream pointer
 in descending order.
 *************************************/
{
  PMT_stream_ptr a, b;

  a = *((PMT_stream_ptr *)arg1);
  b = *((PMT_stream_ptr *)arg2);

  return (b->rate) - (a->rate);
}


void
TP_PAT (FILE *out)
/*************************************
 sync_byte = 1 byte.

 transport_error_indicator = 1 bit      \
 payload_unit_start_indicator = 1 bit    \
 transport_priority = 1 bit               \
 PID = 13 bits                             > 3 bytes.
 transport_scrambling_control = 2 bits    /
 adaptation_field_control = 2 bits       /
 continuity_counter = 4 bits            /

 adaptation_field e/ou data_bytes = until 184 bytes.

 *************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned PID_12_8                     :  5;
    unsigned transport_priority           :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_error_indicator    :  1;

    unsigned PID_7_0                          :  8;

    unsigned continuity_counter           :  4;
    unsigned adaptation_field_control     :  2;
    unsigned transport_scrambling_control :  2;

#else
    unsigned transport_error_indicator    :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_priority           :  1;
    unsigned PID_12_8                     :  5;

    unsigned PID_7_0                          :  8;

    unsigned transport_scrambling_control :  2;
    unsigned adaptation_field_control     :  2;
    unsigned continuity_counter           :  4;
```

```
#endif
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;


  register int i;
  int               tp_bytes = 188; /* Transport_packet byte counter*/



  /* Write sync_byte */
  putc(0x47, out);
  tp_bytes--;


  /* Header */

  /* transport_error_indicator */
  header.bits.transport_error_indicator = 0;

  /* payload_unit_start_indicator */
  header.bits.payload_unit_start_indicator = 1;

  /* transport_priority */
  header.bits.transport_priority = PAT.transport_priority;

  /* Packet Identifier */
  header.bits.PID_12_8 = 0;
  header.bits.PID_7_0  = 0;

  /* transport_scrambling_control */
  header.bits.transport_scrambling_control = PAT.transport_scrambling_control;

  /* adaptation_field_control */
  header.bits.adaptation_field_control = 1;/*payload only*/

  /* continuity_counter */
  header.bits.continuity_counter = 0;/* For the moment ! */

  /* Write 3 bytes header */
  for(i = 0; i < 3; i++) putc(header.byte[i], out);
  tp_bytes -= 3;


  /* payload */
  if(tp_bytes <=0)
  {
    fprintf(stderr,"! Insufficient space for payload (PAT)");
    exit(1);
  }

  TS_PROGRAM_ASSOCIATION_SECTION(&tp_bytes, out);

  if(tp_bytes)
  {
    fprintf(stderr,"\nTP over/underflow (PAT)!");
    exit(1);
  }
}


void
TP_PMT_section (int section, FILE *out)
/***********************************************
 sync_byte = 1 byte.

 transport_error_indicator = 1 bit        \
 payload_unit_start_indicator = 1 bit      \
 transport_priority = 1 bit                 \
 PID = 13 bits                               > 3 bytes.
 transport_scrambling_control = 2 bits      /
 adaptation_field_control = 2 bits         /
 continuity_counter = 4 bits           /

 adaptation_field e/ou data_bytes = until 184 bytes.

 ************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned PID_12_8                    :  5;
    unsigned transport_priority          :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_error_indicator   :  1;

    unsigned PID_7_0                        :  8;

    unsigned continuity_counter          :  4;
    unsigned adaptation_field_control    :  2;
    unsigned transport_scrambling_control :  2;

#else
    unsigned transport_error_indicator   :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_priority          :  1;
    unsigned PID_12_8                    :  5;

    unsigned PID_7_0                        :  8;

    unsigned transport_scrambling_control :  2;
    unsigned adaptation_field_control    :  2;
    unsigned continuity_counter          :  4;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;


  register int i;
  int           tp_bytes = 188; /* Transport_packet byte counter*/



  /* Write sync_byte */
  putc(0x47, out);
  tp_bytes--;


  /* Header */

  /* transport_error_indicator */
  header.bits.transport_error_indicator = 0;

  /* payload_unit_start_indicator */
  header.bits.payload_unit_start_indicator = 1;

  /* transport_priority */
  header.bits.transport_priority = PMT[section].transport_priority;

  /* Packet Identifier */
  i = PMT[section].PID;
  header.bits.PID_12_8 = (i & 0x1F00) >> 8; /*mask 1 1111 0000 0000 */
  header.bits.PID_7_0 = i & 0xFF;           /*mask 0 0000 1111 1111 */

  /* transport_scrambling_control */
                                header.bits.transport_scrambling_control      =
PMT[section].transport_scrambling_control;

  /* adaptation_field_control */
  header.bits.adaptation_field_control = 1;/*payload only*/
  /* continuity_counter */
  header.bits.continuity_counter = 0; /* For the moment ! */

  /* Write header 3 bytes */
  for(i = 0; i < 3; i++) putc(header.byte[i], out);
  tp_bytes -= 3;


  /* payload */
  if(tp_bytes <=0)
  {
    fprintf(stderr,"! Insufficient space for payload (PMT)");
    exit(1);
  }

  TS_PROGRAM_MAP_SECTION(section, &tp_bytes, out);

  if(tp_bytes)
  {
    fprintf(stderr,"\nTP over/underflow (PMT)!");
    exit(1);
  }
}


void
TP_PES (PMT_stream_ptr stream, FILE *out, double PCR)
/*************************************************
 sync_byte = 1 byte.

 transport_error_indicator = 1 bit      \
 payload_unit_start_indicator = 1 bit    \
 transport_priority = 1 bit               \
 PID = 13 bits                             > 3 bytes.
 transport_scrambling_control = 2 bits    /
 adaptation_field_control = 2 bits       /
 continuity_counter = 4 bits            /

 adaptation_field e/ou data_bytes = until 184 bytes.

 *************************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned PID_12_8                     :  5;
    unsigned transport_priority           :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_error_indicator    :  1;

    unsigned PID_7_0                         :  8;

    unsigned continuity_counter           :  4;
    unsigned adaptation_field_control     :  2;
    unsigned transport_scrambling_control :  2;

#else
    unsigned transport_error_indicator    :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_priority           :  1;
    unsigned PID_12_8                     :  5;

    unsigned PID_7_0                         :  8;

    unsigned transport_scrambling_control :  2;
    unsigned adaptation_field_control     :  2;
    unsigned continuity_counter           :  4;

#endif
  };

  union
  {
```

```
  struct header_bits bits;
  char byte[3];
} header;


register int i, unit_start;
int          random_access,
             tp_bytes = 188; /* Transport_packet byte counter*/
long offset;
struct access_unit au;
boolean timestamp_flag,
        PCR_flag;
double PES_PTS,
       PES_DTS,
       PCR_interval;
char str[80];



/* Write sync_byte */
putc(0x47, out);
tp_bytes--;


/* Header */

/* transport_error_indicator */
header.bits.transport_error_indicator = 0;

/* payload_unit_start_indicator */
unit_start = 0;
if(stream->PES_bytes_left == stream->PES_packet_length)  unit_start = 1;
header.bits.payload_unit_start_indicator = unit_start;


/* Checks timestamps (PES_PACKET) and access point (TS_ADAPTATION_FIELD) */
if(  unit_start
    &&
     ( stream->au_start == YES  ||
       stream->au_start == NO && stream->au_bytes_left < stream->PES_payload )
   )
{
  random_access  = 1; /* It's assumed that this happens only w/unit_start.*/
  timestamp_flag = YES;

  if(stream->au_start) /* No part of the au has been transmited.*/
  {
    PES_PTS = stream->au.PTS;
    PES_DTS = stream->au.DTS;
  }
  else /* checks next au */
  {
    offset = ftell(stream->info_file);
    if( fread(&au, sizeof(struct access_unit), 1, stream->info_file) == 1)
    {
      PES_PTS = au.PTS + stream->PMT->startup_delay;
      PES_DTS = au.DTS + stream->PMT->startup_delay;
    }
    else  timestamp_flag = NO;

    fseek(stream->info_file, offset, SEEK_SET);
  }
}
else
{
  random_access  = 0;
  timestamp_flag = NO;
}


/* transport_priority */
header.bits.transport_priority = 0;

/* Packet Identifier */
i = stream->elementary_PID;
header.bits.PID_12_8 = (i & 0x1F00) >> 8; /*mask 1 1111 0000 0000 */
header.bits.PID_7_0 = i & 0xFF;           /*mask 0 0000 1111 1111 */
```

```
  /* PCR management (TS_ADAPTATION_FIELD) */
  PCR_flag = NO;
  if(i == stream->PMT->PCR_PID)
  {
    /* Multiplexed stream semantics restriction:
       PCR coding frequency (0.1 sec). */
    PCR_interval = PCR - stream->PMT->last_PCR;
    if( PCR_interval > 0.1 * SYSTEM_CLK )
    {
      sprintf(str,"Restriction violation in program %d: PCR interval = %g sec (>
0.1).",
                  stream->PMT->program_number, PCR_interval/SYSTEM_CLK);
      Warn(str);
    }

    if(unit_start && random_access) PCR_flag = YES;
    else
    {
      /* Empiric strategy to avoid breaking the restriction. In the limit one
         could ALWAYS transmit a PCR (more overhead). */
      if( 2 * PCR_interval >= 0.1 * SYSTEM_CLK) PCR_flag = YES;
    }

    if(PCR_flag) stream->PMT->last_PCR = PCR;
  }


  /* transport_scrambling_control */
  header.bits.transport_scrambling_control = 0;

  /* adaptation_field_control */
  /* header.bits.adaptation_field_control = 2; adaptation_field only, not used
     in this implementation. */
  if(stream->PES_bytes_left >= (188 - TP_HEADER) && !PCR_flag)
    header.bits.adaptation_field_control = 1;/*payload only*/
  else
    header.bits.adaptation_field_control = 3;/*adaptation_field followed by payload*/

  /* continuity_counter */
  stream->continuity_counter ++;
  if(stream->continuity_counter == 16) stream->continuity_counter = 0;
  header.bits.continuity_counter = stream->continuity_counter;

  /* Write header 3 bytes */
  for(i = 0; i < 3; i++) putc(header.byte[i], out);
  tp_bytes -= 3;


  /* adaptation_field */
  i = header.bits.adaptation_field_control;
  if( i == 2 || i == 3 )
    TS_ADAPTATION_FIELD(stream, i, &tp_bytes, out, random_access, PCR_flag, PCR);


  /* payload */
  if( header.bits.adaptation_field_control == 1 ||
      header.bits.adaptation_field_control == 3     )
  {
    if(tp_bytes <=0)
    {
      fprintf(stderr,"! Insufficient space for payload");
      exit(1);
    }

    PES_PACKET(stream, &tp_bytes, out, timestamp_flag, PES_PTS, PES_DTS);
  }

  if(tp_bytes)
  {
    fprintf(stderr,"\nTP over/underflow !");
    exit(1);
  }

  if(!(stream->PES_bytes_left))
  {
    if(stream->offset > stream->PES_payload) /* last PES packet */
    {
      /* The PES_payload value must be exact */
      i = stream->PES_packet_length - stream->PES_payload;
      stream->PES_packet_length = stream->offset + i;
      stream->PES_payload = stream->PES_packet_length - i;
    }

    stream->PES_bytes_left = stream->PES_packet_length;
  }
}


void
TS_ADAPTATION_FIELD (PMT_stream_ptr stream,
                     int              control,
                     int             *tp_bytes,
                     FILE            *out,
                     int              random_access,
                     boolean          PCR_flag,
                     double           PCR_value)
/*****************************************

 adaptation_field_length = 1 byte

 descontinuity_indicator = 1 bit              \
 random_acess_indicator = 1 bit                \
 elementary_stream_priority_indicator = 1 bit   \
 PCR_flag = 1 bit                                > 1 byte.
 OPCR_flag = 1 bit                              /
 splicing_point_flag = 1 bit                   /
 transport_private_data_flag = 1 bit          /
 adaptation_field_extension_flag = 1 bit     /

 program_clock_reference_base = 33 bits      \
 reserved = 6 bits                            > 6 optional bytes.
 program_clock_reference_extension = 9 bits /

 original_program_clock_reference_base = 33 bits      \
 reserved = 6 bits                                     > 6 optional bytes.
 original_program_clock_reference_extension = 9 bits /

 splice_countdown = 1 optional byte.

 transport_private_data_length = 1 optional byte.
 private_data_byte = n optional bytes.

 adaptation_field_extension_length = 8 bits \
 ltw_flag = 1 bit                            \
 piecewise_rate_flag = 1 bit                  > 2 optional bytes.
 seamless_splice_flag = 1 bit                /
 reserved = 5 bits                          /

 stuffing_byte = n optional bytes.
****************************************/
{
  struct flag_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned adaptation_field_extension_flag      :  1;
    unsigned transport_private_data_flag          :  1;
    unsigned splicing_point_flag                  :  1;
    unsigned OPCR_flag                            :  1;
    unsigned PCR_flag                             :  1;
    unsigned elementary_stream_priority_indicator :  1;
    unsigned random_acess_indicator               :  1;
    unsigned descontinuity_indicator              :  1;

#else
    unsigned descontinuity_indicator              :  1;
    unsigned random_acess_indicator               :  1;
    unsigned elementary_stream_priority_indicator :  1;
    unsigned PCR_flag                             :  1;
    unsigned OPCR_flag                            :  1;
    unsigned splicing_point_flag                 :  1;
    unsigned transport_private_data_flag         :  1;
    unsigned adaptation_field_extension_flag     :  1;

#endif
  };

  union
  {
    struct flag_bits bits;
    char byte;
  } flags;

  struct PCR_bits /* Used to read PCR and OPCR */
  {
    unsigned PCR_base_32_25     :  8;

    unsigned PCR_base_24_17     :  8;

    unsigned PCR_base_16_9      :  8;

    unsigned PCR_base_8_1       :  8;

#ifdef DOS
    unsigned PCR_extension_8    :  1;
    unsigned reserved           :  6;
    unsigned PCR_base_0         :  1;
#else
    unsigned PCR_base_0         :  1;
    unsigned reserved           :  6;
    unsigned PCR_extension_8    :  1;
#endif

    unsigned PCR_extension_7_0 :  8;
  };

  union
  {
    struct PCR_bits bits;
    char byte[6];
  } PCR;


  int adaptation_field_length;


  register int i;
  double aux,
         PCR_base,
         PCR_extension;


  /* Write adaptation_field_length.*/
  if(control == 2) adaptation_field_length = 183;
  else
  {
    adaptation_field_length = 183 - stream->PES_bytes_left;

    if(PCR_flag  &&  adaptation_field_length < 7)  adaptation_field_length = 7;
  }

  if(adaptation_field_length < 0)
  {
    fprintf(stderr,"! adaptation_field_length < 0 !");
    exit(1);
  }

  putc(adaptation_field_length, out);
  (*tp_bytes)--;

  /* 1 stuffing_byte */
  if(!adaptation_field_length) return;


  /*Adaptation field flags.*/
```

```
/* descontinuity_indicator */
flags.bits.descontinuity_indicator = 0;

/* random_acess_indicator */
flags.bits.random_acess_indicator  = random_access;

/* elementary_stream_priority_indicator */
flags.bits.elementary_stream_priority_indicator = 1;

/* PCR_flag */
if(PCR_flag) flags.bits.PCR_flag = 1;
else flags.bits.PCR_flag = 0;

/* OPCR_flag */
flags.bits.OPCR_flag = 0;

/* splicing_point_flag */
flags.bits.splicing_point_flag = 0;

/* transport_private_data_flag */
flags.bits.transport_private_data_flag = 0;

/* adaptation_field_extension_flag */
flags.bits.adaptation_field_extension_flag = 0;

/*Write flags byte.*/
putc(flags.byte, out);
adaptation_field_length--;
(*tp_bytes)--;


/* Program Clock Reference */
if( PCR_flag )
{
  PCR_base = floor(PCR_value / 300);
  PCR_extension = fmod(PCR_value, 300);

  PCR.bits.PCR_base_32_25 = floor(PCR_base / 33554432); /* 2^25 */
  aux = fmod(PCR_base, 33554432);
  PCR.bits.PCR_base_24_17 = floor(aux / 131072);        /* 2^17 */
  aux = fmod(aux, 131072);
  PCR.bits.PCR_base_16_9  = floor(aux / 512);           /* 2^9 */
  aux = fmod(aux, 512);
  PCR.bits.PCR_base_8_1   = floor(aux / 2);
  PCR.bits.PCR_base_0     = fmod(aux, 2);

  PCR.bits.PCR_extension_8   = floor(PCR_extension / 256);
  PCR.bits.PCR_extension_7_0 = fmod(PCR_extension, 256);

  /*Writes PCR 6 bytes.*/
  for(i=0; i<6; i++)  putc(PCR.byte[i], out);
  adaptation_field_length -= 6;
  *tp_bytes -= 6;
}

/* Original Program Clock Reference */
if( flags.bits.OPCR_flag )
{
}

/* splice_countdown */
if( flags.bits.splicing_point_flag )
{
}

/* private data */
if( flags.bits.transport_private_data_flag )
{
}

/* adaptation_field extension */
if( flags.bits.adaptation_field_extension_flag )
{
}

/*Stuffing bytes*/
*tp_bytes -= adaptation_field_length;
```

```
  for(; adaptation_field_length; adaptation_field_length--) putc(0xFF, out);
}


void
TS_PROGRAM_ASSOCIATION_SECTION(int *tp_bytes, FILE *out)
/************************************
 pointer_field = 1 optional byte depending on unit_start.

 table_id = 1 byte

 section_syntax_indicator = 1 bit \
 no_name = 1 bit                   \
 reserved1 = 2 bits                 \
 section_length = 12 bits            > 5 bytes.
 transport_stream_id = 16 bits      /
 reserved2 = 2 bits                /
 version_number = 5 bits          /
 current_next_indicator = 1 bit  /

 section_number = 1 byte;

 last_section_number = 1 byte;

 program_number = 16 bits              \
 reserved = 3 bits                      > N * 4 bytes.
 network_or_program_map_PID = 13 bits /

 CRC_32 = 4 bytes;

 ************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned section_length_11_8       :  4;
    unsigned reserved1                 :  2;
    unsigned no_name                   :  1;
    unsigned section_syntax_indicator  :  1;

    unsigned section_length_7_0        :  8;

    unsigned transport_stream_id_15_8  :  8;

    unsigned transport_stream_id_7_0   :  8;

    unsigned current_next_indicator    :  1;
    unsigned version_number            :  5;
    unsigned reserved2                 :  2;

#else
    unsigned section_syntax_indicator  :  1;
    unsigned no_name                   :  1;
    unsigned reserved1                 :  2;
    unsigned section_length_11_8       :  4;

    unsigned section_length_7_0        :  8;

    unsigned transport_stream_id_15_8  :  8;

    unsigned transport_stream_id_7_0   :  8;

    unsigned reserved2                 :  2;
    unsigned version_number            :  5;
    unsigned current_next_indicator    :  1;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[5];
  } header;
  struct table_bits
  {
    unsigned program_number_15_8              :  8;

    unsigned program_number_7_0               :  8;

#ifdef DOS
    unsigned network_or_program_map_PID_12_8 :  5;
    unsigned reserved                         :  3;
#else
    unsigned reserved                         :  3;
    unsigned network_or_program_map_PID_12_8 :  5;
#endif

    unsigned network_or_program_map_PID_7_0  :  8;
  };

  union
  {
    struct table_bits bits;
    char byte[6];
  } table;


  register int i;
  PAT_program_ptr program;


  /* For the moment, it is assumed that a TP carries only one section, which
     fits the payload.*/
  /*Write pointer_field.*/
  putc(0, out);

  /*Write table_id.*/
  putc(0, out);

  /* section_syntax_indicator */
  header.bits.section_syntax_indicator = 1;

  header.bits.no_name = 0;
  header.bits.reserved1 = PAT.reserved;

  /* section_length */ /* <= 1021 */
  i = 9;
  program = PAT.program;
  while(program != NULL)
  {
    i += 4;
    program = program->next;
  }
  header.bits.section_length_11_8 = (i & 0xF00) >> 8; /*mask 1111 0000 0000 */
  header.bits.section_length_7_0 = i & 0xFF;          /*mask 0000 1111 1111 */

  /* transport_stream_id */
  i = PAT.transport_stream_id;
  header.bits.transport_stream_id_15_8 = floor(i / 256);
  header.bits.transport_stream_id_7_0  = fmod(i, 256);

  header.bits.reserved2 = PAT.reserved;

  /* version_number */
  header.bits.version_number = 0;

  /* current_next_indicator */
  header.bits.current_next_indicator = 1;

  /*Write header 5 bytes.*/
  for(i = 0; i < 5; i++) putc(header.byte[i], out);

  /* section_number */
  putc(0, out);

  /* last_section_number */
  putc(0, out);

  program = PAT.program;
  while(program != NULL)
  {
    /* program_number */
    i = program->number;
    table.bits.program_number_15_8 = floor(i / 256);
    table.bits.program_number_7_0  = fmod(i, 256);

    table.bits.reserved = program->reserved;

    /* network_PID or program_map_PID */
    i = program->PID;
    table.bits.network_or_program_map_PID_12_8 = floor(i / 256);
    table.bits.network_or_program_map_PID_7_0  = fmod(i, 256);

    /*Write table 4 bytes.*/
    for(i = 0; i < 4; i++) putc(table.byte[i], out);

    program = program->next;
    *tp_bytes -= 4;
  }

  /*Writes 4 bytes of CRC_32.*/
  for(i = 0; i < 4; i++) putc(0, out);

  *tp_bytes -= 13; /* 12 ????? */

  /*Stuffing bytes*/
  for(; *tp_bytes > 0; (*tp_bytes)--) putc(0xFF, out);
}


void
TS_PROGRAM_MAP_SECTION(int section, int *tp_bytes, FILE *out)
/*************************************
 pointer_field = 1 optional byte depending on unit_start.

 table_id = 1 byte

 section_syntax_indicator = 1 bit \
 no_name = 1 bit                   \
 reserved1 = 2 bits                 \
 section_length = 12 bits            \
 program_number = 16 bits             \
 reserved2 = 2 bits                    \
 version_number = 5 bits                > 11 bytes.
 current_next_indicator = 1 bit        /
 section_number = 8 bits              /
 last_section_number = 8 bits        /
 reserved3 = 3 bits                 /
 PCR_PID = 13 bits                 /
 reserved4 = 4 bits               /
 program_info_length = 12 bits   /

 N descriptors of several bytes;

 stream_type = 8 bits      \
 reserved = 3 bits          \
 elementary_PID = 13 bits    > N * 5 bytes.
 reserved = 4 bits          /
 ES_info_length = 12 bits  /

 N descriptors of several bytes;

 CRC_32 = 4 bytes;

 *************************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned section_length_11_8      :  4;
    unsigned reserved1                :  2;
    unsigned no_name                  :  1;
    unsigned section_syntax_indicator :  1;
    unsigned section_length_7_0         :  8;

    unsigned program_number_15_8        :  8;

    unsigned program_number_7_0         :  8;

    unsigned current_next_indicator     :  1;
    unsigned version_number             :  5;
    unsigned reserved2                  :  2;

    unsigned section_number             :  8;

    unsigned last_section_number        :  8;

    unsigned PCR_PID_12_8               :  5;
    unsigned reserved3                  :  3;

    unsigned PCR_PID_7_0                :  8;

    unsigned program_info_length_11_8 :  4;
    unsigned reserved4                  :  4;

    unsigned program_info_length_7_0  :  8;

#else
    unsigned section_syntax_indicator :  1;
    unsigned no_name                    :  1;
    unsigned reserved1                  :  2;
    unsigned section_length_11_8        :  4;

    unsigned section_length_7_0         :  8;

    unsigned program_number_15_8        :  8;

    unsigned program_number_7_0         :  8;

    unsigned reserved2                  :  2;
    unsigned version_number             :  5;
    unsigned current_next_indicator     :  1;

    unsigned section_number             :  8;

    unsigned last_section_number        :  8;

    unsigned reserved3                  :  3;
    unsigned PCR_PID_12_8               :  5;

    unsigned PCR_PID_7_0                :  8;

    unsigned reserved4                  :  4;
    unsigned program_info_length_11_8 :  4;

    unsigned program_info_length_7_0  :  8;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[11];
  } header;

  struct ES_bits
  {
    unsigned  stream_type          :  8;

#ifdef DOS
    unsigned elementary_PID_12_8 :  5;
    unsigned reserved1             :  3;

    unsigned elementary_PID_7_0  :  8;

    unsigned ES_info_length_11_8 :  4;
    unsigned reserved2             :  4;
#else
```

```
    unsigned reserved1              :  3;
    unsigned elementary_PID_12_8  :  5;

    unsigned elementary_PID_7_0   :  8;

    unsigned reserved2              :  4;
    unsigned ES_info_length_11_8  :  4;
#endif

    unsigned ES_info_length_7_0   :  8;
  };

  union
  {
    struct ES_bits bits;
    char byte[5];
  } ES;


  register int i;
  PMT_stream_ptr stream;


  /* For the moment, it is assumed that a TP carries only one section, which
     fits the payload.*/
  /*Write pointer_field.*/
  putc(0, out);

  /*Write table_id.*/
  putc(2, out);

  /* section_syntax_indicator */
  header.bits.section_syntax_indicator = 1;

  header.bits.no_name = 0;
  header.bits.reserved1 = PMT[section].reserved;

  /* section_length */ /* <= 1021 */
  i = 13;
  stream = PMT[section].stream;
  while(stream != NULL)
  {
    i += 5;
    stream = stream->next;
  }
  header.bits.section_length_11_8 = (i & 0xF00) >> 8; /*mask 1111 0000 0000 */
  header.bits.section_length_7_0 = i & 0xFF;          /*mask 0000 1111 1111 */

  /* program_number */
  i = PMT[section].program_number;
  header.bits.program_number_15_8 = floor(i / 256);
  header.bits.program_number_7_0  = fmod(i, 256);

  header.bits.reserved2 = PMT[section].reserved;

  /* version_number */
  header.bits.version_number = 0;

  /* current_next_indicator */
  header.bits.current_next_indicator = 1;

  /* section_number */
  header.bits.section_number = 0;

  /* last_section_number */
  header.bits.last_section_number = 0;

  header.bits.reserved3 = PMT[section].reserved;

  /* PCR_PID */
  i = PMT[section].PCR_PID;
  header.bits.PCR_PID_12_8 = floor(i / 256);
  header.bits.PCR_PID_7_0  = fmod(i, 256);

  header.bits.reserved4 = PMT[section].reserved;

  /* program_info_length */
  header.bits.program_info_length_11_8 = 0;
  header.bits.program_info_length_7_0 = 0;

  /*Write header 11 bytes.*/
  for(i = 0; i < 11; i++) putc(header.byte[i], out);

  stream = PMT[section].stream;
  while(stream != NULL)
  {
    /* stream_type */
    ES.bits.stream_type = stream->type;
/*            switch(ES.bits.stream_type)
            {
             case 0x00:
                         fprintf(out,"ITU-T | ISO/IEC Reserved\n");
                         break;
             case 0x01:
                         fprintf(out,"ISO/IEC 11172 Video\n");
                         break;
             case 0x02:
                         fprintf(out,"ITU-T Rec.H262 | ISO/IEC 13818-2 Video\n");
                         break;
             case 0x03:
                         fprintf(out,"ISO/IEC 11172 Audio\n");
                         break;
             case 0x04:
                         fprintf(out,"ISO/IEC 13818-3 Audio\n");
                         break;
             case 0x05:
                                    fprintf(out,"ITU-T  Rec.H222.0|ISO/IEC  13818-1
private_section\n");
                         break;
             case 0x06:
                        fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 PES packets
containing private data\n");
                         break;
             case 0x07:
                         fprintf(out,"ISO/IEC 13522 MHEG\n");
                         break;
             case 0x08:
                         fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 DSM CC\n");
                         break;
             case 0x09:
                               fprintf(out,"ITU-T  Rec.H222.0|ISO/IEC  13818-1/11172-1
auxiliary\n");

                         break;
             default   :
                         if(ES.bits.stream_type>=0x0A && ES.bits.stream_type<=0x7F)
                           fprintf(out,"ITU-T Rec.H222.0|ISO/IEC 13818-1 Reserved\n");
                         else
                         if(ES.bits.stream_type>=0x80 && ES.bits.stream_type<=0xFF)
                           fprintf(out,"User Private\n");
            }
*/
    ES.bits.reserved1 = stream->reserved;

    /* elementary_PID */
    i = stream->elementary_PID;
    ES.bits.elementary_PID_12_8 = (i & 0x1F00) >> 8;/*mask 1 1111 0000 0000 */
    ES.bits.elementary_PID_7_0 = i & 0xFF;          /*mask 0 0000 1111 1111 */

    ES.bits.reserved2 = stream->reserved;

    /* ES_info_length */
    ES.bits.ES_info_length_11_8 = 0;
    ES.bits.ES_info_length_7_0 = 0;

    /*Write 5 bytes of ES_info.*/ /* No descriptors, for the moment !*/
    for(i = 0; i < 5; i++) putc(ES.byte[i], out);

    *tp_bytes -= 5;

    stream = stream->next;
  }
  /*Writes 4 bytes of CRC_32.*/
  for(i = 0; i < 4; i++) putc(0, out);

  *tp_bytes -= 17; /* 16 ??????*/


  /*Stuffing bytes*/
  for(; *tp_bytes; (*tp_bytes)--) putc(0xFF, out);
}


void
PES_PACKET (PMT_stream_ptr stream,
             int            *tp_bytes,
             FILE           *out,
             boolean         timestamp_flag,
             double          PTS,
             double          DTS)
/*************************************

 packet_start_code_prefix = 3 bytes

 stream_id = 1 byte

 PES_packet_length = 2 bytes

 PES header = 3 optional bytes

 PTS e DTS = 0 or 5 or 10 bytes

 ESCR = 6 optional bytes

 ES_rate = 3 optional bytes

 DSM_trick_mode = 1 optional byte

 adicional_copy_info = 1 optional byte

 PES_CRC = 2 optional bytes

 PES_extension flags = 1 optional byte

 PES_private_data = 16 optional bytes

 pack_header field = until 256 bytes

 program_packet_sequence_counter = 2 optional bytes

 P_STD_buffer = 2 optional bytes

 PES_extension_flag_2 = until 128 reserved bytes (optional)

 stuffing_byte (0xFF) = maximum 32 bytes

 *************************************/
 {
   struct header_bits
   {
 #ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned original_or_copy          : 1;
    unsigned copyright                 : 1;
    unsigned data_alignment_indicator  : 1;
    unsigned PES_priority              : 1;
    unsigned PES_scrambling_control    : 2;
    unsigned noname                    : 2;

    unsigned PES_extension_flag        : 1;
    unsigned PES_CRC_flag              : 1;
    unsigned additional_copy_info_flag : 1;
    unsigned DSM_trick_mode_flag       : 1;
    unsigned ES_rate_flag              : 1;
    unsigned ESCR_flag                 : 1;
    unsigned PTS_DTS_flags             : 2;
```

```
#else
    unsigned noname                    : 2;
    unsigned PES_scrambling_control    : 2;
    unsigned PES_priority              : 1;
    unsigned data_alignment_indicator  : 1;
    unsigned copyright                 : 1;
    unsigned original_or_copy          : 1;

    unsigned PTS_DTS_flags             : 2;
    unsigned ESCR_flag                 : 1;
    unsigned ES_rate_flag              : 1;
    unsigned DSM_trick_mode_flag       : 1;
    unsigned additional_copy_info_flag : 1;
    unsigned PES_CRC_flag              : 1;
    unsigned PES_extension_flag        : 1;

#endif
    unsigned PES_header_data_length    : 8;
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;


  struct PTS_DTS_bits
  {
#ifdef DOS
    unsigned marker_bit_1 : 1;
    unsigned PTS_32_30    : 3;
    unsigned noname_1     : 4;

    unsigned PTS_29_22    : 8;

    unsigned marker_bit_2 : 1;
    unsigned PTS_21_15    : 7;

    unsigned PTS_14_7     : 8;

    unsigned marker_bit_3 : 1;
    unsigned PTS_6_0      : 7;

    unsigned marker_bit_4 : 1;
    unsigned DTS_32_30    : 3;
    unsigned noname_2     : 4;

    unsigned DTS_29_22    : 8;

    unsigned marker_bit_5 : 1;
    unsigned DTS_21_15    : 7;

    unsigned DTS_14_7     : 8;

    unsigned marker_bit_6 : 1;
    unsigned DTS_6_0      : 7;

#else
    unsigned noname_1     : 4;
    unsigned PTS_32_30    : 3;
    unsigned marker_bit_1 : 1;

    unsigned PTS_29_22    : 8;

    unsigned PTS_21_15    : 7;
    unsigned marker_bit_2 : 1;

    unsigned PTS_14_7     : 8;

    unsigned PTS_6_0      : 7;
    unsigned marker_bit_3 : 1;

    unsigned noname_2     : 4;
    unsigned DTS_32_30    : 3;
    unsigned marker_bit_4 : 1;
    unsigned DTS_29_22     : 8;

    unsigned DTS_21_15     : 7;
    unsigned marker_bit_5  : 1;

    unsigned DTS_14_7      : 8;

    unsigned DTS_6_0       : 7;
    unsigned marker_bit_6  : 1;

#endif
  };

  union
  {
    struct PTS_DTS_bits bits;
    char byte[10];
  } stamp;


  struct ESCR_bits
  {
#ifdef DOS
    unsigned ESCR_29_28            : 2;
    unsigned marker_bit_1          : 1;
    unsigned ESCR_32_30            : 3;
    unsigned reserved              : 2;

    unsigned ESCR_27_20            : 8;

    unsigned ESCR_14_13            : 2;
    unsigned marker_bit_2          : 1;
    unsigned ESCR_19_15            : 5;

    unsigned ESCR_12_5             : 8;

    unsigned ESCR_extension_8_7    : 2;
    unsigned marker_bit_3          : 1;
    unsigned ESCR_4_0              : 5;

    unsigned marker_bit_4          : 1;
    unsigned ESCR_extension_6_0    : 7;

#else
    unsigned reserved              : 2;
    unsigned ESCR_32_30            : 3;
    unsigned marker_bit_1          : 1;
    unsigned ESCR_29_28            : 2;

    unsigned ESCR_27_20            : 8;

    unsigned ESCR_19_15            : 5;
    unsigned marker_bit_2          : 1;
    unsigned ESCR_14_13            : 2;

    unsigned ESCR_12_5             : 8;

    unsigned ESCR_4_0              : 5;
    unsigned marker_bit_3          : 1;
    unsigned ESCR_extension_8_7    : 2;

    unsigned ESCR_extension_6_0    : 7;
    unsigned marker_bit_4          : 1;

#endif
  };

  union
  {
    struct ESCR_bits bits;
    char byte[6];
  } ESCR;


  struct ES_rate_bits
  {
#ifdef DOS
```

```
    unsigned ES_rate_21_15  : 7;
    unsigned marker_bit_1   : 1;

    unsigned ES_rate_14_7   : 8;

    unsigned marker_bit_2   : 1;
    unsigned ES_rate_6_0    : 7;

#else
    unsigned marker_bit_1   : 1;
    unsigned ES_rate_21_15  : 7;

    unsigned ES_rate_14_7   : 8;

    unsigned ES_rate_6_0    : 7;
    unsigned marker_bit_2   : 1;

#endif
  };

  union
  {
    struct ES_rate_bits bits;
    char byte[3];
  } ES_rate;


  struct DSM_trick_bits
  {
#ifdef DOS
    unsigned bits_3             : 2;
    unsigned bits_2             : 1;
    unsigned bits_1             : 2;
    unsigned trick_mode_control : 3;

#else
    unsigned trick_mode_control : 3;
    unsigned bits_1             : 2;
    unsigned bits_2             : 1;
    unsigned bits_3             : 2;

#endif
  };

  union
  {
    struct DSM_trick_bits bits;
    char byte;
  } DSM_trick;


  struct flag_bits
  {
#ifdef DOS
    unsigned PES_extension_flag_2                  :  1;
    unsigned reserved                              :  3;
    unsigned P_STD_buffer_flag                     :  1;
    unsigned program_packet_sequence_counter_flag :  1;
    unsigned pack_header_field_flag               :  1;
    unsigned PES_private_data_flag                :  1;

#else
    unsigned PES_private_data_flag                :  1;
    unsigned pack_header_field_flag               :  1;
    unsigned program_packet_sequence_counter_flag :  1;
    unsigned P_STD_buffer_flag                     :  1;
    unsigned reserved                              :  3;
    unsigned PES_extension_flag_2                  :  1;

#endif
  };

  union
  {
    struct flag_bits bits;
    char byte;
  } flags;
```

```
double aux,
       ESCR_value,     /* deactivated */
       ES_rate_value; /* deactivated */
register int i, stuffing;
long bytes;


if(stream->PES_bytes_left == stream->PES_packet_length) /* unit_start */
{
  /* packet_start_code_prefix */
  putc(0, out); putc(0, out); putc(1, out);

  /* stream->id */
  putc(stream->id, out);

  /* PES_packet_length */
  putc( (int)(stream->PES_packet_length / 256), out);
  putc( stream->PES_packet_length % 256, out);

  *tp_bytes -= 6;
  stream->PES_bytes_left -= 6;


  /* optional PES header */
  if( stream->id != 0xBC &&  /* program_stream_map */
      stream->id != 0xBF &&  /* private_stream_2 */
      stream->id != 0xBE &&  /* padding_stream */
      stream->id != 0xF0 &&  /* ECM_stream */
      stream->id != 0xF1 &&  /* EMM_stream */
      stream->id != 0xFF    )/* program_stream_directory */
  {
    /* opcional header start (flags) */

    header.bits.noname = 2;

    /* PES_scrambling_control */
    header.bits.PES_scrambling_control = stream->PES_scrambling_control;

    /* PES_priority */
    header.bits.PES_priority = stream->PES_priority;

    /* data_alignment_indicator */
    header.bits.data_alignment_indicator = 0;

    /* copyright */
    header.bits.copyright = stream->copyright;

    /* original_or_copy */
    header.bits.original_or_copy = stream->original_or_copy;

    /* PTS_DTS_flags */
    if(timestamp_flag)
    {
      if(PTS == DTS) i = 2;
      else i = 3;
    }
    else i = 0;
    header.bits.PTS_DTS_flags = i;

    /* ESCR_flag */
    header.bits.ESCR_flag = 0;

    /* ES_rate_flag */
    header.bits.ES_rate_flag = 0;

    /* DSM_trick_mode_flag */
    header.bits.DSM_trick_mode_flag = 0;

    /* additional_copy_info_flag */
    header.bits.additional_copy_info_flag = 0;

    /* PES_CRC_flag */
    header.bits.PES_CRC_flag = 0;

    /* PES_extension_flag */
```

```
header.bits.PES_extension_flag = 0;

/*Initialization of  PES_header_data_length.*/
header.bits.PES_header_data_length = 0;


/* PTS and DTS */
if(header.bits.PTS_DTS_flags)
{
  header.bits.PES_header_data_length += (header.bits.PTS_DTS_flags-1)*5;

  stamp.bits.noname_1 = header.bits.PTS_DTS_flags;

  stamp.bits.marker_bit_1 = 1;
  stamp.bits.marker_bit_2 = 1;
  stamp.bits.marker_bit_3 = 1;

  aux = PTS;

  stamp.bits.PTS_32_30 = floor(aux / 1073741824); /* 2^30 */
  aux = fmod(aux, 1073741824);
  stamp.bits.PTS_29_22 = floor(aux / 4194304);    /* 2^22 */
  aux = fmod(aux, 4194304);
  stamp.bits.PTS_21_15 = floor(aux / 32768);      /* 2^15 */
  aux = fmod(aux, 32768);
  stamp.bits.PTS_14_7  = floor(aux / 128);        /* 2^7  */
  stamp.bits.PTS_6_0 = fmod(aux, 128);

  if(header.bits.PTS_DTS_flags == 3) /* DTS */
  {
    stamp.bits.noname_2 = 1;

    stamp.bits.marker_bit_4 = 1;
    stamp.bits.marker_bit_5 = 1;
    stamp.bits.marker_bit_6 = 1;

    aux = DTS;

    stamp.bits.DTS_32_30 = floor(aux / 1073741824); /* 2^30 */
    aux = fmod(aux, 1073741824);
    stamp.bits.DTS_29_22 = floor(aux / 4194304);    /* 2^22 */
    aux = fmod(aux, 4194304);
    stamp.bits.DTS_21_15 = floor(aux / 32768);      /* 2^15 */
    aux = fmod(aux, 32768);
    stamp.bits.DTS_14_7  = floor(aux / 128);        /* 2^7  */
    stamp.bits.DTS_6_0 = fmod(aux, 128);
  }
}


/* ESCR */
if(header.bits.ESCR_flag)
{
  header.bits.PES_header_data_length += 6;

  ESCR.bits.marker_bit_1 = 1;
  ESCR.bits.marker_bit_2 = 1;
  ESCR.bits.marker_bit_3 = 1;
  ESCR.bits.marker_bit_4 = 1;
  ESCR.bits.reserved = 0;

  aux = floor(ESCR_value / 300);

  ESCR.bits.ESCR_32_30 = floor(aux / 1073741824); /* 2^30 */
  aux = fmod(aux, 1073741824);
  ESCR.bits.ESCR_29_28 = floor(aux / 268435456);   /* 2^28 */
  aux = fmod(aux, 268435456);
  ESCR.bits.ESCR_27_20 = floor(aux / 1048576);     /* 2^20 */
  aux = fmod(aux, 1048576);
  ESCR.bits.ESCR_19_15 = floor(aux / 32768);       /* 2^15 */
  aux = fmod(aux, 32768);
  ESCR.bits.ESCR_14_13 = floor(aux / 8192);        /* 2^13 */
  aux = fmod(aux, 8192);
  ESCR.bits.ESCR_12_5  = floor(aux / 32);          /* 2^5  */
  ESCR.bits.ESCR_4_0 = fmod(aux, 32);

  aux = fmod(ESCR_value, 300);
```

```
    ESCR.bits.ESCR_extension_8_7 = floor(aux / 128);
    ESCR.bits.ESCR_extension_6_0 = fmod(aux, 128);
  }

  /* ES_rate */
  if(header.bits.ES_rate_flag)
  {
    header.bits.PES_header_data_length += 3;

    ES_rate.bits.marker_bit_1 = 1;
    ES_rate.bits.marker_bit_2 = 1;

    aux = ES_rate_value / 50;

    ES_rate.bits.ES_rate_21_15 = floor(aux / 32768);
    aux = fmod(aux, 32768);
    ES_rate.bits.ES_rate_14_7  = floor(aux / 128);
    ES_rate.bits.ES_rate_6_0 = fmod(aux, 128);
  }

  /* DSM_trick_mode */
  if(header.bits.DSM_trick_mode_flag)
  {
    header.bits.PES_header_data_length ++;

    DSM_trick.bits.trick_mode_control = 0; /* ffw */
    DSM_trick.bits.bits_1 = 0;
    DSM_trick.bits.bits_2 = 0;
    DSM_trick.bits.bits_3 = 0;
  }

  /* additional_copy_info */
  if(header.bits.additional_copy_info_flag)
  {
    header.bits.PES_header_data_length ++;
  }

  /* PES_CRC */
  if(header.bits.PES_CRC_flag)
  {
    header.bits.PES_header_data_length += 2;
  }

  /* PES_extension */
  if(header.bits.PES_extension_flag)
  {
    header.bits.PES_header_data_length += 1; /* flags = 0 */

    /* PES_private_data_flag */
    flags.bits.PES_private_data_flag = 0;

    /* pack_header_field_flag */
    flags.bits.pack_header_field_flag = 0;

    /* program_packet_sequence_counter_flag */
    flags.bits.program_packet_sequence_counter_flag = 0;

    /* P_STD_buffer_flag */
    flags.bits.P_STD_buffer_flag = 0;

    flags.bits.reserved = 0;

    /* PES_extension_flag_2*/
    flags.bits.PES_extension_flag_2 = 0;
  }

  /* Stuffing Bytes (for the PES overhead assumptions in this implementation) */
  stuffing = 0;
  i = header.bits.PTS_DTS_flags;
  if(stream->id >= 0xE0 && stream->id <= 0xEF) /* Video */
```

```
{
  if(!i) stuffing = 10;
  if(i == 2) stuffing = 5;
}
else
  if(!i) stuffing = 5;
header.bits.PES_header_data_length += stuffing;


/* Write */

/* opcional header start 3 bytes (flags) */
for(i=0; i<3; i++)  putc(header.byte[i], out);
*tp_bytes -= 3;
stream->PES_bytes_left -= 3;

/* 5 or 10 bytes of PTS and DTS */
if(header.bits.PTS_DTS_flags)
{
  for(i=0; i<5; i++)  putc(stamp.byte[i], out);
  *tp_bytes -= 5;
  stream->PES_bytes_left -= 5;

  if(header.bits.PTS_DTS_flags == 3)
  {
    for(i=5; i<10; i++)  putc(stamp.byte[i], out);
    *tp_bytes -= 5;
    stream->PES_bytes_left -= 5;
  }
}

/* 6 bytes ESCR */
if(header.bits.ESCR_flag)
{
  for(i=0; i<6; i++)  putc(ESCR.byte[i], out);
  *tp_bytes -= 6;
  stream->PES_bytes_left -= 6;
}

/* 3 bytes ES_rate */
if(header.bits.ES_rate_flag)
{
  for(i=0; i<3; i++)  putc(ES_rate.byte[i], out);
  *tp_bytes -= 3;
  stream->PES_bytes_left -= 3;
}

/* DSM_trick_mode */
if(header.bits.DSM_trick_mode_flag)
{
  putc(DSM_trick.byte, out);
  (*tp_bytes)--;
  (stream->PES_bytes_left)--;
}

/* additional_copy_info */
if(header.bits.additional_copy_info_flag)
{
  putc(0, out);
  (*tp_bytes)--;
  (stream->PES_bytes_left)--;
}

/* 2 bytes PES_CRC */
if(header.bits.PES_CRC_flag)
{
  putc(0, out); putc(0, out);
  *tp_bytes -= 2;
  stream->PES_bytes_left -= 2;
}

/* PES_extension */
if(header.bits.PES_extension_flag)
{
  putc(flags.byte, out);
  (*tp_bytes)--;
  (stream->PES_bytes_left)--;
```

```
      }

      /* Stuffing */
      for(i = 0; i < stuffing; i++)  putc(0xFF, out);
      *tp_bytes -= stuffing;
      stream->PES_bytes_left -= stuffing;
    }
  }


  i = *tp_bytes;
  stream->PES_bytes_left -= i;
  if(stream->id == 0xBE) /* padding_stream */
    for(; i; i--) putc(0xFF,out);
  else
  {
    stream->offset -= i;

    /* Fills simulated decoder buffer */
    bytes = stream->au_bytes_left;
    bytes -= i;

    while(bytes <= 0)
    {
      buffer(stream, stream->au_bytes_left);

      if(fread(&(stream->au), sizeof(struct access_unit), 1, stream->info_file) != 1)
      {
        if(stream->offset == 0)  stream->au.length = 1; /* > 0 */
        else
        {
          /* PES packet is adjusted to fit stream length */
          Warn("Error: EOF in access units info !");
          exit(1);
        }
      }

      stream->au.PTS += stream->PMT->startup_delay;
      stream->au.DTS += stream->PMT->startup_delay;

      stream->au_bytes_left = stream->au.length;
      bytes = stream->au_bytes_left + bytes;
    }

    if(bytes == stream->au.length)  stream->au_start = YES;
    else
    {
      stream->au_start = NO;

      buffer(stream, stream->au_bytes_left - bytes);
      stream->au_bytes_left = bytes;
    }


    /* Elementary Stream */
    for(; i; i--) putc(getc(stream->file), out);
  }
  *tp_bytes = 0;
}


void
TP_NULL (FILE *out)
/***************************************
 Null_packet.

 sync_byte = 0x47 (1 byte).

 transport_error_indicator = 1 bit           \
 payload_unit_start_indicator = 0 (1 bit) \
 transport_priority = 1 bit                   \
 PID = 0x1FFF (13 bits)                        > 3 bytes.
 transport_scrambling_control = 0 (2 bits) /
 adaptation_field_control = 1 (2 bits)     /
 continuity_counter = 4 bits               /
 data_bytes = 184 bytes.

 ***********************************/
{
  struct header_bits
  {
#ifdef DOS
    /*Bits are ordered in DOS (lsbf, msbl) differently from UNIX (msbf, lsbl).
      One could access bits directly with masks and shifts but code would be
      harder to read.*/
    unsigned PID_12_8                  :  5;
    unsigned transport_priority        :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_error_indicator :  1;

    unsigned PID_7_0                       :  8;

    unsigned continuity_counter        :  4;
    unsigned adaptation_field_control  :  2;
    unsigned transport_scrambling_control :  2;

#else
    unsigned transport_error_indicator :  1;
    unsigned payload_unit_start_indicator :  1;
    unsigned transport_priority        :  1;
    unsigned PID_12_8                  :  5;

    unsigned PID_7_0                       :  8;

    unsigned transport_scrambling_control :  2;
    unsigned adaptation_field_control  :  2;
    unsigned continuity_counter        :  4;

#endif
  };

  union
  {
    struct header_bits bits;
    char byte[3];
  } header;


  register int i;
  int              tp_bytes = 188; /* Transport_packet byte counter*/



  /* Write sync_byte */
  putc(0x47, out);
  tp_bytes--;


  /* Header */

  /* transport_error_indicator */
  header.bits.transport_error_indicator = 0;

  /* payload_unit_start_indicator */
  header.bits.payload_unit_start_indicator = 0;

  /* transport_priority */
  header.bits.transport_priority = 0;

  /* Packet Identifier */
  header.bits.PID_12_8 = 0x1F;
  header.bits.PID_7_0  = 0xFF;

  /* transport_scrambling_control */
  header.bits.transport_scrambling_control = 0;

  /* adaptation_field_control */
  header.bits.adaptation_field_control = 1;/*payload only*/

  /* continuity_counter */
  header.bits.continuity_counter = 0;
  /* Write 3 bytes header */
  for(i = 0; i < 3; i++) putc(header.byte[i], out);
  tp_bytes -= 3;


  /* payload */
  for(; tp_bytes; tp_bytes--) putc(0, out);
}


void
buffer (PMT_stream_ptr stream, long size)
/************************************
 Fills simulated decoder buffer w/access unit data.
 ************************************/
{
  buffer_ptr ptr;


  ptr = stream->buffer;
  if(ptr == NULL)
  {
    if((stream->buffer = (buffer_ptr)malloc(sizeof(struct buffer_queue))) == NULL)
    { Warn("buffer(): malloc error"); exit(1); }

    stream->buffer->number     = stream->au.number;
    stream->buffer->size       = size;
    stream->buffer->au_length  = stream->au.length;
    stream->buffer->DTS        = stream->au.DTS;
    stream->buffer->next       = NULL;
  }
  else
  {
    while(ptr->next  !=  NULL)  ptr = ptr->next;

    if(stream->au.number == ptr->number)
    {
      ptr->size += size;
    }
    else
    {
      if((ptr->next = (buffer_ptr)malloc(sizeof(struct buffer_queue))) == NULL)
      { Warn("buffer(): malloc error"); exit(1); }

      ptr->next->number     = stream->au.number;
      ptr->next->size       = size;
      ptr->next->au_length  = stream->au.length;
      ptr->next->DTS        = stream->au.DTS;
      ptr->next->next       = NULL;
    }
  }

  if(stream->buffer_space < size)
  {
    Warn("buffer(): overflow");
    exit(1);
  }
  stream->buffer_space -= size;
}


void
clean_buffers (double current_PCR)
/************************************
 Compares DTS of access units, in the simulated decoder buffers, w/the current
 PCR. Discards units that have already been decoded. Checks for underflow.
 ************************************/
{
  /* boolean buffer_status; GLOBAL VAR. */
  buffer_ptr ptr;
  char str[80];
  double PCR_base;
  register int i;


  PCR_base = floor(current_PCR / 300);
```

```
for(i = 0; i < PAT.n_streams; i++)
{

  if(buffer_status) buffer_fullness(ES[i]);

  ptr = ES[i]->buffer;

  while(ptr != NULL && ptr->DTS <= PCR_base)
  {
    if(ptr->au_count == ptr->size) /* Decode */
    {
      ES[i]->buffer_space += ptr->size;
      ES[i]->buffer = ES[i]->buffer->next;
      free(ptr);

      ES[i]->underflow = NO;
      ptr = ES[i]->buffer;
    }
    else /* Underflow */
    {
      if(ES[i]->underflow == NO)
      {
        ES[i]->underflow = YES;

            sprintf(str,"PID = 0x%04X, stream_id = 0x%02X: AU %ld is late (buffer
underflow)",
                    ES[i]->elementary_PID, ES[i]->id, ptr->number);
        Warn(str);
      }

      if(ptr->next == NULL)  ptr = NULL;
      else
      {
        Warn("Error: incomplete AU not last in buffer queue !");
        exit(1);
      }
    }

   }
  }
}


void
buffer_fullness (PMT_stream_ptr stream)
/*****************************************
 Simulated decoder buffer fullness (%).
 *****************************************/
{
  long f, p;
  char str[80];


  if(stream->buffer != NULL)
  {
    f = stream->buffer_size - stream->buffer_space;
    p = (f * 100) / stream->buffer_size;

  }
  else
  {
    f = 0;
    p = 0;
  }

  sprintf(str,"BUFFER FULLNESS [PID 0x%04X, stream_id 0x%02X]: %ld bytes\t(%2ld%%)",
          stream->elementary_PID, stream->id, f, p);
  Warn(str);
}
```

6. CONSIDERAÇÕES FINAIS

Em jeito de síntese final, pode-se dizer que foram estudados os aspectos de sistema MPEG e obtidos conhecimentos gerais ao nível de compressão, através da aplicação prática do conteúdo da norma.

Foi desenvolvido um instrumento que permite efectuar a validação de fluxos MPEG-1 e MPEG-2 na descodificação e de fluxos de transporte MPEG-2 na codificação. Dada a flexibilidade da norma, esta validação é, à partida, parcial.

O codificador não foi testado completamente, ficando em aberto questões relacionadas com a estratégia óptima de multiplexagem.

Seria, no entanto, agora possível recolher diversas estatísticas, tais como, o *overhead* exacto do fluxo de sistema, o nível médio dos *buffers* (entre outras), bem como projectar um conjunto de experiências simples. Ou seja: estaria aberto o caminho para futuros desenvolvimentos.

Poder-se-ia planear, por exemplo, o estudo do impacto da inserção de tabelas de informação PSI, isto é, verificar, de acordo com a estratégia de multiplexagem, a inserção de tabelas iguais com periodicidades diferentes, ou *vice-versa*.

Possibilita-se também a realização de experiências sobre a robustez de esquemas de multiplexagem perante ruído na transmissão, ou a sua adaptabilidade a diferentes modos de transporte (*e.g.* Modo Assíncrono de Transferência - ATM).

Isto tudo para concluir que o *software* - base deste projecto - deveria agora ser submetido a múltiplos testes, no sentido de uma aproximação maior das exigências de aplicação concreta.

Muito trabalho há ainda para ser realizado.

Bibliografia

Ang, P.H. and P.A. Ruetz, and D. Auld (1991). "Video compression makes big gains". *IEEE Spectrum*, Vol. 28, n. 10, pp. 16-19.

Asatani, K. (1994). "Standardization of Network Technologies and Services". *IEEE Communications Magazine*, Vol. 32, n.7, pp. 60-66.

Asthana, P. (1994). "The lessons of optical storage". *IEEE Spectrum*, Vol. 31, n.10, pp. 60-66.

Chiang, T. And D. Anastassiou (1994). "Hierarchical Coding of Digital Television". *IEEE Communications Magazine*, Vol. 32, n.5, pp. 38-45.

Cole, B. (1993). "The technology framework". *IEEE Spectrum*, Vol. 30, n. 3, pp. 32-39.

Comerford, R. (1994).). "The multimedia drive". *IEEE Spectrum*, Vol. 31, n.4, pp. 77-83.

de Prycker, M. (1991). *Asynchronous transfer mode: Solution for broadband ISDN.* England: Ellis Horwood.

Ferreira, A. (1994). *Codificação de Áudio de Alta Qualidade usando MPEG: Enquadramento, Resultados e Perspectivas Futuras*, apresentado no 1° Encontro Nacional do Colégio de Engenharia Electrotécnica. Porto: INESC.

Fonseca, P. (1995). "Grande aliança ou nova guerra ?". *Público*, suplemento Computadores de 5 Junho 1995.

Fox, B. (1995). "The Digital Dawn in Europe". *IEEE Spectrum*, Vol. 32, n. 4, pp. 50-53.

Fox, E.A. (1991). "Advances in Interactive Digital Multimedia Systems". *IEEE Computer*, October 1991, pp. 9-21.

ISO/IEC 11172-1 (s/data). *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 1: Systems*, Draft International Standard. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

ISO/IEC 11172-2 (1991). *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 2: Video*, Committee Draft. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

ISO/IEC 11172-3 (1993). *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio*, Final Text, n. 314. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

ISO/IEC 13818-1 (1994). *Information technology - Generic Coding of Moving Pictures and Associated Audio: Systems, Recommendation H.222.0*, Draft International Standard, n. 721. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

ISO/IEC 13818-2 (1994). *Information technology - Generic Coding of Moving Pictures and Associated Audio: Video, Recommendation H.262*, Draft International Standard, n. 702. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

ISO/IEC 13818-3 (1994). *Information technology - Generic Coding of Moving Pictures and Associated Audio: Audio*, Draft International Standard, n. 703. Switzerland: ISO/IEC JTC 1/SC 29/WG 11.

Jurgen, R.K. (1992). "An abundance of video formats". *IEEE Spectrum*, Vol. 29, n. 3, pp. 26-28.

Le Gall, D. (1991). "MPEG: A Video Compression Standard for multimedia Applications". *Communications of the ACM*, Vol. 34, n. 4, pp. 47-58.

Ninomiya, Y. (1995). "The Japanese Scene". *IEEE Spectrum*, Vol. 32, n. 4, pp. 54-57.

Nunes, M.S. e A.J. Casaca (1992). *Redes Digitais com Integração de Serviços*, 1ª edição. Lisboa: Editorial Presença.

The Grand Alliance (1995). "The U.S. HDTV standard". *IEEE Spectrum*, Vol. 32, n. 4, pp. 36-45.
Warwick, M. (1993), "Video and the view from the desktop". *Communications International*, Vol. 20, n. 3, pp. 48-50.