画像情報特論　授業資料

ストリーミングソフトウェアについて

電子・情報通信学科　甲藤二郎

1. はじめに

画像情報特論のまとめとして、ストリーミングソフトウェア作成のための簡単なガイドラインについて説明する。

2. ストリーミング関連のフリーソフトウェア

インターネット上には、ストリーミング関連のさまざまなフリーソフトウェアが公開されている。表 1 は、それらの代表例をまとめたものである。

表 1: ストリーミング関連のフリーソフトウェア一覧

(a) ビデオ・オーディオ・コーデック関連

名称	URL
mpeg_play (MPEG-1 Video)	http://bmrc.berkeley.edu/frame/research/mpeg/mpeg_play.html
Project Mayo (MPEG-4 Video)	http://www.projectmayo.com/
ITU-T H.26L	http://www.tnt.uni-hannover.de/js/project/vceg/
OggVorbis	http://www.vorbis.com/
その他 (sourceforge.net)	http://sourceforge.net/

(b) ストリーミングシステム関連

名称	URL
OpenH323 (ITU-T H.323)	http://www.openh323.org
Mbone Tools (RTP/SDP/SIP/SAP)	http://www-mice.cs.ucl.ac.uk/multimedia/software/
Apple Open Project (RTSP)	http://developer.apple.com/quicktime/
Project Mayo (RTSP)	http://www.projectmayo.com/
その他 (sourceforge.net)	http://sourceforge.net/

フリーソフトウェアの扱い方で、プログラミングの熟練度がわかる。

- バイナリコードをダウンロードしてインストール (レベル 1)。
- tar ボールをダウンロードして、configure & make (レベル 2)。
- tar ボールをダウンロードしてソースコードを改造。必要に応じて自作 (レベル 3)。

この資料では、レベル 3、すなわちソースコードの改造方法や自作方法について説明する。

自由課題 (マイク、スピーカ、カメラがあるとよい)：

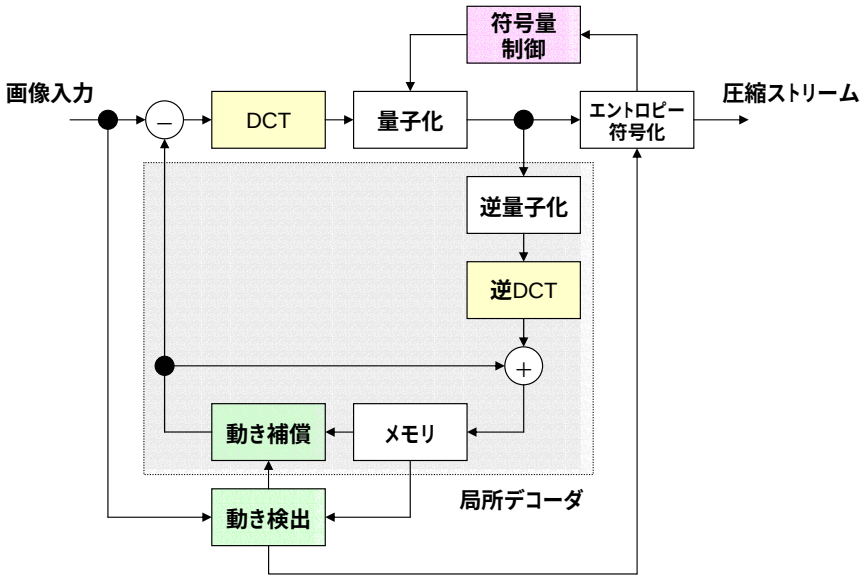
(1) OpenH323 プロジェクトの tar ボールをダウンロードし、make を行い、実行してみよ。

(2) Mbone Tooles の VIC、RAT、SDR の tar ボールをダウンロードし、make を行い、実行してみよ。また、マルチキャスト実験を行ってみよ (LAN 内であればマルチキャストルータの設定は不要)。

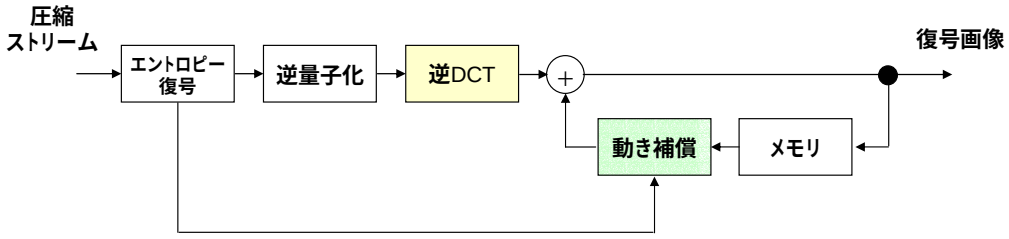
3. エンコーダとデコーダ

3.1. 基本構成

今から 8 年ほど前の mpeg_play の登場以来、ビデオのソフトウェアデコードは当たり前となり、最近ではビデオのソフトウェアエンコードも可能である。図 1 はエンコーダとデコーダの基本構成を、表 2 はソフトウェア記述の基本構成を示している。



(a) ビデオエンコーダ



(b) ビデオデコーダ (エンコーダの局所デコーダに同じ)

図 1: ビデオエンコーダ・デコーダの基本構成

表 2: ソフトウェアエンコーダ・デコーダの基本構成

(a) ビデオエンコーダ

```
main() {
      init();                  // 初期化（メモリ確保、各種パラメータ初期化)
      while(1) {
          read();              // 画像ファイル読み込み
          encode();            // ビデオ・エンコード
          write();             // 圧縮ファイル書き出し
      }
      close();                 // 終了処理（メモリ開放、各種終了処理)
}
```

(b) ビデオデコーダ

```
main() {
      init();                  // 初期化（メモリ確保、各種パラメータ初期化)
      while(1) {
          read();              // 圧縮ファイル読み込み
          decode();            // ビデオ・デコード
          display();           // 表示（WIN32、X11 等)
      }
      close();                 // 終了処理（メモリ開放、表示系の終了処理)
}
```

表 2 の encode( ) 関数と decode( ) 関数の中身をもう少し詳しく書くと表 3 のようになる。さまざまな圧縮アルゴリズムに関するソースコードが公開されているが、関数名の違いこそあれ、構成はほぼ共通している。

表 3: encode( ) 関数と decode( ) 関数の中身

(a) encode( ) 関数

```
encode() {                                  // ピクチャ処理
      picture_header();                     //   ピクチャヘッダ書き出し
      while(slice) {                        // スライス処理
            slice_header();                 //   スライスヘッダ書き出し
            while(macroblock) {             // マクロブロック処理
                  motion_estimation();      //   動き検出
                  motion_compensation();    //   動き補償予測
                  mb_header();              //   マクロブロックヘッダ書き出し
                  while(block) {            // ブロック処理
                        dct();              //   DCT（離散コサイン変換)
                        quantization();     //   量子化
```

```
                        huffman();              //   ハフマン符号書き出し
                        i_quantization();       //   逆量子化
                        i_dct();                //   逆 DCT
                        frame_update();         //   フレームメモリ更新
                  }
            }
      }
}
```

(b) decode( ) 関数

```
decode() {                                      // ピクチャ処理
      picture_header_search();                  //   ピクチャヘッダ探索・復号
      while(slice) {                            // スライス処理
            slice_header_search();              //   スライスヘッダ探索・復号
            while(macroblock) {                 // マクロブロック処理
                  mb_header_decode();           //   マクロブロックヘッダ復号
                  motion_compensation();        //   動き補償予測
                  while(block) {                // ブロック処理
                        i_huffman();            //   ハフマン復号
                        i_quantization();       //   逆量子化
                        i_dct();                //   逆 DCT
                        frame_update();         //   フレームメモリ更新
                  }
            }
      }
}
```

3.2. **詳細**

ソフトウェアコーデックに関して、特に重要な以下の項目について説明する。

- ビット処理
- スタートコード探索
- テーブル参照
- ファイル入力からストリーム入力への変換

3.2.1. **ビット処理**

圧縮ストリームを扱う場合、ビット単位に情報を取り出し、それらを厳密に処理しなければならない。ここではビット処理のための関数群の例を紹介する。

図 2 はデコーダ側におけるビット処理の原理を、表 4 は対応する C プログラムの例 (簡略版) を示して

いる。圧縮データをバッファ (buffer.data) に格納し、バッファ上の位置 (buffer.pointer) 、その位置に対応する unsigned int 型整数値 (buffer.current)、その中のビットオフセット値 (buffer.offset)、を管理・更新しながら、n ビット単位のデータの表示 (show_bits) と取得 (get_bits) を実現している。ただし、ここに示した例は簡略版であり、実際は buffer.data 内のデータ量が少なくなってきたら、適宜補充しなければならない。

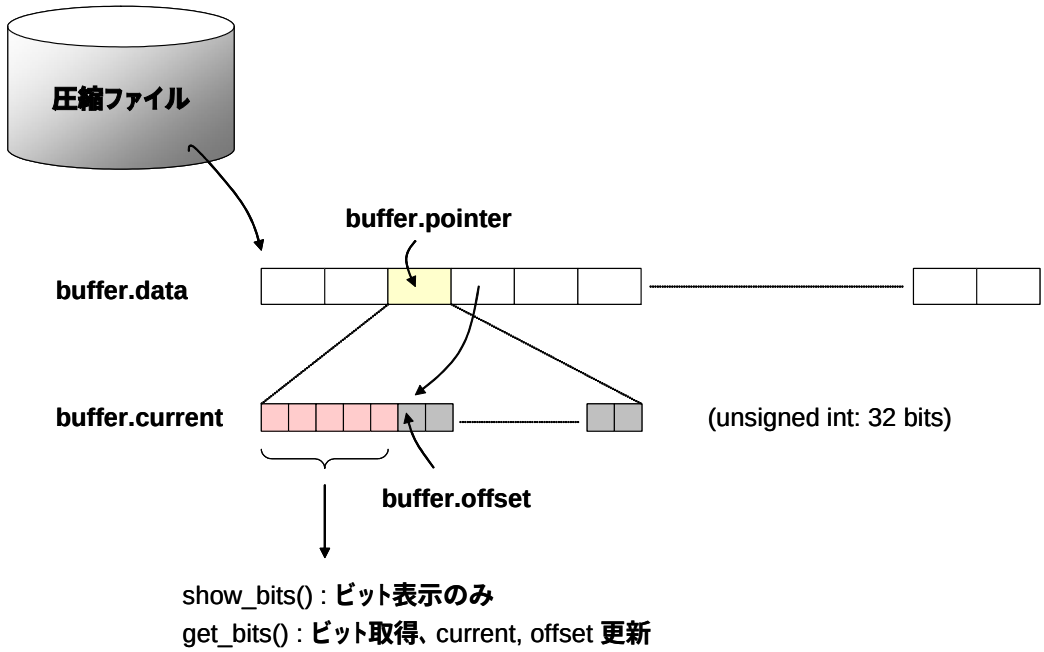


図 2: ビット処理の原理 (デコーダ側)

表 4: ビット処理関数の例

```
// 圧縮ストリームを格納・管理する構造体
struct Buffer {
      unsigned int *data;                 // バッファデータ
      unsigned int pointer;               // バッファデータ上のバイト位置
      unsigned int current;               // 現在の int 型データ
      unsigned int offset;                // 現在の int 型データのビット位置
};
struct Buffer buffer;

// ビットマスク（左詰め）
unsigned int bitmask[32] = {              // ビットマスク（1〜32 ビットマスク）
      0x80000000, 0xc0000000, 0xe0000000, 0xf0000000,
      0xf8000000, 0xfc000000, 0xfe000000, 0xff000000,
      0xff800000, 0xffc00000, 0xffe00000, 0xfff00000,
      0xfff80000, 0xfffc0000, 0xfffe0000, 0xffff0000,
      0xffff8000, 0xffffc000, 0xfffff000, 0xfffff000,
```

```
	0xfffff800, 0xfffffc00, 0xfffffe00, 0xffffff00,
	0xffffff80, 0xffffffc0, 0xffffffe0, 0xfffffff0,
	0xfffffff8, 0xfffffffc, 0xfffffffe, 0xffffffff
};

// n ビット表示する関数 (詳細)
unsigned int show_bits_x(unsigned int num, unsigned int mask, unsigned int shift) {
	int b0,result;

	// current の下位 offset ビットと次のバイトの上位ビットから result を作成
	b0 = buffer.offset + num;
	if(b0 > 32) {
		b0 -= 32;
		result =  ((buffer.current & mask) >> shift) |
			  (buffer.data[buffer.pointer+1] >> (shift + (num-b0)));
	} else {
		result = ((buffer.current & mask) >> shift);
	}
	return result;
}

// n ビット表示する関数 (通常)
unsigned int show_bits(unsigned int num) {
	return show_bits_x(num, bitmask[num-1], 32-num);
}

// n ビット取得する関数 (詳細)
unsigned int get_bits_x(unsigned int num, unsigned int mask, unsigned int shift) {
	int result;

	// 残余ビットと次のバイトの上位ビットから current を更新
	buffer.offset += num;
	if(buffer.offset >= 32) {
		buffer.offset -= 32;
		buffer.pointer++;
		if(buffer.offset > 0) {
			buffer.current |= (buffer.data[buffer.pointer] >> (num-buffer.offset));
		}
		result = (buffer.current & mask) >> shift;
```

```
            buffer.current = buffer.data[buffer.pointer] << buffer.offset;
      } else {
            result = (buffer.current & mask) >> shift;
            buffer.current <<= num;
      }
      return result;
}

// n ビット取得する関数 (通常)
unsigned int get_bits(unsigned int num) {
      return get_bits_x(num, bitmask[num-1], 32-num);
}
```

3.2.2. スタートコード探索 (スタートコード・サーチ)

ビデオビューアではスタートコードサーチは頻繁に実行される。具体例として、以下の二つが挙げられる。

- デコード中の次フレーム開始ポイントの探索 (ピクチャヘッダ)
- RTP におけるパケット廃棄対策としての再同期ポイントの探索 (ピクチャヘッダ、スライスヘッダ、再同期マーカ)

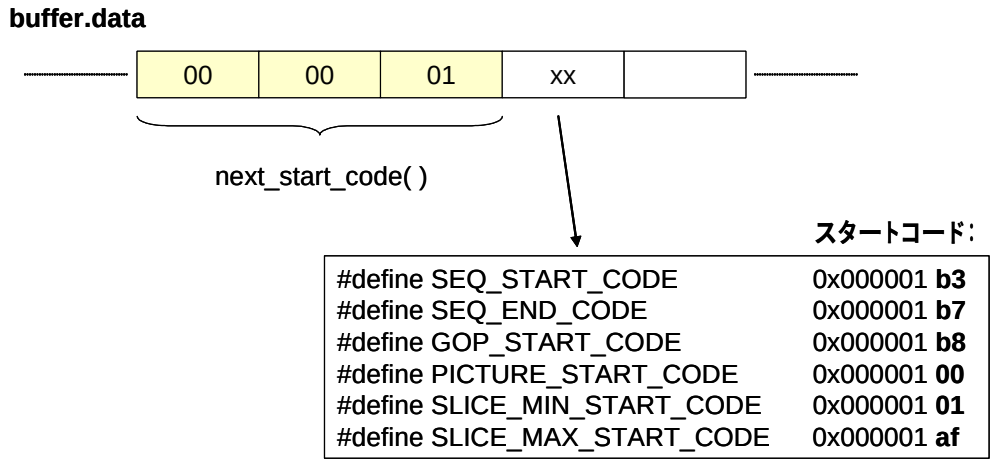


図 3： スタートコードサーチの原理

図 3 は、スタートコードサーチの原理と、MPEG-1 Video の場合のスタートコード一覧を示している。スタートコードの作り方にはルールがあり、MPEG-1 Video の場合、スタートコードは 16 進数で

0x00000100 ～ 0x000001xx

の範囲になければならない。その一方で、符号化ルールとして、スタートコードはバイトアラインされており、かつスタートコード以外は 0x000001 を生成してはならない。このため、圧縮ストリーム中に 0x000001 の 3 バイトの並びを探索すれば、スタートコードサーチを実現できる。表 5 には、これを実

現する next_start_code( ) 関数とその使用例 (header_search() 関数) を示す。

表 5： next_start_code( ) 関数

```
#define SEQ_START_CODE 0x000001b3                    // シーケンス・ヘッダ (開始)
#define SEQ_END_CODE 0x000001b7                      // シーケンス・ヘッダ (終了)
#define GOP_START_CODE 0x000001b8                    // GOP ヘッダ
#define PICTURE_START_CODE 0x00000100                // ピクチャ・ヘッダ
#define SLICE_MIN_START_CODE 0x00000101              // スライス・ヘッダ (最小)
#define SLICE_MAX_START_CODE 0x000001af              // スライス・ヘッダ (最大)
#define EXT_START_CODE 0x000001b5
#define USER_START_CODE 0x000001b2
#define SEQUENCE_ERROR_CODE 0x000001b4

// スタートコードサーチ関数
bool next_start_code() {
      int state;
      int byteoff, data;

      // バイトアライン (強制的にバイト境界に buffer.offset を合わせる)
      byteoff = buffer.offset % 8;
      if(byteoff != 0) flush_bits(8-byteoff);

      // スタートコード・サーチ
      state = 0;                                     // 状態変数 (3: サーチ成功)
      while(buffer.data[i]) {                        // バッファにデータがある限り探索する
            data = get_bits(8);                      // 8bit 取得
            if(data == 0x00) {                       // 0x00 or 0x0000
                  if(state<2) state++;               //   サーチ順調
            } else if(data == 0x01) {                // 0x000001 ?
                  if(state==2) state++;              //   サーチ成功
                  else state = 0;                    //   サーチ失敗
            } else {                                 // 0x00, 0x01 以外
                  state = 0;                         //   サーチ失敗
            }
            if(state == 3) {                         // 0x000001 が見つかった場合のみ
                  buffer.offset = buffer.offset - 24;
                  if(buffer.offset < 0) {
                        buffer.offset += 32;
                        buffer.pointer--;
```

```
                        buffer.current = buffer.data[buffer.pointer] << buffer.offset;
                  } else {
                        buffer.current = buffer.data[buffer.pointer] << buffer.offset;
                  }
                  return OK;                    // サーチ成功 (OK)
            }
            i++;                                // 次のバイトへ
      }
      return ERROR;                             // サーチ失敗 (ERROR)
}

// スタートコードサーチ関数の使用例
bool header_search() {
      bool result;
      unsigned int code;

      // スタートコード・サーチ
      result = next_start_code();
      if(result) code = get_bits(8);
      else return ERROR;
      switch(code) {
      case  SEQ_START_CODE:
            // シーケンスレイヤ処理;
            break;
      case  GOP_START_CODE:
            // GOP レイヤ処理;
            break;
      case  PICTURE_START_CODE:
            // ピクチャレイヤ処理;
            break;
      default:
            return ERROR;
      }
      return OK;
}
```

自由課題：
(3) mpeg_play の tar ボールをダウンロードし、make を行い、実行してみよ。また、ソースファイルの util.c や util.h の中の関数を観察してみよ。

3.3. テーブル参照

ソフトウェアを高速動作させるための典型手法として、

- テーブル参照 (配列参照)
- アセンブラ記述

の二つが挙げられる。テーブル参照は簡単で効果が高く、多くのビデオコーデック・ソフトウェアにおいて、

- DCT (離散コサイン変換)・IDCT (逆離散コサイン変換)
- 量子化・逆量子化
- エントロピー符号化 (ハフマン符号)・復号
- YUV/RGB 変換

等で多用されている。ソフトウェアを高速化したい場合、まずテーブル参照で解決できるかどうかを考えてみるとよい。

自由課題：
(4) mpeg_play のソースファイルの jrevdct.c と floatdct.c の記述を比べてみよ。また、一方を使って make を行い、実行時間を比べてみよ (注： 最近は浮動小数点アクセラレーションが普及しているので、大きな差は出ないかもしれない)。

3.4. ファイル入力からストリーム入力への変換

ソフトウェアコーデックの多くはファイル入力を前提としている。これらをストリーミング対応に改造したい場合は以下の作業を行う。

- ファイル入出力を、関数外部からのデータ入出力に置換する。
  - すべてのファイル入出力関数 (fread 関数等) を探索してコメントアウトする。
  - ファイル入力、ファイル出力それぞれのデータ格納先 (ポインタ or 配列) を探索する。
  - 関数外部と上記のデータ格納先がデータ入出力を行うようにコードを追加する。
- 典型例として、表 6 に示す関数構成とする。ライブラリ化 (DLL 化) してもよい。

表 6: ストリーム対応の関数構成の例

(a) エンコーダ

```
video_init();                  // 初期化（メモリ確保、各種パラメータ初期化）
video_encode(&YUV, &stream);   // ビデオ・エンコード（YUV: 入力画像、stream: 圧縮ストリーム）
video_close();                 // 終了処理（メモリ開放、各種終了処理）
```

(b) デコーダ

```
video_init();                   // 初期化（メモリ確保、各種パラメータ初期化)
video_decode(&stream, &YUV);    // ビデオ・デコード（stream: 圧縮ストリーム、YUV: 復号画像)
video_close();                  // 終了処理（メモリ開放、各種終了処理)
```

さらに、ビデオコーデックは概して制御変数 (グローバル変数) が多いので、C++ のクラスを用いて、表 7 のようにカプセル化してしまってもよい (その後が扱いやすくなる)。

表 7: C++クラスの例

```
class VideoEncoder {
private:
      int *YUV;                 // YUV 入力
      int *stream;              // 圧縮ストリーム
public:
      bool init();              // 初期化（コンストラクタでも可)
      bool encode();            // ビデオ・エンコード
      bool close();             // 終了処理（デコンストラクタでも可)
};

class VideoDecoder {
private:
      int *stream;              // 圧縮ストリーム
      int *YUV;                 // YUV 出力
public:
      bool init();              // 初期化（コンストラクタでも可)
      bool decode();            // ビデオ・デコード
      bool close();             // 終了処理（デコンストラクタでも可)
};
```

自由課題：

(5) mpeg_play や H.26L のソースファイルをダウンロードし、ファイル入出力型からストリーム入出力型に関数定義を変更してみなさい。さらに、新しく定義した関数の外部でファイル入出力を行い、もとのソースと同じ動作をすることを確認しなさい。

4. サーバとビューア

4.1. 基本構成

ストリーミングサーバとストリーミングビューアの基本構成は図 4 のようになる。

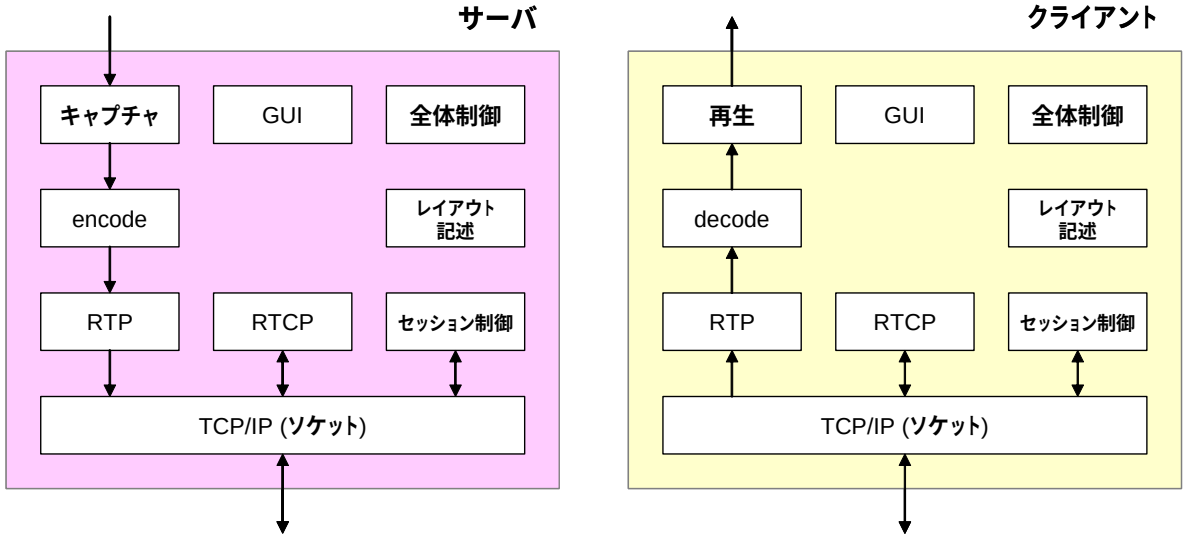


図 4: ストリーミングサーバ、ストリーミングビューアの基本構成

4.2. 個々のパート

個々のパートのポイントをまとめると以下のようになる。

(a) キャプチャ、再生、TCP/IP： 画像情報特論 HP 参照

(b) encode、decode： 3 節参照

(c) RTP/RTCP：

RTP、RTCP を忠実に実装する。RFC1889 には、以下の表 8 に示す RTP ヘッダ、RTCP パケットの構造体記述の例が記載されている。

表 8: RTP ヘッダ、RTCP パケットの構造体記述 (RFC1889)

```
typedef unsigned char  u_int8;
typedef unsigned short u_int16;
typedef unsigned int   u_int32;
typedef          short int16;

// RTP Header
typedef struct {
      unsigned int version:2;        /* protocol version */
      unsigned int p:1;              /* padding flag */
      unsigned int x:1;              /* header extension flag */
```

```
        unsigned int cc:4;              /* CSRC count */
        unsigned int m:1;               /* marker bit */
        unsigned int pt:7;              /* payload type */
        u_int16 seq;                    /* sequence number */
        u_int32 ts;                     /* timestamp */
        u_int32 ssrc;                   /* synchronization source */
        u_int32 csrc[1];                /* optional CSRC list */
} rtp_hdr_t;

// RTCP Common Header
typedef struct {
        unsigned int version:2;         /* protocol version */
        unsigned int p:1;               /* padding flag */
        unsigned int count:5;           /* varies by packet type */
        unsigned int pt:8;              /* RTCP packet type */
        u_int16 length;                 /* pkt len in words, w/o this word */
} rtcp_common_t;

// RTCP Report Block
typedef struct {
        u_int32 ssrc;                   /* data source being reported */
        unsigned int fraction:8;        /* fraction lost since last SR/RR */
        int lost:24;                    /* cumul. no. pkts lost (signed!) */
        u_int32 last_seq;               /* extended last seq. no. received */
        u_int32 jitter;                 /* interarrival jitter */
        u_int32 lsr;                    /* last SR packet from this source */
        u_int32 dlsr;                   /* delay since last SR packet */
} rtcp_rr_t;

// RTCP SDES
typedef struct {
        u_int8 type;                    /* type of item (rtcp_sdes_type_t) */
        u_int8 length;                  /* length of item (in octets) */
        char data[1];                   /* text, not null-terminated */
} rtcp_sdes_item_t;

// RTCP Packet
typedef struct {
        rtcp_common_t common;           /* common header */
```

```
      union {
            // Sender Report (SR)
            struct {
                  u_int32 ssrc;      /* sender generating this report */
                  u_int32 ntp_sec;   /* NTP timestamp */
                  u_int32 ntp_frac;
                  u_int32 rtp_ts;    /* RTP timestamp */
                  u_int32 psent;     /* packets sent */
                  u_int32 osent;     /* octets sent */
                  rtcp_rr_t rr[1];   /* variable-length list */
            } sr;

            // Receiver Report (RR)
            struct {
                  u_int32 ssrc;      /* receiver generating this report */
                  rtcp_rr_t rr[1];   /* variable-length list */
            } rr;

            // Source Description (SDES)
            struct rtcp_sdes {
                  u_int32 src;       /* first SSRC/CSRC */
                  rtcp_sdes_item_t item[1]; /* list of SDES items */
            } sdes;

            // BYE
            struct {
                  u_int32 src[1];    /* list of sources */
            } bye;
      } r;
} rtcp_t;
```

(d) セッション制御、レイアウト記述：

プログラミング的には、セッション制御の記述が最も厄介である。H.323 の場合は ASN.1 PER の符号化モジュールを作成する必要がある。SIP や RTSP の場合は、HTTP の処理ソースが参考になる。一方、レイアウト記述として SMIL を用いる場合は、HTML の処理ソースが参考になると思われる。ただし、ここではセッション制御やレイアウト記述については言及しない。

4.3. **全体構成**

個々のパートの実装方針を決定する。ストリーミングのようなさまざまなモジュールが組み合わさったアプリケーションでは、概して C よりも C++でオブジェクト指向設計を行ったほうがプログラミングは行いやすい。ここでは参考例として、以下のような設計方針とする。

ストリーミングサーバ
コールバック： キャプチャ
タイムイベント＋マルチスレッド： encode
マルチスレッド(送信・受信別)： RTP/UDP/IP、RTCP (受信)
タイムイベント： RTCP (送信)

ストリーミングビューア
マルチスレッド： RTP/UDP/IP、RTCP (受信)
タイムイベント： RTCP (送信)
タイムイベント＋マルチスレッド： decode
イベント呼び出し： 表示

自由課題：
(6) 上記の設計指針のサブセットとして、ダミーデータを送受信する RTP/RTCP 送受信プログラムを作成してみよ。
(7) 上記の設計指針のサブセットとして、ビデオデータをキャプチャし、encode し、ファイルに書き出すプログラムを作成してみよ。
(8) 上記の設計指針のサブセットとして、圧縮データを定期的に読み出し、decode し、ディスプレイに表示するプログラムを作成してみよ。

自由課題 (発展版)：
(9) インターネット放送プロトタイプ： (6)+(7)+(8) として、ビデオデータをキャプチャ＆ encode し、スライス単位に区切って RTP パケットを送出するサーバプログラムと、受け取った RTP パケットから圧縮データを再構成し、decode ＆表示するビューアプログラムを作成してみよ。
(10) インターネット TV 電話プロトタイプ： 同じく(6)+(7)+(8) とし、さらに encode と decode を共存させ、双方向でビデオの送受信を行うプログラムを作成せよ。
(11) (10) のプログラムにパケット廃棄対策と TCP フレンドリの機構を取り入れてみよ。

以上