デジタルテレビジョン放送における
CAS 処理の実装と最適化

デジタル放送総合技術研究開発機構

2012 年 9 月 15 日

目次

1 概要

1.1 目的

デジタルテレビジョン放送においては、CAS によるコンテンツ保護が行われている。テレビジョン放送は今日我々の身近なものとなっており、その放送に用いられている CAS の仕組みは興味深いものである。しかしながら、この CAS 処理を具体的に示した文献等は少なく、その実際の処理を理解し辛いのが現状である。

そこで、デジタルテレビジョン放送に用いられている CAS 処理の実装を試みる。CAS 処理の仕様は社団法人電波産業会 (ARIB) により規格として公開されており、その規格に基づいて実装を行う。また、実装の最適化について考察する。

1.2 前提

実装には C 言語を用いる。コードは ISO/IEC 9899:1999 規格に準拠する。

2 CAS の基礎

2.1 CAS とは

CAS (Conditional Access System : 限定受信システム) とは、デジタル放送におけるコンテンツ保護の仕組みである。暗号化技術を用いて、特定の者以外が放送を視聴することを防止する。

2.2 ARIB 限定受信方式

日本のデジタルテレビジョン放送には、ARIB (Association of Radio Industries and Businesses : 一般社団法人電波産業会) により規格策定された ARIB 限定受信方式が用いられている。

ARIB 限定受信方式は 2000 年 12 月 1 日の BS デジタル放送の開始と共に導入された。その後、2002 年 3 月 1 日に開始した 110 度 CS デジタル放送、2003 年 12 月 1 日に開始された地上デジタル放送にも用いられている。

ARIB 限定受信方式では、視聴の制御に IC カード (スマートカード) を用いる。以下、CAS 処理に用いる IC カードを CAS カードと呼ぶ。

3 デジタル放送の基礎

3.1 ISDB

日本のデジタル放送は ISDB (Integrated Services Digital Broadcasting : 統合ディジタル放送サービス) と呼ばれる方式で行われている。日本放送協会 (NHK) を中心として開発され、衛星デジタル放送用の ISDB-S、地上デジタル放送用の ISDB-T 等がある。

3.1.1 伝送形式

日本のデジタル放送の伝送には、MPEG-2 TS (Transport Stream) が用いられている。MPEG-2 TS は、映像や音声などを多重化するための規格であり、ISO/IEC 13818-1 及び ITU-T 勧告 H.222.0 において標準化されている。

3.2 MPEG-2 TS

3.2.1 TS パケット

MPEG-2 TS は、TS パケット (Transport Packet) と呼ばれる固定長のパケットを単位とする。

TS パケットの先頭 4 バイトはヘッダとなる。ヘッダの各ビットの内容を表 1 に示す。

表 1　TS パケットヘッダ

ビット位置	ビット長	内容
+0	8	同期ワード
+8	1	トランスポート・エラー・インジケータ
+9	1	ペイロード・ユニット・スタート・インジケータ
+10	1	トランスポート・プライオリティ
+11	13	パケット識別子 (PID)
+24	2	トランスポート・スクランブリング・コントロール
+26	2	アダプテーション・フィルード・コントロール
+28	4	巡回カウンタ

同期ワードは 47h の固定値であり、この値を探すことでパケットの先頭を検出できる。

一つの TS パケットの大きさは 188 バイトと規定されているが、各パケットの先頭や末尾にデータを付加する場合もある。代表的なものとして、末尾に FEC (Forward Error Correction : 前方誤り訂正) と呼ばれる 16 バイトの誤り訂正情報を付加したものや、先頭に 4 バイトのタイムスタンプ情報を付加したものがある。

3.2.2 ES

符号化された映像や音声などのそれぞれ一連のデータを ES (Elementary Stream) と呼ぶ。ES は適当な大きさに分割されパケット化されるが、それを PES (Packetized Elementary Stream) と呼ぶ。この分割の単位は具体的に規定されていないが、運用上においてはピクチャ単位など意味のある単位ごとに分割する場合が多い。PES はさらに TS パケットに分割される。

3.2.3 PSI

関連する映像や音声のまとまりをプログラムと呼び、TS に含まれるプログラムに関する情報を PSI (Program Specific Information) と呼ぶ。

4　暗号化の概要

4.1　MULTI2

日本のデジタル放送は、日立製作所により開発された MULTI2 暗号が用いられている。MULTI2 はブロック長 64 ビット、鍵長 64 ビットのブロック暗号である。MULTI2 は 1988 年に特許出願・登録され (特許番号第 2760799 号)、2008 年に特許期限切れとなっている。

暗号化は TS パケット単位で行い、TS パケットのペイロード部を対象とする。

4.2　ECM

放送の送出側において、映像や音声などのデータを MULTI2 で暗号化する。この時暗号化に用いる鍵を Ks (スクランブル鍵) と呼ぶ。Ks は Kw (ワーク鍵) によって暗号化され、ECM (Entitlement Control Message) の形で TS に埋め込まれる。

受信機は、TS 中の ECM を CAS カードに送信する。カードは自身に記憶されている Kw を用いて ECM を復号し、Ks を返す。カードから Ks が得られれば、受信機はこの Ks を用いて暗号化された映像や音声などを復号する。

4.3　EMM

視聴に契約が必要な放送では、契約を行った者が視聴できるようにするため、EMM によってその放送の復号に必要な情報が伝送される。

CAS カードは、それぞれの ID ごとにユニークな鍵 Km (マスター鍵) を持つ。送出側では Kw や 契約情報を、対象とするカードの Km を用いて暗号化し、カード ID と共に EMM (Entitlement Management Message) の形で TS に埋め込む。受信機は、ID によって自身の CAS カードに対する EMM を取り出し、CAS カードに送信する。CAS カードは EMM を Km を用いて復号し、Kw や契約情報をカード内に記憶する。

5　IC カード処理

5.1　IC カードの基礎

IC カードの基本動作として、IC カードへコマンドが送信され、それを IC カードが処理し、レスポンスを返す。この送受信されるデータを APDU (Application Protocol Data Unit：応用プロトコルデータ単位) と呼ぶ。IC カードのコマンドは ISO/IEC 7816 によって規格化されている。CAS カードのコマンドとレスポンスは ISO/IEC 7816-4:1995 規格に準拠している。

コマンドは 4 バイトのヘッダ部と可変長の本体部からなる。コマンドのヘッダ部の構成を表 2 に示す [2, 5.3.1]。

表 2　コマンドヘッダ

表記	内容	バイト長
CLA	クラスバイト	1
INS	命令バイト	1
P1	パラメータバイト 1	1
P2	パラメータバイト 2	1

CLA はコマンドクラスを表す。コマンドクラスは、コマンドが ISO/IEC 7816-4 に準拠するか否かなどを表す。CLA の最上位ビットが 0 である場合、ISO/IEC 7816-4 に規定されているコマンドであり、これをユーザコマンドと呼ぶ。CLA の最上位ビットが 1 である場合、ISO/IEC 7816-4 に規定されていないコマンドであり、これを準システムコマンドと呼ぶ。

INS は命令コードを表す。命令コードはコマンドが行う命令を表す。

P1 及び P2 は命令のパラメータを表す。パラメータを使用しない場合は値を 0 とする。

コマンド及びレスポンスは、それぞれデータを含む場合と含まない場合があり、その組み合わせによって異なるコマンド APDU の構成となる。コマンドとレスポンスそれぞれにデータを含む場合と含まない場合の本体部の構成を表 3 に示す。

表 3　データの有無によるフィルードの構成

コマンドデータ	レスポンスデータ	構成
なし	なし	
なし	あり	Le
あり	なし	Lc DATA
あり	あり	Lc DATA Le

Le はレスポンス長を表す。レスポンスとして受け取る最大長をバイト単位で表す。

Lc は データ長を表す。送信するデータ長をバイト単位で表す。

DATA は送信するデータを表す。

Lc は 1 又は 3 バイトであり、カードが拡張 Lc に対応する場合、先頭バイトの値が 0 である時続く 2 バイトでサイズを表す。さもなくば 1 バイトでサイズを表す。

Le は 1,2 又は 3 バイトであり、カードが拡張 Le に対応する場合、拡張 Lc が存在する時 2 バイトでサイズを表し、さもなくば先頭バイトの値が 0 である時続く 2 バイトでサイズを表す。カードが拡張 Le に対応しないか、拡張 Lc が存在せず且つ先頭バイトが 0 でない場合、1 バイトでサイズを表す。Le の値が 0 である場合、短縮 Le では 256、拡張 Le では 65536 を表す。

レスポンス APDU の構成を表 4 に示す [2, 5.3.3]。

表 4　レスポンス APDU

表記	内容	バイト長
DATA	データフィールド	可変長
SW1	ステータスバイト 1	1
SW2	ステータスバイト 2	1

レスポンスにデータが含まれない場合、DATA は存在しない。SW1 と SW2 はステータスワードと呼ばれ、コマンドが正常終了したかなどを表す。

5.2　CAS カードの APDU

CAS カードのコマンド APDU の構成を表 5 に示す。

表 5　コマンド APDU

分類	名称	バイト長	値
ヘッダ	CLA	1	90h
	INS	1	
	P1	1	00h
	P2	1	00h
ボディ	Lc	1	DATA 長
	DATA	Lc	
	Le	1	00h

レスポンス APDU の構成を表 6 に示す。

表 6　レスポンス APDU

分類	名称	バイト長	備考
ボディ	DATA	可変長	オプション
トレーラ	SW1	1	
	SW2	1	

5.3　CAS カードコマンド

CAS カードで用いられるコマンドを示す。

5.3.1　初期設定条件コマンド

CAS カードに初期設定条件コマンドを送信することにより、そのカードの ID や種別などを得る。初期設定条件コマンドの構成を表 7 に示す。

表 7　初期設定条件コマンド

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	CLA	CLA コード	90h
+1	1	INS	INS コード	30h
+2	1	P1	パラメータ 1	00h
+3	1	P2	パラメータ 2	00h
+4	1	Le	レスポンスデータ長	00h

初期設定条件レスポンスの構成を表 8 に示す。

表 8　初期設定条件レスポンス

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	Data	プロトコルユニット番号	00h
+1	1		ユニット長	
+2	2		IC カード指示	
+4	2		リターンコード	
+6	2		CA_system_id	
+8	6		カード ID	
+14	1		カード種別	
+15	1		メッセージ分割長	
+16	32		デスクランブラシステム鍵	
+48	8		デスクランブラ CBC 初期値	
+56	1		System_management_id 数	N
+57 2	2 2		System_management_id(1) ... System_management_id(N)	
+57+2N	1	SW1		
+58+2N	1	SW2		

5.3.2　ECM 受信コマンド

ECM 受信コマンドを送信することにより、ECM から Ks などを得る。ECM 受信コマンドの構成を表 9 に示す。

表 9　ECM 受信コマンド

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	CLA	CLA コード	90h
+1	1	INS	INS コード	34h
+2	1	P1	パラメータ 1	00h
+3	1	P2	パラメータ 2	00h
+4	1	Lc	コマンドデータ長	N
+5	N	Data	ECM	
N+5	1	Le	レスポンスデータ長	00h

ECM 受信レスポンスの構成を表 10 に示す。

表 10　ECM 受信レスポンス

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	Data	プロトコルユニット番号	00h
+1	1		ユニット長	
+2	2		IC カード指示	
+4	2		リターンコード	
+6	8		Ks (odd)	
+14	8		Ks (even)	
+22	1		録画制御	
+23	1	SW1		
+24	1	SW2		

5.3.3　EMM 受信コマンド

EMM データを送信する。EMM 受信コマンドの構成を表 11 に示す。

表 11　EMM 受信コマンド

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	CLA	CLA コード	90h
+1	1	INS	INS コード	36h
+2	1	P1	パラメータ 1	00h
+3	1	P2	パラメータ 2	00h
+4	1	Lc	コマンドデータ長	N
+5	N	Data	EMM データ	
N+5	1	Le	レスポンスデータ長	00h

EMM 受信レスポンスの構成を表 12 に示す。

表 12　EMM 受信レスポンス

バイト位置	バイト長	フィールド	内容	値 (16 進)
+0	1	Data	プロトコルユニット番号	00h
+1	1		ユニット長	
+2	2		IC カード指示	
+4	2		リターンコード	
+6	1	SW1		
+7	1	SW2		

6　CAS 処理の実装

CAS 処理の実装を試みる。

6.1　MULTI2 の実装

MULTI2 暗号の復号処理の実装を行う。

6.1.1　鍵データ構造

初めに、ブロックを表す構造体 `multi2_block_t` と、システム鍵を表す構造体 `multi2_system_key_t` を定義する。

```
typedef struct multi2_block_t {
    uint32_t left;
    uint32_t right;
} multi2_block_t;

typedef struct multi2_system_key_t {
    uint32_t key[8];
} multi2_system_key_t;
```

6.1.2　ラウンド関数

MULTI2 では、4 種類のラウンド関数の繰り返しによって暗号化を行う。ラウンド関数はそれぞれ π_1 π_2 π_3 π_4 と呼ばれる。それぞれのラウンド関数の計算式を以下に示す [1, 3.1.4]。

π_1

$$\pi_1(T) = T[left] \| (T[left] \oplus T[right])$$

π_2

$$
\begin{aligned}
x &= T[right] \\
y &= x + k_1 \\
z &= Rot_1(y) + y - 1 \\
\pi_2 k_1(T) &= (T[left] \oplus (Rot_4(z) \oplus z)) \| T[right]
\end{aligned}
$$

π_3

$$
\begin{aligned}
x &= T[left] \\
y &= x + k_2 \\
z &= Rot_2(y) + y + 1 \\
a &= Rot_8(z) \oplus z \\
b &= a + k_3 \\
c &= Rot_1(b) - b \\
\pi_3 k_2, k_3(T) &= T[left] \| (T[right] \oplus (Rot_{16}(c) \oplus (c \vee x)))
\end{aligned}
$$

π_4

$$
\begin{aligned}
x &= T[right] \\
y &= x + k_4 \\
\pi_4 k_4(T) &= (T[left] \oplus (Rot_2(y) + y + 1)) \| T[right]
\end{aligned}
$$

- T は関数への入力を表す。
- $[left]$ はブロックの左 32 ビットを表す。
- $[right]$ はブロックの右 32 ビットを表す。
- $\oplus$ はビット単位の排他的論理和を表す。
- $+$ は 2^{32} を法とした加算を表す。
- $-$ は 2^{32} を法とした減算を表す。
- Rot_n は左巡回 $_n$ ビットシフトを表す。
- $\vee$ はビット単位の論理和を表す。
- $\|$ はブロックの結合を表す。

この式に従い、それぞれのラウンド関数を以下のように実装した。

```
uint32_t rotate(uint32_t x, int shift)
{
    return (x << shift) | (x >> (32 - shift));
}

void round_pi1(multi2_block_t *block)
{
    block->right ^= block->left;
}
```

```
void round_pi2(multi2_block_t *block, uint32_t k1)
{
    uint32_t y = block->right + k1;
    uint32_t z = rotate(y, 1) + y - 1;
    block->left ^= rotate(z, 4) ^ z;
}

void round_pi3(multi2_block_t *block, uint32_t k2, uint32_t k3)
{
    uint32_t y = block->left + k2;
    uint32_t z = rotate(y, 2) + y + 1;
    uint32_t a = rotate(z, 8) ^ z;
    uint32_t b = a + k3;
    uint32_t c = roate(b, 1) - b;
    block->right ^= rotate(c, 16) ^ (c | block->left);
}

void round_pi4(multi2_block_t *block, uint32_t k4)
{
    uint32_t y = block->right + k4;
    block->left ^= rotate(y, 2) + y + 1;
}
```

rotate はビット単位の左巡回シフトを行う関数である。各ラウンド関数では、渡されたブロックのデータと鍵を用いて処理を行う。

6.1.3 鍵スケジュール

次に、鍵スケジュールを実装する。鍵スケジュールの計算式を以下に示す [1, 3.1.3]。

$$
\begin{aligned}
S_K &= s_1 \| s_2 \| \ldots \| s_8 \\
a_1 &= \pi_2 s_1 \cdot \pi_1(D_K) \\
w_1 &= a_1[left] \\
a_2 &= \pi_3 s_2, s_3(a_1) \\
w_2 &= a_2[right] \\
a_3 &= \pi_4 s_4(a_2) \\
w_3 &= a_3[left] \\
a_4 &= \pi_1(a_3) \\
w_4 &= a_4[right] \\
a_5 &= \pi_2 s_5(a_4) \\
w_5 &= a_5[left] \\
a_6 &= \pi_3 s_6, s_7(a_5)
\end{aligned}
$$

$$
\begin{aligned}
w_6 &= a_6[right] \\
a_7 &= \pi_4 s_8(a_6) \\
w_7 &= a_7[left] \\
a_8 &= \pi_1(a_7) \\
w_8 &= a_8[right] \\
W_K &= w_1 \| w_2 \| \ldots \| w_8
\end{aligned}
$$

この式に従い、鍵スケジュールを以下のように実装した。

```
void key_schedule(
    multi2_system_key_t *work_key,
    const multi2_system_key_t *sys_key,
    multi2_block_t *data_key)
{
    round_pi1(data_key);

    round_pi2(data_key, sys_key->key[0]);
    work_key->key[0] = data_key->left;

    round_pi3(data_key, sys_key->key[1], sys_key->key[2]);
    work_key->key[1] = data_key->right;

    round_pi4(data_key, sys_key->key[3]);
    work_key->key[2] = data_key->left;

    round_pi1(data_key);
    work_key->key[3] = data_key->right;

    round_pi2(data_key, sys_key->key[4]);
    work_key->key[4] = data_key->left;

    round_pi3(data_key, sys_key->key[5], sys_key->key[6]);
    work_key->key[5] = data_key->right;

    round_pi4(data_key, sys_key->key[7]);
    work_key->key[6] = data_key->left;

    round_pi1(data_key);
    work_key->key[7] = data_key->right;
}
```

6.1.4 復号

次に、MULTI2 の復号を実装する。

システム鍵などの状態を格納する構造体 multi2_t を以下のように定義した。

```
typedef struct multi2_t {
    multi2_block_t initial_cbc;
    multi2_system_key_t system_key;
    multi2_system_key_t work_key_odd, work_key_even;
} multi2_t;
```

ブロックやシステム鍵にデータの設定や取得を行う関数を実装する。

```
uint32_t load_uint32(const uint8_t *src)
{
    return ((uint32_t)src[0] << 24) |
           ((uint32_t)src[1] << 16) |
           ((uint32_t)src[2] <<  8) |
           ((uint32_t)src[3]);
}

void store_uint32(uint8_t *dst, uint32_t src)
{
    dst[0] = (uint8_t) (src >> 24);
    dst[1] = (uint8_t)((src >> 16) & 0xFF);
    dst[2] = (uint8_t)((src >>  8) & 0xFF);
    dst[3] = (uint8_t)( src        & 0xFF);
}

void load_block(multi2_block_t *block, const uint8_t *data)
{
    block->left  = load_uint32(&data[0]);
    block->right = load_uint32(&data[4]);
}

void store_block(uint8_t *data, const multi2_block_t *block)
{
    store_uint32(&data[0], block->left);
    store_uint32(&data[4], block->right);
}
```

```
void load_system_key(multi2_system_key_t *sys_key, const uint8_t *data)
{
    int i;

    for (i = 0; i < 8; i++)
    {
        sys_key->key[i] = load_uint32(&data[i * 4]);
    }
}
```

次に、MULTI2 を初期化する関数と、スクランブル鍵を設定する関数を定義する。

```
void multi2_initialize(
    multi2_t *multi2,
    const uint8_t *system_key_data,
    const uint8_t *initial_cbc_data)
{
    load_system_key(&multi2->system_key, system_key_data);
    load_block(&multi2->initial_cbc, initial_cbc_data);
}

void multi2_set_scramble_key(
    multi2_t *multi2,
    const uint8_t *scramble_key)
{
    multi2_block_t key_odd, key_even;

    load_block(&key_odd,  &scramble_key[0]);
    load_block(&key_even, &scramble_key[8]);

    key_schedule(&multi2->work_key_odd,  &multi2->system_key, &key_odd);
    key_schedule(&multi2->work_key_even, &multi2->system_key, &key_even);
}
```

multi2_initialize 関数では、システム鍵と初期 CBC を設定する。multi2_set_scramble_key 関数では、スクランブル鍵を設定する。

暗号化されたデータを復号する関数を実装する。

```
void multi2_decrypt(
    multi2_t *multi2,
    uint8_t *data,
    size_t data_size,
```

```
    uint8_t scrambling_ctrl)
{
    const multi2_system_key_t *work_key;
    multi2_block_t cbc_data, src_data;
    size_t data_pos;

    if ((scrambling_ctrl != 2) || (scrambling_ctrl != 3))
        return;

    if (scrambling_ctrl == 3)
        work_key = &multi2->work_key_odd;
    else
        work_key = &multi2->work_key_even;

    cbc_data = multi2->initial_cbc;

    for (data_pos = 0; data_pos + 8 <= data_size ; data_pos += 8)
    {
        int round;

        load_block(&src_data, &data[data_pos]);

        for (round = 0; round < MULTI2_ROUND; round++)
        {
            round_pi4(&src_data, work_key->key[7]);
            round_pi3(&src_data, work_key->key[5], work_key->key[6]);
            round_pi2(&src_data, work_key->key[4]);
            round_pi1(&src_data);
            round_pi4(&src_data, work_key->key[3]);
            round_pi3(&src_data, work_key->key[1], work_key->key[2]);
            round_pi2(&src_data, work_key->key[0]);
            round_pi1(&src_data);
        }

        src_data.left  ^= cbc_data.left;
        src_data.right ^= cbc_data.right;

        load_block(&cbc_data, &data[data_pos]);
        store_block(&data[data_pos], &src_data);
    }
```

```
    if (data_pos < data_size) {
        int round;
        uint8_t remain_data[8];
        size_t i;

        for (round = 0; round < MULTI2_ROUND; round++)
        {
            round_pi1(&cbc_data);
            round_pi2(&cbc_data, work_key->key[0]);
            round_pi3(&cbc_data, work_key->key[1], work_key->key[2]);
            round_pi4(&cbc_data, work_key->key[3]);
            round_pi1(&cbc_data);
            round_pi2(&cbc_data, work_key->key[4]);
            round_pi3(&cbc_data, work_key->key[5], work_key->key[6]);
            round_pi4(&cbc_data, work_key->key[7]);
        }

        store_block(remain_data, &cbc_data);

        for (i = 0; data_pos + i < data_size; i++)
        {
            data[data_pos + i] ^= remain_data[i];
        }
    }
}
```

MULTI2_ROUND はラウンド数を表す。

全体の復号処理の流れとしては、multi2_initialize 関数で初期化を行い、multi2_set_scramble_key 関数でスクランブル鍵を設定した後、multi2_decrypt 関数で暗号化されたデータを復号する。

6.2 カード通信の実装

CAS カードとの通信処理を実装する。

6.2.1 IC カード通信

IC カードとの通信を行う関数は標準 C 言語規格には存在せず、それぞれの動作環境に用意された関数を利用する。ここでは、IC カードとの通信を行う以下の関数を仮定して実装を行う。

```
int smart_card_transmit(
    const void *send_data, size_t send_size,
```

```
    void *receive_data, size_t *receive_size);
```

send_data は送信するデータのポインタである。send_size は送信するデータのサイズをバイト単位で表す。receive_data は受信したデータを受け取るメモリ領域のポインタである。receive_size は、呼び出し前に receive_data のサイズを設定し、呼び出し後は受信したデータのサイズが格納される変数へのポインタである。

Microsoft Windows 環境においては、同様の機能を果たす API 関数として SCardTransmit がある。smart_card_transmit をこれらの環境依存の関数に置き換えることは容易である。

なお、ここでは簡潔にするためエラー処理を省略している。実用のコードにおいては、適宜エラー処理を行うべきである。

6.2.2 カード情報の取得

まず、CAS カードから ID などの情報を取得する。CAS カードの情報を格納する構造体を以下のように定義した。

```
typedef struct cas_card_info_t {
    uint16_t ca_system_id;
    uint8_t  card_id[6];
    uint8_t  card_type;
    uint8_t  message_partition_length;
    uint8_t  system_key[32];
    uint8_t  initial_cbc[8];
} cas_card_info_t;
```

CAS カードに初期設定条件コマンドを送信し、カードの情報を得る。

```
void get_card_info(cas_card_info_t *card_info)
{
    uint8_t init_setting_command[] =
    {
        0x90,  /* CLA */
        0x30,  /* INS */
        0x00,  /* P1 */
        0x00,  /* P2 */
        0x00   /* Le */
    };
    uint8_t receive_data[256];
    size_t receive_size;

    receive_size = sizeof(receive_data);

    smart_card_transmit(
```

```
        init_setting_command, sizeof(init_setting_command),
        receive_data, &receive_size);

    card_info->ca_system_id =
        (uint16_t)((receive_data[6] << 8) | receive_data[7]);
    memcpy(card_info->card_id, &receive_data[8], 6);
    card_info->card_type = receive_data[14];
    card_info->message_partition_length = receive_data[15];
    memcpy(card_info->system_key, &receive_data[16], 32);
    memcpy(card_info->initial_cbc, &receive_data[48], 8);
}
```

6.2.3 ECM の復号

CAS カードに ECM 受信コマンドを送信し、ECM から Ks を取得する。

```
void receive_ecm(
    const void *ecm_data,
    size_t ecm_size,
    uint8_t *ks_data)
{
    uint8_t send_data[5 + ecm_size + 1];
    uint8_t receive_data[256];
    size_t receive_size;

    send_data[0] = 0x90;                        /* CLA */
    send_data[1] = 0x34;                        /* INS */
    send_data[2] = 0x00;                        /* P1 */
    send_data[3] = 0x00;                        /* P2 */
    send_data[4] = (uint8_t)ecm_size;           /* Lc */
    memcpy(&send_data[5], ecm_data, ecm_size);  /* DATA */
    send_data[5 + ecm_size] = 0x00;             /* Le */

    receive_size = sizeof(receive_data);

    smart_card_transmit(
        send_data, 5 + ecm_size + 1,
        receive_data, &receive_size);

    memcpy(ks_data, &receive_data[6], 16);
}
```

6.3 全体の処理

MULTI2 処理とカード処理を組み合わせ、CAS 処理全体を実装する。

6.3.1 初期化処理

CAS カードから情報を取得し、その情報で MULTI2 を初期化する。

```
cas_card_info_t card_info;
multi2_t multi2;

get_card_info(&card_info);
multi2_initialize(&multi2, card_info.system_key, card_info.initial_cbc);
```

6.3.2 ECM 処理

ECM を CAS カードに送信して Ks を取得し、MULTI2 のスクランブル鍵を設定する。

```
uint8_t ks_data[16];

receive_ecm(ecm_data, ecm_size, ks_data);
multi2_set_scramble_key(&multi2, ks_data);
```

この処理は ECM の内容が変化する度に行う。

6.3.3 復号処理

以上の準備が済めば、暗号化された TS パケットのペイロード部を `multi2_decrypt` で復号する。

7 処理の最適化

7.1 SIMD

SIMD (Single Instruction Multiple Data) を用いていくつかのブロックをまとめて処理することにより、処理の高速化が可能であると推測される。そこで、Intel x86 CPU の SIMD 拡張命令セットである SSE2 (Streaming SIMD Extensions 2) を用いた最適化を試みる。

ここでは、Microsoft Visual C++ や Intel C++ Compiler 等で利用可能な SSE2 組み込み関数を用いた。

7.2 ラウンド関数

ラウンド関数の最適化を試みる。

```
__m128i rotate_sse2(__m128i value, int shift)
{
    return _mm_or_si128(_mm_slli_epi32(value, shift),
```

```
                        _mm_srli_epi32(value, 32 - shift));
}

void round_pi1_sse2(__m128i *left, __m128i *right)
{
    *right = _mm_xor_si128(*right, *left);
}

void round_pi2_sse2(__m128i *left, __m128i *right, __m128i k1)
{
    __m128i t;

    t = _mm_add_epi32(*right, k1);
    t = _mm_sub_epi32(_mm_add_epi32(rotate_sse2(t, 1), t),
                      _mm_set1_epi32(1));
    t = _mm_xor_si128(rotate_sse2(t, 4), t);

    *left = _mm_xor_si128(*left, t);
}

void round_pi3_sse2(__m128i *left, __m128i *right, __m128i k2, __m128i k3)
{
    __m128i t;

    t = _mm_add_epi32(*left, k2);
    t = _mm_add_epi32(_mm_add_epi32(rotate_sse2(t, 2), t),
                      _mm_set1_epi32(1));
    t = _mm_xor_si128(rotate_sse2(t, 8), t);
    t = _mm_add_epi32(t, k3);
    t = _mm_sub_epi32(rotate_sse2(t, 1), t);
    t = _mm_xor_si128(rotate_sse2(t, 16), _mm_or_si128(t, *left));

    *right = _mm_xor_si128(*right, t);
}

void round_pi4_sse2(__m128i *left, __m128i *right, __m128i k4)
{
    __m128i t;

    t = _mm_add_epi32(*right, k4);
```

```
    t = _mm_add_epi32(_mm_add_epi32(rotate_sse2(t, 2), t),
                      _mm_set1_epi32(1));

    *left = _mm_xor_si128(*left, t);
}
```

各ラウンド関数を、SSE2 命令を用いて 128 ビット単位でまとめて行うようにしたものである。

7.3 最適化の効果

今回行った SSE2 命令を用いた実装では、4 つのブロックをまとめて処理することから、ラウンド関数が 4 倍程度高速化することが期待される。

実際に比較を行った結果、MULTI2 復号処理全体として、処理速度が 3 倍から 4 倍弱向上することを確認できた。

8 まとめ

今回 CAS 処理の実装を通して、デジタル放送におけるコンテンツ保護の仕組みを学んだ。具体的な実装を行うことで、処理内容の正確な理解が可能である。

参考文献

[1] 社団法人電波産業会, デジタル放送におけるアクセス制御方式 ARIB STD B-25 6.0 版 (2011)

[2] ISO/IEC, ISO/IEC 7816-4:1995 Identification cards —Integrated circuit cards — Part 4: Organization, security and commands for interchange (1995)