

ストリームジョイント

外部仕様書

1 9 9 9 年 3 月 1 6 日

Ver . 5.45

1 9 9 9 年 6 月 1 1 日

Ver . 5.50

2 0 0 0 年 2 月 2 5 日

Ver . 5.53

変 更 履 歴

年月日	バ ー ジ ョ ン	変 更 内 容
1999.03.16	5.45	新規作成。
1999.06.11	5.50	ライブラリのバージョンアップに伴いバージョン番号を更新。
2000.02.21	5.53	ライブラリのバージョンアップに伴いバージョン番号を更新。

目 次

1. 概 要	1
2. ストリームジョイント	2
2.1 ストリームジョイントとは	2
2.2 データ供給モジュール	3
2.3 データ消費モジュール	4
2.4 データ加工モジュール	5
3. リングバッファ型ストリームジョイント	6
4. メモリ型ストリームジョイント	8
5. 汎用型ストリームジョイント	10
5.1 汎用型ストリームジョイント作成方法	11
6. データ仕様	14
6.1 定 数	15
6.2 データ型	16
7. 関数仕様	17
7.1 共通関数	18
7.2 リングバッファ型ストリームジョイント	23
7.3 メモリ型ストリームジョイント	24
7.4 汎用型ストリームジョイント	25

1. 概 要

ストリームジョイントライブラリは、モジュール間でストリームデータを受け渡すためのライブラリです。ストリームジョイントを用いることにより、独立性の高いストリーム処理モジュールを開発することができます。例えば、下記のようにミキサモジュールを開発する場合、そのミキサモジュールは、入出力のためのストリームジョイントのみを意識して開発します。その先に接続されるモジュールに依存しないため、ミキサモジュールを様々なシステムに再利用することが可能です。

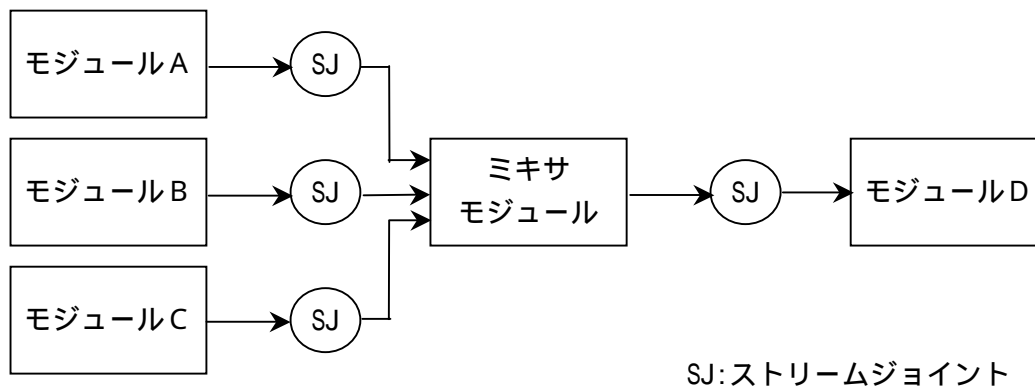


図 ストリームジョイントを用いたモジュール接続例

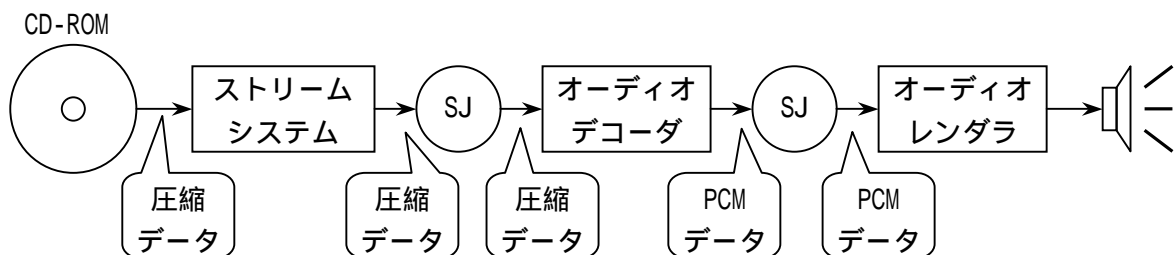


図 ストリームジョイントの使用例

2. ストリームジョイント

2.1 ストリームジョイントとは

ストリームジョイントは、モジュール間でストリームデータを受け渡すための仕組みです。データラインとフリーラインの2つのラインを持ちます。データラインは有効なデータを保持し、フリーラインは使用済みデータを保持します。ユーザは、各ラインからデータを取得(Get)したり、挿入(Put)したすることができます。各ラインは、FIFOとして動作しますので、各データは挿入順に取得できます。

データは、各ラインからデータチャンクとして取得できます。データチャンクは、データ領域へのポインタとサイズを持っています。取得したデータチャンクのうち、使用しなかったデータは、戻す(Unget)することができます。データチャンクは、以下のように定義されます。

```
/* データチャンク */
typedef struct {
    char *data;      /* データ領域へのポインタ */
    long len;        /* データ領域の大きさ */
} SJCK;
```

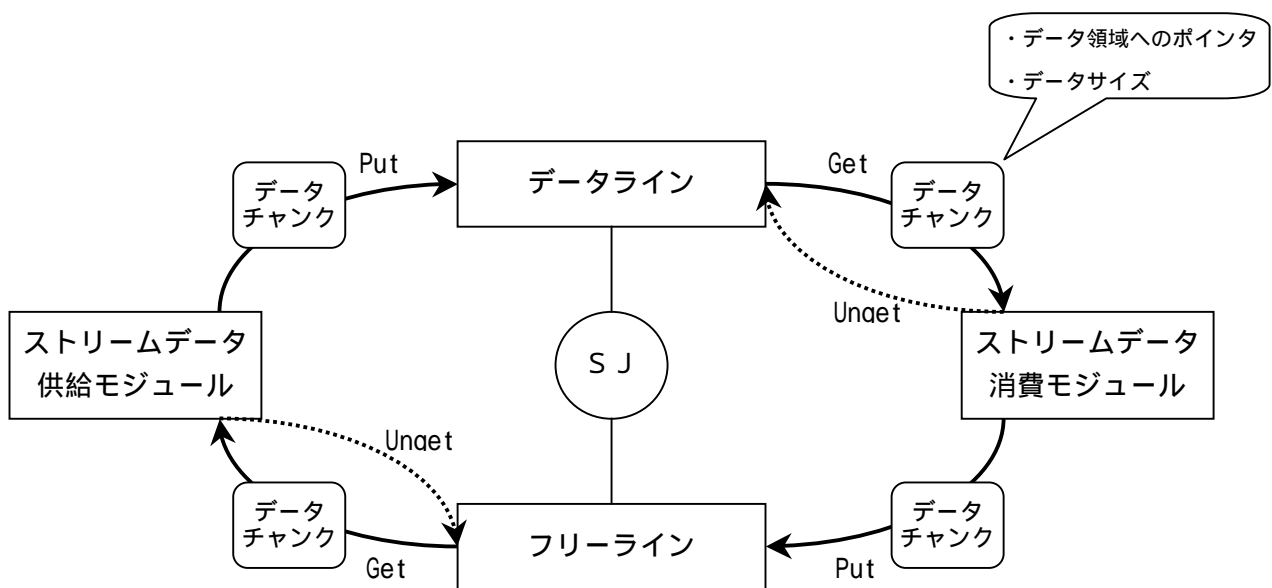
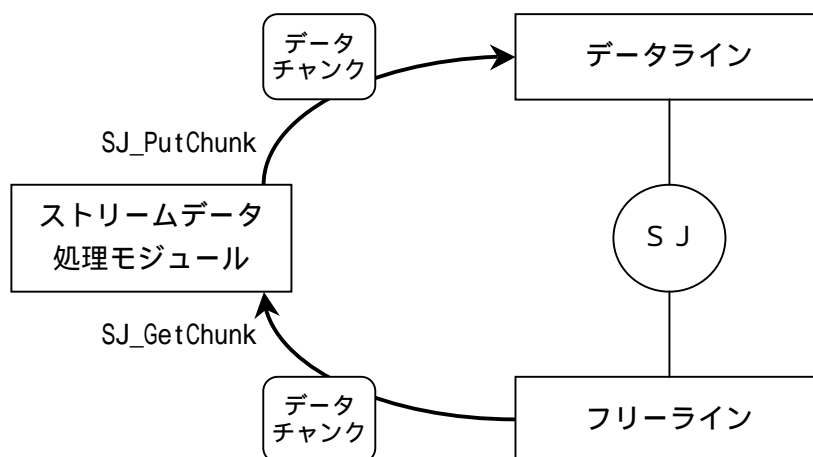


図 ストリームジョイントの仕組み

2.2 データ供給モジュール

データ供給モジュールは、フリーラインから使用済みのデータを受け取り、新しいデータを詰めて、データラインに渡します。データの供給する手順は以下の通りです。

- (1) 供給できるデータサイズ(nbyte)を指定して、フリーラインからデータチャンクを受け取る。
得られるデータチャンクのサイズは、取得できる連続データの最大サイズとなる。
従って、nbyte であることは保証されない。
SJ_GetChunk(sj, SJ_LIN_FREE, nbyte, &ck);
- (2) 受け取ったデータチャンクにデータをコピーする。
memcpy(ck.data, user_data, ck.len);
- (3) データラインにデータチャンクを挿入する。
SJ_PutChunk(sj, SJ_LIN_DATA, &ck);



< プログラム例 >

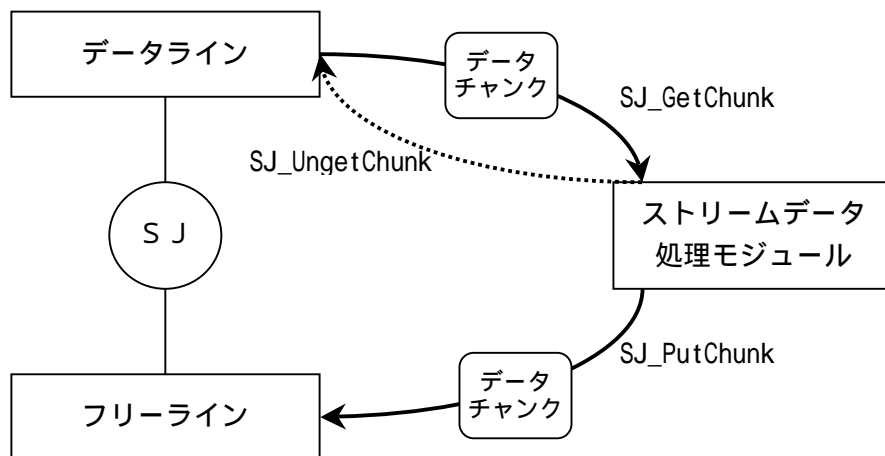
```
SJ sj;      /* ストリームジョイント */
SJCK ck;    /* データチャンク */

SJ_GetChunk(sj, SJ_LIN_FREE, 1024, &ck); /* 使用済みデータの取得 */
user_supply_func(ck.data, ck.len);      /* データの供給 */
SJ_PutChunk(sj, SJ_LIN_DATA, &ck);      /* データラインへの挿入 */
```

2.3 データ消費モジュール

データ消費モジュールは、データラインから有効なデータを受け取り、消費したデータをフリーラインに渡します。データを受け取る手順は以下の通りです。

- (1) データラインから必要なデータサイズ(nbyte)を指定して、データチャンクを受け取る。
SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck);
- (2) 受け取ったデータチャンクを処理する。(nused は使用したデータサイズ)
nused = user_consume_func(ck.data, ck.len);
- (3) データチャンクを処理の終わったチャンクと終わらなかったチャンクに分解する。
SJ_SplitChunk(&ck, nused, &ck, &ck2);
- (4) 処理の終わったデータチャンクをフリーラインに渡す。
SJ_PutChunk(sj, SJ_LIN_FREE, &ck);
- (5) 処理できなかったデータをデータチャンクに戻す
SJ_UngetChunk(sj, SJ_LIN_DATA, &ck2);



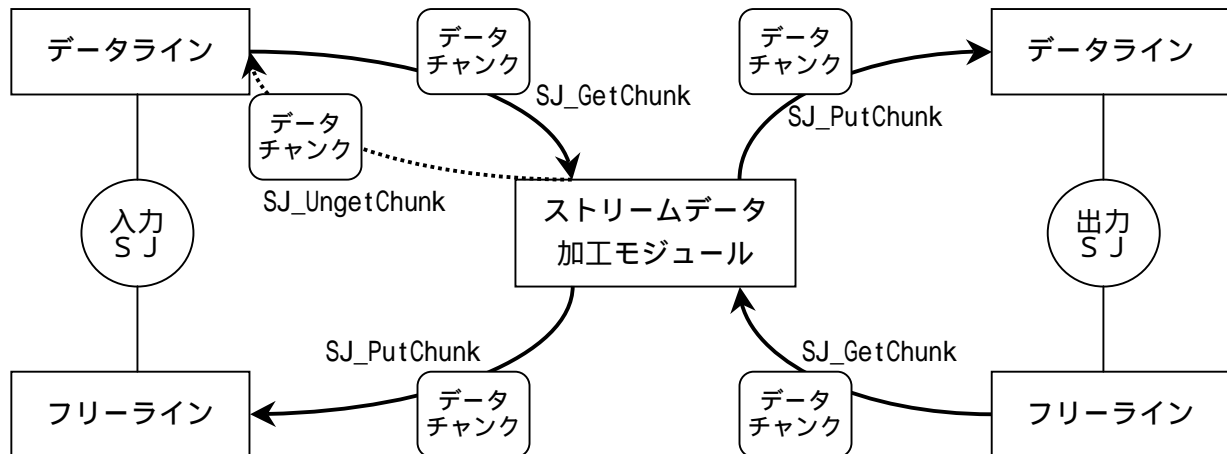
< プログラム例 >

```
SJ sj;      /* ストリームジョイント */
SJCK ck;    /* データチャンク */

SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck); /* 有効なデータの取得 */
nused = user_consume_func(ck.data, ck.len); /* データの消費 */
SJ_SplitChunk(&ck, nused, &ck, &ck2); /* データチャンクの分解 */
SJ_PutChunk(sj, SJ_LIN_FREE, &ck); /* フリーラインへの挿入 */
SJ_UngetChunk(sj, SJ_LIN_DATA, &ck2); /* データラインへの返却 */
```

2.4 データ加工モジュール

データ加工モジュールは、データ消費モジュールとデータ供給モジュールを組み合わせたものになります。



< プログラム例 >

```

SJ  sji;          /* 入力ストリームジョイント */
SJ  sjo;          /* 出力ストリームジョイント */
SJCK cki, cki2;   /* 入力 SJ 用データチャンク */
SJCK cko, cko2;   /* 出力 SJ 用データチャンク */
long nused;       /* 消費されたデータ量 */
long ngen;        /* 供給されたデータ量 */

SJ_GetChunk(sji, SJ_LIN_DATA, nbyte1, &cki); /* 有効なデータの取得 */
SJ_GetChunk(sjo, SJ_LIN_FREE, nbyte2, &cko); /* 空きデータ領域の取得 */

user_proc_func(&cki, &cko, &nused, &ngen); /* データの加工 */

SJ_SplitChunk(&cki, nused, &cki, &cki2); /* 入力データチャンクの分解 */
SJ_PutChunk(sji, SJ_LIN_FREE, &cki); /* フリーラインへの挿入 */
SJ_UngetChunk(sji, SJ_LIN_DATA, &cki2); /* データラインへの返却 */

SJ_SplitChunk(&cko, ngen, &cko, &cko2); /* 出力データチャンクの分解 */
SJ_PutChunk(sjo, SJ_LIN_DATA, &cko); /* データラインへの挿入 */
SJ_UngetChunk(sjo, SJ_LIN_FREE, &cko2); /* フリーラインへの返却 */

```


3. リングバッファ型ストリームジョイント

リングバッファ型ストリームジョイントは、ストリームジョイントのAPIによりリングバッファにアクセスできます。

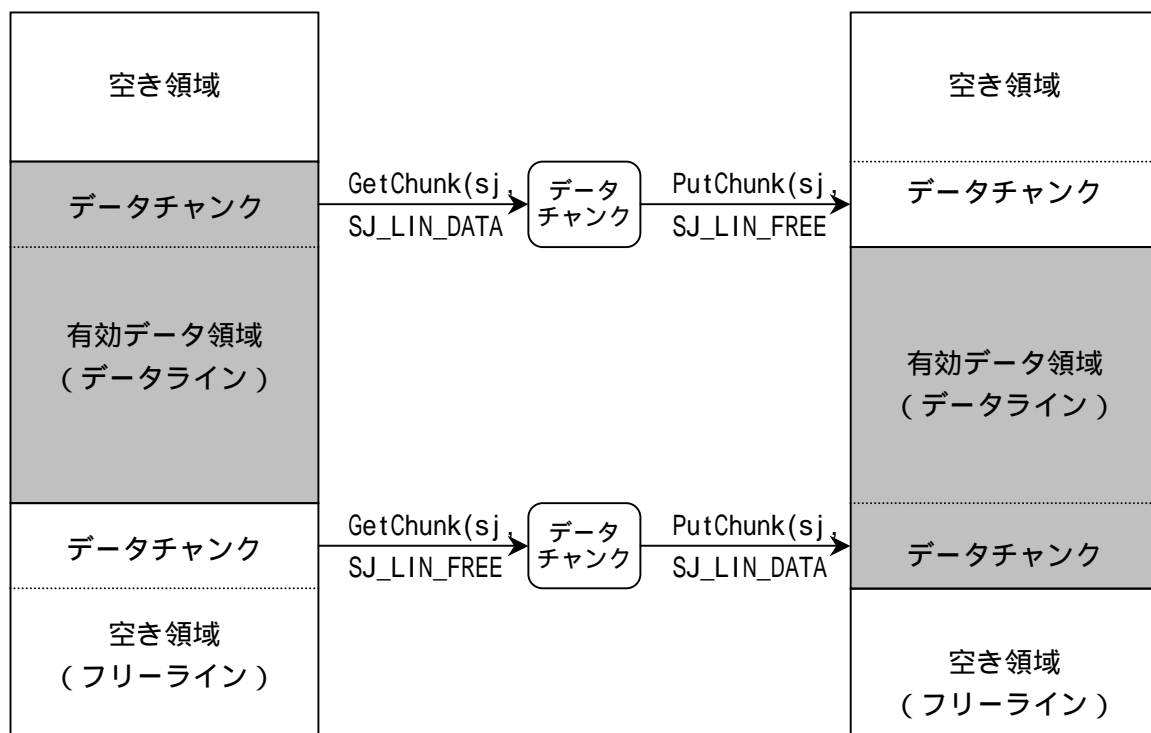


図 リングバッファ型ストリームジョイント

リングバッファ型ストリームジョイントを作成するときに、第3引数にエクストラ領域を指定できます。エクストラ領域は、常にリングバッファの先頭からのデータがコピーされています。従って、エクストラ領域サイズ分、データが連続していることが保証されます。例えば、ブロック単位でデータを処理しなければならない場合、ブロックサイズをエクストラ領域サイズとして指定します。エクストラ領域により、リングバッファであるにもかかわらず、バッファの折り返しを気にすることなく処理することができます。

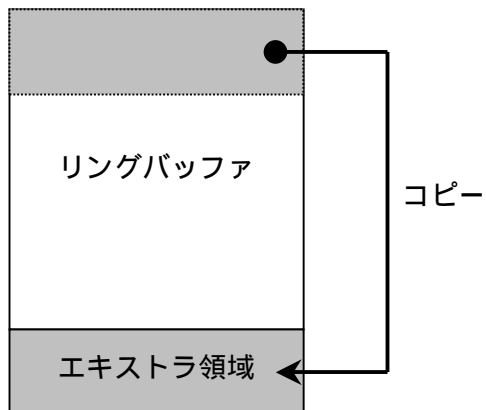


図 エクストラ領域

以下にサンプルプログラムを示します。

< サンプルプログラム >

```
void main(long ac, char *av[])
{
    SJ          sj;
    SJCK        ck, ck1, ck2;
    static char buf[1024+32];
    FILE        *fp, *fp2;
    long        nbyte, rsize;

    if ( (fp=fopen("test2a.bin", "rb")) == NULL ) {
        exit(1);
    }
    if ( (fp2=fopen("test2b.bin", "wb")) == NULL ) {
        exit(1);
    }

    SJRBF_Init();                                /* 初期化          */
    sj = SJRBF_Create(buf, 1024, 32);            /* ハンドルの生成  */
    for (;;) {
        if ( SJ_GetNumData(sj, SJ_LIN_DATA) == 0 && feof(fp) )
            break;
        /* 読み込み処理 */
        nbyte = rand() % 1025;
        SJ_GetChunk(sj, SJ_LIN_FREE, nbyte, &ck);
        rsize = fread(ck.data, sizeof(char), ck.len, fp);
        SJ_SplitChunk(&ck, rsize, &ck1, &ck2);
        SJ_PutChunk(sj, SJ_LIN_DATA, &ck1);
        SJ_UngetChunk(sj, SJ_LINE_FREE, &ck2);
        /* 書き出し処理 */
        nbyte = rand() % 1025;
        SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck);
        fwrite(ck.data, ck.len, sizeof(char), fp2);
        SJ_PutChunk(sj, SJ_LIN_FREE, &ck);
    }
    SJ_Destroy(sj);
    fclose(fp);
    fclose(fp2);
    getchar();
}
```

4. メモリ型ストリームジョイント

メモリ型ストリームジョイントは、ストリームジョイントのAPIによりメモリ上のデータにアクセスできます。メモリ上にあらかじめ読み込まれた音声や映像データを再生するときに使用します。基本的にはリングバッファ型ストリームジョイントと同じですが、下記の点は異なります。

- (1) 初期状態でデータラインにデータチャンクが存在する。
SJ_Reset 関数でデータラインにすべてのデータが存在することになる。
- (2) エクストラ領域が存在しない。
- (3) 通常、フリーラインからのデータチャンクの取得はしない。

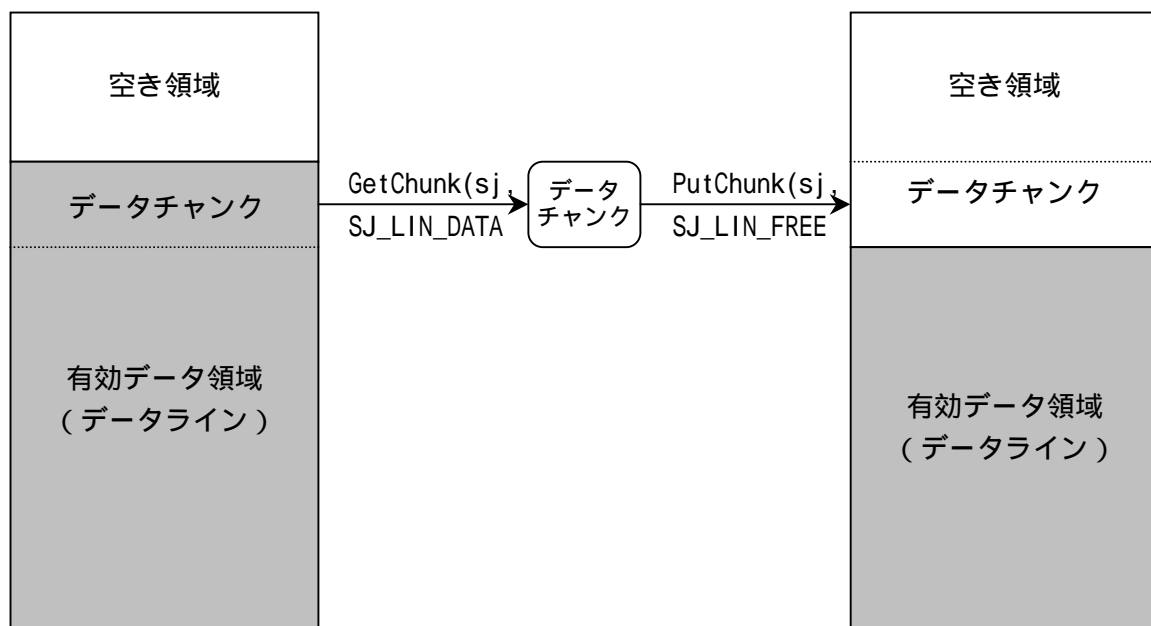


図 メモリ型ストリームジョイント

以下にサンプルプログラムを示します。

< サンプルプログラム >

```
void main(long ac, char *av[])
{
    SJ          sj;
    SJCK        ck, ck1, ck2;
    static char data[1024*1024];    /* 1 Mバイトのデータ領域 */
    FILE        *fp, *fp2;
    long        nbyte, rsize, dtsize;

    /* データの読み込み */
    fp = fopen("test2a.bin", "rb");
    dtsize = fread(data, sizeof(char), sizeof(data), fp);
    fclose(fp);

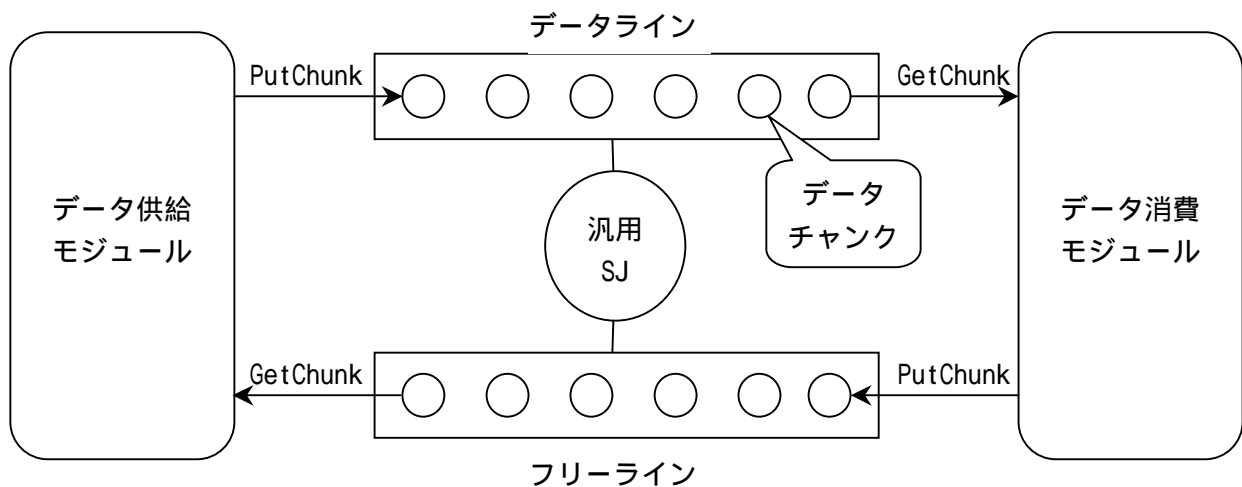
    /* データ書き出し用のファイルを開く */
    fp2 = fopen("test2b.bin", "wb");

    SJMEM_Init();                                /* 初期化 */
    sj = SJMEM_Create(data, dtsize);             /* ハンドルの生成 */
    for (;;) {
        /* 書き出し処理 */
        nbyte = (rand() % 1025) + 1;
        SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck);
        if ( ck.len == 0 )
            break;
        fwrite(ck.data, ck.len, sizeof(char), fp2);
        SJ_PutChunk(sj, SJ_LIN_FREE, &ck);
    }
    SJ_Destroy(sj);
    fclose(fp2);
}
```

5. 汎用型ストリームジョイント

汎用型ストリームジョイントは、リングバッファ型やメモリ型のストリームジョイントとはことなり、ストリームジョイント自体では記憶領域を持ちません。データ供給モジュールは汎用ストリームジョイントにデータチャンクを与え、データ消費モジュールはその供給されたデータチャンクを直接受け取ります。不要なデータコピーをせずに、効率良くデータをストリーミングできます。

汎用型ストリームジョイントは、ベルトコンベヤのようなものです。データ供給モジュールは、バケツにデータを詰めて、データラインのベルトコンベヤに乗せます。バケツはデータチャンクを意味します。データ消費モジュールは、ベルトコンベアに乗せられた順にデータを受け取ります。受け取ったデータを消費すると、そのバケツ（データチャンク）をフリーラインのベルトコンベアに乗せます。データ供給モジュールは、フリーラインから空のバケツを受け取り、データを詰めて再度データラインに乗せます。



また、GetChunk によって取り出したバケツは、UngetChunk 関数によって使用しなかった分を戻すことができます。

さらに、汎用型ストリームジョイントは、ライン上データチャンクを結合する機能を持ちます。この機能は、データチャンクをライン上に乗せられたときに、直前に乗せられたデータチャンクと領域が連続していれば、新たに乗せたデータチャンクを直前のデータチャンクに結合する機能です。つまり、新しいバケツをコンベアに乗せた時に、そのバケツの内容と直前のバケツの内容を一緒にして新しいバケツをコンベアに乗せることになります。

この機能は、音声データを再生するためのリングバッファなどに利用できます。供給する側と消費する側のデータサイズが異なる場合、供給されたデータを次々に結合しておき、消費する側は結合されたデータを少しずつ取り出すことになります。

結合するか否かの設定は、ストリームジョイントを作成するときに指定します。

5.1 汎用型ストリームジョイント作成方法

汎用型ストリームジョイントを作成する場合、そのストリームジョイントが扱えるデータチャンクの最大数に応じた作業領域を与える必要があります。例えば、100個のデータチャンクを扱う場合は、下記のようになります。

```
#define MAX_NCHUNK (100)
SJ sj;
static char sj_work[SJUNI_CALC_WORK(MAX_NCHUNK)];

sj = SJUNI_Create(SJUNI_MODE_SEPA, sj_work, sizeof(sj_work));
```

また、リングバッファのようにデータを結合する場合、下記のように指定します。リングバッファは、バッファ領域が最大3つに分解されるため、最大データチャンク数は3となります。

```
#define MAX_NCHUNK (3)
SJ sj;
static char sj_work[SJUNI_CALC_WORK(MAX_NCHUNK)];

sj = SJUNI_Create(SJUNI_MODE_JOIN, sj_work, sizeof(sj_work));
```

< サンプルプログラム (汎用型によるリングバッファ) >

```
void main(long ac, char *av[])
{
    SJ          sj;
    SJCK        ck, ck1, ck2;
    static char buf[1024];
    static char sjwork[SJUNI_CALC_WORK(3)];
    FILE        *fp, *fp2;
    long        nbyte, rsize;

    if ( (fp=fopen("test2a.bin", "rb")) == NULL ) {
        exit(1);
    }
    if ( (fp2=fopen("test2b.bin", "wb")) == NULL ) {
        exit(1);
    }

    SJUNI_Init(); /* 初期化 */
    sj = SJUNI_Create(SJUNI_MODE_JOIN, sjwork, sizeof(sjwork)); /* ハンドルの生成 */
    ck.data = buf, ck.len = sizeof(buf);
    SJ_PutChunk(sj, SJ_LIN_FREE, &ck); /* バッファ領域を供給 */
    for (;;) {
        if ( SJ_GetNumData(sj, SJ_LIN_DATA) == 0 && feof(fp) )
            break;
        /* 読み込み処理 */
        nbyte = rand() % 1025;
        SJ_GetChunk(sj, SJ_LIN_FREE, nbyte, &ck);
        rsize = fread(ck.data, sizeof(char), ck.len, fp);
        SJ_SplitChunk(&ck, rsize, &ck1, &ck2);
        SJ_PutChunk(sj, SJ_LIN_DATA, &ck1);
        SJ_UngetChunk(sj, SJ_LINE_FREE, &ck2);
        /* 書き出し処理 */
        nbyte = rand() % 1025;
        SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck);
        fwrite(ck.data, ck.len, sizeof(char), fp2);
        SJ_PutChunk(sj, SJ_LIN_FREE, &ck);
    }
    SJ_Destroy(sj);
    fclose(fp);
    fclose(fp2);
    getchar();
}
```

< サンプルプログラム（汎用型によるメモリ型ストリームジョイント） >

```
void main(long ac, char *av[])
{
    SJ          sj;
    SJCK        ck, ck1, ck2;
    static char sjwork[SJUNI_CALC_WORK(3)];
    static char data[1024*1024];    /* 1 Mバイトのデータ領域 */
    FILE        *fp, *fp2;
    long        nbyte, rsize, dtsize;

    /* データの読み込み */
    fp = fopen("test2a.bin", "rb");
    dtsize = fread(data, sizeof(char), sizeof(data), fp);
    fclose(fp);

    /* データ書き出し用のファイルを開く */
    fp2 = fopen("test2b.bin", "wb");

    SJUNI_Init();                                /* 初期化 */
    sj = SJUNI_Create(SJUNI_MODE_JOIN, sjwork, sizeof(sjwork)); /* ハンドルの生成 */
    ck.data = data, ck.len = dtsize;
    SJ_PutChunk(sj, SJ_LIN_DATA, &ck);          /* データ領域を供給 */
    for (;;) {
        /* 書き出し処理 */
        nbyte = (rand() % 1025) + 1;
        SJ_GetChunk(sj, SJ_LIN_DATA, nbyte, &ck);
        if ( ck.len == 0 )
            break;
        fwrite(ck.data, ck.len, sizeof(char), fp2);
        SJ_PutChunk(sj, SJ_LIN_FREE, &ck);
    }
    SJ_Destroy(sj);
    fclose(fp2);
}
```


6. データ仕様

ライブラリのデータ一覧を以下に示す。

表 6 - 1 データ一覧

データ名		機 能	番号
定 数			
	SJ_LIN_~	データライン	1.1
	SJ_ERR_~	エラーコード	1.2
	SJ_ERR_~	エラーコード	1.3
データ型			
	UUID	ライブラリ識別子	2.1
	SJ	ストリームジョイントハンドル	2.2
	SJCK	データチャンク	2.3

6.1 定 数

Title	Data Name	Data	No
データ	SJ_LIN_~	データライン	1.1

チャンクを供給するラインを意味します。

定数名	説 明
SJ_LIN_DATA	データライン 有効なデータを取り出すための識別子
SJ_LIN_FREE	フリーライン 使用済みデータを取り出すための識別子

Title	Data Name	Data	No
データ	SJ_ERR_~	エラーコード	1.2

MWD_SJ_ERR_OK の値は 0 で、その他のエラーコードは負の値となります。

定数名	説 明
SJ_ERR_OK	正常終了
SJ_ERR_FATAL	致命的なエラー（通常起きない）
SJ_ERR_INTERNAL	内部エラー（通常起きない）
SJ_ERR_PRM	パラメータエラー

Title	Data Name	Data	No
データ	SJUNI_MODE_~	結合モード	1.3

汎用型ストリームジョイントを生成する時のパラメータです。SJ_PutChunk 関数でデータチャンクを供給したときに結合するか否かを決めます。

定数名	説 明
SJUNI_MODE_SEPA	データチャンクを結合しない
SJUNI_MODE_JOIN	データチャンクを結合する

6.2 データ型

Title	Data Name	Data	No
データ	UUID	ライブラリ識別子	2.1

空間的にも時間的にも一意に定義されるライブラリ識別子。Visual C++に付属の UUIDGEN.EXE などを実行するとコードが得られるので、これを以下のメンバーに従って各ライブラリ内で定義しています。

メンバー	型 名	説 明
Data1	long	ID1
Data2	short	ID2
Data3	short	ID3
Data4[8]	char	ID4

```
C:¥>uuidgen
```

```
211b2800-a6a9-11d1-8f3f-0060089448bc
```

```
#define SmD_LID { ¥
```

```
0x211b2800, 0xa6a9, 0x11d1, 0x8f, 0x3f, 0x00, 0x60, 0x08, 0x94, 0x48, 0xbc¥
```

```
}¥
```

Title	Data Name	Data	No
データ	SJ	ストリームジョイントハンドル	2.2

ストリームジョイントを制御するためのハンドルです。

Title	Data Name	Data	No
データ	SJCK	データチャンク	2.3

メンバー	型 名	説 明
data	char *	開始アドレス
len	long	バイト数

7. 関数仕様

ライブラリの関数一覧を以下に示す。

表 7-1 関数一覧

関数名	機 能	番号
共通関数		
SJ_Destroy	ストリームジョイントの消去	1.1
SJ_Reset	ストリームジョイントのリセット	1.2
SJ_GetChunk	データチャンクの取得	1.3
SJ_UngetChunk	データチャンクを戻す	1.4
SJ_PutChunk	データチャンクを渡す	1.5
SJ_GetNimData	読み書き可能なデータ容量の取得	1.6
SJ_GetUuid	ストリームジョイント UUID の取得	1.7
SJ_EntryErrFunc	エラー処理関数の登録	1.8
SJ_SplitChunk	データチャンクの分解	1.9
リングバッファ		
SJRBF_Init	初期化	2.1
SJRBF_Create	ストリームジョイントの生成	2.2
メモリ		
SJMEM_Init	初期化	3.1
SJMEM_Create	ストリームジョイントの生成	3.2
汎用		
SJUNI_Init	初期化	4.1
SJUNI_Create	ストリームジョイントの生成	4.2

7.1 共通関数

ストリームジョイントのタイプによらない共通の関数です。

Title	Function Name	Function	No
関 数	SJ_Destroy	ストリームジョイントの消去	1.1

[書 式] void SJ_Destroy(SJ sj);

[入 力] sj : ストリームジョイント

[出 力] なし

[関数値] なし

[機 能] ストリームジョイントを消去する。

Title	Function Name	Function	No
関 数	SJ_Reset	ストリームジョイントのリセット	1.2

[書 式] void SJ_Reset(SJ sj);

[入 力] sj : ストリームジョイント

[出 力] なし

[関数値] なし

[機 能] ストリームジョイントをリセットする。

Title 関 数	Function Name SJ_GetChunk	Function データチャンクの取得	No 1.3
--------------	------------------------------	------------------------	-----------

[書 式] void SJ_GetChunk(SJ sj, long id, long nbyte, SJCK *ck);

[入 力] sj : ストリームジョイント

id : チャンク入出力 ID

nbyte : 要求データ量(単位: バイト)

[出 力] ck : データチャンク(データの先頭アドレス/サイズ)

[関数値] なし

[機 能] バッファに対して書き込みや読み出しを行う時に、ストリームジョイントから読み書き可能な領域(データチャンク)を取得する。

Title 関 数	Function Name SJ_UngetChunk	Function データチャンクを戻す	No 1.4
--------------	--------------------------------	------------------------	-----------

[書 式] void SJ_UngetChunk(SJ sj, long id, SJCK *ck);

[入 力] sj : ストリームジョイント

id : チャンク入出力 ID

ck : データチャンク(データの先頭アドレス/サイズ)

[出 力] なし

[関数値] なし

[機 能] 取得したデータチャンクに対して、書き込みや読み出しを実際に行えなかった時に、読み書きできなかった領域(データチャンク)をストリームジョイントに戻す。

Title 関 数	Function Name SJ_PutChunk	Function データチャンクを渡す	No 1.5
--------------	------------------------------	------------------------	-----------

[書 式] void SJ_PutChunk(SJ sj, long id, SJCK *ck);

[入 力] sj : ストリームジョイント

id : チャンク入出力 ID

ck : データチャンク(データの先頭アドレス/サイズ)

[出 力] なし

[関数値] なし

[機 能] 取得したデータチャンクに対して、書き込みや読み出しを実際に行った時に、読み書きできた領域(データチャンク)をストリームジョイントに渡す。

Title 関 数	Function Name SJ_GetNumData	Function 読み書き可能なデータ容量の取得	No 1.6
--------------	--------------------------------	-----------------------------	-----------

[書 式] long SJ_GetNumData(SJ sj, long id);

[入 力] sj : ストリームジョイント

id : チャンク入出力 ID

[出 力] なし

[関数値] 読み書き可能なデータ容量(単位: バイト)

・書き込む場合: 空き容量

・読み出す場合: データ容量

[機 能] ストリームジョイントにデータを書き込む場合は、バッファの空き容量を取得し、読み出す場合は、バッファのデータ容量を取得する。

Title 関 数	Function Name SJ_GetUuid	Function ストリームジョイント UUID の取得	No 1.7
--------------	-----------------------------	---------------------------------	-----------

[書 式] UUID *SJ_GetUuid(SJ sj);
[入 力] sj :ストリームジョイント
[出 力] なし
[関数値] ストリームジョイントの UUID
[機 能] ストリームジョイント UUID を取得する。

Title 関 数	Function Name SJ_EntryErrFunc	Function エラー処理関数の登録	No 1.8
--------------	----------------------------------	------------------------	-----------

[書 式] void SJ_EntryErrFunc(SJ sj, long id,
void (*func)(void *obj, long ecode), void *obj);
[入 力] sj :ストリームジョイント
id :チャンク入出力 ID
func :エラー発生時に呼び出される関数
obj :func 関数の第 1 引数
[出 力] なし
[関数値] なし
[機 能] エラーが発生した時に呼び出される関数を登録する。
func に NULL を設定すると、エラー処理関数が未登録になる。

Title 関 数	Function Name SJ_SplitChunk	Function データチャンクの分解	No 1.9
--------------	--------------------------------	------------------------	-----------

[書 式] void SJ_SplitChunk(SJCK *ck, long nbyte,
SJCK *ck1, SJCK *ck2);

[入 力] ck : 分解対象となるデータチャンク
nbyte : ck1 のサイズ(単位: バイト)

[出 力] ck1 : 分解された前半のデータチャンク
ck2 : 分解された後半のデータチャンク

[関数値] データチャンク ck2 のサイズ(単位: バイト)

[機 能] データチャンク ck を 2 つのデータチャンク ck1 と ck2 に分解する。
ck1 は、nbyte の長さになる。
ck1 のサイズが nbyte に満たない場合は分解されず、ck1 に ck が代入され、
ck2 のサイズは 0 となる。
ck1 と ck は、同じものを指定しても良い。
単に分解する場合は、以下のように記述する。

[例]

SJ_SplitChunk(&ck, 100, &ck, &ck2);

7.2 リングバッファ型ストリームジョイント

Title	Function Name	Function	No
関 数	SJRBFin	初期化	2.1

[書 式] void SJRBFin(void);

[入 力] なし

[出 力] なし

[関数値] なし

[機 能] リングバッファタイプのストリームジョイントを初期化する。

Title	Function Name	Function	No
関 数	SJRBFCreat	ストリームジョイントの生成	2.2

[書 式] SJ SJRBFCreat(char *buf, long bsize, long xsize);

[入 力] buf : リングバッファ

bsize : リングバッファのサイズ(単位: バイト)

xsize : エキストラバッファサイズ(単位: バイト)

[出 力] なし

[関数値] ストリームジョイント

[機 能] リングバッファタイプのストリームジョイントを生成する。

7.3 メモリ型ストリームジョイント

Title	Function Name	Function	No
関 数	SJMEM_Init	初期化	3.1

[書 式] void SJMEM_Init(void);

[入 力] なし

[出 力] なし

[関数値] なし

[機 能] メモリタイプのストリームジョイントを初期化する。

Title	Function Name	Function	No
関 数	SJMEM_Create	ストリームジョイントの生成	3.2

[書 式] SJ SJMEM_Create(char *data, long bsize);

[入 力] data : データ領域

bsize : データサイズ(単位: バイト)

[出 力] なし

[関数値] ストリームジョイント

[機 能] メモリタイプのストリームジョイントを生成する。

7.4 汎用型ストリームジョイント

Title 関 数	Function Name SJUNI_Init	Function 初期化	No 4.1
--------------	-----------------------------	-----------------	-----------

[書 式] void SJUNI_Init(void);
[入 力] なし
[出 力] なし
[関数値] なし
[機 能] 汎用タイプのストリームジョイントを初期化する。

Title 関 数	Function Name SJUNI_Create	Function ストリームジョイントの生成	No 4.2
--------------	-------------------------------	---------------------------	-----------

[書 式] SJ SJUNI_Create(long mode, char *work, long wksize);
[入 力] mode : 結合モード (SJUNI_MODE_SEPA, SJUNI_MODE_JOIN)
work : 作業領域
wksize : 作業領域のサイズ
[出 力] なし
[関数値] ストリームジョイント
[機 能] 汎用タイプのストリームジョイントを生成する。
データチャンクを分離したまま扱う場合は、SJUNI_MODE_SEPA を指定し、
データチャンクを結合する場合は、SJUNI_MODE_JOIN を指定する。
ストリームジョイントの扱う最大データチャンク数より作業領域を確保し、work と wksize
に引数として渡す。

[用 例]
#define MAX_NCK (256)

char work[SJUNI_CALC_WORK(MAX_NCK)];

sj = SJUNI_Create(SJUNI_MODE_SEPA, work, sizeof(work));