EPSON
EXCEED YOUR VISION

FSA ライブラリ
マニュアル

セイコーエプソン株式会社

Rev.1.0

評価ボード・キット、開発ツールご使用上の注意事項

1. 本評価ボード・キット、開発ツールは、お客様での技術的評価、動作の確認および開発のみに用いられることを想定し設計されています。それらの技術評価・開発等の目的以外には使用しないで下さい。本品は、完成品に対する設計品質に適合していません。

2. 本評価ボード・キット、開発ツールは、電子エンジニア向けであり、消費者向け製品ではありません。お客様において、適切な使用と安全に配慮願います。弊社は、本品を用いることで発生する損害や火災に対し、いかなる責も負いかねます。通常の使用においても、異常がある場合は使用を中止して下さい。

3. 本評価ボード・キット、開発ツールに用いられる部品は、予告無く変更されることがあります。

本資料のご使用につきましては、次の点にご留意願います。

本資料の内容については、予告無く変更することがあります。

1. 本資料の一部、または全部を弊社に無断で転載、または、複製など他の目的に使用することは堅くお断りいたします。

2. 本資料に掲載される応用回路、プログラム、使用方法等はあくまでも参考情報であり、これらに起因する第三者の知的財産権およびその他の権利侵害あるいは損害の発生に対し、弊社はいかなる保証を行うものではありません。また、本資料によって第三者または弊社の知的財産権およびその他の権利の実施権の許諾を行うものではありません。

3. 特性値の数値の大小は、数直線上の大小関係で表しています。

4. 製品および弊社が提供する技術を輸出等するにあたっては「外国為替および外国貿易法」を遵守し、当該法令の定める手続きが必要です。大量破壊兵器の開発等およびその他の軍事用途に使用する目的をもって製品および弊社が提供する技術を費消、再販売または輸出等しないでください。

5. 本資料に掲載されている製品は、生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本（当該）製品をこれらの用途に用いた場合のいかなる責任についても負いかねます。

6. 本資料に掲載されている会社名、商品名は、各社の商標または登録商標です。

©SEIKO EPSON CORPORATION 2015, All rights reserved.

目　次

1. 複素 FFT/実数 FFT/DCT-4/スペクトル解析 1
1.1 概要 1
1.2 ファイル構成 2
1.3 コンフィグレーション 3
1.4 API 関数 7
1.5 ビヘイビアモデルで使用する API 関数 12
1.6 注意事項（重要） 13
1.7 パフォーマンス 14
1.8 要求メモリ 15
2. FIR フィルタ 16
2.1 概要 16
2.2 ファイル構成 17
2.3 コンフィグレーション 17
2.4 フィルタ係数定義 18
2.5 API 関数 20
2.6 ビヘイビアモデルで使用する API 関数 23
2.7 パフォーマンス 24
2.8 要求メモリ 24
3. IIR フィルタ 25
3.1 概要 25
3.2 ファイル構成 25
3.3 コンフィグレーション 26
3.4 フィルタ係数定義 27
3.5 API 関数 29
3.6 ビヘイビアモデルで使用する API 関数 31
3.7 パフォーマンス 32
3.8 要求メモリ 32
4. ソート 33
4.1 概要 33
4.2 ファイル構成 33
4.3 ファイル構成 34
4.4 API 関数 35
4.5 ビヘイビアモデルで使用する API 関数 43
4.6 パフォーマンス 44
4.7 要求メモリ 44
5. 三角関数 45
5.1 概要 45
5.2 ファイル構成 45
5.3 コンフィグレーション 46
5.4 API 関数 47

5.5 ビヘイビアモデルで使用する API 関数 50
5.6 パフォーマンス 51
5.7 要求メモリ 51
6. 三角関数 2 52
6.1 概要 52
6.2 ファイル構成 52
6.3 コンフィグレーション 53
6.4 API 関数 54
6.5 ビヘイビアモデルで使用する API 関数 56
6.6 パフォーマンスおよび要求メモリ 57
6.7 演算誤差 57
7. 相互相関係数 58
7.1 概要 58
7.2 ファイル構成 58
7.3 コンフィグレーション 59
7.4 API 関数 61
7.5 ビヘイビアモデルで使用する API 関数 62
7.6 パフォーマンスおよび要求メモリ 63
8. 対数 64
8.1 概要 64
8.2 ファイル構成 64
8.3 コンフィグレーション 65
8.4 API 関数 66
8.5 ビヘイビアモデルで使用する API 関数 67
8.6 パフォーマンス 68
8.7 要求メモリ 68
8.8 演算精度 68
9. 分散、標準偏差 69
9.1 概要 69
9.2 ファイル構成 69
9.3 コンフィグレーション 70
9.4 API 関数 71
9.5 ビヘイビアモデルで使用する API 関数 74
9.6 パフォーマンス 75
9.7 要求メモリ 75
10. 平方根、逆数平方根 76
10.1 概要 76
10.2 ファイル構成 76
10.3 コンフィグレーション 77
10.4 API 関数 79
10.5 ビヘイビアモデルで使用する API 関数 81
10.6 パフォーマンス 82
10.7 要求メモリ 82

改訂履歴表 83

1. 複素 FFT/実数 FFT/DCT-4/スペクトル解析

1.1 概要

FSA ライブラリー複素 FFT・実数 FFT・DCT-4・スペクトル解析は、以下に示す複素／実数 FFT、DCT-4 およびスペクトル解析を高速に演算する関数群を提供します。FsaCFFT、FsaRFFT は、各々１つのオブジェクト・コードで複数ポイントの複素／実数 FFT に対応した関数です。複数ポイントの FFT が必要な場合は、これら関数が便利です。一方、FsaXXXXpCFFT、FsaXXXXpRFFT は、各々指定されたポイント数の FFT に専用化されたオブジェクト・コードを採用します。汎用関数より若干ですがより高速です。

FsaPowSpectAnalyzeXXXX は、窓掛けから実数 FFT、パワースペクトル算出までを１つにまとめたスペクトル解析用の関数です。

表 1.1　サポート関数一覧

関数	概仕様
FsaCFFT	32～1024 ポイントまでの指定された複素 FFT を処理する関数
FsaRFFT	64～2048 ポイントまでの指定された実数 FFT を処理する関数
FsaDCT4	64～2048 ポイントまでの指定された DCT-4 を処理する関数
Fsa16pCFFT	16 ポイントの複素 FFT を行う関数
Fsa32pCFFT	32 ポイントの複素 FFT を行う関数
Fsa64pCFFT	64 ポイントの複素 FFT を行う関数
Fsa128pCFFT	128 ポイントの複素 FFT を行う関数
Fsa256pCFFT	256 ポイントの複素 FFT を行う関数
Fsa512pCFFT	512 ポイントの複素 FFT を行う関数
Fsa1024pCFFT	1024 ポイントの複素 FFT を行う関数
Fsa32pRFFT	32 ポイントの実数 FFT を行う関数
Fsa64pRFFT	64 ポイントの実数 FFT を行う関数
Fsa128pRFFT	128 ポイントの実数 FFT を行う関数
Fsa256pRFFT	256 ポイントの実数 FFT を行う関数
Fsa512pRFFT	512 ポイントの実数 FFT を行う関数
Fsa1024pRFFT	1024 ポイントの実数 FFT を行う関数
Fsa2048pRFFT	2048 ポイントの実数 FFT を行う関数
FsaPowSpectAnalyze32	32 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze64	64 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze128	128 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze256	256 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze512	512 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze1024	1024 ポイントのフレームでスペクトル解析を行う関数
FsaPowSpectAnalyze2048	2048 ポイントのフレームでスペクトル解析を行う関数

1.2 ファイル構成

表 1.2 ソースファイル一覧

ソースファイル	記述内容
fsafft.h	ライブラリ関数宣言
fsafftd.h	コンフィグレーションのためのパラメータ定義
fsafft.c	FFT ライブラリ関数定義
fsadct4.c	DCT-4 ライブラリ関数定義
fsafftk.h	複素 FFT カーネル部分および実数 FFT のポスト処理の FSA プログラム
fsaffts.h	ビットスクランブル部分の FSA プログラム
fsafftc.c	SIN/COS テーブル等の定数配列定義
fsapsana.c	スペクトル解析ライブラリ関数定義
hamming.c	サンプル用のハミング窓関数の定数配列定義

1.3 コンフィグレーション

fsafftd.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

必要な関数のみを残して、コメントアウトします。_CFFT_GENERIC_XXXX、_RFFT_GENERIC_XXXX および _DCT4_GENERIC_XXXX では、各々FsaCFFT、FsaRFFT でサポートする FFT のポイント数および FsaDCT4 でサポートする DCT-4 のポイント数を指定できます。すべてのポイント数定義をコメントアウトするとその関数は実装されません。(未使用関数のコメントアウトを行わないと、コードサイズ、定数テーブルサイズが冗長になる場合があります。)

表 1.3 #define 定義と実装関数

<table>
<tr><th>定義</th><th>実装関数</th><th>定義</th><th>実装関数</th></tr>
<tr><td>_CFFT_16</td><td>Fsa16pCFFT</td><td>_RFFT_32</td><td>Fsa32pRFFT</td></tr>
<tr><td>_CFFT_32</td><td>Fsa32pCFFT</td><td>_RFFT_64</td><td>Fsa64pRFFT</td></tr>
<tr><td>_CFFT_64</td><td>Fsa64pCFFT</td><td>_RFFT_128</td><td>Fsa128pRFFT</td></tr>
<tr><td>_CFFT_128</td><td>Fsa128pCFFT</td><td>_RFFT_256</td><td>Fsa256pRFFT</td></tr>
<tr><td>_CFFT_256</td><td>Fsa256pCFFT</td><td>_RFFT_512</td><td>Fsa512pRFFT</td></tr>
<tr><td>_CFFT_512</td><td>Fsa512pCFFT</td><td>_RFFT_1024</td><td>Fsa1024pRFFT</td></tr>
<tr><td>_CFFT_1024</td><td>Fsa1024pCFFT</td><td>_RFFT_2048</td><td>Fsa2048pRFFT</td></tr>
<tr><td>_CFFT_GENERIC_32</td><td rowspan="6">有効にしたポイント数の複素 FFT が FsaCFFT でサポートされる。</td><td>_RFFT_GENERIC_64</td><td rowspan="6">有効にしたポイント数の実数 FFT が FsaRFFT でサポートされる。</td></tr>
<tr><td>_CFFT_GENERIC_64</td><td>_RFFT_GENERIC_128</td></tr>
<tr><td>_CFFT_GENERIC_128</td><td>_RFFT_GENERIC_256</td></tr>
<tr><td>_CFFT_GENERIC_256</td><td>_RFFT_GENERIC_512</td></tr>
<tr><td>_CFFT_GENERIC_512</td><td>_RFFT_GENERIC_1024</td></tr>
<tr><td>_CFFT_GENERIC_1024</td><td>_RFFT_GENERIC_2048</td></tr>
<tr><td>_DCT4_GENERIC_64</td><td rowspan="6">有効にしたポイント数の DCT-4 が FsaDCT4 でサポートされる。</td><td rowspan="6" colspan="2"></td></tr>
<tr><td>_DCT4_GENERIC_128</td></tr>
<tr><td>_DCT4_GENERIC_256</td></tr>
<tr><td>_DCT4_GENERIC_512</td></tr>
<tr><td>_DCT4_GENERIC_1024</td></tr>
<tr><td>_DCT4_GENERIC_2048</td></tr>
</table>

表 1.4 #define 定義と実装関数

定義	実装関数
_PSANA_32	FsaPowSpectAnalyze32
_PSANA_64	FsaPowSpectAnalyze64
_PSANA_128	FsaPowSpectAnalyze128
_PSANA_256	FsaPowSpectAnalyze256
_PSANA_512	FsaPowSpectAnalyze512
_PSANA_1024	FsaPowSpectAnalyze1024
_PSANA_2048	FsaPowSpectAnalyze2048

fsafftd.hの記述例 （FFT 関数部分）

```
//---------------------------------------------------------------------------------
// Specify complex FFT function to use.
// The function is disabled if it comments out.
// For instance, make _CFFT_1024 valid to use Fsa1024pCFFT().
//#define _CFFT_1024
//#define _CFFT_512
//#define _CFFT_256
//#define _CFFT_128
//#define _CFFT_64
//#define _CFFT_32
#define _CFFT_16
//---------------------------------------------------------------------------------

//---------------------------------------------------------------------------------
// Specify complex FFT point that should be supported by FsaCFFT().
// The function is disabled if it comments out.
// For instance, make _CFFT_GENERIC_1024 valid for FsaCFFT() to support
// 1024-point complex FFT.
#define _CFFT_GENERIC_1024
#define _CFFT_GENERIC_512
#define _CFFT_GENERIC_256
#define _CFFT_GENERIC_128
#define _CFFT_GENERIC_64
#define _CFFT_GENERIC_32
//---------------------------------------------------------------------------------

//---------------------------------------------------------------------------------
// Specify real FFT function to use.
// The function is disabled if it comments out.
// For instance, make _RFFT_2048 valid to use Fsa2048pRFFT().
//#define _RFFT_2048
//#define _RFFT_1024
//#define _RFFT_512
//#define _RFFT_256
//#define _RFFT_128
//#define _RFFT_64
#define _RFFT_32
//---------------------------------------------------------------------------------

//---------------------------------------------------------------------------------
// Specify real FFT point that can be supported by FsaRFFT().
// The function is disabled if it comments out.
// For instance, make _RFFT_GENERIC_2048 valid for FsaRFFT() to support
// 2048-point real FFT.
#define _RFFT_GENERIC_2048
#define _RFFT_GENERIC_1024
#define _RFFT_GENERIC_512
#define _RFFT_GENERIC_256
#define _RFFT_GENERIC_128
#define _RFFT_GENERIC_64
//---------------------------------------------------------------------------------
```

fsafftd.h の記述例（DCT-4 関数部分）

```
//---------------------------------------------------------------------------
// Specify DCT-4 point that can be supported by FsaDCT4().
// The function is disabled if it comments out.
// For instance, make _DCT4_GENERIC_2048 valid for FsaDCT4() to support
// 2048-point DCT-4.
#define _DCT4_GENERIC_2048
#define _DCT4_GENERIC_1024
#define _DCT4_GENERIC_512
#define _DCT4_GENERIC_256
#define _DCT4_GENERIC_128
#define _DCT4_GENERIC_64
//---------------------------------------------------------------------------
```

fsafftd.h の記述例（スペクトル解析関数部分）

```
//---------------------------------------------------------------------------
// Specify power spectrum analyzing function to use.
// The function is disabled if it comments out.
// For instance, make _PSANA_2048 valid to use FsaPowSpectAnalyze2048().
//#define _PSANA_2048
//#define _PSANA_1024
//#define _PSANA_512
#define _PSANA_256
#define _PSANA_128
#define _PSANA_64
#define _PSANA_32
//---------------------------------------------------------------------------
```

sin/cos 定数テーブルのデータ語長

FFT 演算の sin/cos 定数テーブルの語長を、16 ビットあるいは 32 ビットから選択できます。要求される演算精度、メモリサイズに応じて設定してください。定数テーブルは、指定した語長に応じてサイズが増減します。

fsafftd.h の記述例

```
// The bit length of sin/cos table can be set here.
// Notice: 16 or 32-bit is selectable in the length.
#define _FFT_COSIN_BITS                          16
//---------------------------------------------------------------------------
```

複素 FFT、実数 FFT、DCT-4 関数の入出力データ語長

上記の関数の入出力データの語長を、16 ビットあるいは 32 ビットから選択できます。ただ、基本は 32 を定義してください。

fsafftd.h の記述例

```
// The bit length of data can be set here
// Notice: 16 or 32-bit is selectable in the length. But it has to be 32 basically
#define _FFT_DATA_BITS                           32
//---------------------------------------------------------------------------
```

スペクトル解析関数の入力データ語長

スペクトル解析関数は、入力データの語長を 16 ビットあるいは 32 ビットから選択できます。要求される演算精度、メモリサイズに応じて設定してください。但し、16 を設定しても窓掛け以降の計算は 32 ビットで行われます。

fsafftd.h の記述例

```
// The input data bit length of FsaPowSpectAnalyzeXXXX can be set here.
// Notice: 16 or 32-bit is selectable in the length.
#define _PSANA_INDATA_BITS                       32
//---------------------------------------------------------------------------
```

1.4 API 関数

FsaCFFT

インクルード

```
#include "fsafft.h"
```

形式　`int FsaCFFT(FSAREG *pFsaReg, int n, void *pData)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*n*	複素 FFT のポイント数、32、64、128、256、512、1024 の何れかを指定
(in/out)*pData*	入出力データのポインタ（データは実数・虚数の点順次）、FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_FFT_DATA_BITS の定義で short 型か long 型に設定されます。また、実行する FFT のポイント数に応じたアドレスアライメントが必要です。（詳細は後述）

戻り値

成功すると 0 が返されます。

説明

pData で指定されたメモリ上のデータに対して、n で指定されるポイント数の複素 FFT を行います。入力データは変換後のデータに上書きされます。（in-place）なお、変換後のデータのスケールは、n 倍になります。

FsaRFFT

インクルード

```
#include "fsafft.h"
```

形式　`int FsaRFFT(FSAREG *pFsaReg, int n, void *pData)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*n*	実数 FFT のポイント数、64、128、256、512、1024、2048 の何れかを指定
(in/out)*pData*	入出力データのポインタ、FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_FFT_DATA_BITS の定義で short 型か long 型に設定されます。また、実行する FFT のポイント数に応じたアドレスアライメントが必要です。（詳細は後述）

戻り値

成功すると 0 が返されます。

説明

pData で指定されたメモリ上のデータに対して、n で指定されるポイント数の実数 FFT を行います。入力データは変換後のデータに上書きされます。（in-place）なお、変換後のデータのスケールは、n 倍になります。

FsaDCT4

インクルード

```
#include “fsafft.h”
```

形式

```
int FsaDCT4(FSAREG *pFsaReg, int n, void *pData)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*n*	DCT-4 のサイズ、64、128、256、512、1024、2048 の何れかを指定
(in/out)*pData*	入出力データのポインタ、FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_FFT_DATA_BITS の定義で short 型か long 型に設定されます。また、実行する DCT-4 のサイズに応じたアドレスアライメントが必要です。(詳細は後述)

戻り値

成功すると 0 が返されます。

説明

pData で指定されたメモリ上のデータに対して、n で指定されるサイズ(ポイント数)の DCT-4 を行います。入力データは変換後のデータに上書きされます。(in-place)

FsaXXXXpCFFT

インクルード

```
#include “fsafft.h”
```

形式

```
void Fsa16pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa32pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa64pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa128pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa256pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa512pCFFT(FSAREG *pFsaReg, void *pData)
void Fsa1024pCFFT(FSAREG *pFsaReg, void *pData)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in/out)*pData*	入出力データのポインタ（データは実数・虚数の点順次)、FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_FFT_DATA_BITS の定義で short 型か long 型に設定されます。また、実行する FFT のポイント数に応じたアドレスアライメントが必要です。(詳細は後述)

戻り値

なし

説明

pData で指定されたメモリ上のデータに対して複素 FFT を行います。入力データは変換後のデータに上書きされます。(in-place) なお、変換後のデータのスケールは、複素 FFT のポイント数を n としたとき、n 倍になります。

FsaXXXXpRFFT

インクルード

```
#include "fsafft.h"
```

形式

```
void Fsa32pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa64pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa128pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa256pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa512pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa1024pRFFT(FSAREG *pFsaReg, void *pData)
void Fsa2048pRFFT(FSAREG *pFsaReg, void *pData)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in/out)*pData*	入出力データのポインタ、FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_FFT_DATA_BITS の定義で short 型か long 型に設定されます。また、実行する FFT のポイント数に応じたアドレスアライメントが必要です。（詳細は後述）

戻り値

なし

説明

pData で指定されたメモリ上のデータに対して実数 FFT を行います。入力データは変換後のデータに上書きされます。（in-place）なお、変換後のデータのスケールは、実数 FFT のポイント数を n としたとき、n 倍になります。

FsaPowSpectAnalyzeXXXX

インクルード

```
#include “fsafft.h”
```

形式

```
void FsaPowSpectAnalyze32(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze64(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze128(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze256(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze512(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze1024(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
void FsaPowSpectAnalyze2048(
        FSAREG *pFsaReg, const short *pi16Win, void *pInData, long *pi32Out)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*pi16Win*	Q14 フォーマットの窓関数テーブル配列のポインタ。FSA がアクセス可能なメモリのポインタでなければばならない。
(in)*pInData*	入力データのポインタ。FSA がアクセス可能なメモリのポインタでなければばならない。データ型は、fsafftd.h の_PSANA_INDATA_BITS の定義で short 型か long 型に設定されます。
(out)*pi32Out*	パワースペクトルが格納されるポインタ。FSA がアクセス可能なメモリのポインタでなければばならない。フレームサイズは n のとき、n/2 ×4 バイトの領域が必要です。

戻り値

なし

説明

pInData で指定されたメモリ上のデータに対して、pi16Win で指定された窓関数処理を行い、所定のポイント数の実数 FFT により周波数スペクトルに分解し、そのパワースペクトルを pi32Out で指定されたメモリに格納します。pInData と pi32Out が同一アドレスであっても問題ありません。

窓関数は左右対称である必要があり、pi16Win で指定する窓関数テーブルは、左半分のデータのみで問題ありません。ハミング窓の場合は、hamming.c に記載された定数配列を使用できます。その他の窓関数を使用する場合は、hamming.c の記述を参考にして、窓関数を定義してください。

1.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvFFTConstMap

インクルード

```
#include “fsafft.h”
```

形式

```
unsigned long FSABhvFFTConstMap(unsigned long ulAddr)
```

引数

(in)*ulAddr*　　マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvFFTMemMap 関数は、FFT ライブラリ関数の実行に必要な定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、FFT ライブラリ関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

FSABhvHammingWindowMap

インクルード

なし

この関数のプロトタイプ宣言を記載したヘッダーファイルは存在しませんので、使用する場合、プロトタイプ宣言を適当な場所に追記してください。

形式

```
unsigned long FSABhvHammingWindowMap(unsigned long ulAddr)
```

引数

(in)*ulAddr*　　マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvHammingWindowMap 関数は、hamming.c で定義されたハミング窓関数の定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、ハミング窓関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに窓関数の定数配列をマッピングします。

1.6 注意事項（重要）

１）終了検出

ライブラリ関数は、必要な FSA のレジスタ設定を行って FSA を起動するだけです。したがって、関数を抜けた時点で、FSA 処理が終了したことを保証するものではありません。処理の終了検出は、wait_fsa_finish 関数で行う必要があります。

wait_fsa_finish 関数の使用例

```
FsaPowSpectAnalyze256(pFsaReg, ai16HammingWindow, ai16FrameData, ai32Spectrum);

wait_fsa_finish(pFsaReg);   // FSA の処理が終了するまでホスト CPU が Halt します。
```

２）入出力データポインタのアライメント

複素 FFT、実数 FFT、DCT-4 関数の引数 pData は、入力データのサイズでアライメントされたアドレスである必要があります。例えば、1024 ポイントの複素 FFT を行う場合、2048 ワードアライメントされたアドレスでなければなりません。2048 ワードアライメントとは、仮にデータ語長が 4 バイト（_FFT_DATA_BITS の#define 定義が 32）のとき、2048 x 4 = 2^{13} ですので、pData のポインタアドレスの少なくとも下位 13 ビットがすべて 0 となるアドレスです。表 1.5 にポイント数に応じた入出力データポインタのワードアライメント要件を示します。

なお、FsaPowSpectAnalyzeXXXX 関数の引数 pData にはアライメント制約はありません。

表 1.5 入出力データポインタのアライメント

ポイント数	複素 FFT	実数 FFT	DCT-4
16	32	-	-
32	64	32	-
64	128	64	64
128	256	128	128
256	512	256	256
512	1024	512	512
1024	2048	1024	1024
2048	-	2048	2048

３）SIN/COS 定数テーブルのアライメント

SIN/COS 定数テーブルもポイント数に応じたアライメントが必要です。ただ、このアライメント制約はリンカによって自動的に解決されます。

1.7 パフォーマンス

FSA の命令ステップ数を表 1.6 に示します。表中の数値はビヘイビアモデルによる理論的な評価値です。メモリアクセスの競合によりこの数値より 1.5～2 倍程度遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックするオーバーヘッドが若干必要です。

表 1.6 FSA の命令ステップ数

ライブラリ関数	ポイント数							
	16	32	64	128	256	512	1024	2048
FsaCFFT		752	1733	3998	9093	20166	44627	
Fsa1024pCFFT							42070	
Fsa512pCFFT						18951		
Fsa256pCFFT					8488			
Fsa128pCFFT				3725				
Fsa64pCFFT			1634					
Fsa32pCFFT		677						
Fsa16pCFFT	286							
FsaRFFT			940	2138	4716	10516	22998	50276
Fsa2048pRFFT								47708
Fsa1024pRFFT							21773	
Fsa512pRFFT						9902		
Fsa256pRFFT					4435			
Fsa128pRFFT				1992				
Fsa64pRFFT			859					
Fsa32pRFFT		380						
FsaDCT4			1436	3121	6674	14425	30810	65895
FsaPowSpectAnalyze2048								79156
FsaPowSpectAnalyze1024							36435	
FsaPowSpectAnalyze512						16688		
FsaPowSpectAnalyze256					7555			
FsaPowSpectAnalyze128				3402				
FsaPowSpectAnalyze64			1490					
FsaPowSpectAnalyze32		654						

1.8 要求メモリ

表 1.7 に汎用関数（FsaCFFT、FsaRFFT、FsaDCT4）の要求メモリサイズをまとめました。C17 プログラム、SIN/COS 定数テーブル、その他定数テーブルは、サポートする FFT、DCT-4 のポイント数に応じて増減します。表 1.7 の要求メモリはフルサポート時のサイズです。サポートするポイント数を限定すれば、このサイズより小さくなります。

表中の FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。なお、ポイント数に応じた入出力データの RAM 領域は別途必要になります。

表 1.7 FsaCFFT、FsaRFFT、FsaDCT4 関数の要求メモリサイズ

項目	FsaCFFT	FsaRFFT	FsaDCT4
FSA プログラム（byte）	244	476	508
C17 プログラム（byte）	210	212	584
SIN/COS 定数テーブル（word）	1024	2048	1120
その他定数テーブル（byte）	92	92	92
FSA スタックメモリ（byte）	0	32	16

表 1.8 および表 1.9 は、各々FsaXXXXpCFFT および FsaXXXXpRFFT の要求メモリサイズです。ポイント数に応じた入出力データの RAM 領域は別途必要になります。

表 1.8 FsaXXXpCFFT 関数の要求メモリサイズ

項目	複素 FFT ポイント数						
	16	32	64	128	256	512	1024
FSA プログラム（byte）	172	196	220	232	340	412	764
C17 プログラム（byte）	20	20	20	20	20	20	20
SIN/COS 定数テーブル（word）	16	32	64	128	256	512	1024
FSA スタックメモリ（byte）	0	0	0	0	0	0	0

表 1.9 FsaXXXpRFFT 関数の要求メモリサイズ

項目	実数 FFT ポイント数						
	32	64	128	256	512	1024	2048
FSA プログラム（byte）	372	396	420	432	540	612	964
C17 プログラム（byte）	20	20	20	20	20	20	20
SIN/COS 定数テーブル（word）	32	64	128	256	512	1024	2048
FSA スタックメモリ （byte）	32	32	32	32	32	32	32

表 1.10 は、FsaPowSpectAnalyzeXXXX 関数の要求メモリサイズです。フレームサイズに応じた入出力データの RAM 領域は別途必要になります。

表 1.10 FsaPowSpectAnalyzeXXXX 関数の要求メモリサイズ

項目	フレームサイズ						
	32	64	128	256	512	1024	2048
FSA プログラム（byte）	436	460	484	496	604	684	1036
C17 プログラム（byte）	92	92	92	92	92	92	92
SIN/COS 定数テーブル（word）	32	64	128	256	512	1024	2048
窓関数テーブル（byte）	32	64	128	256	512	1024	2048
FSA スタックメモリ（byte）	164	292	548	1060	2084	4132	8228

2. FIR フィルタ

2.1 概要

FIR フィルタライブラリは、図 2.1 に示すような構成の直接形 FIR フィルタ構成により FIR フィルタを実現するライブラリです。主に、Z バッファの更新（データ移動）を行う FsaFIRFifoUpdate 関数と積和演算を行う FsaFIRFilter 関数の 2 つで構成されます。

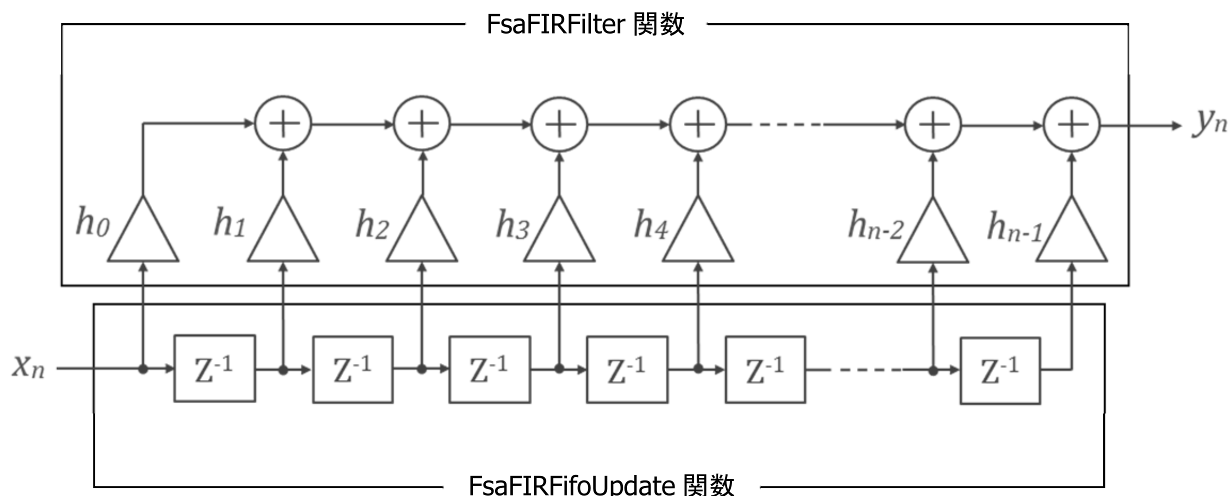

図 2.1 直接形 FIR フィルタ

Z バッファの更新部分が独立しているので、図 2.2 に示すように、Z バッファを共用して複数の異なる特性のフィルタ処理を施すような使い方も可能です。

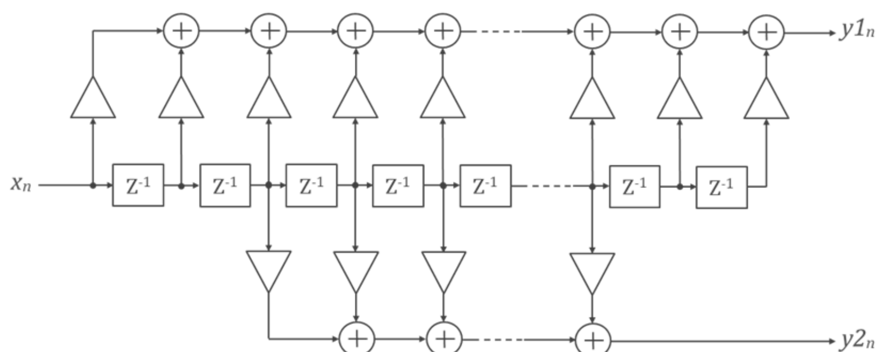

図 2.2 Z バッファを共用した直接形 FIR フィルタの例

2.2 ファイル構成

表 2.1 ソースファイル一覧

ソースファイル	記述内容
fsafirft.h	API 関数宣言、コンフィグレーション定義
fsafirft.c	API 関数定義
fsafirdef.h	フィルタ係数のマクロ定義
fsafircoef.c	フィルタ係数配列の定義

ライブラリ本体のソースファイルは、fsafirft.h と fsafirft.c です。fsafirdef.h と fsafircoef.c は、フィルタ係数の定義とその定数配列の記述方法を示すサンプルです。

2.3 コンフィグレーション

fsafirft.h の#define 定義を変更することで、下記のコンフィグレーションが可能です。

フィルタ係数の型指定

フィルタ係数を short 型としてライブラリを構成する場合、下記の定義を無効にします。long 型の場合は有効にしてください。

fsafirft.h の記述

```
// If the below definition is valid, the coefficients are long-type.
// If not, it is short-type.
#define FIRFT_COEF_LONG
//------------------------------------------------------------------------------
```

2.4 フィルタ係数定義

fsafirdef.h および fsafircoef.c に、フィルタ係数の定義のサンプルが記述されています。適宜修正して使用してください。なお、fsafircoef.c はライブラリ本体には組み込まれませんので、別途プロジェクトに追加してください。

フィルタ係数

下記はタップ数 N=49 のカイザー窓による LPF のフィルタ係数のマクロ定義です。ここで定義されたマクロに基づき、fsafircoef.c にて short 型あるいは long 型の定数配列が構成されます。

なお、フィルタのタップ数は必ず奇数にしてください。原則的に FIR フィルタのフィルタ係数は左右対称ですので、下記のサンプルのように、N=49 の場合は前半の 25 個係数のみ定義します。

fsafirdef.h の記述

```
// Filter length : N = 49
// Kinf of filter : LPF
// Window function : Kaiser
// Normalized cutoff frequency : 0.10
// Attenuation : 50 [dB]
#define c00          4.262579827562175e-04
#define c01          1.057464549874083e-03
#define c02          1.515867962922970e-03
#define c03          1.283938716678974e-03
#define c04         -1.419458250029041e-18
#define c05         -2.204249653434355e-03
#define c06         -4.526273576420264e-03
#define c07         -5.654596601451008e-03
#define c08         -4.310334519600676e-03
#define c09          3.288290749325822e-18
#define c10          6.367555978466844e-03
#define c11          1.238791425256506e-02
#define c12          1.482296367964778e-02
#define c13          1.092855408889568e-02
#define c14         -5.426410134126872e-18
#define c15         -1.553665688460399e-02
#define c16         -3.011089367120460e-02
#define c17         -3.634335900517877e-02
#define c18         -2.746018419900938e-02
#define c19          7.139943781628102e-18
#define c20          4.422375302942928e-02
#define c21          9.778504534048262e-02
#define c22          1.492675566326206e-01
#define c23          1.864469311647859e-01
#define c24          2.000000000000000e-01
//--------------------------------------------------------------------------------
```

フィルタ係数の小数点位置の設定

フィルタ係数は fsafircoef.c に記述された下記のマクロによって、short 型あるいは long 型に変換されます。

fsafircoef.c の記述

```
#ifdef FIRFT_COEF_LONG
#define FLOAT2INT(a, b)          ((long)((a) * (long long)(b)))
#else
#define FLOAT2INT(a, b)          ((short)((a) * (long long)(b)))
#endif

#ifdef FIRFT_COEF_LONG
const long g_aiFIRFilterCons[] __FSA_CONSTANTS__ = {
#else
const short g_aiFIRFilterCons[] __FSA_CONSTANTS__ = {
#endif
      FLOAT2INT(c00, (1LL << FIRFT_FIXED_POINT)),
      FLOAT2INT(c01, (1LL << FIRFT_FIXED_POINT)),
      FLOAT2INT(c02, (1LL << FIRFT_FIXED_POINT)),
      FLOAT2INT(c03, (1LL << FIRFT_FIXED_POINT)),
      FLOAT2INT(c04, (1LL << FIRFT_FIXED_POINT)),
      FLOAT2INT(c05, (1LL << FIRFT_FIXED_POINT)),
```

そのため、定義したフィルタ係数が整数型でオーバーフローしない最大の小数点位置を決定し、下記のように定義します。但し、最大でも 32 を超えないようにしてください。

fsafirdef.h の記述

```
// Fixed point position of filter coefficients.
#ifdef FIRFT_COEF_LONG
#define FIRFT_FIXED_POINT                    (17 + 15)
#else
#define FIRFT_FIXED_POINT                    17
#endif
//----------------------------------------------------------------------------
```

2.5 API 関数

FsaFIRInit

インクルード

```
#include  "fsafirft.h"
```

形式

```
void FsaFIRInit(
        FSAFIRHDR *pstFsaFirHdr,
        short ai16Fifo[],
        const void *pCoef,
        const unsigned int uiFifoLen,
        const unsigned int uiTaps,
        const unsigned int uiShift
)
```

引数

(in/out)*pstFsaFirHdr*	FSAFIRHDR 構造体のポインタ 外部で定義した FSAFIRHDR 構造体変数のポインタを渡します。
(in)*ai16Fifo*	FIFO バッファのポインタ Z バッファとして使用する FIFO バッファの領域を指定します。少なくともフィルタタップ数以上のサイズが必要です。FIFO バッファは FSA がアクセス可能なメモリに配置してください。
(in)*pCoef*	フィルタ係数配列のポインタ 　フィルタ係数配列のポインタを渡します。フィルタ係数配列は、FSA がアクセス可能なメモリに配置してください。
(in)*uiFifoLen*	FIFO バッファサイズ FIFO バッファのサイズを与えます。設定範囲は 3～65535 です。
(in)*uiTaps*	フィルタタップ数 3～513 範囲の奇数のフィルタタップ数を与えます。
(in)*uiShift*	フィルタ係数の小数点以下ビット数 pCoef で設定したフィルタ係数の小数点以下のビット数を与えます。設定範囲は 0～32 です。

戻り値

なし

説明

FsaFIRInit は、FSAFIRHDR 構造体の設定を行う関数です。この関数で設定した pstFsaFirHdr を FsaFIRFifoUpdate 関数および FsaFIRFilter 関数に渡すことで、これらの関数はその設定に基づき FIFO バッファの更新、FIR フィルタ処理を実行します。

FsaFIRFifoUpdate

インクルード

```
#include "fsafirft.h"
```

形式

```
void FsaFIRUpdate(FSAREG *pFsaReg, FSAFIRHDR *pstFsaFirHdr, short i16InData)
```

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*pstFsaFirHdr* FSAFIRHDR 構造体のポインタ
(in)*i16InData* 入力データ

戻り値

なし

説明

FsaFIRInit 関数で設定された FIFO バッファの先頭に、i16InData で与えられたデータを書き込みます。その際、最も古い FIFO バッファのデータは上書きされます。

FIFO バッファの先頭とは、FsaFIRInit 関数の引数 ai16Fifo の ai16Fifo[0]に該当します。また、最も古いデータとは、ai16Fifo[uiFifoLen - 1]を指します。

FsaFIRFilter

インクルード

```
#include "fsafirft.h"
```

形式

```
void FsaFIRFilter(
        FSAREG *pFsaReg,
        FSAFIRHDR *pstFsaFirHdr,
        short *pi16OutData
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*pstFsaFirHdr*	FSAFIRHDR 構造体のポインタ
(out)*pi16OutData*	出力データのポインタ FSA がアクセス可能なメモリ上のポインタを指定してください。

戻り値

なし

説明

FsaFIRInit 関数で決定された FIFO バッファの先頭からデータを読み込み、指定されたフィルタ係数を使って FIR フィルタ処理を行います。通常は下記に示すように、FsaFIRFifoUpdate 関数とペアで実行して、FIR フィルタを構成します。

FsaFIRFilter 関数は、必要な FSA のレジスタ設定を行って FSA を起動すると直ちに関数を終了します。そのため、関数から復帰した時点では、*pi16OutData が更新されていない可能性があります。したがって、原則的に、*pi16OutData にアクセス前に、wait_fsa_finish 関数を実行してください。

```
FSAFIRHDR g_stFsaFirHdr;
short g_i16OutData __FSA_WORK__;
short g_ai16Fifo[49];
const short g_ai16Coef[25] = {. . . . .};

FsaFIRInit(&g_stFsaFirHdr, g_ai16Fifo, g_ai16Coef, 49, 49, 17);

while(1) {
     short i16InData, i16OutData;
     . . . . . . . .

     i16InData = GetNewData();
     . . . . . . . .

     FsaFIRFifoUpdate(pFsaReg, &g_stFsaFirHdr, i16InData);
     FsaFIRFilter(pFsaReg, &g_stFsaFirHdr, &g_i16OutData);
     wait_fsa_finish(pFsaReg);
     i16OutData = g_i16OutData;
     . . . . . . . .

}
```

2.6 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvFIRConstMap

インクルード

```
#include “fsafirdef.h”
```

形式 `unsigned long FSABhvFIRConstMap(unsigned long ui32Addr)`

引数

(in)*ui32Addr* マッピング仮想アドレス

戻り値

ui32Addr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvFIRConstMap 関数は、fsafircoef.c に記述されたフィルタ係数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。関数の定義も fsafircoef.c にあります。

FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、FsaFIRFilter 関数を実行する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスにフィルタ係数テーブルをマッピングします。

2.7 パフォーマンス

FsaFIRFifoUpdate 関数および FsaFIRFilter 関数の FSA の命令ステップ数を表 2.2 に示します。なお、表 2.2 に示した FSA の命令ステップ数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 2.2　FSA の命令ステップ数（N=49）

ライブラリ関数	FSA の命令ステップ数
FsaFIRFifoUpdate	98
FsaFIRFilter	58

2.8 要求メモリ

表 2.3　プログラムおよびワークメモリのサイズ（byte）

種別	C17	FSA	備考
プログラム	316	80	
FSAFIRHDR 構造体	12	-	
FIFO バッファ	-	2 x n	n は FIR フィルタのタップ数
short 型フィルタ係数配列	-	n + 1	long 型は 2 倍です。
FSA スタックメモリ(*1)	-	0	

(*1) FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

3. IIR フィルタ

3.1 概要

IIR フィルタライブラリは、#define 定義により静的に指定された数の入出力サンプル系列に対して、IIR フィルタ処理を行うライブラリです。IIR フィルタは、図 3.1 に示すような構成の 2 次の直接形 IIR フィルタを基本セクションとし、これを縦続接続することでフィルタを構成します。

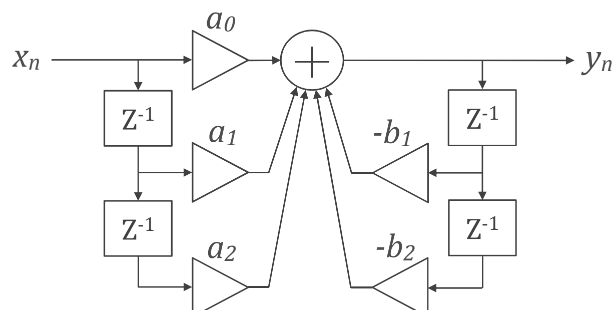

図 3.1 IIR フィルタの基本セクション（2 次の直接形 IIR フィルタ）

フィルタ段数を 3 段にした場合の構成を図 3.2 に示します。IIR フィルタライブラリは 1 段から 6 段までの IIR フィルタ構成に対応可能です。

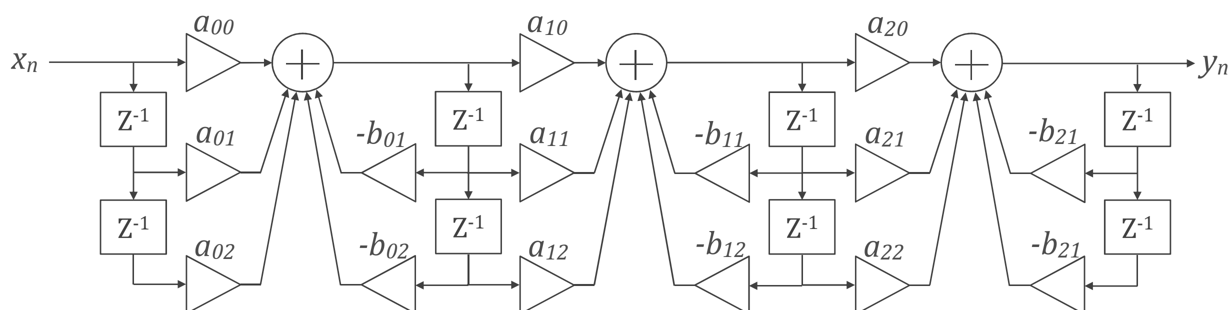

図 3.2 IIR フィルタの構成（フィルタ段数 3 の例）

また、IIR フィルタライブラリは、IIR フィルタで適当な帯域制限を行った上で、入力するサンプル系列を 1/2 や 1/3 にダウンサンプリングして出力することが可能です。

3.2 ファイル構成

表 3.1 ソースファイル一覧

ソースファイル	記述内容
fsaiirft.h	API 関数宣言、コンフィグレーション定義
fsaiirft.c	API 関数定義
fsaiirdef.h	フィルタ係数のマクロ定義
fsaiircoef.c	フィルタ係数配列の定義

ライブラリ本体のソースファイルは、fsaiirft.h と fsaiirft.c です。fsaiirdef.h と fsaiircoef.c は、フィルタ係数の定義とその定数配列の記述方法を示すサンプルです。

3.3 コンフィグレーション

fsaiirft.h の#define 定義を変更することで、下記のコンフィグレーションが可能です。

入力サンプル数

1 回の FsaIIRFilter 関数コールで処理するサンプル数を設定します。

設定可能な値は、1 から IIRFT_OUT_SAMPLE の設定値の 256 倍までです。

fsaiirft.h の記述

```
// Number of input sample.
#define IIRFT_IN_SAMPLE                          500
//--------------------------------------------------------------------------------
```

出力サンプル数

出力サンプル数を指定します。IIRFT_IN_SAMPLE より小さい値を指定すると、フィルタ処理の後の出力がダウンサンプルされます。但し、IIRFT_IN_SAMPLE は IIRFT_OUT_SAMPLE で割り切れる数値にしてください。なお、IIRFT_OUT_SAMPLE の設定上限は 256 です。

fsaiirft.h の記述

```
// Number of output sample.
#define IIRFT_OUT_SAMPLE                         250
// Notice: IIRFT_IN_INPUTS / IIRFT_OUT_SAMPLE must be devided.
// Notice: IIRFT_OUT_SAMPLE must be less than 256.
//--------------------------------------------------------------------------------
```

3.4 フィルタ係数定義

fsaiirdef.h および fsaiircoef.c にフィルタ係数の定義のサンプルが記述されています。適宜修正して使用してください。なお、fsaiircoef.c はライブラリ本体には組み込まれませんので、別途プロジェクトに追加してください。

フィルタ段数

フィルタ段数を指定します。1 段～6 段まで指定可能です。

fsaiirdef.h の記述

```
// Filter step setting.
#define IIRFT_FILTER_STEP                    3
// Notice: From 1 to 6 is selectable.
//------------------------------------------------------------------------------
```

フィルタ係数

下記は、図 3.2 に示した 3 段数の IIR フィルタの場合の定義です。

fsaiirdef.h の記述

```
#if IIRFT_FILTER_STEP == 1
// When the filter steps is 1, write the coefficients here.
//------------------------------------------------------------------------------

#elif IIRFT_FILTER_STEP == 2
// When the filter steps is 2, write the coefficients.
//------------------------------------------------------------------------------

#elif IIRFT_FILTER_STEP == 3
// When the filter steps is 3, write the coefficients.
#define a00            2.36698945973440944e+00*1.47222390730827796e-01
#define a01            4.73397891946881888e+00*1.47222390730827796e-01
#define a02            2.36698945973440944e+00*1.47222390730827796e-01
#define b01            -4.80831431653081953e-01
#define b02            8.74726820040162556e-01
#define a10            9.77330602428059469e-01*1.64171591923048144e-01
#define a11            1.95466120485611894e+00*1.64171591923048144e-01
#define a12            9.77330602428059469e-01*1.64171591923048144e-01
#define b11            -9.92472893712932680e-01
#define b12            6.34272577055837505e-01
#define a20            3.44200807444426937e-01*4.26584615457994087e-01
#define a21            3.44200807444426937e-01*4.26584615457994087e-01
#define a22            0.00000000000000000e+00*4.26584615457994087e-01
#define b21            -7.06338461831976239e-01
#define b22            0.00000000000000000e+00
//------------------------------------------------------------------------------
```

フィルタ係数の小数点位置の設定

フィルタ係数は fsaiircoef.c に記述された下記のマクロによって、long 型に変換されます。

fsaiircoef.c の記述

```
#define FLOAT2LONG(a, b)          ((long)((a) * (long)(b)))

const long ai32IIRFilterCons[] __FSA_CONSTANTS__ = {
      FLOAT2LONG(a00, (0x1L << IIRFT_FIXED_POINT)),
      FLOAT2LONG(a01, (0x1L << IIRFT_FIXED_POINT)),
      FLOAT2LONG(a02, (0x1L << IIRFT_FIXED_POINT)),
      FLOAT2LONG(b01, (0x1L << IIRFT_FIXED_POINT)),
      FLOAT2LONG(b02, (0x1L << IIRFT_FIXED_POINT))
```

そのため、定義したフィルタ係数が long 型でオーバーフローしない最大の小数点位置を決定し、下記のように定義します。但し、最大でも 32 を超えないようにしてください。

fsaiirdef.h の記述

```
// Fixed point position of filter coefficients.
#define IIRFT_FIXED_POINT                  30
//-----------------------------------------------------------------------------
```

3.5 API 関数

FsaIIRInit

インクルード

```
#include "fsaiirft.h"
```

形式

```
void FsaIIRInit(
        FSAIIRHDR *pstFsaIirHdr,
        long ai32ZBuf[],
        const long ai32Coef[],
        const unsigned int uiStep,
        const unsigned int uiShift
)
```

引数

(in/out)*pstFsaIirHdr*　FSAIIRHDR 構造体のポインタ
外部で定義した FSAIIRHDR 構造体変数のポインタを渡します。

(in)*ai32ZBuf*　Z バッファのポインタ
Z バッファとして使用する領域を指定します。必要なサイズはフィルタ段数に依存します。Z バッファは FSA がアクセス可能なメモリに配置してください。

表 3.2　フィルタ段数と必要な Z バッファのサイズ

フィルタ段数	Z バッファのサイズ（ワード）
1	4
2	6
3	8
4	10
5	12
6	14

(in)*ai32Coef*　フィルタ係数配列のポインタ
フィルタ段数に応じた係数配列のポインタを渡します。フィルタ係数配列は、FSA がアクセス可能なメモリに配置してください。

(in)*uiStep*　IIR フィルタ段数
1 段から 6 段まで設定可能です。

(in)*uiShift*　フィルタ係数の小数点以下ビット数
ai32Coef で設定したフィルタ係数の小数点以下のビット数を与えます。設定範囲は 0～32 です。

戻り値

なし

説明

FsaIIRInit は、FSAIIRHDR 構造体の設定を行う関数です。この関数で設定した pstFsaIirHdr を FsaIIRFilter 関数に渡すことで、FsaIIRFilter 関数はその設定に基づき IIR フィルタ処理を実行します。

FsaIIRFilter

インクルード

```
#include “fsaiirft.h”
```

形式

```
void FsaIIRFilter(
        FSAREG *pFsaReg,
        FSAIIRHDR *pstFsaIirHdr,
        short *pi16In,
        short *pi16Out
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*pstFsaIirHdr*	FSAIIRHDR 構造体のポインタ
(in)*pi16In*	入力データのポインタ
(out)*pi16Out*	出力データのポインタ

戻り値

なし

説明

pstFsaIirHdr の設定に基づき IIR フィルタ処理を行います。但し、1 回の関数コールで、pi16In から入力するデータ数は、fsaiirft.h に記述された#define IIRFT_IN_SAMPLE により静的に決定されます。同様に pi16Out に出力するデータ数は、#define IIRFT_OUT_SAMPLE で決定されます。

pi16In、pi16Out は、何れも FSA がアクセス可能なメモリを指定する必要があります。

3.6 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvIIRConstMap

インクルード

```
#include “fsaiirdef.h”
```

形式

```
unsigned long FSABhvIIRConstMap(unsigned long ui32Addr)
```

引数

(in)*ui32Addr*　　マッピング仮想アドレス

戻り値

ui32Addr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvIIRConstMap 関数は、fsaiircoef.c に記述されたフィルタ係数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、FsaIIRFilter 関数を実行する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスにフィルタ係数テーブルをマッピングします。

3.7 パフォーマンス

FsaIIRFilter 関数の FSA の命令ステップ数を表 3.3 に示します。なお、表 3.3 に示した FSA の命令ステップ数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 3.3　FSA の命令ステップ数

条件			FSA の命令ステップ数
IIRFT_IN_SAMPLE	IIRFT_OUT_SAMPLE	IIRFT_FILTER_STEP	
500	250	3	12530
500	250	4	15534

3.8 要求メモリ

表 3.4　プログラムおよびワークメモリのサイズ（byte）

種別	C17	FSA	備考
プログラム	120	112	
FSAIIRHDR 構造体	12	-	
Z バッファ	-	(n + 1) x 8	n は IIR フィルタの段数
フィルタ係数配列	-	n x 20	n は IIR フィルタの段数
FSA スタックメモリ(*1)	-	64	

(*1) FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

4. ソート

4.1 概要

本 FSA ライブラリは、指定したデータ配列（short 型 or long 型）のソート、最小値、最大値の取得等の機能を提供します。

表 4.1 サポート関数一覧

関数	概仕様
FsaSort16	short 型データ配列のソートを行います。
FsaGetMin16	short 型データ配列の最小値とそのインデックスを求めます。
FsaGetMax16	short 型データ配列の最大値とそのインデックスを求めます。
FsaGetMinMax16	short 型データ配列の最小値と最大値を同時に求めます。
FsaSort32	long 型データ配列のソートを行います。
FsaGetMin32	long 型データ配列の最小値とそのインデックスを求めます。
FsaGetMax32	long 型データ配列の最大値とそのインデックスを求めます。
FsaGetMinMax32	long 型データ配列の最小値と最大値を同時に求めます。

4.2 ファイル構成

表 4.2 ソースファイル一覧

ソースファイル	記述内容
fsasort.h	ライブラリ関数宣言
fsasort.c	ライブラリ関数定義

4.3 ファイル構成

fsasort.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

実装する関数を選択します。使用する関数の定義だけ残してコメントアウトします。

表 4.3 #define 定義と実装関数

定義	実装関数
_FSASORT16	FsaSort16
_FSAGETMIN16	FsaGetMin16
_FSAGETMAX16	FsaGetMax16
_FSAGETMINMAX16	FsaGetMinMax16
_FSASORT32	FsaSort32
_FSAGETMIN32	FsaGetMin32
_FSAGETMAX32	FsaGetMax32
_FSAGETMINMAX32	FsaGetMinMax32

fsatrig.h の記述例

```
// Function selection
// Disable the definitions except the function you will use.
// #define _FSASORT16
#define _FSAGETMIN16
#define _FSAGETMAX16
// #define _FSAGETMINMAX16
#define _FSASORT32
// #define _FSAGETMIN32
// #define _FSAGETMAX32
// #define _FSAGETMINMAX32
```

ソート順序の選択

FsaSort16 および FsaSort32 のソート順序を下記の_FSASORT_ORDER の定義で設定します。降順で行う場合は 0 を、昇順で行う場合は 1 とします。

fsatrig.h の記述例

```
// Sort order setting in FsaSort16 and FsaSort32
// In case of descending order, the value should be 0.
// In case of Ascending order, the value should be 1.
#define _FSASORT_ORDER                                  0
```

4.4 API 関数

FsaSort16

インクルード

```
#include “fsasort.h”
```

形式

```
void FsaSort16(
        FSAREG *pFsaReg,
        short *pi16Dst,
        short *pi16Src,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi16Dst*	出力データ配列のポインタ
(in)*pi16Src*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

なし

説明

pi16Src で与えられた要素数 ui16Num 個の short 型データ配列のソートを行い、pi16Dst が示すアドレスに格納します。ソートの順序は fsasort.h の_FSASORT_ORDER の定義で設定してください。

ui16Num の指定可能範囲は、1～256 です。pi16Src および pi16Dst は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMin16

インクルード

```
#include "fsasort.h"
```

形式

```
unsigned short FsaGetMin16(
        FSAREG *pFsaReg,
        short *pi16Min,
        short *pi16Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi16Min*	最小値が格納されるポインタ
(in)*pi16Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

最小値のインデックス番号

説明

pi16Data で与えられた要素数 ui16Num 個の short 型データ配列の最小値を*pi16Min に格納します。同時、その最小値が入力データの何番目かを示すインデックス番号を、戻り値として返します。入力データの先頭が最小値の場合は、戻り値は 0 です。同じ値の最小値が複数存在する場合は、番号の若いインデックスが返されます。

ui16Num の指定可能範囲は、2～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMax16

インクルード

```
#include “fsasort.h”
```

形式

```
unsigned short FsaGetMax16(
        FSAREG *pFsaReg,
        short *pi16Max,
        short *pi16Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi16Max*	最大値が格納されるポインタ
(in)*pi16Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

最大値のインデックス番号

説明

pi16Data で与えられた要素数 ui16Num 個の short 型データ配列の最大値を*pi16Max に格納します。同時、その最大値が入力データの何番目かを示すインデックス番号を、戻り値として返します。入力データの先頭が最大値の場合は、戻り値は 0 です。同じ値の最大値が複数存在する場合は、番号の若いインデックスが返されます。

ui16Num の指定可能範囲は、2～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMinMax16

インクルード

```
#include "fsasort.h"
```

形式

```
void FsaGetMinMax16(
        FSAREG *pFsaReg,
        short ai16MinMax[2],
        short *pi16Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*ai16MinMax[2]*	最小値と最大値が格納される配列のポインタ
(in)*pi16Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

なし

説明

pi16Data で与えられた要素数 ui16Num 個の short 型データ配列の最小値を ai16MinMax[0]に、最大値を ai16MinMax[1]に格納します。

ui16Num の指定可能範囲は、2～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaSort32

インクルード

```
#include “fsasort.h”
```

形式

```
void FsaSort32(
        FSAREG *pFsaReg,
        long *pi32Dst,
        long *pi32Src,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi32Dst*	出力データ配列のポインタ
(in)*pi32Src*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

なし

説明

pi32Src で与えられた要素数 ui16Num 個の long 型データ配列のソートを行い、pi32Dst が示すアドレスに格納します。ソートの順序は fsasort.h の_FSASORT_ORDER の定義で設定してください。

ui16Num の指定可能範囲は、1～256 です。pi32Src および pi32Dst は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMin32

インクルード

```
#include “fsasort.h”
```

形式

```
unsigned short FsaGetMin32(
        FSAREG *pFsaReg,
        long *pi32Min,
        long *pi32Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi32Min*	最小値が格納されるポインタ
(in)*pi32Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

最小値のインデックス番号

説明

pi32Data で与えられた要素数 ui16Num 個の long 型データ配列の最小値を*pi32Min に格納します。同時、その最小値が入力データの何番目かを示すインデックス番号を、戻り値として返します。入力データの先頭が最小値の場合は、戻り値は 0 です。同じ値の最小値が複数存在する場合は、番号の若いインデックスが返されます。

ui16Num の指定可能範囲は、2～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMax32

インクルード

```
#include "fsasort.h"
```

形式

```
unsigned short FsaGetMax32(
        FSAREG *pFsaReg,
        long *pi32Max,
        long *pi32Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi32Max*	最大値が格納されるポインタ
(in)*pi32Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

最大値のインデックス番号

説明

pi32Data で与えられた要素数 ui16Num 個の long 型データ配列の最大値を*pi32Max に格納します。同時、その最大値が入力データの何番目かを示すインデックス番号を、戻り値として返します。入力データの先頭が最大値の場合は、戻り値は 0 です。同じ値の最大値が複数存在する場合は、番号の若いインデックスが返されます。

ui16Num の指定可能範囲は、2～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaGetMinMax32

インクルード

```
#include "fsasort.h"
```

形式

```
void FsaGetMinMax32(
        FSAREG *pFsaReg,
        long ai32MinMax[2],
        long *pi32Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*ai32MinMax[2]*	最小値と最大値が格納される配列のポインタ
(in)*pi32Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

なし

説明

pi32Data で与えられた要素数 ui16Num 個の long 型データ配列の最小値を ai32MinMax[0]に、最大値を ai32MinMax[1]に格納します。

ui16Num の指定可能範囲は、2～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

4.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

本ライブラリは定数テーブルを使用しないので、メモリ割り当て用の関数はありません。

4.6 パフォーマンス

入力データ配列の要素数が 200 の場合の FSA の命令ステップ数を表 4.4 に示します。表中の数値はビヘイビアモデルによる評価値です。実機ではメモリアクセスの競合により、この数値より若干遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 4.4 FSA の命令ステップ数

ライブラリ関数	ステップ数
FsaSort16	121394
FsaGetMin16	805
FsaGetMax16	805
FsaGetMinMax16	1201
FsaSort32	121393
FsaGetMin32	804
FsaGetMax32	804
FsaGetMinMax32	1200

4.7 要求メモリ

表 4.5、表 4.6 に要求メモリサイズをまとめます。FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

表 4.5 short 型データ配列用ライブラリの要求メモリサイズ（単位：byte）

項目	FsaSort16	FsaGetMin16	FsaGetMax16	FsaGetMinMax16
C17 プログラム	92	140	140	138
FSA プログラム	88	52	52	52
定数テーブル	0	0	0	0
FSA スタックメモリ	0	0	0	0

表 4.6 long 型データ配列用ライブラリの要求メモリサイズ（単位：byte）

項目	FsaSort32	FsaGetMin32	FsaGetMax32	FsaGetMinMax32
C17 プログラム	92	144	144	146
FSA プログラム	80	48	48	48
定数テーブル	0	0	0	0
FSA スタックメモリ	0	0	0	0

5. 三角関数

5.1 概要

本 FSA ライブラリは、以下の三角関数を提供します。

- 三角関数　sin、cos
- 逆三角関数　atan2、asin

表 5.1　サポート関数一覧

関数	概仕様
FsaCos	cos(x)を返す関数
FsaSin	sin(x)を返す関数
FsaCosSin	cos(x)、sin(x)を同時に求める関数
FsaAtan2	atan2(y, x)を返す関数
FsaAsin	asin(x)を返す関数

5.2 ファイル構成

表 5.2　ソースファイル一覧

ソースファイル	記述内容
fsatrig.h	ライブラリ関数宣言
fsatrig.c	ライブラリ関数定義
fsaleftshift32.c	32 ビット左シフト用定数テーブル

fsaleftshift32.c は、他のライブラリからも使用される共用の定数配列です。本ライブラリには組み込まれませんので、別途ターゲットプロジェクトに追加する必要があります。

5.3 コンフィグレーション

fsatrig.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

実装する関数を選択します。使用する関数の定義だけ残してコメントアウトします。

表 5.3 #define 定義と実装関数

定義	実装関数
_FSACOS	FsaCos
_FSASIN	FsaSin
_FSACOSSIN	FsaCosSin
_FSAATAN2	FsaAtan2
_FSAASIN	FsaAsin

fsatrig.h の記述例

```
// Function selection
// Disable the definitions except the function you will use.
#define _FSACOS
#define _FSASIN
//#define _FSACOSSIN
#define _FSAATAN2
//#define _FSAASIN
```

5.4 API 関数

FsaCos

インクルード

```
#include “fsatrig.h”
```

形式 `long FsaCos(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 余弦を求める角度（ラジアン）

戻り値

cos(x)の値

説明

入力の角度 x は小数点以下 29 ビットのラジアンです。入力範囲は約±4 ラジアン（0x80000000 から 0x7FFFFFFF）で、度で表現すると約±230 度に相当します。

戻り値の余弦は小数点以下 30 ビットの数値です。数値の範囲は数学的な定義から、±1.0（0xC0000000 から 0x40000000）です。（ただし微小な計算誤差を除く）

FsaSin

インクルード

```
#include “fsatrig.h”
```

形式 `long FsaSin(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 正弦を求める角度（ラジアン）

戻り値

sin(x)の値

説明

入力の角度 x は小数点以下 29 ビットのラジアンです。入力範囲は約±4 ラジアン（0x80000000 から 0x7FFFFFFF）で、度で表現すると約±230 度に相当します。

戻り値の正弦は小数点以下 30 ビットの数値です。数値の範囲は数学的な定義から、±1.0（0xC0000000 から 0x40000000）です。（ただし微小な計算誤差を除く）

FsaCosSin

インクルード

```
#include  “fsatrig.h”
```

形式 `void FsaCosSin(FSAREG *pFsaReg, long alCosSin[2], long x)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*alCosSin[2]*	cos(x)、sin(x)が格納される配列
(in)*x*	余弦、正弦を求める角度（ラジアン）

戻り値

なし

説明

FsaCosSin 関数は余弦と正弦を同時に求めます。入力の角度 x は小数点以下 29 ビットのラジアンです。入力範囲は約±4 ラジアン（0x80000000 から 0x7FFFFFFF）で、度で表現すると約±230 度に相当します。

出力は、alCosSin[0]に cos(x)を、alCosSin[1]に sin(x)を格納します。出力の余弦および正弦は小数点以下 30 ビットの数値です。数値の範囲は数学的な定義から、±1.0（0xC0000000 から 0x40000000）です。（ただし微小な計算誤差を除く）

FsaCosSin 関数を使用する場合の注意点

- 結果を受け取る配列 ai32CosSin[2]は、FSA がアクセス可能なメモリに配置されている必要があります。
- FsaCosSin 関数は、必要な FSA のレジスタ設定を行って FSA を起動すると直ちに関数を終了します。そのため、関数から復帰した時点では、FSA 処理は終了していない可能性があります。したがって、alCosSin[2]から演算結果をリードする前に、wait_fsa_finish 関数を実行し、FSA 処理の終了待ちを行ってください。

wait_fsa_finish 関数の使用例

```
FsaCosSin(pFsaReg, ai32CosSin, x);

wait_fsa_finish(pFsaReg);    // FSA の処理が終了するまでホスト CPU が Halt します。

i32Cos = ai32CosSin[0];
i32Sin = ai32CosSin[1];
```

FsaAtan2

インクルード

```
#include "fsatrig.h"
```

形式　`long FsaAtan2(FSAREG *pFsaReg, long y, long x)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*y*	atan2(y, x) の y
(in)*x*	atan2(y, x) の x

戻り値

atan2(y, x) の値（ラジアン）

説明

x と y で指定される値を使って atan を求めて返します。入力の x と y の固定小数点位置は、同一であれば任意です。戻り値は小数点以下 29 ビット、単位ラジアンの角度です。

$x > 0, y > 0$： 0〜+π/2 の角度
$x > 0, y < 0$： 0〜-π/2 の角度
$x < 0, y > 0$： +π/2〜+π の角度
$x < 0, y < 0$： -π/2 〜-π の角度
$x = 0, y > 0$： +π/2
$x = 0, y < 0$： -π/2
$x > 0, y = 0$： 0
$x < 0, y = 0$： +π

標準 math ライブラリと異なり、$x = y = 0$ の場合は例外的に 0 を返します。

FsaAsin

インクルード

```
#include "fsatrig.h"
```

形式　`long FsaAsin(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*x*	asin(x)の x（正弦の値）

戻り値

asin(x) の値（ラジアン）

説明

入力 x は小数点以下 29 ビットで表現された正弦です。戻り値は小数点以下 29 ビット、単位ラジアンの角度です。

入力範囲は約±4.0（0x80000000 から 0x7FFFFFFF）です。標準 math ライブラリでは±1.0 の範囲外の場合はエラーを返しますが、FsaAsin 関数は、-1 より小さい負の入力に対しては-π/2、+1 より大きい正の入力に対しては+π/2 に相当する戻り値を返します。

出力は数学的な定義から、-π/2 から+π/2 を 29 ビット右シフトした値です。（ただし微小な計算誤差を除く）

5.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvTrigConstMap

インクルード

```
#include “fsatrig.h”
```

形式 `unsigned long FSABhvTrigConstMap(unsigned long ulAddr)`

引数

ulAddr マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvTrigConstMap 関数は、三角関数ライブラリで使用する定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、三角関数ライブラリの関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

FSABhvLeftShift32ConstMap

インクルード

```
#include “fsatrig.h”
```

形式 `unsigned long FSABhvLeftShift32ConstMap(unsigned long ulAddr)`

引数

ulAddr マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

この関数は、FsaAtan2、FsaAsin 関数で使用する 32 ビット左シフトの定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、FsaAtan2、FsaAsin 関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに 32 ビット左シフトの定数配列をマッピングします。

5.6 パフォーマンス

FSA の命令ステップを表 5.4 に示します。表中の数値はビヘイビアモデルによる評価値です。実機ではメモリアクセスの競合により、この数値より若干遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 5.4 FSA の命令ステップ

ライブラリ関数	命令ステップ
FsaCos	47
FsaSin	47
FsaCosSin	118
FsaAtan2	203
FsaAsin	249

5.7 要求メモリ

表 5.5 に要求メモリサイズをまとめます。表のサイズは、各々の関数を単独で組み込んだ場合のメモリリソースを示しています。FsaCos 関数と FsaSin 関数は FSA プログラムを共有することができます。また、FsaCos 関数、FsaSin 関数、FsaCosSin 関数は、定数テーブルを共有することができます。

FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

表 5.5 要求メモリサイズ（単位：byte）

項目	FsaCos	FsaSin	FsaCosSin	FsaAtan2	FsaAsin
C17 プログラム	100	48	80	144	100
FSA プログラム	152	152	216	340	352
定数テーブル	40	40	40	104	72
FSA スタックメモリ	4	4	8	20	16

6. 三角関数 2

6.1 概要

FSA 三角関数ライブラリ 2 は、π/2 で正規化された角度を入力とする sin/cos 関数を提供するもので。各々16 ビット版と 32 ビット版の関数があります。16 ビット版はミニマックス近似、32 ビット版はマクローリン近似で計算します。

表 6.1 サポート関数一覧

関数	概仕様
FsaSin32	Q30 フォーマットの入力 x（$-2.0 \leqq x < 2.0$）を引数で渡し、Q30 フォーマットの $sin(\pi x/2)$を返します。
FsaCos32	Q30 フォーマットの入力 x（$-2.0 \leqq x < 2.0$）を引数で渡し、Q30 フォーマットの $cos(\pi x/2)$ を返します。
FsaSin16	Q14 フォーマットの入力 x（$-2.0 \leqq x < 2.0$）を引数で渡し、Q14 フォーマットの $sin(\pi x/2)$を返します。
FsaCos16	Q14 フォーマットの入力 x（$-2.0 \leqq x < 2.0$）を引数で渡し、Q14 フォーマットの $cos(\pi x/2)$ を返します。

6.2 ファイル構成

表 6.2 ソースファイル一覧

ソースファイル	記述内容
fsatrig2.h	ライブラリ関数宣言
fsatrig2.c	ライブラリ関数定義

6.3 コンフィグレーション

fsatrig2.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

実装する関数を選択します。使用する関数の定義だけ残してコメントアウトします。

表 6.3 #define 定義と実装関数

定義	実装関数
_FSASIN32	FsaSin32
_FSACOS32	FsaCos32
_FSASIN16	FsaSin16
_FSACOS16	FsaCos16

fsatrig2.h の記述例

```
// Function selection
// Disable the definitions except the function you will use.
#define _FSASIN32
#define _FSACOS32
// #define _FSASIN16
// #define _FSACOS16
//-------------------------------------------------------------------------------
```

6.4 API 関数

FsaSin32

インクルード

```
#include “fsatrig2.h”
```

形式 `long FsaSin32(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*x*	Q30 フォーマット固定小数点数（-2.0≦x<2.0）を指定

戻り値

Q30 フォーマット固定小数点数（-1.0～1.0）で表現された正弦が返されます。

説明

sin(*πx*/2)を計算します。

FsaCos32

インクルード

```
#include “fsatrig2.h”
```

形式 `long FsaCos32(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*x*	Q30 フォーマット固定小数点数（-2.0≦x<2.0）を指定

戻り値

Q30 フォーマット固定小数点数（-1.0～1.0）で表現された余弦が返されます。

説明

cos(*πx*/2)を計算します。

FsaSin16

インクルード

```
#include “fsatrig2.h”
```

形式 `short FsaSin16(FSAREG *pFsaReg, short x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* Q14 フォーマット固定小数点数（-2.0≦x<2.0）を指定

戻り値

Q14 フォーマット固定小数点数（-1.0～1.0）で表現された正弦が返されます。

説明

$sin(\pi x/2)$を計算します。

FsaCos16

インクルード

```
#include “fsatrig2.h”
```

形式 `short FsaCos16(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* Q14 フォーマット固定小数点数（-2.0≦x<2.0）を指定

戻り値

Q14 フォーマット固定小数点数（-1.0～1.0）で表現された余弦が返されます。

説明

$cos(\pi x/2)$を計算します。

6.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvTrig2ConstMap

インクルード

```
#include “fsatrig2.h”
```

形式 `unsigned long FSABhvTrig2ConstMap(unsigned long ulAddr)`

引数

(in)*ulAddr* マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvTrig2ConstMap 関数は、三角関数ライブラリ 2 で使用する定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、三角関数ライブラリ 2 の関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

6.6 パフォーマンスおよび要求メモリ

FSA の命令ステップ数と要求メモリを表 6.4 に示します。表中の数値はビヘイビアモデルによる評価値です。実機ではメモリアクセスの競合により、この数値より若干遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

表 6.4 FSA の命令ステップ数および要求メモリ

ライブラリ関数	FSA 命令ステップ数	要求メモリ（バイト）			
		C17 プログラム	FSA プログラム	定数テーブル	FSA スタックメモリ
FsaSin32	35	100	92	32	4
FsaCos32	40	100	100	32	4
FsaSin16	28	100	88	24	4
FsaCos16	29	100	92	24	4

6.7 演算誤差

ランタイムライブラリの sin/cos 関数を真値としたときの誤差の絶対値の最大値を表 6.5 に示します。

表 6.5 誤差の絶対値の最大値

ライブラリ関数	誤差の絶対値の最大値
FsaSin32	5.960×10^{-8}
FsaCos32	5.818×10^{-9}
FsaSin16	6.915×10^{-5}
FsaCos16	6.915×10^{-5}

7. 相互相関係数

7.1 概要

本ライブラリは、下記の定義で示される相互相関係数を計算する関数を提供するものです。このとき、x_i、y_i は共に n 個のデータ列で、$\bar{x}$、$\bar{y}$は各々のデータ列の相加平均です。

$$\frac{\sum_{i=0}^{n}(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum_{i=0}^{n}(x_i-\overline{x})^2}\sqrt{\sum_{i=0}^{n}(y_i-\overline{y})^2}}$$

相関係数（correlation coefficient）とは、２つの確率変数の間の相関（類似性の度合い）を示す統計学的指標である。原則、単位は無く、-1 から 1 の間の実数値をとり、1 に近いときは２つの確率変数には正の相関があるといい、-1 に近ければ負の相関があるという。0 に近いときはもとの確率変数の相関は弱い。因みに 1 もしくは -1 となる場合は ２つの確率変数は線形従属の関係にある。

表 7.1 サポート関数一覧

関数	概仕様
FsaCalcCorrCoef8	8 個のデータ列 x_i、y_i の相関係数を計算
FsaCalcCorrCoef16	16 個のデータ列 x_i、y_i の相関係数を計算
FsaCalcCorrCoef32	32 個のデータ列 x_i、y_i の相関係数を計算
FsaCalcCorrCoef64	64 個のデータ列 x_i、y_i の相関係数を計算
FsaCalcCorrCoef128	128 個のデータ列 x_i、y_i の相関係数を計算
FsaCalcCorrCoef256	256 個のデータ列 x_i、y_i の相関係数を計算

7.2 ファイル構成

表 7.2 ソースファイル一覧

ソースファイル	記述内容
fsacorr.h	ライブラリ関数宣言、コンフィグレーションのためのパラメータ定義
fsacorr.c	ライブラリ関数定義
fsacorrk.h	相互相関を計算する FSA プログラム

7.3 コンフィグレーション

fsacorr.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

実装する関数を選択します。使用する関数の定義だけ残してコメントアウトします。

表 7.3 #define 定義と実装関数

定義	実装関数
CORR_ARRAY_SIZE_8	FsaCalcCorrCoef8
CORR_ARRAY_SIZE_16	FsaCalcCorrCoef16
CORR_ARRAY_SIZE_32	FsaCalcCorrCoef32
CORR_ARRAY_SIZE_64	FsaCalcCorrCoef64
CORR_ARRAY_SIZE_128	FsaCalcCorrCoef128
CORR_ARRAY_SIZE_256	FsaCalcCorrCoef256

fsacorr.h の記述

```
// Specify the size of the input array.
// The function is disabled if it comments out.
// For instance, make CORR_ARRAY_SIZE_8 valid to use FsaCalcCorrCoef8().
//#define CORR_ARRAY_SIZE_8
//#define CORR_ARRAY_SIZE_16
//#define CORR_ARRAY_SIZE_32
//#define CORR_ARRAY_SIZE_64
#define CORR_ARRAY_SIZE_128
//#define CORR_ARRAY_SIZE_256
//--------------------------------------------------------------------------------
```

入力データ列のデータ型指定

入力データ配列のデータ型を short と long から選択できます。short にする場合は、下記の定義を有効にしてください。

fsacorr.h の記述

```
// If the below definition is valid, the input data is short-type.
// If not, it is long-type.
#define CORR_DATA_SHORT
//--------------------------------------------------------------------------------
```

7. 相互相関係数

関数戻り値（相関係数）のデータ型指定

関数戻り値のデータ型を short と long から選択できます。short にする場合は、下記の定義を有効にしてください。

fsacorr.h の記述

```
// If the below definition is valid, the return value is short-type.
// If not, it is long-type.
#define CORR_RETURN_SHORT
//---------------------------------------------------------------------------------
```

正規化処理の指定

入力データの正規化処理を行うか否かを選択できます。正規化を行うと処理サイクル数が増加しますが、相関係数の計算精度が上がります。正規化を行う場合は、下記の定義を有効にしてください。

fsacorr.h の記述

```
// If the below definition is valid, the input data is normalized.
#define CORR_NORMALIZE_ENABLE
//---------------------------------------------------------------------------------
```

7.4 API 関数

FsaCalcCorrCoefXXX

インクルード

```
#include “fsacorr.h”
```

形式

```
short FsaCalcCorrCoe8(FSAREG *pFsaReg, void *pX, void *pY)
short FsaCalcCorrCoe16(FSAREG *pFsaReg, void *pX, void *pY)
short FsaCalcCorrCoe32(FSAREG *pFsaReg, void *pX, void *pY)
short FsaCalcCorrCoe64(FSAREG *pFsaReg, void *pX, void *pY)
short FsaCalcCorrCoe128(FSAREG *pFsaReg, void *pX, void *pY)
short FsaCalcCorrCoe256(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe8(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe16(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe32(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe64(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe128(FSAREG *pFsaReg, void *pX, void *pY)
long FsaCalcCorrCoe256(FSAREG *pFsaReg, void *pX, void *pY)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(in)*pX*	データ列 X（FSA がアクセス可能なメモリ上）のポインタ
(in)*pY*	データ列 Y（FSA がアクセス可能なメモリ上）のポインタ

戻り値

相関係数が返されます。戻り値のデータ型が short 型の場合は Q14 フォーマット、long 型の場合は Q30 フォーマットになります。

説明

FSA がアクセス可能なメモリ上のデータ列 X、Y の相関係数を計算し、戻り値として返します。データ列 X、Y の少なくとも一方がオールゼロの場合、相関係数も 0 になります。

7.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvCorrConstMap

インクルード

```
#include "fsacorr.h"
```

形式
```
unsigned long FSABhvCorrConstMap(unsigned long ulAddr)
```

引数

(in)*ulAddr*　　マッピング仮想アドレス

戻り値

ulAddr +「マッピングされた定数配列のサイズ」のアドレス

説明

FSABhvCorrConstMap 関数は、相互相関係数の計算で使用する定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、相互相関係数ライブラリを使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

7.6 パフォーマンスおよび要求メモリ

FSA の命令ステップ数と要求メモリを表 7.4 および表 7.5 に示します。FSA の実行サイクルは、FSA プログラム、定数テーブル、ワークメモリを各々別バンクの RAM に配置して、ビヘイビアモデルで動作させた場合の評価値です。(同一メモリバンクアクセスによるウェイトが考慮されています。) 但し、実際のライブラリ関数の実行サイクル数は、ホスト CPU（C17 等）が FSA をキックしたり、計算結果を関数の戻り値として出力したりするオーバーヘッドが加わります。

表 7.4 FSA の命令ステップ数および要求メモリ（正規化処理なし）

ライブラリ関数	FSA 実行サイクル(*1)	要求メモリ（バイト）			
		C17 プログラム	FSA プログラム	定数テーブル	FSA スタックメモリ(*2)
FsaCalcCorrCoef8	287	80	292	24	16 x n + 8
FsaCalcCorrCoef16	371	↑	↑	↑	32 x n + 8
FsaCalcCorrCoef32	535	↑	↑	↑	64 x n + 8
FsaCalcCorrCoef64	859	↑	↑	↑	128 x n + 8
FsaCalcCorrCoef128	1503	↑	↑	↑	256 x n + 8
FsaCalcCorrCoef256	2787	↑	↑	↑	512 x n + 8

(*1) データ依存の処理が含まれるため、数サイクル前後します。
(*2) n は入力データ型が short の場合は 2、long の場合は 4 です。FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。なお、入力のデータ列を格納するバッファ領域は別途必要です。

表 7.5 FSA の命令ステップ数および要求メモリ（正規化処理あり）

ライブラリ関数	FSA 実行サイクル(*1)	要求メモリ（バイト）			
		C17 プログラム	FSA プログラム	定数テーブル	FSA スタックメモリ(*2)
FsaCalcCorrCoef8	472	80	392	28	16 x n + 8
FsaCalcCorrCoef16	608	↑	↑	↑	32 x n + 8
FsaCalcCorrCoef32	864	↑	↑	↑	64 x n + 8
FsaCalcCorrCoef64	1384	↑	↑	↑	128 x n + 8
FsaCalcCorrCoef128	2408	↑	↑	↑	256 x n + 8
FsaCalcCorrCoef256	4464	↑	↑	↑	512 x n + 8

8. 対数

8.1 概要

本ライブラリは、底が e、2、10 の対数を計算するライブラリです。入力データの小数点位置は 0～24（Q0～Q24）に対応し、後述するコンフィグレーションによって設定します。出力の小数点位置は 24（Q24）固定です。

表 8.1 サポート関数一覧

関数	概仕様
FsaLog	底が e の対数値を求める関数
FsaLog2	底が 2 の対数値を求める関数
FsaLog10	底が 10 の対数値を求める関数

8.2 ファイル構成

表 8.2 ソースファイル一覧

ソースファイル	記述内容
fsalog.h	ライブラリ関数宣言、およびパラメータ定義
fsalog.c	ライブラリ関数定義

8.3　コンフィグレーション

fsalog.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

入力データフォーマット

本ライブラリ関数の入出力データはいずれも符号付き 32bit ですが、それらデータに対する小数点の位置を 0 から 24 の範囲で指定することが可能です。

fsalog.h の記述例

```
// The fixed point position of the input should be set here.
// From 0 up to 24 can be set for the number.
#define FSALOG_FIXED_POINT                            16
//------------------------------------------------------------------------------
```

8.4 API 関数

FsaLog

インクルード

```
#include "fsalog.h"
```

形式 `long FsaLog(FSAREG *pFsaReg, long x)`

引数
(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 入力データ

戻り値

底が e の対数値を返します。

説明

底が e の入力 x の対数値を計算します。入力値が 0 以下の場合は、0 を出力します。コンフィグレーションにより、入力データに対する固定小数点位置を指定することが可能です。出力データの小数点位置は 24（Q24）固定です。

FsaLog2

インクルード

```
#include "fsalog.h"
```

形式 `long FsaLog2(FSAREG *pFsaReg, long x)`

引数
(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 入力データ

戻り値

底が 2 の対数値を返します。

説明

底が 2 の入力 x の対数値を計算します。入力値が 0 以下の場合は、0 を出力します。コンフィグレーションにより、入力データに対する固定小数点位置を指定することが可能です。出力データの小数点位置は 24（Q24）固定です。

FsaLog10

インクルード

```
#include "fsalog.h"
```

形式 `long FsaLog10(FSAREG *pFsaReg, long x)`

引数
(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 入力データ

戻り値

底が 10 の対数値を返します。

説明

底が 10 の入力 x の対数値を計算します。入力値が 0 以下の場合は、0 を出力します。コンフィグレーションにより、入力データに対する固定小数点位置を指定することが可能です。出力データの小数点位置は 24（Q24）固定です。

8.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvLogConstMap

インクルード

```
#include "fsadspl.h"
```

形式 `unsigned long FSABhvLogConstMap(unsigned long ui32Addr)`

引数

(in)*ui32Addr* マッピング仮想アドレス

戻り値

ui32Addr +「マッピングされた定数配列のサイズ」のアドレスが返される。

説明

FSABhvLogConstMap 関数は、対数の計算に使用する定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、対数ライブラリの関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

8.6 パフォーマンス

表 8.3 のサイクル数は C17、FSA 合わせたサイクル数を表しています。

表 8.3 FsaLog サイクル数

ライブラリ関数	サイクル数
FsaLog	210

8.7 要求メモリ

表 8.4 に要求メモリサイズをまとめます。FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

表 8.4 要求メモリサイズ（単位：byte）

項目	FsaLog
C17 プログラム	216
C17 定数テーブル	1028
FSA プログラム	128
FSA 定数テーブル	104
FSA スタックメモリ	12

8.8 演算精度

本ライブラリ関数の演算精度を表 8.5 に示します。

表 8.5 演算精度

底	演算精度
2	小数点以下 18bit
e	小数点以下 19bit
10	小数点以下 20bit

9. 分散、標準偏差

9.1 概要

本 FSA ライブラリは、指定したデータ配列（short 型 or long 型）から、相加平均、分散、標準偏差を計算する表 9.1 に示す関数を提供します。

表 9.1 サポート関数一覧

関数	概仕様
FsaCalcAve16	short 型データ配列の相加平均を計算します。
FsaCalcVar16	short 型データ配列の相加平均および分散を計算します。
FsaCalcStd16	short 型データ配列の相加平均および標準偏差を計算します。
FsaCalcAve32	long 型データ配列の相加平均を計算します。
FsaCalcVar32	long 型データ配列の相加平均および分散を計算します。
FsaCalcStd32	long 型データ配列の相加平均および標準偏差を計算します。

9.2 ファイル構成

表 9.2 ソースファイル一覧

ソースファイル	記述内容
fsastat.h	ライブラリ関数宣言
fsastat.c	ライブラリ関数定義
fsaleftshift32.c	32 ビット左シフト用定数テーブル

fsaleftshift32.c は、他のライブラリからも使用される共用の定数配列です。本ライブラリには組み込まれませんので、別途ターゲットプロジェクトに追加する必要があります。

9.3 コンフィグレーション

fsastat.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

実装する関数を選択します。使用する関数の定義だけ残してコメントアウトします。

表 9.3 #define 定義と実装関数

定義	実装関数
_FSAAVE16	FsaCalcAve16
_FSAVAR16	FsaCalcVar16
_FSASTD16	FsaCalcStd16
_FSAAVE32	FsaCalcAve32
_FSAVAR32	FsaCalcVar32
_FSASTD32	FsaCalcStd32

fsatrig.h の記述例

```
// Function selection
// Disable the definitions except the function you will use.
#define _FSAAVE16
// #define _FSAVAR16
#define _FSASTD16
// #define _FSAAVE32
// #define _FSAVAR32
// #define _FSASTD32
```

9.4 API 関数

FsaCalcAve16

インクルード

```
#include “fsastat.h”
```

形式

```
short FsaCalcAve16(FSAREG *pFsaReg, short *pi16Data, unsigned short ui16Num)
```

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*pi16Data* 入力データ配列のポインタ
(in)*ui16Num* 入力データ配列の要素数

戻り値

データ配列の相加平均

説明

short 型データ配列の相加平均求めて、戻り値として返します。

ui16Num の指定可能範囲は、1～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaCalcVar16

インクルード

```
#include “fsastat.h”
```

形式

```
short FsaCalcVar16(
        FSAREG *pFsaReg,
        long *pi32Var,
        short *pi16Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(out)*pi32Var* 分散が格納されるポインタ
(in)*pi16Data* 入力データ配列のポインタ
(in)*ui16Num* 入力データ配列の要素数

戻り値

データ配列の相加平均

説明

short 型データ配列の分散を求め、*pi32Var に格納します。戻り値は、FsaCalcAve16 と同様の相加平均が返されます。pi32Var に NULL ポインタを指定すると、分散の格納は行われません。ui16Num の指定可能範囲は、1～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

9. 分散、標準偏差

FsaCalcStd16

インクルード

```
#include “fsastat.h”
```

形式
```
short FsaCalcStd16(
        FSAREG *pFsaReg,
        long *pi32Std,
        short *pi16Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(out)*pi32Std* 標準偏差が格納されるポインタ
(in)*pi16Data* 入力データ配列のポインタ
(in)*ui16Num* 入力データ配列の要素数

戻り値

データ配列の相加平均

説明

short 型データ配列の標準偏差を求め、*pi32Std に格納します。戻り値は、FsaCalcAve16 と同様の相加平均が返されます。pi32Std に NULL ポインタを指定すると、標準偏差の格納は行われません。

ui16Num の指定可能範囲は、1～65535 です。pi16Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaCalcAve32

インクルード

```
#include “fsastat.h”
```

形式
```
long FsaCalcAve32(FSAREG *pFsaReg, long *pi32Data, unsigned short ui16Num)
```

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*pi32Data* 入力データ配列のポインタ
(in)*ui16Num* 入力データ配列の要素数

戻り値

データ配列の相加平均

説明

long 型データ配列の相加平均求めて、戻り値として返します。

ui16Num の指定可能範囲は、1～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaCalcVar32

インクルード

```
#include “fsastat.h”
```

形式

```
long FsaCalcVar32(
        FSAREG *pFsaReg,
        long long *pi64Var,
        long *pi32Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi64Var*	分散が格納されるポインタ
(in)*pi32Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

データ配列の相加平均

説明

long 型データ配列の分散を求め、*pi64Var に格納します。戻り値は、FsaCalcAve32 と同様の相加平均が返されます。pi64Var に NULL ポインタを指定すると、分散の格納は行われません。ui16Num の指定可能範囲は、1～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

FsaCalcStd32

インクルード

```
#include “fsastat.h”
```

形式

```
long FsaCalcStd32(
        FSAREG *pFsaReg,
        long *pi32Std,
        long *pi32Data,
        unsigned short ui16Num
)
```

引数

(in)*pFsaReg*	FSA レジスタ構造体のポインタ
(out)*pi32Std*	標準偏差が格納されるポインタ
(in)*pi32Data*	入力データ配列のポインタ
(in)*ui16Num*	入力データ配列の要素数

戻り値

データ配列の相加平均

説明

long 型データ配列の標準偏差を求め、*pi32Std に格納します。戻り値は、FsaCalcAve32 と同様の相加平均が返されます。pi32Std に NULL ポインタを指定すると、標準偏差の格納は行われません。
ui16Num の指定可能範囲は、1～65535 です。pi32Data は、FSA がアクセス可能なメモリを指定する必要があります。

9.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvLeftShift32ConstMap

インクルード

```
#include “fsastat.h”
```

形式　`unsigned long FSABhvLeftShift32ConstMap(unsigned long ulAddr)`

引数

(in)*ulAddr*　　マッピング仮想アドレス

戻り値

ulAddr +「マッピングした定数配列のサイズ」のアドレス

説明

この関数は、32 ビット左シフト用の定数配列（124 バイト）を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、本ライブラリ関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

9.6 パフォーマンス

入力データ配列の要素数が 300 の場合の FSA の命令ステップ数を表 9.4 に示します。表中の数値はビヘイビアモデルによる評価値です。実機ではメモリアクセスの競合により、この数値より若干遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 9.4　FSA の命令ステップ数

ライブラリ関数	ステップ数
FsaCalcAve16	411
FsaCalcVar16	1125
FsaCalcStd16	1227
FsaCalcAve32	411
FsaCalcVar32	1201
FsaCalcStd32	1522

9.7 要求メモリ

表 9.5、表 9.6 に要求メモリサイズをまとめます。表のサイズは、各々の関数を単独で組み込んだ場合の要求リソースを示しています。

FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。表中の n は入力配列の要素数です。

表 9.5　要求メモリサイズ（単位：byte）

項目	FsaCalcAve16	FsaCalcVar16	FsaCalcStd16
C17 プログラム	126	228	364
FSA プログラム	80	168	216
定数テーブル	124	124	124
FSA スタックメモリ	0	2 x n	2 x n

表 9.6　要求メモリサイズ（単位：byte）

項目	FsaCalcAve32	FsaCalcVar32	FsaCalcStd32
C17 プログラム	124	246	382
FSA プログラム	80	244	320
定数テーブル	124	124	124
FSA スタックメモリ	0	4 x n + 4	4 x n + 4

10. 平方根、逆数平方根

10.1 概要

本 FSA ライブラリは、逆数平方根および平方根を高速に演算する以下の関数を提供します。これら関数は、後述するコンフィグレーションによって、使用の可否を選択することができます。

なお、平方根はある与えられた値 x に対して $x = a^2$ となるような a を指し、$\sqrt{x}$ で表すのに対して、その$\sqrt{x}$ の逆数を逆数平方根と呼び、$1/\sqrt{x}$ で表します。。

表 10.1　FSA の命令ステップ数

関数	概仕様
FsaSqrtInv	逆数平方根を求める関数
FsaSqrt	平方根を求める関数
FsaSumSqrt16	short 型データの二乗和平方根を求める関数
FsaSumSqrt32	long 型データの二乗和平方根を求める関数

10.2 ファイル構成

表 10.2　ソースファイル一覧

ソースファイル	記述内容
fsasqrt.h	ライブラリ関数宣言、およびパラメータ定義
fsasqrt.c	ライブラリ関数定義
fsaleftshift32.c	32 ビット左シフト用定数テーブル

fsaleftshift32.c は、他のライブラリからも使用される共用の定数配列です。本ライブラリには組み込まれませんので、別途ターゲットプロジェクトに追加する必要があります。

10.3 コンフィグレーション

fsasqrt.h の#define 定義を変更することで、用途に応じたコンフィグレーションが可能です。

実装関数の選択

本ライブラリでサポートする関数を選択します。必要な定義だけ残してコメントアウトします。

表 10.3 #define 定義と実装関数

定義	実装関数
_FSASQRTINV	FsaSqrtInv
_FSASQRT	FsaSqrt
_FSASUMSQRT16	FsaSumSqrt16
_FSASUMSQRT32	FsaSumSqrt32

fsasqrt.h の記述例

```
// Make necessary function only valid in the following definitions.
#define _FSASQRTINV
#define _FSASQRT
// #define _FSASUMSQRT16
// #define _FSASUMSQRT32
```

FsaSqrtInv 関数の入力データフォーマット

FsaSqrtInv 関数の入力データの固定小数点位置を指定します。0 から 30 まで指定することが可能です。ただし、固定小数点位置は必ず偶数を指定する必要があります。

fsasqrt.h の記述例

```
// _FSASQRTINV_FIXED_POINT means the number of bit of the decimal point in the
// input of FsaSqrtInv function.
// The smaller the number is, the accuracy is higher.
// 0 - 30 is selectable. But the number must be even.
#define _FSASQRTINV_FIXED_POINT                                  0
```

FsaSqrt 関数の出力データフォーマット

FsaSqrt 関数の入力データが整数値と仮定した場合の出力データの小数点以下ビット数を指定します。0 から 15 まで指定することが可能です。小数点以下ビット数が多い方が、より高精度な平方根の算出ができますが、処理サイクルは増加します。

fsasqrt.h の記述例

```
// _FSASQRT_FIXED_POINT means the number of bit of the decimal point in the output
// of FsaSqrt function.
// The bigger the number is, the accuracy is higher. But the required cycle
// increases. 0 - 15 is selectable.
#define _FSASQRT_FIXED_POINT                                     15
```

FsaSqrt 関数の計算アルゴリズムの選択

FsaSqrt 関数において、ニュートンラプソン法を用いて平方根を計算する場合は、下記の定義を有効にします。ニュートンラプソン法を用いた方がより高速ですが、プログラムサイズは大きくなります。演算精度は、ニュートンラプソン法を用いた場合とそうでない場合で、大きな違いはありません。

fsasqrt.h の記述例

```
// Calculating algorithm selection of FsaSqrt function.
// When the following definition is valid, Newton-Raphson method is used.
// Newton-Raphson method gives higher performance. But the code size is bigger.
#define _FSASQRT_NEWTON_RAPHSON
```

10.4 API 関数

FsaSqrtInv

インクルード

```
#include "fsasqrt.h"
```

形式 `long FsaSqrtInv(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 入力データ

戻り値

逆数平方根 1/√x の演算結果を返します。入力が 1 未満の場合は、1.0（0x40000000）を返します。

説明

ニュートンラプソン法により、x の逆数平方根 1/√x を求め、戻り値として返します。戻り値は小数点以下 30 ビットの Q30 フォーマットです。x の固定小数点位置は、コンフィグレーションの章の説明にあるように、0〜30 の範囲（但し、偶数値）で指定することができます。

FsaSqrt

インクルード

```
#include "fsasqrt.h"
```

形式 `long FsaSqrt(FSAREG *pFsaReg, long x)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*x* 入力データ

戻り値

平方根√x の演算結果を返します。入力が 0 以下の場合は、0 を返します。

説明

x の平方根√x を求め、戻り値として返します。戻り値の固定小数点位置は、コンフィグレーションの章の説明にあるように、x が整数値と仮定した場合の小数点以下のビット数を 0〜15 の範囲で指定することができます。

FsaSumSqrt16

インクルード

```
#include “fsasqrt.h”
```

形式 `long FsaSumSqrt16(FSAREG *pFsaReg, short *pi16Data, unsigned short ui16Num)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*pi16Data* 入力データ x_i のポインタ
(in)*ui16Num* 入力データの要素数 *n*

戻り値

下記に示す二乗和平方根を返します。

$$\sqrt{\sum_{i=0}^{n-1} x_i^2}$$

説明

pi16Data で与えられた要素数 ui16Num 個の short 型データ配列の二乗和平方根を求め、戻り値として返します。ui16Num の指定可能範囲は、1～256 です。

FsaSumSqrt32

インクルード

```
#include “fsasqrt.h”
```

形式 `long FsaSumSqrt32(FSAREG *pFsaReg, long *pi32Data, unsigned short ui16Num)`

引数

(in)*pFsaReg* FSA レジスタ構造体のポインタ
(in)*pi32Data* 入力データ x_i のポインタ
(in)*ui16Num* 入力データの要素数 *n*

戻り値

下記に示す二乗和平方根を返します。

$$\sqrt{\sum_{i=0}^{n-1} x_i^2}$$

説明

pi32Data で与えられた要素数 ui16Num 個の long 型データ配列の二乗和平方根を求め、戻り値として返します。ui16Num の指定可能範囲は、1～256 です。

10.5 ビヘイビアモデルで使用する API 関数

この章で説明する API 関数は、FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いて、PC 上で FSA プログラムをエミュレーションする場合に必要なメモリ割り当て用の関数です。

実機では、メモリ割り当てはリンカにより解決されますので、これらの API 関数を実装する必要はありません。また、実機用のオブジェクトには、これら関数は組み込まれません。

FSABhvSqrtConstMap

インクルード

`#include “fsasqrt.h”`

形式 `unsigned long FSABhvSqrtConstMap(unsigned long ui32Addr)`

引数

(in)*uni32Addr* マッピング仮想アドレス

戻り値

uni32Addr +「マッピングした定数配列のサイズ」のアドレス

説明

FSABhvSqrtConstMap 関数は、平方根、逆数平方根の計算に使用する定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、FsaSqrt 関数、FsaSqrtInv 関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに定数配列をマッピングします。

FSABhvLeftShift32ConstMap

インクルード

`#include “fsasqrt.h”`

形式 `unsigned long FSABhvLeftShift32ConstMap(unsigned long ulAddr)`

引数

(in)*ulAddr* マッピング仮想アドレス

戻り値

ulAddr +「マッピングした定数配列のサイズ」のアドレス

説明

この関数は、ニュートンラプソン法を使用しない FsaSqrt 関数および FsaSumSqrt16、FsaSumSqrt32 関数で使用する 32 ビット左シフトの定数配列を、ビヘイビアモデル上の指定した仮想アドレスにマッピングする関数です。FSA ビヘイビアモデルライブラリ（fsabhv.lib）を用いた PC によるエミュレーション環境で、上述した関数を使用する場合は、予めこの関数を用いてビヘイビアモデル上の仮想アドレスに 32 ビット左シフトの定数配列をマッピングします。

10.6 パフォーマンス

FSA の命令ステップ数を表 10.4 に示します。表中の数値はビヘイビアモデルによる評価値です。実機ではメモリアクセスの競合により、この数値より若干遅くなる場合もあります。また、この表に示した処理サイクル数とは別に、ホスト CPU が FSA をキックする処理サイクルが若干加算されます。

表 10.4 FSA の命令ステップ数

ライブラリ関数	ステップ数
FsaSqrtInv	78 ~ 130(*1)
FsaSqrt （ニュートンラプソン法）	81 ~ 136(*1)
FsaSqrt	100 ~ 191(*2)
FsaSumSqrt16	469(*3)
FsaSumSqrt32	578(*3)

(*1) 命令ステップ数は、入力値に応じて変化します。
(*2) 命令ステップ数は、出力値の固定小数点位置設定に応じて変化します。小数点以下ビット数が大きいほど、命令ステップ数は増加します。
(*3) 入力データ配列の要素数が 256 のときの命令ステップ数です。

10.7 要求メモリ

表 10.5 に要求メモリサイズをまとめます。FSA スタックメモリとは、FSA が一時的な作業領域として使用するメモリです。具体的には、FSA がアクセス可能なメモリの最上位の領域が使用されます。

表 10.5 要求メモリサイズ（単位：byte）

項目	FsaSqrtInv	FsaSqrt (ニュートン法)	FsaSqrt	FsaSumSqrt16	FsaSumSqrt32
FSA プログラム	176	188	44	88	84
C17 プログラム	120	110	100	112	112
定数テーブル	24	28	0(*4)	0(*4)	0(*4)
FSA スタックメモリ	4	4	0	8	8

(*4) 左シフトのための汎用定数テーブル（124 バイト）が別途必要です。

改訂履歴表

付－1

Rev. No.	日付	ページ	種別	改訂内容（旧内容を含む） および改訂理由
Rev 1.0	2015/01/09	全ページ	新規	新規制定

セイコーエプソン株式会社

マイクロデバイス事業部　デバイス営業部

東京　〒191-8501　東京都日野市日野 421-8

TEL（042）587-5313（直通）　FAX（042）587-5116

大阪　〒541-0059　大阪市中央区博労町 3-5-1　エプソン大阪ビル 15F

TEL（06）6120-6000（代表）　FAX（06）6120-6100

ドキュメントコード：412863600

2015 年 1 月 作成Ⓥ