お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

SuperH RISC engine C/C++ コンパイラパッケージ

アプリケーションノート：<コンパイラ活用ガイド>DSP 活用 編

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラ V.9 における SH-DSP コア (SH2-DSP, SH3-DSP，SH4AL-DSP，SH-MOBILE シリーズ) 向け DSP 記述の使用方法、注意点を説明します。

目次

1. SH-DSP について

SH-DSP コアは 16 ビット固定小数点を扱う、
積和演算
繰り返し処理
に最適な DSP ユニットを搭載しているので、マルチメディア演算に必要な JPEG 処理、音声処理、フィルタ処理などを高速に実行することができます。

従来の SH コア（**図** 1.1 は SH-1 コアの例）では、パイプラインのとおり、乗算器の動作時間の 3 サイクルが積和演算の性能を決定してしまいます。また、仮に乗算器の動作時間を 1 サイクルに改善しても命令データ転送によるパイプラインのストールがおこり、長期平均時間は 2.5 サイクルになってしまいます。
SH-DSP コアでは、DSP ユニットの動作時間の 1 サイクル化とデータバス用に XY のバスを設けたことにより、積和演算の 1 サイクル化を実現しています（**図** 1.2）。この場合、長期平均時間も 1 サイクルとなります。

コード例

```
clrmac
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
rts
```

IF **：命令フェッチ**(32 ビット)
if **：命令フェッチ (バスサイクルなし)**
ID **：デコード**
EX **：実行／アドレス計算**
MA **：メモリアクセス**
mul **：乗算器動作**
WB **：ライトバック**

パイプライン動作例

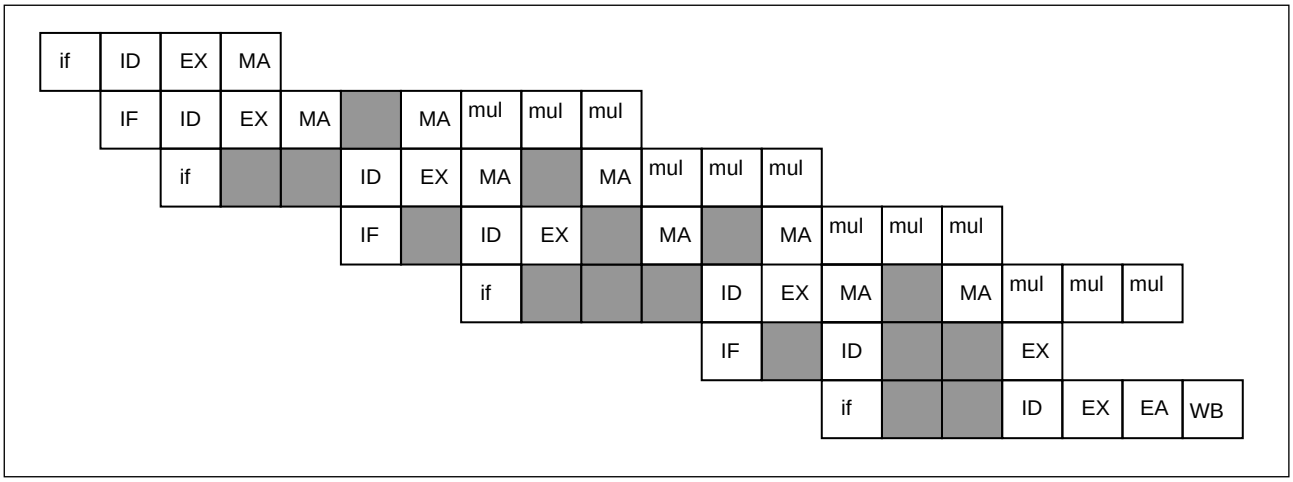
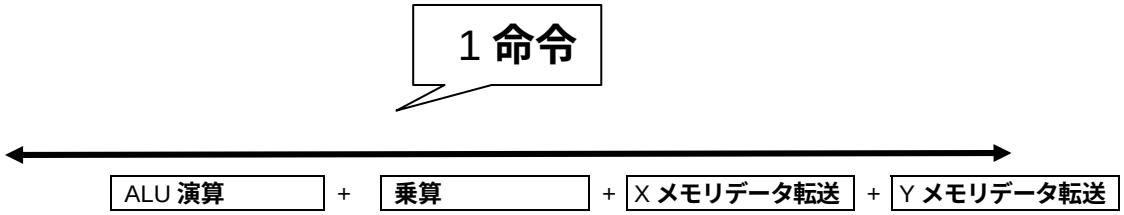


図 1.1　SH **コアの積和命令**



コード例

命令 1			MOVX.W@R4+,X0	MOVY.W@R6+,Y0
命令 2		PMULSX0,Y0,M0	MOVX.W@R4+,X1	MOVY.W@R6+,Y1
命令 3	PADD A0,M0,A0	PMULSX1,Y1,M1	MOVX.W@R4+,X0	MOVYW@R6+,Y0
命令 4	PADD A0,M1,A0	PMULS X0,Y0,M0	MOVX.W@R4+,X1	MOVYW@R6+,Y1

パイプライン動作例

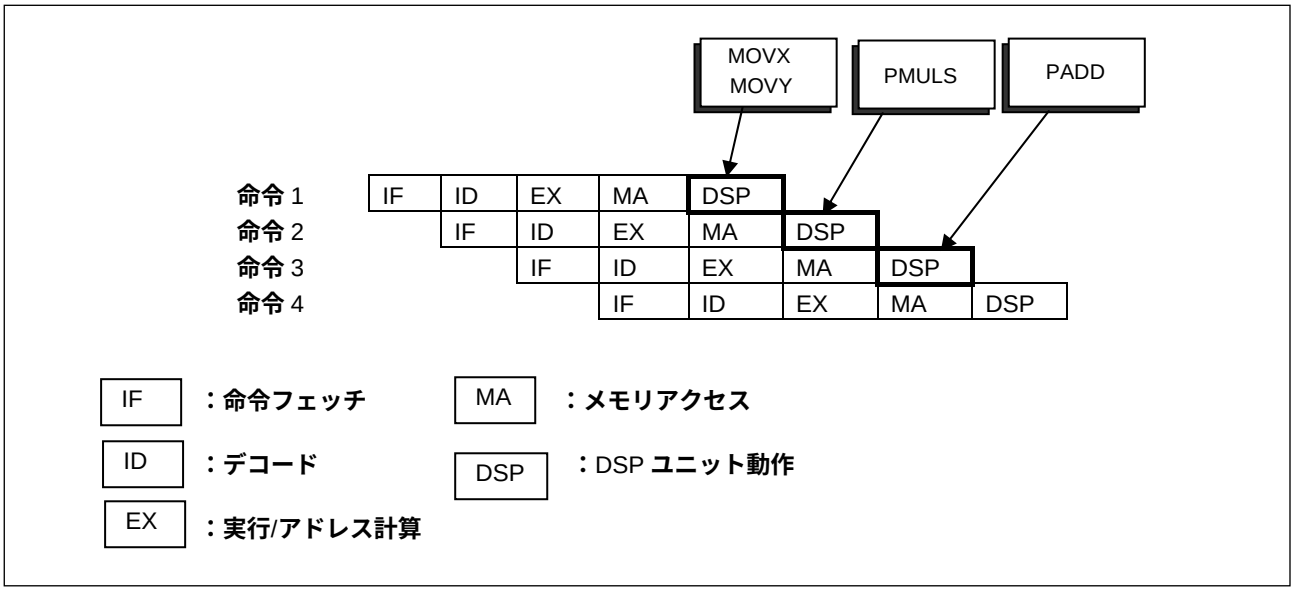


図 1.2　SH-DSP **コアの積和命令の例**

また、SH-DSP コアでは、繰り返し処理によるパイプラインの乱れを低減するためのハードウェア機構が搭載されています。

従来の SH コアでは、ループ処理には条件分岐命令を使用します。条件分岐命令はパイプラインの乱れを発生させ、処理にオーバヘッドをだしてしまいます。

SH-DSP コアでは、このループ処理によるパイプラインの乱れをゼロにするゼロオーバヘッドの機構があります。あらかじめ、ループの開始終了のアドレスとループ回数をセットしておくだけで、条件分岐の処理を行わずにループ処理が完了します。ソフト的にクリティカルな処理はループ処理になっている場合が多く、ソフト処理の高速化に有効なハードウェア機構です。

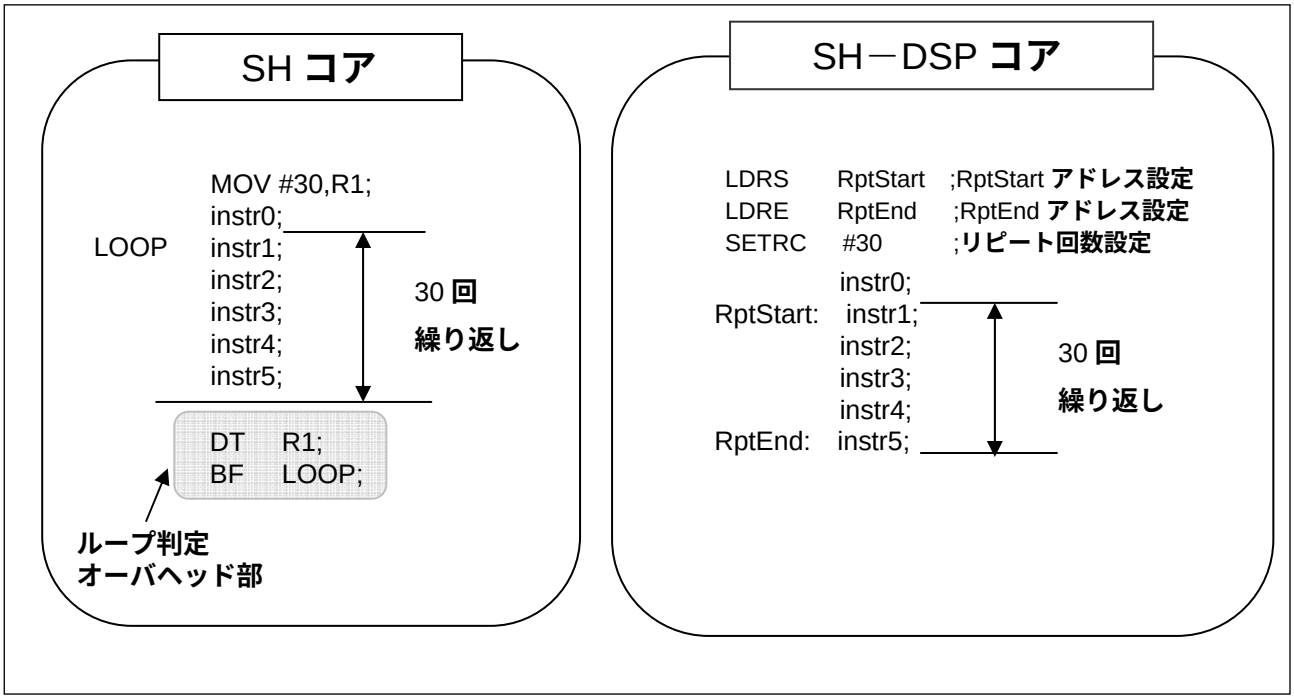


図 1.3　繰り返し処理

SH-DSP コアは、**図** 1.4 に示すように実行条件の判定、ALU 演算、符号付き乗算、X メモリアクセス、Y メモリアクセスの 5 つの命令を並行して実行することができます。これらの命令を組み合わせることによって、さまざまな積和演算の処理を高速に行うことができるわけです。

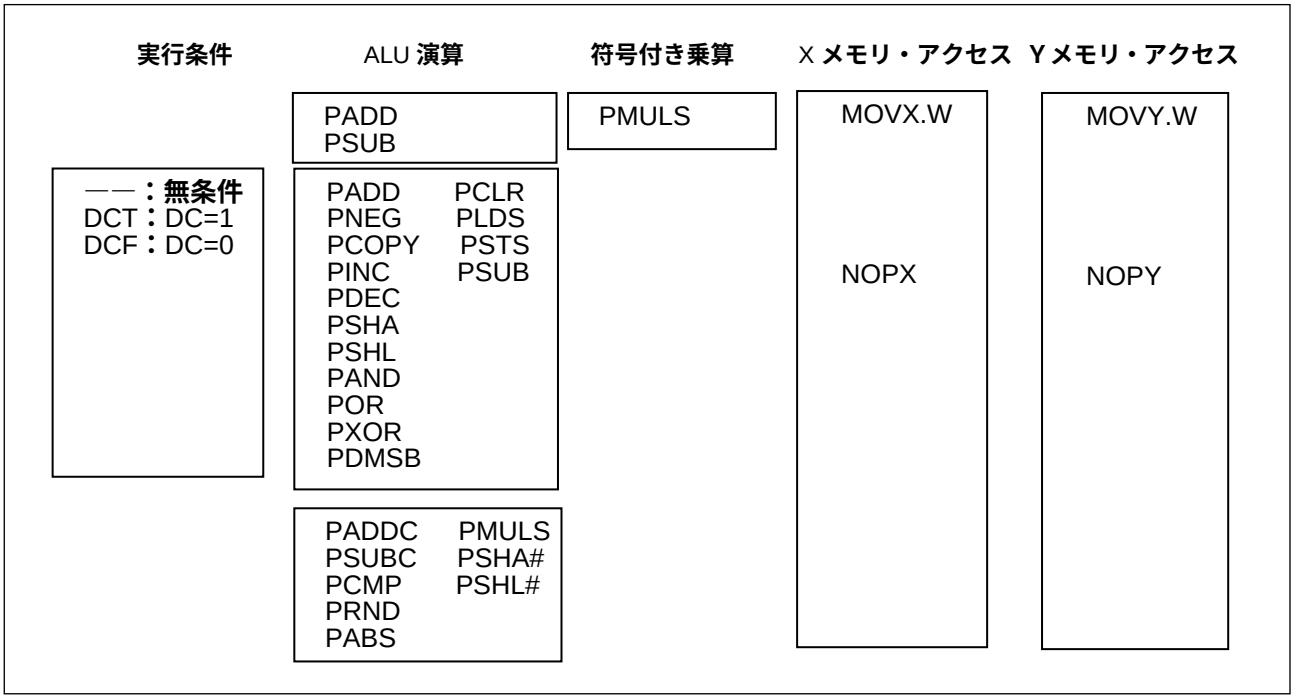


図 1.4　DSP **命令（並列命令）**

2. DSP ライブラリ

2.1 概要

SH2-DSP、SH3-DSP、および SH4AL-DSP(以降、併せて単に SH-DSP と呼ぶこととします)で使用できるデジタル信号処理(DSP)ライブラリについて説明します。本ライブラリは標準的な DSP 関数を含んでおり、単独または連続的に使用することによって、DSP 演算を行うことができます。

本ライブラリは以下の関数を含んでいます。

- 高速フーリエ変換
- 窓関数
- フィルタ
- 畳み込みと相関
- その他

本ライブラリ関数は高速フーリエ変換とフィルタを除いてリエントラントです。

本ライブラリを使用するときは、**表** 2.1 に示すファイルをインクルードしてください。また、**表** 2.2 に示すように、CPU およびコンパイルオプションに対応するライブラリをリンクしてください。

本ライブラリを呼び出した際、関数が正常に終了した場合は EDSP_OK を、異常があった場合は EDSP_BAD_ARG もしくは EDSP_NO_HEAP をリターン値として返します。リターン値の詳細については各関数の説明を参照してください。

表 2.1 DSP **ライブラリ用のインクルードファイル**

ライブラリの種類	**内容**	**インクルードファイル**
DSP **ライブラリ**	DSP **演算を行うライブラリです。**	<ensigdsp.h>
		<filt_ws.h>*1

【注】 *1 **フィルタ関数を使用する場合、ユーザプログラム内で１回のみインクルードしてください。**

表 2.2 DSP **ライブラリー覧**

CPU	**オプション**		**ライブラリ名**
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP SH4AL-DSP	-pic=0	-endian=big	sh3dspnb.lib
	-pic=1	-endian=big	sh3dsppb.lib
	-pic=0	-endian=little	sh3dspnl.lib
	-pic=1	-endian=little	sh3dsppl.lib

2.1.1 データフォーマット

本ライブラリはデータを符号付き 16 ビット固定小数点数として扱います。符号付き 16 ビット固定小数点数は**図** 2.1(a)に示すように、小数点が最上位ビット(MSB)の右側に固定されたデータフォーマットとなっており、$-1 \sim 1-2^{-15}$ の範囲の値を表現できます。

本ライブラリでは、データの受け渡しは short 型のデータフォーマットを使用します。したがって、C/C++プログラムから本ライブラリを使用する場合、データを符号付き 16 ビット固定小数点数で表現する必要があります。

(例) +0.5は符号付き16ビット固定小数点数で表現するとH'4000です。したがってライブラリ関数に渡すshort型実引数はH'4000となります。

本ライブラリ内部の演算では、符号付き 32 ビット固定小数点数と符号付き 40 ビット固定小数点数も使用します。符号付き 32 ビット固定小数点数は**図** 2.1(b)に示すデータフォーマットとなっており、$-1 \sim 1-2^{-31}$ の範囲の値を表現できます。符号付き 40 ビット固定小数点数は**図** 2.1(c)に示すように 8 ビットのガードビットが付加されたデータフォーマットとなっており、$-2^{8} \sim 2^{8}-2^{-31}$ の範囲の値を表現できます。

符号付き 16 ビット固定小数点数の乗算結果は符号付き 32 ビット固定小数点数で保持します。DSP 命令を用いた固定小数点乗算では、H'8000×H'8000 の場合だけオーバフローが発生することに注意してください。また乗算結果の最下位ビット(LSB)は常に 0 になります。乗算結果を次の演算に使用する場合、上位 16 ビットを取り出し、符号付き 16 ビット固定小数点数に変換します。このときアンダフローや精度低下が発生する可能性があります。

本ライブラリの積和演算では、加算結果を符号付き 40 ビット固定小数点数で保持します。加算のときにオーバフローが発生しないように注意してください。

演算の際、オーバフローが発生すると正しい結果が得られません。オーバフローを防ぐためには、係数や入力データをスケーリングする必要があります。本ライブラリには、スケーリングの機能が組み込まれています。スケーリングの詳細については各関数の説明を参照してください。

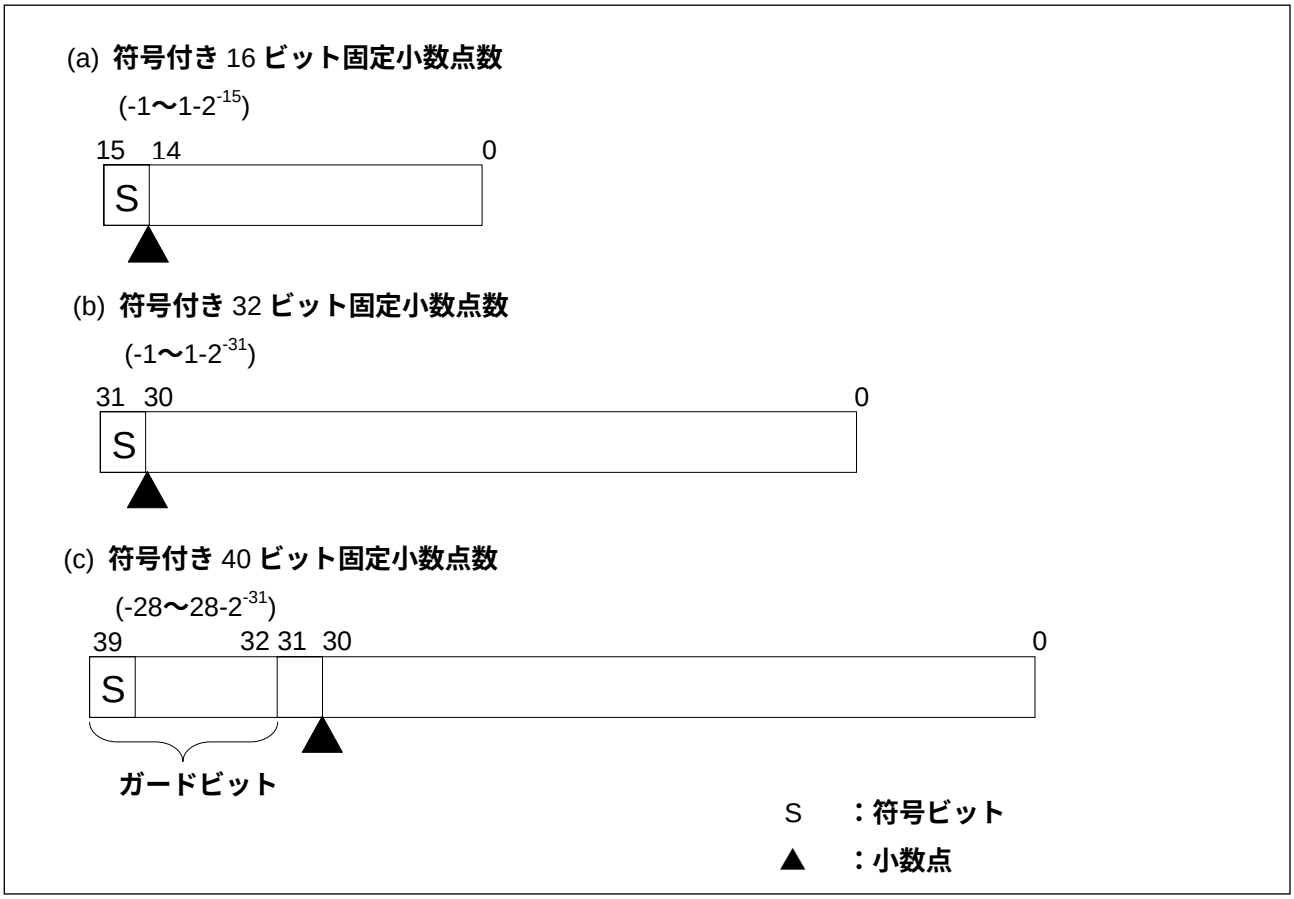


図 2.1 **データフォーマット**

2.1.2 効率

本ライブラリ関数は、SH-DSP 上で高速に実行するように最適化しています。

ライブラリを効率よく活用するために、開発するシステムのメモリマップを決める際には、できるだけ以下の 2 つの推奨事項に従ってください。

プログラムコードセグメントは、1サイクルでの32ビットリードをサポートしているメモリに配置する。

データセグメントは、1サイクルでの16(または32)ビットリード・ライトをサポートしているメモリに配置する。

使用するマイコンが、ライブラリコードとデータを配置するのに十分な容量の 32 ビットメモリを内蔵している場合は、その 32 ビットメモリに配置するのが最適です。その他のメモリを使用しなければならない場合は、可能な限り上記の推奨事項に従ってください。

2.2 DSP ライブラリ関数の詳細

2.2.1 高速フーリエ変換

(1) 関数一覧

表 2.3 DSP ライブラリ関数一覧(高速フーリエ変換)

項番	項目	関数名	説明
1	not-in-place **複素数** FFT	FftComplex	not-in-place **複素数** FFT **を実行します。**
2	not-in-place **実数** FFT	FftReal	not-in-place **実数** FFT **を実行します。**
3	not-in-place **複素数逆** FFT	IfftComplex	not-in-place **複素数逆** FFT **を実行します。**
4	not-in-place **実数逆** FFT	IfftReal	not-in-place **実数逆** FFT **を実行します。**
5	in-place **複素数** FFT	FftInComplex	in-place **複素数** FFT **を実行します。**
6	in-place **実数** FFT	FftInReal	in-place **実数** FFT **を実行します。**
7	in-place **複素数逆** FFT	IfftInComplex	in-place **複素数逆** FFT **を実行します。**
8	in-place **実数逆** FFT	IfftInReal	in-place **実数逆** FFT **を実行します。**
9	**対数絶対値**	LogMagnitude	**複素数データを対数絶対値に変換します。**
10	FFT **回転係数生成**	InitFft	FFT **回転係数を生成します。**
11	FFT **回転係数解放**	FreeFft	FFT **回転係数の格納に使用したメモリを解放します。**

【注】 not-in-place、in-place **については「**(5) FFT **構造」を参照してください。**

これらの関数は、ユーザが定義したスケーリングを使って、順方向高速フーリエ変換と逆方向高速フーリエ変換を実行します。

順方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{-2j\pi n/N}\cdot x_n$$

ここで、s はスケーリングが行われるステージの数、N はデータ数を示しています。

逆方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{2j\pi n/N}\cdot x_n$$

スケーリングについては**「**(4) **スケーリング」**を参照してください。

(2) 複素数データ配列フォーマット

FFT および IFFT の複素数データ配列は、実数を X メモリに、虚数を Y メモリに配置します。ただし、実数 FFT の出力データと実数 IFFT の入力データの配置は異なっています。実数、虚数を格納する配列をそれぞれ x,y とすると、x[0]には DC 成分の実数成分が入り、y[0]には DC 成分の虚数成分ではなく Fs/2 成分の実数成分が入ります(DC 成分と Fs/2 成分はどちらも実数で、虚数成分は 0 です)。

(3) 実数データ配列フォーマット

FFT および IFFT の実数データ配列フォーマットには、以下の 3 種類があります。

単一の配列に格納し、任意のメモリブロックに配置。

単一の配列に格納し、Xメモリに配置。

2つの配列に分けて格納。それぞれの配列のサイズはN/2で、配列の前半はXメモリに配置し、後半はYメモリに配置。

FftReal は 1 番目の指定方法のみです。IfftReal、FftInReal および IfftInReal は 2 番目か 3 番目の方法をユーザが選択します。

(4) スケーリング

基数 2 の FFT は各ステージで信号強度が倍になり、ピーク信号振幅も倍になります。そのため、高強度信号を変換する際にオーバフローが発生することがありますが、各ステージで信号を 1/2 にすることにより(これをスケーリングといいます)オーバフローを防ぐことができます。しかし、スケーリングしすぎると不要な量子化雑音が発生する可能性があります。

オーバフローや量子化雑音とスケーリングの最適なバランスは入力信号の特性に大きく依存します。スペクトルが大きなピークを持つ信号はオーバフローを防ぐために最大スケーリングが必要になりますが、インパルス信号ではスケーリングの必要はほとんどありません。

すべてのステージでスケーリングするのが最も安全な方法です。入力データが強度 2^{30} 未満であれば、この方法でオーバフローを防ぐことができます。本ライブラリでは、各ステージごとにスケーリングを行うかどうかを指定できます。したがって、スケーリング指定を精密に行うことによって、オーバフローと量子化雑音の影響を最小限に抑えることができます。

スケーリングの方法を指定するために、各 FFT 関数の引数に scale が含まれています。scale は最下位ビットから 1 ビットずつが各ステージに対応しています。対応する scale のビットが 1 に設定されているすべてのステージで、2 の除算を実行します。

本ライブラリは実行速度を上げるために基数 4 の FFT を使用しています。scale は最下位ビットから 2 ビットずつが各ステージに対応しています。どちらか 1 ビットが 1 に設定されていれば、2 の除算を実行します。両方が 1 に設定されていれば 4 の除算を実行します。つまり、2 つの基数 2 の FFT ステージが 1 つの基数 4 の FFT ステージに置き換えられたのと同じことになります。しかし、基数 2 の FFT よりも基数 4 の FFT の方が量子化雑音の発生する可能性があります。

以下に scale の例を示します。

scale = H’FFFFFFFF(またはsize-1)はすべての基数2のFFTステージでスケーリングを行います。すべての入力データの強度が2^{30}未満であれば、オーバフローは発生しません。

scale = H’55555555は1つおきの基数2のFFTステージでスケーリングを行います。

scale = 0はスケーリングを行いません。

これらの scale の値は、ensigdsp.h で EFFTALLSCALE(H’FFFFFFFF)、EFFTMIDSCALE(H’55555555)、EFFTNOSCALE(0)と定義されています。

(5) FFT 構造

本ライブラリの FFT 構造には not-in-place FFT と in-place FFT の 2 種類があります。

not-in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM のユーザが指定した別の場所に格納します。

一方 in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM の同じ場所に格納します。この方法を用いると FFT の実行時間は増加しますが、使用メモリスペースが削減できます。

入力データを FFT 関数のほかにも使用する場合は、not-in-place FFT を使用してください。また、メモリスペースを節約したい場合は、in-place FFT を使用してください。

(6) 各関数の説明

(a) not-in-place 複素数 FFT

■説明

【書式】
```
int FftComplex (short op_x[], short op_y[], const short  ip_x[],
                const short ip_y[], long size, long scale)
```

【引数】

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 複素数高速フーリエ変換を実行します。

【備考】 本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define MAX_FFT_SAMP  64
#define MIN_CFFT_SIZE   4
long ip_scale=0xffffffff;
long size = MIN_CFFT_SIZE;


#pragma section X
short ip_x[MAX_FFT_SAMP];
short op_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
short op_y[MAX_FFT_SAMP];
#pragma section

/* サイクル数測定用データ */
#define TWOPI 6.283185307 /* data */

void main()
{
    int i,j;
    long n_samp;

    n_samp=MAX_FFT_SAMP; /* data */

    for (j = 0; j < n_samp; j++){
         ip_x[j] = cos(j * TWOPI/n_samp) * 8188;
         ip_y[j] = sin(j * TWOPI/n_samp) * 8188;
  }
    if(InitFft(n_samp) != EDSP_OK){
       printf("Initfft != err end");
    }
    if(FftComplex(op_x,op_y,ip_x,ip_y,n_samp,EFFTALLSCALE) != EDSP_OK){
       printf("FftComplex error¥n");
    }
    FreeFft();

    for(i=0;i<n_samp;i++){
       printf("[%d] op_x=%d  op_y=%d  ¥n",i,op_x[i],op_y[i]);
    }
}
```

インクルードヘッダ

X **メモリ** Y **メモリに配置する変数は** #pragma section **でセクション内に定義します。**

FFT**用データ作成**

FFT **の初期化関数データ数で初期化を行います。必須です。データ数は** FFT **のデータサイズと同じで** 2 **のべき乗であることが必要です。**

FFT **計算に使用したテーブルの** free **を行います。**
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT **を行うのであれば、**
FreeFft **を行　わずにそのまま** FFT **の関数を使用します。**

(b) not-in-place 実数 FFT

■説明

【書式】
```
int FftReal (short op_x[], short op_y[], const short ip[],
             long size, long scale)
```

【引数】

op_x[]	正の出力データの実数成分
op_y[]	正の出力データの虚数成分
ip[]	実数入力データ
size	FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 実数高速フーリエ変換を実行します。

【備考】 op_x と op_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_S/2$ での出力データの値は実数なので、$F_S/2$ での実数出力は op_y[0]に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define VLEN  64
#define TWOPI 6.28318530717959

/* global data declarations */
#pragma section X
short output_x[VLEN];
#pragma section Y
short output_y[VLEN];
#pragma section

void main()
{
    short i;
    int k;
    short input[VLEN];
    short output[VLEN];

/* generate two sinusoids */
    k = VLEN / 8;
    for (i = 0; i < VLEN; i++)
       input[i] = floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);
    k = VLEN * 3 / 8;
    for (i = 0; i < VLEN; i++)
        input[i] += floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);

/* do FFT */
    if (InitFft(VLEN) != EDSP_OK)
        printf("InitFft problem¥n");
    if (FftReal(output_x, output_y, input, VLEN, EFFTALLSCALE) != EDSP_OK)
        printf("FftReal problem¥n");
    FreeFft();
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は
#pragma section でセクション内に定義します。

FFT 用データ作成部

FFT の初期化関数
データ数で初期化を行います。必須です。
データ数は FFT のデータサイズと同じで
2 のべき乗であることが必要です。

FFT 計算に使用したテーブルの
free を行います。
これを行わないとメモリを無駄に
使用してしまいます。
次に同じデータ数で FFT を行うの
であれば、FreeFft を行わずに
そのまま FFT の関数を使用します。

(c) not-in-place 複素数逆 FFT

■説明

【書式】
```
int IfftComplex (short op_x[], short op_y[], const short ip_x[],
                 const short ip_y[], long size, long scale)
```

【引数】

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 複素数逆高速フーリエ変換を実行します。

【備考】 本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];        /* input array */
short opi_x[MAX_IFFT_SIZE];        /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
            opi_x[j]=0;
            opi_y[j]=0;
        }
        if(IfftComplex(opi_x, opi_y, ipi_x, ipi_y, max_size,
                        EFFTALLSCALE)!= EDSP_OK){
          printf("IfftComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
          printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
        }
        FreeFft();
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部（FftComplex を実行するためのデータです。）

FFT の初期化関数
データ数で初期化を行います。
必須です。データ数は FFT のデータサイズと同じで 2 のべき乗であることが必要です。
逆 FFT 計算にも必要な処理です。

いったん FFT 計算を行って
その結果を逆 FFT 関数の入力値と
するための処理なので通常は不要です。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。

(d) not-in-place 実数逆 FFT

■説明

【書式】
```
int IfftReal (short op_x[], short scratch_y[], const short ip_x[],
              const short ip_y[], long size, long scale,
              int op_all_x)
```

【引数】

op_x[]	実数出力データ
scratch_y[]	スクラッチメモリまたは実数出力データ
ip_x[]	正の入力データの実数成分
ip_y[]	正の入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

【内容】 実数逆高速フーリエ変換を実行します。

【備考】 ip_x と ip_y には size/2 の正の入力データを格納してください。負の入力データは正の入力データの共役複素数です。また、0 と $F_S/2$ での入力データの値は実数なので、$F_S/2$ での実数入力は ip_y[0] に格納してください。
出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは op_x に格納されます。op_all_x=0 の場合、最初の size/2 の出力データは op_x に格納され、残りの size/2 の出力データは scratch_y に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
ip_x、ip_y はそれぞれ size/2 のデータを格納してください。op_x は op_all_x の値によって、size または size/2 のデータが格納されます。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];         /* input array */
short opi_x[MAX_IFFT_SIZE];         /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }

    if (InitFft(max_size) != EDSP_OK){
        printf("InitFft error end ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }

        if(IfftReal(opi_x, opi_y, ipi_x, ipi_y, max_size,
                                                EFFTALLSCALE,1)!= EDSP_OK){
            printf("IfftReal err end  ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
        }

        FreeFft();
    }
}
```

インクルードヘッダ

X **メモリ** Y **メモリに配置する変数は** #pragma section **でセクション内に定義します。**

FFT **用データ作成部**
(FftReal **を計算するためのデータです。)**

FFT **の初期化関数**
データ数で初期化を行います。
必須です。データ数は FFT **の**
データサイズと同じで
2 **のべき乗であること。**
逆 FFT **にも必要です。**

いったん FFT **計算を行ってその結果を逆** FFT **関数の入力値とするための処理なので通常は不要です。**

FFT **計算に使用したテーブルの** free **を行います。**
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT **を行うのであれば、**
FreeFft **を行わずにそのまま** FFT **の関数を使用します。**

(e) in-place 複素数 FFT

■説明

【書式】 `int FftInComplex (short data_x[], short data_y[], long size, long scale)`

【引数】

`data_x[]`	入出力データの実数成分
`data_y[]`	入出力データの虚数成分
`size`	FFT のサイズ
`scale`	スケーリング指定

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`size < 4` ・`size` が 2 の累乗ではありません ・`size > max_fft_size`

【内容】 in-place 複素数高速フーリエ変換を実行します。

【備考】 複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に `InitFft` を呼び出して、回転係数と `max_fft_size` を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
`scale` は下位 $\log_2$(`size`)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=0xffffffff;

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section

void main()
{
    int i,j;
    long max_size;
    long n_samp;

    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = sin(j * TWOPI/n_samp) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }
    if(FftInComplex(ip_x, ip_y, n_samp,EFFTALLSCALE ) != EDSP_OK){
        printf("FftInComplex error¥n");
    }
    FreeFft();

    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部

FFT の初期化関数
データ数で初期化を行います。必須です。
データ数は FFT のデータサイズと同じで 2 のべき乗でなければなりません。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。

(f) in-place 実数 FFT

■説明

【書式】
```
int FftInReal (short data_x[], short data_y[], long size,
               long scale, int ip_all_x)
```

【引数】

data_x[]	入力時は実数データ、出力時は正の出力データの実数成分
data_y[]	入力時は実数データまたは未使用、出力時は正の出力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定
ip_all_x	入力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・ip_all_x ≠ 0 または 1

【内容】 in-place 実数高速フーリエ変換を実行します。

【備考】 入力データのフォーマットは、ip_all_x で指定します。ip_all_x=1 の場合、
全入力データは data_x から取り出します。ip_all_x=0 の場合、前半の size/2 の入力データは data_x から、後半の size/2 の入力データは data_y から取り出します。
本関数実行後、data_x と data_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_S/2$ での出力データの値は実数なので、$F_S/2$ での実数出力は data_y[0] に格納されます。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
data_y は size/2 のデータを格納します。data_x は ip_all_x の値によって size または size/2 のデータを格納します。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;
/*long ip_scale=0xffffffff;*/

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section


void main()
{
    int i,j;
    long max_size;
    long n_samp;
    int ip_all_x;

    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = 0;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }

    ip_all_x = 1;
    if(FftInReal(ip_x, ip_y, n_samp,EFFTALLSCALE ,ip_all_x) != EDSP_OK){
        printf("FftInReal error¥n");
    }

    FreeFft();

    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部

FFT の初期化関数
データ数で初期化を行います。必須です。
データ数は FFT のデータサイズと同じで 2 のべき乗である必要があります。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。

(g) in-place 複素数逆 FFT

■説明

【書式】
```
int IfftInComplex (short data_x[], short data_y[], long size,
                   long scale)
```

【引数】

data_x[]	入出力データの実数成分
data_y[]	入出力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 in-place 複素数逆高速フーリエ変換を実行します。

【備考】 複素数データ配列の配置については「(2) 複素数データ配列フォーマット」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];        /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }

    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end ¥n");
        }

        if(IfftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("IfftInComplex err  end ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
        }

       FreeFft();
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部（FftInComplex の入力値となるデータ）

FFT の初期化関数
データ数で初期化を行います。必須です。
データ数は FFT のデータサイズと同じで 2 のべき乗でなければなりません。
逆 FFT にも必要です。

いったん FFT 計算を行ってその結果を逆 FFT 関数の入力値とするための処理なので通常は不要です。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。

(h) in-place 実数逆 FFT

■説明

【書式】
```
int IfftInReal (short data_x[], short data_y[],
                long size, long scale, int op_all_x)
```

【引数】

data_x[]	入力時は正の入力データの実数成分、出力時は実数データ
data_y[]	入力時は正の入力データの虚数成分、出力時は実数データまたは未使用
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

【内容】 in-place 実数逆高速フーリエ変換を実行します。

【備考】 data_x と data_y には size/2 の正の入力データを格納してください。負の入力データは正の入力データの共役複素数です。また、0 と $F_S/2$ での入力データの値は実数なので、$F_S/2$ での実数入力は data_y[0]に格納してください。
出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは data_x に格納されます。op_all_x=0 の場合、前半の size/2 の出力データは data_x に格納され、後半の size/2 の出力データは data_y に格納されます。
複素数と実数データ配列の配置については**「(2) 複素数データ配列フォーマット」「(3) 実数データ配列フォーマット」**を参照してください。
data_y は size/2 のデータを格納します。data_x は、op_all_x の値によって size または size/2 のデータが格納されます。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については**「(4) スケーリング」**を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];        /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }

    if(InitFft(max_size) != EDSP_OK){
          printf("InitFft error end ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end ¥n");
        }
        if(IfftInReal(ipi_x, ipi_y, max_size, EFFTALLSCALE,1) != EDSP_OK){
            printf("IfftInReal err  end ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
        }

        FreeFft();
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部（FftInReal 関数の入力値として使用します。）

FFT の初期化関数
データ数で初期化を行います。必須です。
データ数は FFT のデータサイズと同じで 2 のべき乗である必要があります。
逆 FFT にも必要です。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。

(i) 対数絶対値

■説明

【書式】 `int LogMagnitude (short output[], const short ip_x[], const short ip_y[], long no_elements, float fscale)`

【引数】

output[]	実数出力 z
ip_x[]	入力の実数成分 x
ip_y[]	入力の虚数成分 y
no_elements	出力データ数 N
fscale	出力スケーリング係数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・no_elements > 32767 ・$\lvert fscale \rvert \geq 2^{15}/(10\log_{10}2^{31})$

【内容】 複素数入力データの対数絶対値をデシベル単位で計算し、スケーリング結果を出力配列に書き込みます。

【備考】 $z(n)=10fscale \cdot \log_{10}(x(n)^2+y(n)^2) \quad 0 \leq n < N$

複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];        /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;
    short output[MAX_IFFT_SIZE];

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }

    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }
        if(LogMagnitude(output, ipi_x,ipi_y, max_size/2, 2) != EDSP_OK){
            printf("LogMagnitude err  end  ¥n");
        }
        for (j = 0; j < max_size/2; j++){
          printf("[%d]  output=%d  ¥n",j, output[j]);
        }

        FreeFft();
    }

}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

FFT 用データ作成部

FFT の関数 LogMagnitude 関数で使用するデータを作成します。

FFT 計算に使用したテーブルの free を行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数で FFT を行うのであれば、
FreeFft を行わずにそのまま FFT の関数を使用します。
LogMagnitude には直接関係ありません。

(j) 回転係数生成

■説明

【書式】 `int InitFft (long max_size)`

【引数】

`max_size`	必要になる FFT の最大サイズ

【戻り値】

`EDSP_OK`	成功
`EDSP_NO_HEAP`	malloc で確保できるメモリスペースが不十分
`EDSP_BAD_ARG`	以下のいずれかの場合です ・max_size < 2 ・max_size が 2 の累乗ではありません ・max_size > 32768

【内容】 FFT 関数で使用する回転係数（1/4 サイズ）を生成します。

【備考】 回転係数は malloc によって確保されるメモリに格納されます。
回転係数が生成されると max_fft_size グローバル変数が更新されます。max_fft_size は FFT の最大許容サイズを示します。
本関数は最初の FFT 関数を呼び出す前に必ず一度呼び出してください。
max_size は 8 以上としてください。
回転係数は max_size で指定した変換サイズで生成されます。max_size より小さいサイズの FFT 関数を実行したときも同じ回転係数を使用します。
回転係数のアドレスは内部変数内に格納されています。ここはユーザプログラムでアクセスしないでください。
本関数はリエントラントではありません。

(k) 回転係数開放

■説明

【書式】 `void FreeFft (void)`

【引数】 なし

【戻り値】 なし

【内容】 回転係数の格納に使用したメモリを解放します。

【備考】 max_fft_size グローバル変数を 0 にします。FreeFft を実行した後再び FFT 関数を実行するときには、その前に必ず InitFft を実行してください。
本関数はリエントラントではありません。

2.2.2 窓関数

(1) 関数一覧

表 2.4 DSP ライブラリ関数一覧(窓関数)

項番	項目	関数名	説明
1	ブラックマン窓	GenBlackman	ブラックマン窓を生成します。
2	ハミング窓	GenHamming	ハミング窓を生成します。
3	ハニング窓	GenHanning	ハニング窓を生成します。
4	三角窓	GenTriangle	三角窓を生成します。

(2) 各関数の説明

(a) ブラックマン窓

■説明

【書式】 `int GenBlackman (short output[], long win_size)`

【引数】
`output[]`	出力データ `W(n)`
`win_size`	窓サイズ `N`

【戻り値】
`EDSP_OK`	成功
`EDSP_BAD_ARG`	`win_size` ≤ 1

【内容】 ブラックマン窓を生成し、`output` に出力します。

【備考】 実際のデータにこの窓をかけるときは `VectorMult` を使用します。
使用する関数を以下に示します。

$$W(n) = (2^{15} - 1)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \qquad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>          } インクルードヘッダ
#define MAXN    10

void main()
{
   int i;
    long len;
    short output[MAXN];

    len=MAXN ;
    if(GenBlackman(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
   for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(b) ハミング窓

■説明

【書式】 int GenHamming (short output[], long win_size)

【引数】

output[]	出力データ W(n)	
win_size	窓サイズ N	

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

【内容】 ハミング窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = (2^{15} - 1)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define MAXN  10

void main()
{

    int i;
    long len;
   short output[MAXN];

    len=MAXN ;
    if(GenHamming(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

インクルードヘッダ

(c) ハニング窓

■説明

【書式】 `int GenHanning (short output[], long win_size)`

【引数】

`output[]`	出力データ `W(n)`
`win_size`	窓サイズ `N`

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`win_size` ≤ 1

【内容】 ハニング窓を生成し、`output` に出力します。

【備考】 実際のデータにこの窓をかけるときは `VectorMult` を使用します。
使用する関数を以下に示します。

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \qquad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
```

インクルードヘッダ

```
#define MAXN  10

void main()
{
    int i;
   long len;
    short output[MAXN];

    len=MAXN ;
    if(GenHanning(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }

}
```

(d) 三角窓

■説明

【書式】 int GenTriangle (short output[], long win_size)

【引数】

output[]		出力データ W(n)
win_size		窓サイズ N

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

【内容】 三角窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \qquad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
```
} インクルードヘッダ

```
#define MAXN    10
void main()
{
   int i;
    long len;
    short output[MAXN];

    len=MAXN ;
    if(GenTriangle(output, len) != EDSP_OK){
       printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
       printf("output=%d ¥n",output[i]);
    }
}
```

2.2.3 フィルタ

(1) 関数一覧

表 2.5 DSP ライブラリ関数一覧(フィルタ)

項番	項目	関数名	説明
1	FIR	Fir	有限インパルス応答フィルタ処理を実行します。
2	単一データ用 FIR	Fir1	単一データ用有限インパルス応答フィルタ処理を実行します。
3	IIR	Iir	無限インパルス応答フィルタ処理を実行します。
4	単一データ用 IIR	Iir1	単一データ用無限インパルス応答フィルタ処理を実行します。
5	倍精度 IIR	Diir	倍精度無限インパルス応答フィルタ処理を実行します。
6	単一データ用倍精度 IIR	Diir1	単一データ用倍精度無限インパルス応答フィルタ処理を実行します。
7	適応 FIR	Lms	適応 FIR フィルタ処理を実行します。
8	単一データ用適応 FIR	Lms1	単一データ用適応 FIR フィルタ処理を実行します。
9	FIR 作業領域割り付け	InitFir	FIR フィルタ用に作業領域を割り付けます。
10	IIR 作業領域割り付け	InitIir	IIR フィルタ用に作業領域を割り付けます。
11	倍精度 IIR 作業領域割り付け	InitDIir	DIIR フィルタ用に作業領域を割り付けます。
12	適応 FIR 作業領域割り付け	InitLms	LMS フィルタ用に作業領域を割り付けます。
13	FIR 作業領域解放	FreeFir	InitFir で割り付けられた作業領域を解放します。
14	IIR 作業領域解放	FreeIir	InitIir で割り付けられた作業領域を解放します。
15	倍精度 IIR 作業領域解放	FreeDIir	InitDIir で割り付けられた作業領域を解放します。
16	適応 FIR 作業領域解放	FreeLms	InitLms で割り付けられた作業領域を解放します。

【注】 本関数のいずれかを使用する場合、ユーザプログラム内で 1 回のみ filt_ws.h をインクルードしてください。

(2) 係数のスケーリング

フィルタ処理を行うと飽和、または量子化雑音が発生する可能性があります。これらはフィルタ係数のスケーリングを行うことによって最小限に抑えることができます。しかし、飽和と量子化雑音の影響をよく考えてスケーリングを行わなければなりません。係数が大きすぎると飽和が、小さすぎると量子化雑音が発生する可能性があります。

FIR(有限インパルス応答)フィルタの場合、以下の式が成り立つようにフィルタ係数を設定すれば飽和は起こりません。

coeff[i] ≠ H'8000 (すべての i について)

$\sum|coeff| < 2^{24}$

res_shift = 24

coeff はフィルタ係数、res_shift は出力で行われる右シフトのビット数です。

しかし、多くの入力信号の場合、もっと小さい res_shift の値(またはもっと大きな coeff の値)を使用しても飽和する可能性は少なく、量子化雑音も大幅に削減できます。また入力値に H'8000 が含まれている可能性があれば、すべての coeff の値は H'8001～H'7FFF の範囲になるように設定してください。

IIR(無限インパルス応答)フィルタは再帰的な構造になっています。そのため上述したようなスケーリング方法は適していません。

LMS(最小 2 乗平均)適応フィルタは FIR フィルタと同様です。しかし、係数を適応するときに飽和を引き起こす場合があります。その場合は、係数に H'8000 を含まないように設定してください。

(3) 作業領域

デジタルフィルタでは、ある処理から次の処理へ保持しておかなければならない情報があります。これらの情報は、最小オーバーヘッドでアクセスすることができるメモリに格納します。本ライブラリでは、Y-RAM 領域を作業領域として使用します。作業領域はフィルタ処理を実行する前に Init 関数を呼び出して初期化してください。

作業領域メモリはライブラリ関数によってアクセスされます。なお、ユーザプログラムから作業領域を直接アクセスしないでください。

(4) メモリの使用

SH-DSP を効率よく使うために、フィルタ係数は X メモリに配置してください。入出力データは任意のメモリセグメントに配置することができます。

フィルタ係数は#pragma section 命令を用いて X メモリに配置してください。

各フィルタは Init 関数を用いてグローバルバッファから作業領域を割り付けます。グローバルバッファは Y メモリに配置します。

(5) 各関数の説明

(a) FIR

■説明

【書式】
```
int Fir (short output[], const short input[], long no_samples,
         const short coeff[], long no_coeffs, int res_shift,
         short *workspace)
```

【引数】

output[]	出力データ y
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数 h
no_coeffs	係数の数(フィルタの長さ)K
res_shift	各出力に適用される右シフト
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 有限インパルス応答(FIR)フィルタ処理を実行します。

【備考】 最新の入力データは作業領域に保持されます。input をフィルタ処理した結果は output に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。
本関数を呼び出す前に InitFir を呼び出し、フィルタの作業領域を初期化してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define NFN 8 /* number of functions */
#define FIL_COUNT   32  /* データ数 */
#define N   32

#pragma section X
static short coeff_x[FIL_COUNT];
#pragma section

short data[FIL_COUNT] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,};

short coeff[8] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000,};

void main()
{
    short *work, i;
    short output[N];
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* 係数設定 */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = NFN;/* 係数の数設定 */
    nsamp = FIL_COUNT;/* サンプル数設定 */
    rshift = 12;

    if (InitFir(&work, ncoeff) != EDSP_OK){
        printf("Init Problem¥n");
    }
    if(Fir(output, data, nsamp, coeff_x, ncoeff, rshift, work) != EDSP_OK){
        printf("Fir Problem¥n");
    }
    if (FreeFir(&work, ncoeff) != EDSP_OK){
        printf("Free Problem¥n");
    }
    for(i=0;i<nsamp;i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }

}
```

インクルードヘッダ

フィルタ係数を X メモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
Fir 関数発行の前に必須です。
Y メモリをワークエリアとして（係数の数）*2+8 バイト使用します。

Fir 計算に使用したワークエリアの解放。
Fir 使用後は必ず行います。
この関数を発行しないと、メモリが無駄に使用されます。

(b) **単一データ用 FIR**

■説明

【書式】 `int Fir1 (short *output, short input, const short coeff[], long no_coeffs, int res_shift, short *workspace)`

【引数】

`output`	出力データ `y(n)` へのポインタ
`input`	入力データ `x(n)`
`coeff[]`	フィルタ係数 `h`
`no_coeffs`	係数の数(フィルタの長さ)`K`
`res_shift`	各出力に適用される右シフト
`workspace`	作業領域へのポインタ

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_coeffs ≤ 2` ・`res_shift < 0` ・`res_shift > 25`

【内容】 単一データ用に有限インパルス応答(FIR)フィルタ処理を実行します。

【備考】 最新の入力データは作業領域に保持されます。`input` をフィルタ処理した結果は`*output` に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 `y(n)` は `res_shift` ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。
関数を呼び出す前に `InitFir` を呼び出し、フィルタの作業領域を初期化してください。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define NFN 8  /* number of functions */
#define MAXSH 25
#define N   32

#pragma section X
static short coeff_x[NFN];
#pragma section

short data[32] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[8] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

void main()
{
    short *work, i;
    short output[N];
    int ncoeff, rshift;

    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* 係数設定 */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    rshift = 12;
    ncoeff = NFN;/* 係数の数設定 */

    if (InitFir(&work, NFN) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++) {
        if(Fir1(&output[i], data[i], coeff_x, ncoeff, rshift, work) !=
    EDSP_OK){
            printf("Fir1 Problem¥n");
        }
        printf(" output[%d]=%d ¥n",i,output[i]);
    }
    if (FreeFir(&work, NFN) != EDSP_OK){
        printf("Free Problem¥n");
    }
}
```

インクルードヘッダ

フィルタ係数を X メモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
Fir1 関数発行の前に必須です。
Y メモリをワークエリアとして（係数の数）*2+8
バイト使用します。

Fir1 **はデータ数 1 の** Fir **関数と同じです。**
Fir1 **を複数回発行する場合には、**InitFir **と** FreeFir **は最初と最後の 1 回ずつ発行してください。**

(c) IIR

■説明

【書式】 `int Iir (short output[], const short input[], long no_samples, const short coeff[], long no_sections, short *workspace)`

【引数】

output[]	出力データ y_{K-1}
input[]	入力データ x_0
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・a_{0k} < 0 ・a_{0k} > 16

【内容】 無限インパルス応答(IIR)フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。

各バイカッドの出力は以下の式で与えられます。

$d_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)] \cdot 2^{-15}$

$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)] \cdot 2^{-a0k}$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}$ ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。

output に input と同じ配列を指定した場合、input は上書きされます。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K  4
#define NUMCOEF  (6*K)
#define N 50

#pragma section X
static short coeff_x[NUMCOEF];
#pragma section

static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};


static short input[50] = {32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000 };

void main()
{
    short *work, i;
    short output[N];

    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
    }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
   if (Iir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

インクルードヘッダ

フィルタ係数を X メモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

フィルタ係数は 6 個を 1 セクションとして設定してください、セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Iir 関数発行の前に必須です。
Y メモリをワークエリアとして
((フィルタセクションの数) *2*2)
バイト使用します。

Iir 計算に使用したワークエリアの解放。
Iir 使用後は必ず行ってください。
この関数を発行しないと、メモリが無駄に使用されてしまいます。

(d) **単一データ用 IIR**

■説明

【書式】 `int Iir1 (short *output, short input, const short coeff[], long no_sections, short *workspace)`

【引数】

output	出力データ $y_{K-1}(n)$ へのポインタ
input	入力データ $x_0(n)$
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・a_{ok} < 0 ・a_{ok} > 16

【内容】 単一データ用に無限インパルス応答(IIR)フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。

各バイカッドの出力は以下の式で与えられます。

$$d_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)] \cdot 2^{-15}$$

$$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)] \cdot 2^{-a0k}$$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K   4
#define NUMCOEF  (6*K)
#define N   50

#pragma section X
static short coeff_x[NUMCOEF];
#pragma section

static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000 };

short keisu[5]={ 1,2,20,4,5 };

void main()
{
    short *work, i;
    short output[N];

    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
    }

    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++){
        if (Iir1(&output[i], input[i], coeff_x, K, work) != EDSP_OK){
            printf("EDSP_OK not returned¥n");
        }
        printf("output[%d]:%d ¥n" ,i,output[i]);
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem¥n");
    }
}
```

インクルードヘッダ

フィルタ係数を X メモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

フィルタ係数は 6 個を 1 セクションとして設定し、セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Iir1 関数発行の前に必須です。
Y メモリをワークエリアとして（フィルタセクションの数）*2*2 バイト使用します。

Iir1 はデータ数 1 の Iir 関数と同じです。
Iir1 を複数回発行する場合には、InitIir と FreeIir は最初と最後の 1 回ずつ発行します。

(e) 倍精度 IIR

■説明

【書式】
```
int DIir (short output[], const short input[], long no_samples,
          const long coeff[], long no_sections, long *workspace)
```

【引数】

output[]	出力データ y_{K-1}
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・a_{0k} < 3 ・k < K-1 で a_{0k} > 32 ・k = K-1 で a_{0k} > 48

【内容】 倍精度無限インパルス応答フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。
各バイカッドの出力は、以下の方程式で与えられます。

$$d_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{29}x(n)] \cdot 2^{-31}$$

$$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^{2}$$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)の入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。
coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。
DIir は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは正または負の最大値となります。最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。
本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。
遅延ノード $d_k(n)$ は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。
DIir は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k < K-1 のときは 31、k=K-1 のときは 47 に設定してください。
DIir より Iir の方が実行速度は速いので、倍精度計算の必要がなければ Iir を使用してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <filt_ws.h>
#include <ensigdsp.h>

#define K  5
#define NUMCOEF  (6*K)
#define N  50

#pragma section X
static long coeff_x[NUMCOEF];
#pragma section

static long coeff[60] =
      {31,1254686956, -496866304, 347415747, 694831502, 347415746,
       31,-113001278,-1523568505, 893094203,1786188388, 893094206,
       31,1254686956, -496866304, 347415747, 694831502, 347415746,
       31,-113001278,-1523568505, 893094203,1786188388, 893094206,
       47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[100] = {
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000,
                        32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;
    long nsamp;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work,K) != EDSP_OK){
        printf("InitDIir Problem¥n");
    }
    if(DIir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("DIir Problem¥n");
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("FreeDIir Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

インクルードヘッダ

フィルタ係数を X メモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

フィルタ係数は 6 個を 1 セクションとして設定します。セクションの先頭要素は右シフト数でフィルタ係数ではありません。

右シフト数は
最後のセクション以外 31。
最後のセクションのみ 47。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Dlir 関数発行の前に必須です。
Y メモリをワークエリアとして（フィルタセクションの数）*4*2
バイト使用します。

Dlir 計算に使用したワークエリアの解放。
Dlir 使用後は必ず行ってください。
この関数を発行しないと、メモリが無駄に使用されてしまいます。

(f) 単一データ用倍精度 IIR

■説明

【書式】
```
int DIir1  (short output[], const short input[], long no_samples,
        const long coeff[], long no_sections, long *workspace)
```

【引数】

output	出力データ $y_{K-1}(n)$ へのポインタ
input	入力データ $x_0(n)$
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・a_{0k} < 3 ・k < K-1 で a_{0k} > 32 ・k = K-1 で a_{0k} > 48

【内容】 単一データ用に倍精度無限インパルス応答フィルタ処理を実行します。

【備考】 フィルタはバイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。

各バイカッドの出力は、以下の方程式で与えられます。

$d_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{29}x(n)] \cdot 2^{-31}$

$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^2$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)への入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション(k=K-1)からの出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} \ldots b_{2K-1}$

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

DIir1 は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir1 と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは、正または負の最大値となります。最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。

遅延ノード $d_k(n)$ は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。

DIir1 は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k < K-1 のときは 31、k=K-1 のときは 47 に設定してください。

DIir1 より Iir1 の方が実行速度は速いので、倍精度計算の必要がなければ Iir1 を使用してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K  5
#define NUMCOEF  (6*K)
#define N   50

#pragma section X
static long coeff_x[NUMCOEF];
#pragma section

static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[N]   = {32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];

    if(InitDIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++){
        if(DIir1(&output[i], input[i], coeff_x, K, work) !=EDSP_OK){
            printf("DIir1 error¥n");
        }
        printf("output[%d]:%d ¥n" ,i,output[i]);
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("Free DIir error¥n");
    }
}
```

インクルードヘッダ

フィルタ係数は 6 個を 1 セクションとして設定します。セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数を X メモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているので Y メモリは使用しないでください。

右シフト数は
最後のセクション以外 31。
最後のセクションのみ 47。

フィルタ係数を X メモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Dlir1 関数発行の前に必須です。
Y メモリをワークエリアとして（フィルタセクションの数）*4*2 バイト使用します。

Dlir1 はデータ数 1 の Dlir 関数と同じです。
Dlir1 を複数回発行する場合には、InitDlir と FreeDlir は最初と最後の 1 回ずつで良いです。

(g) 適応 FIR

■説明

【書式】

```
int Lms  (short output[], const short input[],
          const short ref_output[], long no_samples,
          short coeff[], long no_coeffs, int res_shift,
          short conv_fact, short  *workspace)
```

【引数】

output[]	出力データ y
input[]	入力データ x
ref_output[]	所望の出力値 d
no_samples	入力データの数 N

	coeff[]	適応フィルタ係数 h
	no_coeffs	係数の数 K
	res_shift	各出力に適用される右シフト
	conv_fact	収束係数 2μ
	workspace	作業領域へのポインタ
【戻り値】	EDSP_OK	成功
	EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 最小 2 乗平均アルゴリズム(LMS)を使って、実数適応 FIR フィルタ処理を実行します。

【備考】 FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\, x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

フィルタ係数の更新は、Widrow-Hoff アルゴリズムを使用します。

$h_{n+1}(k) = h_n(k) + 2\mu e(n) x(n-k)$

ここで e(n) は所望する信号と実際の出力の誤差です。

$e(n) = d(n) - y(n)$

2μe(n)x(n-k) の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算結果とも上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。

係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。

conv_fact は通常正に設定してください。また H'8000 には設定しないでください。

本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。

output に input または ref_output と同じ配列を指定した場合、input または ref_output は上書きされます。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[K] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                        -10781, 239, 13655, 11202, 2180,
                        -687, -2883, -7315, -6527, 196,
                        4278, 3712, 3367, 4101, 2703,
                        591, 695, -1061, -5626, -4200,
                        3585, 9285, 11796, 13416, 12994,
                        10231, 5803, -449, -6782, -11131,
                        -10376, -2968, 2588, -1241, -6133};

void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
```

インクルードヘッダ

フィルタ係数をXメモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

フィルタ係数をXメモリ上の変数に設定します。

```
 for (i = 0; i < N; output[i++] = 0) ;
 ncoeff = K;/* 係数の数設定 */
 nsamp = N;/* サンプル数設定 */


for (i = 0; i < K; i++){
       coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
       printf("Init Problem¥n");
    }
    if(Lms(output, data, ref, nsamp, coeff_x, ncoeff, RSHIFT,TWOMU, work) !=
                                                          EDSP_OK){
       printf("Lms Problem¥n");
    }
    if (FreeLms(&work, K) != EDSP_OK){
       printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
       printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
LMS 関数発行の前に必須です。
Y メモリをワークエリアとして
（係数の数）*2+8
バイト使用します。

LMS **計算に使用したワークエリアの解放。**
LMS **使用後は必ず行ってください。**
この関数を発行しないと、
メモリが無駄に使用されてしまいます。

(h) 単一データ用適応 FIR

■説明

【書式】
```
int Lms1  (short *output, short input, short ref_output,
           short coeff[],long no_coeffs, int res_shift,
           short conv_fact, short *workspace)
```

【引数】

output	出力データ y(n) へのポインタ
input	入力データ x(n)
ref_output	所望の出力値 d(n)
coeff[]	適応フィルタ係数 h
no_coeffs	係数の数 K
res_shift	各出力に適用される右シフト
conv_fact	収束係数 2μ
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 最小 2 乗平均アルゴリズム (LMS) を使って、単一データ用に実数適応 FIR フィルタ処理を実行します。

【備考】 FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

フィルタ係数の更新は Widrow-Hoff アルゴリズムを使用します。

$h_{n+1}(k) = h_n(k) + 2\mu e(n) x(n-k)$

ここで e(n) は所望する信号と実際の出力の誤差です。

$e(n) = d(n) - y(n)$

$2\mu e(n) x(n-k)$ の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算とも、上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。

係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。

conv_fact は通常正に設定してください。また H'8000 には設定しないでください。

本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];
#pragma section

short data[N] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[K] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                        -10781, 239, 13655, 11202, 2180,
                        -687, -2883, -7315, -6527, 196,
                        4278, 3712, 3367, 4101, 2703,
                        591, 695, -1061, -5626, -4200,
                        3585, 9285, 11796, 13416, 12994,
                        10231, 5803, -449, -6782, -11131,
                        -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;
     ncoeff = K;/* 係数の数設定 */
    nsamp = N;/* サンプル数設定 */
```

インクルードヘッダ

X メモリに配置する変数は #pragma section で セクション内に定義します。

フィルタ係数を X メモリ上の変数に設定します。

```
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<nsamp;i++){
        if(Lms1(&output[i], data[i], ref[i], coeff_x, ncoeff, RSHIFT, TWOMU,
                                                        work) != EDSP_OK){
            printf("Lms1 Problem¥n");
       }
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
LMS1 関数発行の前に必須です。
Y メモリをワークエリアとして
（係数の数）*2+8
バイト使用します。

LMS1 計算に使用したワークエリアの解放。
LMS1 使用後は必ず行ってください。
この関数を発行しないと、メモリが無駄に使用されてしまいます。

(i) FIR 作業領域割り付け

■説明

【書式】 int InitFir (short **workspace, long no_coeffs)

【引数】

workspace	作業領域へのポインタへのポインタ
no_coeffs	係数の数 K

【戻り値】

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_coeffs ≤ 2

【内容】 Fir と Fir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitFir で割り付けられた作業領域を操作することができます。
ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(j) IIR 作業領域割り付け

■説明

【内容】 int InitIir (short **workspace, long no_sections)

【引数】

workspace	作業領域へのポインタへのポインタ
no_sections	2 次フィルタセクションの数 K

【戻り値】

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_sections < 1

【内容】 Iir と Iir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Iir と Iir1 だけが InitIir で割り付けられた作業領域を操作することができます。ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(k) 倍精度 IIR 作業領域割り付け

■説明

【書式】 int InitDIir (long **workspace, long no_sections)

【引数】

workspace	作業領域へのポインタへのポインタ
no_sections	2 次フィルタセクションの数 K

【戻り値】

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_sections < 1

【内容】 DIir と DIir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
DIir と DIir1 だけが InitDIir で割り付けられた作業領域を操作することができます。
本関数はリエントラントではありません。

(l) 適応 FIR 作業領域割り付け

■説明

【書式】 `int InitLms (short **workspace, long no_coeffs)`

【引数】

`workspace`		作業領域へのポインタへのポインタ
`no_coeffs`		係数の数 K

【戻り値】

`EDSP_OK`	成功
`EDSP_NO_HEAP`	workspace の使用できるメモリスペースが不十分
`EDSP_BAD_ARG`	no_coeffs ≤ 2

【内容】 Lms と Lms1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitLms で割り付けられた作業領域を操作することができます。
ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(m) FIR 作業領域開放

■説明

【書式】 `int FreeFir (short **workspace, long no_coeffs)`

【引数】

`workspace`		作業領域へのポインタへのポインタ
`no_coeffs`		係数の数 K

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	no_coeffs ≤ 2

【内容】 InitFir で割り付けられた作業領域を解放します。

【備考】 本関数はリエントラントではありません。

(n) IIR 作業領域開放

■説明

【書式】 `int FreeIir (short **workspace, long no_sections)`

【引数】

`workspace`		作業領域へのポインタへのポインタ
`no_sections`		2 次フィルタセクションの数 K

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	no_sections < 1

【内容】 InitIir で割り付けられた作業領域を解放します。

【備考】 本関数はリエントラントではありません。

(o) 倍精度 IIR 作業領域開放

■説明

【書式】 `int FreeDIir (long **workspace, long no_sections)`

【引数】

`workspace`		作業領域へのポインタへのポインタ
`no_sections`		2 次フィルタセクションの数 K

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`no_section ≤ 2`

【内容】 `InitDIir` で割り付けられた作業領域メモリを解放します。

【備考】 本関数はリエントラントではありません。

(p) 適応 FIR 作業領域開放

■説明

【書式】 `int FreeLms (short **workspace, long no_coeffs)`

【引数】

`workspace`		作業領域へのポインタへのポインタ
`no_coeffs`		係数の数 K

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`no_coeffs < 1`

【内容】 `InitLms` で割り付けられた作業領域メモリを解放します。

【備考】 本関数はリエントラントではありません。

2.2.4 畳み込みと相関

(1) 関数一覧

表 2.6 DSP ライブラリ関数一覧(畳み込み)

項番	項目	関数名	説明
1	**完全畳み込み**	ConvComplete	2 **つの配列の完全な畳み込みを計算します。**
2	**周期的畳み込み**	ConvCyclic	2 **つの配列の周期的な畳み込みを計算します。**
3	**部分的畳み込み**	ConvPartial	2 **つの配列の部分的な畳み込みを計算します。**
4	**相関**	Correlate	2 **つの配列の相関を計算します。**
5	**周期的相関**	CorrCyclic	2 **つの配列の周期的な相関を計算します。**

これらの関数を使用する際は、2 つの入力配列のうち 1 つは X メモリに、もう 1 つは Y メモリに配置してください。出力配列はどのメモリに配置してもかまいません。

(2) 各関数の説明

(a) 完全畳み込み

■説明

【書式】
```
int ConvComplete (short output[], const short ip_x[],
                  const short ip_y[], long x_size,
                  long y_size, int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・res_shift < 0 ・res_shift > 25

【内容】 2 つの入力配列 x,y を完全に畳み込み、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left\lfloor \sum_{i=0}^{X-1} x(i)\, y(m-i) \right\rfloor \cdot 2^{-res\_shift} \qquad 0 \le m < X+Y-1$$

入力配列外のデータは 0 として読み込まれます。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、(xsize+ysize-1)以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX   8
#define NY   8
#define NOUT  NX+NY-1

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

short w1[5] = {-1, -32768, 32767, 2, -3, };
short x1[5] = {1, 32767, -32767, -32767, -2, };

void main()
{
    short i;
    short output[NOUT];
    int xsize, ysize, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w1[i%5];
    }
  for(i=0;i<NY;i++){
        daty[i] = x1[i%5];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;

    if(ConvComplete(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<NX;i++){
        printf("#%3d  dat_x:%6d  dat_y:%6d ¥n",i,datx[i],daty[i]);
    }
    for(i=0;i<NOUT;i++){
        printf("#%3d  output:%d ¥n",i,output[i]);
    }

}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

畳み込み計算に使用するデータ設定

(b) 周期的畳み込み

■説明

【書式】 `int ConvCyclic (short output[], const short ip_x[], const short ip_y[], long size, int res_shift)`

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 1 ・res_shift < 0 ・res_shift > 25

【内容】 2 つの入力配列 x,y を周期的に畳み込み、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|m - i + N|_N \right) \right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$ は剰余(i % N)を意味します。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、size 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x2[5] = {1, 32767, -32767, -32767, -2, };
short w2[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i;
    short output[N];
    int size, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w2[i];
       daty[i] = x2[i];
    }
   size = N ;
    rshift = 15;
    if(ConvCyclic(output, datx, daty, size, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  ip_x:%6d  ip_y:%6d  output:%6d ¥n",i,datx[i],daty[i],
                                                              output[i]);
    }
}
```

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

畳み込み計算に使用するデータ設定

(c) 部分的畳み込み

■説明

【書式】
```
int ConvPartial  (short output[], const short ip_x[],
                  const short ip_y[],long x_size, long y_size,
                  int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・res_shift < 0 ・res_shift > 25

【内容】 本関数は 2 つの入力配列 x,y を畳み込み、結果を出力配列 z に書き出します。

【備考】 入力配列外のデータから引き出された出力は含まれていません。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m \le |A-B|$$

ただし、配列の個数は a < b で、A は a のサイズ、B は b のサイズです。
入力配列外のデータは 0 として読み込まれます。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、(|xsize-ysize|+1)以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define   NX    5
#define   NY    5
short x3[5] = {1, 32767, -32767, -32767, -2, };
short w3[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

void main()
{
    short i;
    short output[NY+NX];
    int ysize, xsize, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w3[i];
    }
    for(i=0;i<NY;i++){
        daty[i] = x3[i];
    }

   xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvPartial(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NX;i++){
        printf("ip_x=%d  ¥n",datx[i]);
    }
    for(i=0;i<NY;i++){
        printf("ip_y=%d  ¥n",daty[i]);
    }
    for(i=0;i<(NY+NX);i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

畳み込み計算に使用するデータ設定

(d) **相関**

■説明

【書式】
```
int Correlate (short output[], const short ip_x[],
               const short ip_y[], long x_size, long y_size,
               long no_corr, int x_is_larger, int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
no_corr	計算する相関の数 M
x_is_larger	X=Y のときの配列指定
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・no_corr < 1 ・res_shift < 0 ・res_shift > 25 ・x_is_larger ≠ 0 または 1

【内容】 2 つの入力配列 x,y の相関を求め、結果を出力配列 z に書き出します。

【備考】 以下の式では配列の個数は a < b で、A は a のサイズとします。
x_is_larger=0 とすると x を a とし、x_is_larger=1 とすると x を b とします。
a 配列より b 配列が少ない場合の動作は保証致しません。
入力配列 x,y の大小と x_is_larger は矛盾のないように設定をお願いします。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(i+m) \right] \cdot 2^{-res\_shift} \qquad 0 \le m < M$$

A < X + M となっても差し支えありません。この場合、入力配列外のデータは 0 を使用します。
res_shift=0 は通常の整数計算に、res_shift=15 は小数計算に相当します。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、no_corr 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define NY   5
#define NX   5
#define M    4
#define MAXM  NX+NY
short x4[5] = {1, 32767, -32767, -32767, -2, };
short w4[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

void main()
{
    short i;
    int ysize, xsize, ncorr, rshift;
    short output[MAXM];
    int x_is_larger;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w4[i%5];
    }
    for(i=0;i<NY;i++){
       daty[i] = x4[i%5];
    }

    /* test working of stack */
    ysize = NY;
    xsize = NX;
    ncorr = M;
    rshift = 15;

    x_is_larger=0;
    for (i = 0; i < MAXM; output[i++] = 0);

    if (Correlate(output, datx, daty, xsize, ysize, ncorr,x_is_larger,rshift)
                                                          != EDSP_OK){
       printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<MAXM;i++){
       printf("[%d]:output=%d¥n",i,output[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

計算に使用するデータ設定

(e) 周期的相関

■説明

【書式】
```
int CorrCyclic (short output[], const short ip_x[],
                const short ip_y[], long size, int reverse,
                int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
reverse	反転フラグ
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 1 ・res_shift < 0 ・res_shift > 25 ・reverse ≠ 0 または 1

【内容】 周期的に配列 x,y の相関を求め、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|i + m|_N \right) \right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$は剰余(i % N)を意味します。reverse=1 の場合、出力のデータは反転され、実際の計算は以下のようになります。

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\, x\left(|i + m|_N \right) \right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、size 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5

short x5[5] = {1, 32767, -32767, -32767, -2, };
short w5[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i;
    short output[N];
    int size, rshift;
    int reverse;
    int result;

    /* TEST CYCLIC CORRELATION OF X WITH Y */
    reverse=0;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w5[i];
        daty[i] = x5[i];
    }

    /* test working of stack */
    size = N;
    rshift = 15;

    if (CorrCyclic(output, datx, daty, size, reverse, rshift) != EDSP_OK){
        printf("EDSP_OK not returned - this one¥n");
    }

    for(i=0;i<N;i++){
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

計算に使用するデータ設定

2.2.5 その他

(1) 関数一覧

表 2.7 DSP ライブラリ関数一覧(その他)

項番	項目	関数名	説明
1	H'8000→H'8001 置換	Limit	H'8000 のデータを H'8001 に置き換えます。
2	X メモリ→Y メモリコピー	CopyXtoY	配列を X メモリから Y メモリにコピーします。
3	Y メモリ→X メモリコピー	CopyYtoX	配列を Y メモリから X メモリにコピーします。
4	X メモリへコピー	CopyToX	配列を指定した場所から X メモリにコピーします。
5	Y メモリへコピー	CopyToY	配列を指定した場所から Y メモリにコピーします。
6	X メモリからコピー	CopyFromX	配列を X メモリから指定した場所にコピーします。
7	Y メモリからコピー	CopyFromY	配列を Y メモリから指定した場所にコピーします。
8	白色ガウス雑音	GenGWnoise	白色ガウス雑音を生成します。
9	マトリックス乗算	MatrixMult	2 つのマトリックスの乗算をします。
10	乗算	VectorMult	2 つのデータの乗算をします。
11	平均 2 乗値	MsPower	2 乗平均強度を求めます。
12	平均	Mean	平均を求めます。
13	平均と偏差	Variance	平均と偏差を求めます。
14	最大値	MaxI	整数配列の最大値を求めます。
15	最小値	MinI	整数配列の最小値を求めます。
16	最大絶対値	PeakI	整数配列の最大絶対値を求めます。

(2) 各関数の説明

(a) H'8000→H'8001 置換

■説明

【書式】 `int Limit (short data[], long no_elements, int data_is_x)`

【引数】

`data[]`	データ配列
`no_elements`	データ数
`data_is_x`	データ配置指定

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_elements < 1` ・`data_is_x ≠ 0` または `1`

【内容】 値が H'8000 の入力データを H'8001 に置き換えます。これにより、DSP 命令の固定小数点乗算の際にオーバフローが発生しないようにします。

【備考】 この処理を行っても積和演算の加算でオーバフローが発生する可能性はあります。
`data_is_x=1` のときは `data` は X メモリに、`data_is_x=0` のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i;
    int size;
    int src_x;

    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = dat[i%4];
        daty[i] = dat[i%4];
        printf("BEFORE NO %d datx daty :%d:%d ¥n",i,datx[i], daty[i]);
    }

    size = N;
    src_x = 1;

    if (Limit(datx, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned¥n");
    }

    src_x = 0;
    if (Limit(daty, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++) {
        printf("After NO %d datx daty :%d:%d¥n",i,datx[i], daty[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

データ設定

X メモリを使用する場合

Y メモリを使用する場合

(b) Xメモリ→Yメモリコピー

■説明

【書式】 `int CopyXtoY (short op_y[], const short ip_x[], long n)`

【引数】

`op_y[]`		出力配列
`ip_x[]`		入力配列
`n`		データ数

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`n < 1`

【内容】 配列を `ip_x` から `op_y` へコピーします。

【備考】 `ip_x` はXメモリに、`op_y` はYメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

  for(i=0;i<N;i++){
        daty[i]=0;
        datx[i]=dat[i%4];
    }

    if(CopyXtoY(daty, datx, N) != EDSP_OK){
        printf("CopyXtoY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

データ設定

(c) Ｙメモリ→Ｘメモリコピー

■説明

【書式】 `int CopyYtoX (short op_x[], const short ip_y[], long n)`

【引数】

	op_x[]	出力配列
	ip_y[]	入力配列
	n	データ数

【戻り値】

	EDSP_OK	成功
	EDSP_BAD_ARG	n < 1

【内容】 配列を ip_y から op_x へコピーします。

【備考】 ip_y はＹメモリ、op_x はＸメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  5
static short dat[N] = { -32768, 32767, -32768, 0,3};

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }

    if(CopyYtoX(datx, daty, N)!= EDSP_OK){
         printf("CopyYtoX error!¥n");
    }

    printf("no_elements %d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  po_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

インクルードヘッダ

Ｘメモリ Ｙメモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

(d) X メモリへコピー

■説明

【書式】 `int CopyToX (short op_x[], const short input[], long n)`

【引数】

`op_x[]`		出力配列
`input[]`		入力配列
`n`		データ数

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`n < 1`

【内容】 配列 `input` を `op_x` へコピーします。

【備考】 `op_x` は X メモリに、`input` は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section X
static short datx[N];
#pragma section

void main()
{
    int   i;
    short data[N];

    for(i=0;i<N;i++){
       data[i]=dat[i];
  }
    if(CopyToX(datx, data, N) !=EDSP_OK){
       printf("CopyToX Problem¥n");
    }
    printf("no_elements %d¥n",N);
    for(i=0;i<N;i++){
       printf("#%2d  op_x:%6d  input:%6d ¥n",i,datx[i],data[i]);
    }
}
```

インクルードヘッダ

X メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

(e) Ｙメモリへコピー

■説明

【書式】 `int CopyToY (short op_y[], const short input[], long n)`

【引数】

`op_y[]`		出力配列
`input[]`		入力配列
`n`		データ数

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`n < 1`

【内容】 配列 `input` を `op_y` へコピーします。

【備考】 `op_y` はＹメモリに、`input` は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;
    short data[N] ;

    for(i = 0; i < N; i++){
        data[i] = dat[i%4] ;
    }
    if(CopyToY(daty, data, N) != EDSP_OK){
        printf("CopyToY Problem¥n");
    }
    printf("no_elements %ld ¥n",N);
    for(i = 0; i < N; i++){
        printf("#%2d  op_y:%6d  input:%6d ¥n",i,daty[i],data[i]);
    }
}
```

インクルードヘッダ

Ｙメモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

(f) X メモリからコピー

■説明

【書式】 `int CopyFromX (short output[], const short ip_x[], long n)`

【引数】

`output[]`	出力配列
`ip_x[]`	入力配列
`n`	データ数

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	`n < 1`

【内容】 配列 `ip_x` を `output` へコピーします。

【備考】 `ip_x` は X メモリに、`output` は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section X
static short datx[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
      datx[i]=dat[i];
    }
    if(CopyFromX(out_dat,datx, N) != EDSP_OK){
       printf("CopyFromX Problem¥n");
    }
    for(i=0;i<N;i++){
       printf("#%3d output:%6d  ip_x:%6d ¥n",i,out_dat[i],datx[i]);
    }
    printf("no_elements:%ld¥n",N);
}
```

インクルードヘッダ

X メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

(g) Ｙメモリからコピー

■説明

【書式】 `int CopyFromY (short output[], const short ip_y[], long n)`

【引数】

output[]	出力配列
ip_y[]	入力配列
n	データ数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

【内容】 配列 ip_y を output へコピーします。

【備考】 ip_y はＹメモリに、output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyFromY(out_dat,daty, N)!= EDSP_OK){
        printf("CopyFormY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_y:%6d ¥n",i,out_dat[i],daty[i]);
    }
}
```

インクルードヘッダ

Ｙメモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

(h) 白色ガウス雑音

■説明

【書式】 `int GenGWnoise (short output[], long no_samples, float variance)`

【引数】

`output[]`	白色雑音データの出力
`no_samples`	出力データ数
`variance`	ノイズ分布の偏差 σ^2

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_samples < 1` ・`variance ≤ 0.0`

【内容】 平均が 0 で、ユーザが指定した偏差を持つ白色ガウス雑音を生成します。

【備考】 出力データは 2 つ 1 組で生成されます。1 組の出力データを生成するために rand 関数を使用し、x の 2 乗合計が 1 未満になる組が求められるまで-1～1 の間で 1 組の乱数 γ_1、γ_2 を生成します。そして、1 組の出力データ o_1、o_2 が以下の式で計算されます。

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

データ数を奇数に設定した場合、最後の組の 2 番目のデータは破棄されます。
本関数が呼び出している標準ライブラリの rand 関数はリエントラントではないので、生成される乱数 γ_1、γ_2 の順番が常に同じになるとは限りません。しかし、生成される白色雑音 o_1、o_2 の特性に影響を及ぼすことはありません。
本関数は浮動小数点演算を使用しています。浮動小数点演算は処理速度が遅くなるので、本関数は評価用として使うことをおすすめします。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>          } インクルードヘッダ

#define MAXG  4.5    /* approx. saturating level for N(0,1) random variable */
#define N_SAMP  10   /* number of samples generated in a frame
   */

void main()
{
    short out[N_SAMP];
    float var;
    int   i;

    var = 32768 / MAXG * 32768 / MAXG;
  if(GenGWnoise(out, N_SAMP, var) !=EDSP_OK){
       printf("GenGWnoise Problem¥n");
    }
    for(i=0;i<N_SAMP;i++){
       printf("#%2d  out:%6d  ¥n",i,out[i]);
    }
}
```

(i) マトリックス乗算

■説明

【書式】
```
int MatrixMult  (void *op_matrix, const void *ip_x,
                 const void *ip_y, long m, long n, long p,
                 int x_first, int res_shift)
```

【引数】

op_matrix	出力の第一データへのポインタ
ip_x	入力 x の第一データへのポインタ
ip_y	入力 y の第一データへのポインタ
m	マトリックス 1 の行数
n	マトリックス 1 の列数、マトリックス 2 の行数
p	マトリックス 2 の列数
x_first	マトリックス乗算の順番指定
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・m,n,または p < 1 ・res_shift < 0 ・res_shift > 25 ・x_first ≠ 0 または 1

【内容】 2 つのマトリックス x,y の乗算を行い、結果を op_matrix に配置します。

【備考】 x_first=1 の場合、x･y を計算します。このとき、ip_x は m×n、ip_y は n×p、op_matrix は m×p となります。
x_first=0 の場合、y･x を計算します。このとき、ip_y は m×n、ip_x は n×p、op_matrix は m×p となります。
積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
各マトリックスは通常の C 様式 (行優先順) で配置されます。

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

任意の配列サイズを指定できるようにするために、配列パラメタは void*で指定します。これらのパラメタは short 変数を指すようにしてください。
入力配列 ip_x,ip_y と出力配列 op_matrix は別々に用意してください。
ip_x は X メモリに、ip_y は Y メモリに、op_matrix は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
#define NN  N*N
short m1[16] = {  1, 32767, -32767, 32767,
                        1, 32767, -32767, 32767,
                              1, 32767, -32767, 32767,
                              1, 32767, -32767, 32767, };
short m2[16] = {  -1, 32767, -32767, -32767,
                        -1, 32767, -32767, -32767,
                        -1, 32767, -32767, -32767,
                              -1, 32767, -32767, -32767, };

#pragma section X
static short datx[NN];
#pragma section Y
static short daty[NN];
#pragma section

void main()
{
    short i, j;
    short output[NN];
    int m, n, p, rshift, x_first;
    long sum;

    for (i = 0; i < NN; output[i++] = 0) ;
    /* copy data into X and Y RAM */
    for(i=0;i<NN;i++) {
        datx[i] = m1[i%16];
        daty[i] = m2[i%16];
    }

    m = n = p = N;
    rshift = 15;
    x_first = 1;

    if (MatrixMult(output, datx, daty, m, n, p, x_first, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NN;i++) {
        printf("output[%d]=%d¥n",i,output[i]);
    }

}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

データ設定

(j) **乗算**

■説明

【書式】 `int VectorMult (short output[], const short ip_x[], const short ip_y[],long no_elements, int res_shift)`

【引数】

output[]	出力
ip_x[]	入力 1
ip_y[]	入力 2
no_elements	データ数
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・res_shift < 0 ・res_shift > 16

【内容】 ip_x,ip_y から 1 つずつデータを取り出して乗算を行い、結果を output に配置します。

【備考】 出力は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。
なお、オーバフローしたときは正または負の最大値となります。
本関数はデータの乗算を行います。内積を計算する場合は m(マトリックス 1 の行数) と p(マトリックス 2 の列数)を 1 に設定して MatrixMult を使用してください。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
```
インクルードヘッダ

```
#define N  4
#define RSHIFT  15
short y[4] = {1, 32767, -32767, 32767, };
short x[4] = {-1, 32767, -32767, -32767, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
```
X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義します。

```
void main()
{
    short i, n ;
    short output[N];
    int size, rshift;


    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = x[i];
        daty[i] = y[i];
    }
```
データ設定

```
    size = N;
    rshift = RSHIFT;
    for (i = 0; i < N; output[i++] = 0) ;
    if (VectorMult(output, datx, daty, size, rshift) != EDSP_OK) {
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_x:%6d  ip_y:%6d
                        ¥n",i,output[i],datx[i],daty[i]);
    }
}
```

(k) 平均 2 乗値

■説明

【書式】 `int MsPower (long *output, const short input[], long no_elements, int src_is_x)`

【引数】

`output`		出力へのポインタ
`input[]`		入力 x
`no_elements`		データ数 N
`src_is_x`		データ配置指定

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_elements < 1` ・`src_is_x ≠ 0` または 1

【内容】 入力データの平均 2 乗値を求めます。

【備考】 $\text{平均2乗値} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$

除算結果は最も近い整数値に丸められます。
演算の結果は 63 ビットで保持されます。`no_elements` が 2^{32} 以上の場合、オーバフローが発生することがあります。
`src_is_x=1` のときは `input` は X メモリに、`src_is_x=0` のときは Y メモリに配置してください。
`output` は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section

void main()
{
    int    i;
    long   output[1];
    int    src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    src_x = 1;
    if (MsPower(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:x=%d¥n",output[0]);

    src_x = 0;
    if (MsPower(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:y=%d¥n",output[0]);
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

X メモリを使用する場合
src_x=1

Y メモリを使用する場合
src_x=0

(l) **平均値**

■説明

【書式】 `int Mean (short *mean, const short input[], long no_elements, int src_is_x)`

【引数】

mean	input の平均 $\bar{x}$ へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 入力データの平均値を求めます。

【備考】 $\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$

除算結果は最も近い整数値に丸められます。
演算結果は 32 ビットで保持されます。no_elements が 2^{16}-1 よりも大きい場合、オーバフローが発生することがあります。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
    short    i,output[1];
    int      size;
    int      src_x;
    int      flag = 1;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
       daty[i] = dat[i];
    }
    /* test working of stack */
    src_x = 1;
    if (Mean(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Mean:x=%d¥n",output[0]);

    src_x = 0;
    if (Mean(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Mean:y=%d¥n",output[0]);
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

X メモリを使用する場合
src x=1

Y メモリを使用する場合
src_x=0

(m) **平均と偏差**

■説明

【書式】 `int Variance (long *variance, short *mean, const short input[], long no_elements, int src_is_x)`

【引数】

variance	入力の偏差 σ^2 へのポインタ
mean	データの平均 $\bar{x}$ へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 input の平均と偏差を求めます。

【備考】 $\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$ $\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \bar{x}^2$

除算結果は最も近い整数値に丸められます。
$\bar{x}$ は 32 ビットで保持されます。また、オーバフローのチェックはしません。
no_elements が 2^{16}-1 よりも大きい場合、オーバフローが発生することがあります。
σ^2 は 63 ビットで保持されます。オーバフローのチェックはしません。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section

void main()
{
    long       size,var[1];
    short      mean[1];
    int        i ;
    int        src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* test working of stack */
    size = N;
    src_x = 1;
    if (Variance(var, mean, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);

    src_x = 0;
    if (Variance(var, mean, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);
}
```

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

X メモリを使用する場合
src_x=1

Y メモリを使用する場合
src_x=0

(n) **最大値**

■説明

【書式】 `int MaxI (short **max_ptr, short input[], long no_elements, int src_is_x)`

【引数】

max_ptr	最大データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 配列 input の最大値を検索して、そのアドレスを max_ptr に返します。

【備考】 複数のデータが同じ最大値を持つ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[131] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section

void main()
{
    short    *outp,**outpp;
    int      size,i;
    int      src_x;

 /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* MAXI */
    size = N;
    outpp = &outp;
    src_x = 1;

    if (MaxI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Max:x = %d¥n",**outpp);

    src_x = 0;
    if (MaxI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Max:y = %d¥n",**outpp);
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

X メモリを使用する場合
src_x=1

Y メモリを使用する場合
src_x=0

(o) **最小値**

■説明

【書式】 `int MinI (short **min_ptr, short input[], long no_elements, int src_is_x)`

【引数】

`min_ptr`	最小データへのポインタへのポインタ
`input[]`	入力
`no_elements`	データ数
`src_is_x`	データ配置指定

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_elements < 1` ・`src_is_x ≠ 0` または `1`

【内容】 配列 `input` の最小値を検索して、そのアドレスを `min_ptr` に返します。

【備考】 複数のデータが同じ最小値を持つ場合、`input` の先頭に最も近いデータのアドレスが返されます。
`src_is_x=1` のときは `input` は X メモリに、`src_is_x=0` のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N 10
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section

void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* MINI */
    /* test working of stack */
    size = N;
    outpp = &outp;

    src_x = 1;
    if (MinI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Min:x=%d¥n",**outpp);

    src_x = 0;
    if (MinI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Min:y=%d¥n",**outpp);
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

X メモリを使用する場合
src_x=1

Y メモリを使用する場合
src_x=0

(p) **最大絶対値**

■説明

【書式】 `int PeakI (short **peak_ptr, short input[], long no_elements, int src_is_x)`

【引数】

`peak_ptr`	最大絶対値データへのポインタへのポインタ
`input[]`	入力
`no_elements`	データ数
`src_is_x`	データ配置指定

【戻り値】

`EDSP_OK`	成功
`EDSP_BAD_ARG`	以下のいずれかの場合です ・`no_elements < 1` ・`src_is_x ≠ 0` または `1`

【内容】 配列 `input` の最大絶対値を検索して、そのアドレスを `peak_ptr` に返します。

【備考】 複数のデータが同じ最大絶対値を持つ場合、`input` の先頭に最も近いデータのアドレスが返されます。
`src_is_x=1` のときは `input` は X メモリに、`src_is_x=0` のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section

void main()
{
    short    *outp,**outpp;
    int      size,i;
    int      src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    size = N;
    outpp = &outp;

    src_x = 1;
    if (PeakI(outpp, datx, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:x=%d¥n",**outpp);

    src_x = 0;
    if (PeakI(outpp, daty, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:y=%d¥n",**outpp);
}
```

インクルードヘッダ

X メモリ Y メモリに配置する変数は #pragma section でセクション内に定義する。

データ設定

X メモリを使用する場合
src_x=1

Y メモリを使用する場合
src_x=0

2.3 DSP ライブラリの性能について

(1) DSP ライブラリの実行サイクル数について

DSP ライブラリの実行サイクル数は以下のとおりです。

測定条件はエミュレータ(SH-DSP,60MHz)にて測定、プログラムセクションは X-ROM または Y-ROM に割り付けました。

表 2.8 DSP ライブラリの実行サイクル数一覧表（1）

分類	DSP ライブラリ関数名	実行サイクル数 (Cycle)	備考
高速フーリエ変換	FftComplex	29,330	**サイズ**: 256 **スケーリング**: 0xFFFFFFFF
	FftReal	25,490	
	IfftComplex	30,380	
	IfftReal	29,240	
	FftInComplex	26,540	
	FftInreal	25,260	
	IfftInComplex	27,590	
	IfftInReal	27,470	
	LogMagnitude	1,778,290	
	InitFft	3,116,640	
	FreeFft	780	
フィルタ	Fir	23,010	**係数の数** : 64 **データの数** : 200 **収束係数** 2μ = 32767
	Fir1	280	
	Lms	97,710	
	Lms1	790	
	InitFir	1,400	
	InitLms	1,400	
	FreeFir	90	
	FreeLms	90	
	Iir	23,530	**データ数** : 200 **フィルタセクションの数** : 5
	Iir1	360	
	DIir	309,010	
	DIir1	1,860	
	InitIir	280	
	InitDIir	280	
	FreeIir	90	
	FreeDIir	270	
窓関数	GenBlackman	789,950	**データ数** : 100
	GenHamming	418,330	
	GenHanning	447,250	
	GenTriangle	744,220	

分類	DSP ライブラリ関数名	実行サイクル数 (Cycle)	備考
畳み込	ConvComplete	21,890	データ数 : 100
	ConvCyclic	14,790	
	ConvPartial	370	
	Correlate	11,930	
	CorrCyclic	15,790	
その他	Limit	480	データ数 : 100
	CopyXtoY	130	
	CopyYtoX	130	
	CopyToX	1,270	
	CopyToY	1,270	
	CopyFromX	1,320	
	CopyFromY	1,320	
	GenGWnoise	2,878,410	
	MatrixMult	2,337,460	
	VectorMult	1,500	
	MsPower	370	
	Mean	270	
	Variance	820	
	MaxI	540	
	MinI	520	
	PeakI	740	

(2) C言語と DSP ライブラリのソースコードの比較

FFT 関数の一部(バタフライ計算を行っている部分)の関数について C 言語で書かれたものと DSP ライブラリのソースを示します。

DSP ライブラリでは movx,movy,padd などの DSP 特有の命令を使用することで性能 Up を実現しています。

C 言語のソースコード

```
void R4add(short *arp, short *brp, short *aip, short *bip, int grpinc, int numgrp) {
     short  tr,ti;
     int    grpind;

     for(grpind=0;grpind<numgrp;grpind++) {
            tr     =      *brp;
            ti     =      *bip;
            *brp = sub(*arp,ti);
            *bip = add(*aip,tr);
            *arp = add(*arp,ti);
            *aip = sub(*aip,tr);
            arp += grpinc;
            aip += grpinc;
            brp += grpinc;
            bip += grpinc;
      }
}
```

DSP ライブラリのソースコード

```
_R4add:
        MOV.L Ix,@-R15
        MOV.L Iy,@-R15

        MOV.L @(2*4,R15),Ix
        SHLL  Ix
        MOV   Ix,Iy
        MOV.L @(3*4,R15),R1

        REPEAT r4alps,r4alpe
        ADD   #-1,R1
        SETRC R1
                                          movx.w  @ar,X0          movy.w @bi,Y0
        padd  X0,Y0,A0
        psub  X0,Y0,A1                    movx.w  @br,X0          movy.w @ai,Y0
        padd  X0,Y0,A0                    movx.w  A0,@ar+Ix
                          pneg    X0,X0   movx.w  A1,@br+Ix
        padd  X0,Y0,A1                                            movy.w A0,@bi+Iy
                                          movx.w  @ar,X0          movy.w @bi,Y0

        .ALIGN      4
r4alps  padd  X0,Y0,A0                                            movy.w A1,@ai+Iy
        psub  X0,Y0,A1                    movx.w  @br,X0          movy.w @ai,Y0
        padd  X0,Y0,A0                    movx.w  A0,@ar+Ix
                          pneg   X0,X0    movx.w  A1,@br+Ix
        padd  X0,Y0,A1                                            movy.w A0,@bi+Iy
r4alpe                                    movx.w  @ar,X0          movy.w @bi,Y0
                                                                  movy.w A1,@ai+Iy
        MOV.L @R15+,Iy
        RTS
        MOV.L @R15+,Ix
```

(3) FFT の各関数についての性能

フーリエ変換の各関数は以下のように分類されます。

表 2.9 高速フーリエ変換

	Not-in-place 方式	In-place 方式
複素数フーリエ変換	FftComplex	FftInComplex
実数フーリエ変換	FftReal	FftReal

表 2.10 逆高速フーリエ変換

	Not-in-place 方式	In-place 方式
複素数フーリエ変換	IfftComplex	IfftInComplex
実数フーリエ変換	IfftReal	IfftInReal

■ in-place 方式と not-in-place 方式の違いについて

in-place 方式は入力データとの配列をそのまま出力データの配列として使用する方式です。したがって入力データは出力データによって上書きされるため保存されません。
not-in-place 方式は入力データと出力データを別々に用意して関数呼び出しを行う方式です。入力データと出力データが別々なので関数を呼び出した後も入力データはそのまま保存されます。
in-place 方式と not-in-place 方式の性能上の差はほとんどないのでメモリの使用量に従って使用する関数を決定してください。
in-place 方式では not-in-place 方式に比べて使用するメモリ量が半分で済みます。

■ スケーリングについて

FFT 計算の各段階において、計算は積和の形式で実行されるためオーバフローが起きやすくなっています。オーバフローが起きるとすべて最大値または最小値となるため計算結果を正しく評価することができません。
スケーリングはそのオーバフローを防ぐために FFT 計算の各段階において、どの程度 2 で割る（右シフトを行う）かを表す目安です。

表 2.11 スケーリング値とその特長

スケーリング値	特長
FFTNOSCALE	**まったくシフトしない。オーバフローが起きやすい**
EFFTMIDSCALE	**段階の 1 つおきにシフトする。**
EFFTALLSCALE	**すべての段階でシフト処理を行う。 オーバフローが起きにくい。**

スケーリングの性能に及ぼす影響は大きくありません。したがって、スケーリングを決定する場合は性能ではなく、データの特性から決定してください。

(4) フィルタの各関数について

■ Fir と Lms の使用方法

FirフィルタとLmsフィルタの係数の数とサイクル数の関係を**図**3.11に示します。
Lmsの計算速度は適応アルゴリズムを使用しているため、Firよりも遅くなります。データの波形が安定しているシステムでは、Lmsはフィルタ係数を決定するために使用し、決定した後はFirフィルタに置き換えてください。
データのスケーリングのために、右シフトの数を指定することができます。SH-DSPライブラリの内部で積和演算を行いますので、データによってはオーバフローが起こります。その場合は右シフト数を適当に変更して、出力される値を参考にしながら決定してください。

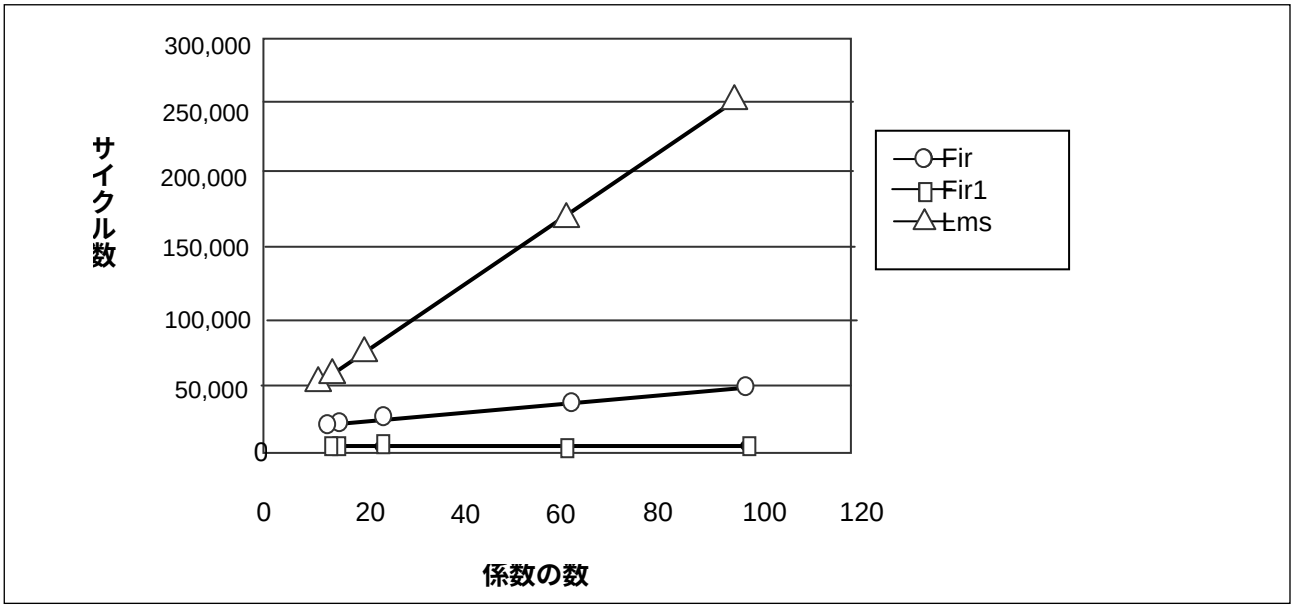


図 2 .2 **係数の数とサイクル数の関係**

■ Iir と DIir について

性能を優先する場合は DIir ではなく、Iir を使用してください。SH-DSP ライブラリの内部で積和演算を行いますのでデータによってはオーバフローが起こります。その場合には右シフト数を適当に変更して、出力される値を見ながら決定してください。
データのスケーリングのために、右シフトの数を指定することができます。ただし、右シフト数の指定方法はフィルタ係数の配列の一部として設定します。詳細は「3.13.6(5)(c)IIR,(e)**倍精度** IIR」を参照してください。

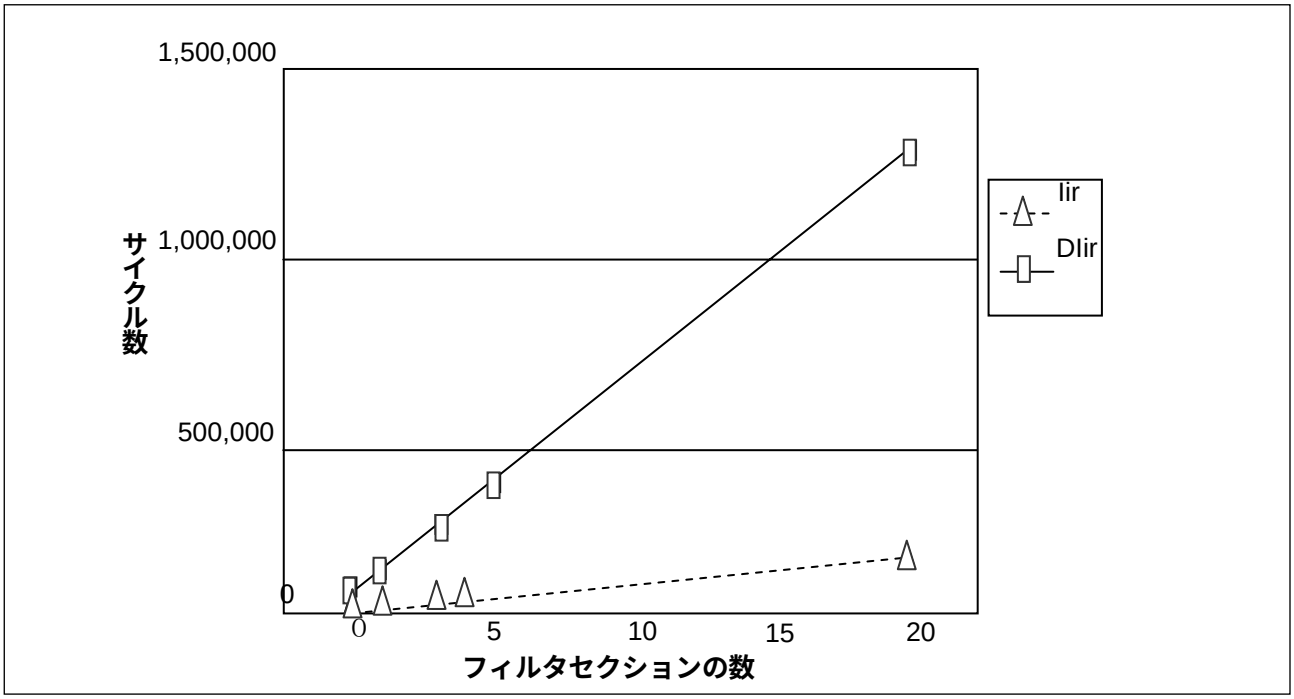


図 2.3 **フィルタセクションの数とサイクル数の関係**

■ フィルタ関数の使い分け

Fir フィルタは直線位相応答の特長を持ち、常に安定しているので位相の歪みが許されないオーディオ、ビデオなどのアプリケーションに適しています。一方、Iir フィルタはフィードバックを持つフィルタであり、Fir よりも少ない係数で結果を得ることができ、高速なので、時間制約のあるような場合に適しています。しかし、Iir フィルタは不安定になることがあるので使用に際しては十分注意が必要です。

3. DSP-C 言語について

■説明

SuperH RISC engine C/C++コンパイラでは、C 言語の拡張言語である DSP-C 言語をサポートしています。DSP-C 言語を用いることで、DSP 命令を C コンパイラで生成することができます。

DSP-C 言語は SuperH RISC engine C/C++コンパイラのコンパイルオプション「dspc」を指定した場合に有効となります。

3.1 固定小数点データ型

通常の C 言語で固定小数点数を表す場合は整数型を代用しますが、DSP-C 言語では固定小数点データ型を用いることで、小数値をそのまま記述し固定小数点数として扱うことができます。

SuperH RISC engine C/C++コンパイラでは固定小数点データ型の使用状況により、適切な DSP 演算命令を生成します。固定小数点データ型の内部表現を**表** 3.1 に示します。

表 3.1 **固定小数点データ型の内部表現**

型	Size (メモリ上の Size)	境界調整数 (byte)	データの範囲		定数添字
			最小値	最大値	
_ _fixed	16 bit (16 bit)	2	-1.0	$1.0-2^{-15}$ (0.999969482421875)	r
long _ _fixed	32 bit (32 bit)	4	-1.0	$1.0-2^{-31}$ (0.99999999953433871269226074218 75)	R
_ _accum	24 bit (32 bit)	4	-256.0	$256.0-2^{-15}$ (255.999969482421875)	a
Long _ _accum	40 bit (64 bit)	4	-256.0	$256.0-2^{-31}$ (255.99999999953433871269226074218 75)	A

■注意事項

（1） _ _accumおよびlong _ _accum型の場合、メモリ格納は右詰めとなり、先頭部分は符号拡張されます。

例) (_ _accum)128.5a ⇒ 00 40 40 00
例) (long _ _accum)(-256.0A) ⇒ FF FF FF 80 00 00 00 00

（2） 従来方法とDSP-Cの比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8

short input[NUM] = {0x1000, 0x2000, 0x4000,
                    0x6000,
                    0xf000, 0xe000, 0xc000,
                    0xa000};
short result[NUM];

void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = input[i] + 0x1000;
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8

fixed input[8] = { 0.125r, 0.25r, 0.5r,    0.75r,-0.125r,
                   -0.25r, -0.5r, -0.75r};

__fixed result[NUM];

void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = input[i] + 0.125r;
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

（3） 積和演算の例
整数型で代用した場合、積の桁をあわせる必要がありますが、固定小数点データ型は必要ありません。

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};

short y_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000,
                      0xc000, 0xa000};

int result;
i
nt func(short *x_input, short *y_input)
{
      int i;
      int temp = 0;
      for (i = 0; i< NUM ;i++) {
            temp += (x_input[i] * y_input[i]) >> 15;
      }
      return (temp);
}

void main()
{
      result = func(x_input, y_input);
      printf("%f¥n", result/32768.0);
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc -fixed_noround
#include <stdio.h>
#define NUM 8

__X __fixed x_input[NUM] = { 0.125r,    0.25r,   0.5r,   0.75r,
                            -0.125r,   -0.25r, -0.5r, -0.75r};

__Y __fixed y_input[NUM] = { 0.125r,    0.25r,   0.5r,   0.75r,
                            -0.125r,   -0.25r, -0.5r, -0.75r};

__accum result;

void func(__accum *result_p,
          __X __fixed *x_input,
          __Y __fixed *y_input)
{
      int i;
      __accum temp = 0.0a;
      for (i = 0; i< NUM ;i++) {
            temp += x_input[i] * y_input[i];
      }
      *result_p = temp;
}

void main()
{
      func(&result, x_input, y_input);
      printf("%a¥n", result);
}
```

3.2 メモリ修飾子

変数に X／Y メモリ修飾子を付加することで、通常のメモリアクセス命令よりも効率の良い X／Y メモリ専用アクセス命令の生成を促します。

X／Y メモリへの格納を明示的に指定するために、以下の修飾子を使用します。

_ _X ：X メモリにデータを格納
_ _Y ：Y メモリにデータを格納

SuperH RISC engine C/C++コンパイラは、_ _X/_ _Y メモリ修飾子の付いたオブジェクトを**表** 3.2 に示すセクションに出力します。これらのセクションは、リンク時にそれぞれ X／Y メモリに割り付けるようにしてください。

表 3.2 **メモリ修飾子の仕様**

名称	セクション	内容
定数領域	**$**XC	const **型のデータ** (X **メモリに格納**)
	$YC	const **型のデータ** (X **メモリに格納**)
初期化データ領域	**$**XD	**初期値のあるデータ** (X **メモリに格納**)
	$YD	**初期値のあるデータ** (Y **メモリに格納**)
未初期化データ領域	**$**XB	**初期値のないデータ** (X **メモリに格納**)
	$YB	**初期値のないデータ** (Y **メモリに格納**)

ただし、X／Y メモリが RAM 上にのみ存在する場合があります。この場合の ROM 化の際は注意する必要があります。

■使用例

（1）_ _X/_ _Y メモリ修飾子使用時のメモリ格納例

```
_ _X   int          a;      // Xメモリに格納
int   _ _X          b;      // Xメモリに格納
_ _Y   int          * c;    // Yメモリ上のint型へのポインタ(メモリは不定)
int    _ _Y         * d;    // Yメモリ上のint型へのポインタ(メモリは不定)
int    *_ _Y        e;      / int型へのポインタ(Yメモリに格納)
_ _X int *_ _Y      f;      // Xメモリ上のint型へのポインタ(Yメモリに格納)
```

（2）ROM 上の定数領域、初期化データ領域を X／Y RAM へコピー

リンク時には ROM 上に置き、プログラムの実行開始時に X／Y RAM 上にコピーします。
したがって、最適化リンケージエディタの rom オプションを用いて、ROM 上と X／Y RAM に、二重に領域をとる必要があります。

リンク時サブコマンド例)

```
rom=$XC=XC,$XD=XD,$YC=YC,$YD=YD
start P,C,D,$XC,$XD,$YC,$YD/400,$XB,XC,XD/05007000,$YB,YC,YD/05017000
```

また ROM から X／Y RAM へのコピーは、標準ライブラリ関数_INITSCT()を使うと容易にできます。

（使用例）_INITSCT()

```
#include <_h_c_lib.h>

void PowerON_Reset(void)
{
      _INITSCT();
      main();
      sleep();
}

#pragma section $DSEC
static const struct {
   void *rom_s;
   void *rom_e;
   void *ram_s;
} DTBL[] = {{__sectop("$XC"), __secend("$XC"), __sectop("XC")},
                {__sectop("$XD"), __secend("$XD"), __sectop("XD")},
                {__sectop("$YC"), __secend("$YC"), __sectop("YC")},
                {__sectop("$YD"), __secend("$YD"), __sectop("YD")}};
#pragma section
```

（3）定数領域、初期化データ領域を使用しない

_ _X／_ _Y メモリ修飾子を付けたオブジェクトに対して const 指定、初期化データを付けないようにすれば、ROM 上と X／Y RAM 上に二重に領域を取る必要はありません。

たとえば、以下の例のように動的に初期化するようにすることで、初期化データをなくすことができます。

（使用例）

```
#define NUM 8

__X __fixed x_input[NUM];
__Y __fixed y_input[NUM];

__fixed x_input[NUM] = {   0.125r,   0.25r,   0.5r,   0.75r,   -0.125r,   -0.25r,   -0.5r,   -0.75r};
__fixed y_input[NUM] = {   0.125r,   0.25r,   0.5r,   0.75r,   -0.125r,   -0.25r,   -0.5r,   -0.75r};

void xy_init()
{
        int i;

        for (i = 0; i< NUM; i++) {
              x_input[i] = x_init[i];
              y_input[i] = y_init[i];
        }
}

void main()
{
          xy_init();
              :
              :
}
```

（4） 従来方法と DSP-C の比較

C 言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000,0xf000, 0xe000,
                      0xc000, 0xa000};

short y_input[NUM] = {0x2000, 0x4000, 0xe000,
                      0xf000, 0x6000, 0x2000,
                      0xe000, 0xf000};
short result[NUM];

void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] - y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n",
        result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8

__X __fixed x_input[NUM] = { 0.125r, 0.25r, 0.5r, 0.75r,
                             -0.125r, -0.25r, -0.5r, -0.75r};

__Y __fixed y_input[NUM] = {0.25r, 0.5r,   -0.25r, -0.125r,
                            0.75r, 0.25r, -0.25r, -0.125r};
__fixed result[NUM];

void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] - y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.3 飽和修飾子

演算結果がオーバフローした場合にその結果を最大値／最小値に置き換える飽和演算を DSP-C では飽和修飾子をつけるだけで可能にします。

飽和演算を指定するために以下の修飾子を使用します。

_ _sat

飽和修飾子は、_ _fixed 型、long _ _fixed 型データのみに使用することができます。それ以外の型に指定した場合は、エラーになります。

なお、式の中に 1 つ以上の飽和修飾子(_ _sat 指定)指定されたデータがあれば、飽和演算を行います。

■使用例

（1） _ _sat指定例

```
_ _fixed          a;
_ _sat _ _fixed   b;
_ _fixed          c;

a = -0.75r ;
b = -0.75r ;
c = a + b   ;   // c = -1.0r  になります。
```

（2） 従来方法とDSP-Cの比較

C 言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000, 0x6000,
                      0xf000, 0xe000, 0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000, 0x6000,
                      0xf000, 0xe000, 0xc000, 0xa000};
short result[NUM];

void func(void)
{
    int i;
    int temp;
    for (i = 0; i < NUM; i++) {
        temp = x_input[i] + y_input[i];
        if (temp > 32767) {
            temp = 32767;
        }
        else if (temp < -32768) {
            temp = -32768;
        }
        result[i] = temp;
    }
}

void main(void)
{
    int i;
    func();
      :
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8

__sat __X __fixed x_input[NUM] = { 0.125r,   0.25r,   0.5r,   0.75r,
                                  -0.125r,  -0.25r,  -0.5r, -0.75r};
__sat __Y __fixed y_input[NUM] = { 0.125r,   0.25r,   0.5r,   0.75r,
                                  -0.125r, -0.25r, -0.5r, -0.75r};
__fixed result[NUM];

void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] + y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.4 循環修飾子

モジュロアドレッシングを指定するために、以下の修飾子を使用します。

_ _circ

モジュロアドレッシングは、メモリ修飾子(_ _X／_ _Y)指定のある_ _fixed 型の１次元配列およびポインタのみに指定することができます。それ以外の条件で使用した場合はエラーになります。

■使用例

（1） 従来方法とDSP-Cの比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
#define BUFFER_SIZE 4

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                      0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short y_input[BUFFER_SIZE] = {0x2000, 0x4000,
                              0x2000, 0x1000};
short result[NUM];

void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] + y_input[i%(BUFFER_SIZE)];
    }
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#include <machine.h>
#define NUM 8
#define BUFFER_SIZE 4

__X __fixed x_input[NUM] = { 0.125r,   0.25r,   0.5r,   0.75r,
                            -0.125r, -0.25r, -0.5r, -0.75r};
__circ __Y __fixed y_input[BUFFER_SIZE] = {0.25r, 0.5r,
                                           0.25r, 0.125r};
__fixed result[NUM];

void func()
{
    int i;
    set_circ_y(y_input, sizeof(y_input));
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] + y_input[i];
    }
    clr_circ();
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

■注意事項

（1） モジュロアドレッシングは、組み込み関数 set_circ_x()またはset_circ_y()とclr_circ()の間にある１次元配列およびポインタが対象となります。

（2） 複数の配列を同時にモジュロアドレッシング指定をした場合、および上記組み込み関数の間以外で_ _circ指定のある配列またはポインタを参照した場合、動作は保証しません。

（3） 負方向へのモジュロアドレッシングを指定した場合、動作は保証しません。

（4） モジュロアドレッシングを行うデータは、リンク時にアドレスの上位16bitが同じになるように配置する必要があります。なお、配列内容の直接参照はできません。

（5） 以下のいずれかに該当する場合、動作は保証しません。(ウォーニングを出力する場合もあり)

optimize=0を指定している

_ _circポインタをローカル変数以外に指定している

_ _circポインタにvolatileを指定している

_ _circポインタの更新のみで参照がない

組み込み関数 set_circ_xまたはset_circ_yとclr_circの間に関数呼び出しがある

3.5 型変換

型変換の規則を**表** 3.3 に示します。

表 3.3 **型変換の規則**

変換	仕様
_ _fixed -> long _ _fixed	**下位** 16bit **ゼロクリア。**
_ _accum -> long _ _accum	**値は変わらない。**
long _ _fixed -> _ _fixed	**下位** 16bit **切り捨て。**
long _ _accum -> _ _accum	**小数の精度が落ちる。**
_ _fixed -> _ _accum	**上位** 8bit **符号拡張。**
long _ _fixed -> long _ _accum	**値は変わらない。**
_ _fixed -> long _ _accum	**上位** 8bit **符号拡張。 下位** 16bit **ゼロクリア。** **値は変わらない。**
long _ _fixed -> _ _accum	**上位** 8bit **符号拡張。 下位** 16bit **は切り捨て。** **小数の精度が落ちる。**
_ _accum -> _ _fixed	**上位** 8bit **切り捨て。** 9bit **目を符号** bit **とする。** **整数部分が** 0 **の時は値は不変。**
long _ _accum -> long _ _fixed	
_ _accum -> long _ _fixed	
long _ _accum -> _ _fixed	**上位** 8bit **切り捨て。 下位** 16bit **切り捨て。** 9bit **目を符号** bit **とする。整数部分が** 0 **のときは値は不変。** **小数の精度が落ちる。**
_ _fixed ->**符号付き整数型**	-1.0r, -1.0R **の場合は、**-1, **それ以外の場合は** 0**。**
long _ _fixed ->**符号付き整数型**	
_ _accum ->**符号付き整数型**	**小数部切り捨て。**
long _ _accum ->**符号付き整数型**	**変換後の値は**,-256～255 **の間の整数。**
_ _fixed ->**符号なし整数型**	-1.0r, -1.0R **の場合は変換後の型の最大値、**
long _ _fixed ->**符号なし整数型**	**それ以外の場合は** 0**。**
_ _accum ->**符号なし整数型**	**小数部切り捨て。**
long _ _accum ->**符号なし整数型**	**正の値の場合、変換後の値は、**0～255 **の間の整数。** **負の値の場合、(変換前の値**+1+**変換後の型の最大値)。**
符号付き整数型-> _ _fixed	**変換前の最上位** bit **を変換後の最上位** bit **にする。**
符号付き整数型->long _ _fixed	**その他の** bit **はすべて** 0**。**
符号付き整数型-> _ _accum	**値の下位** 9bit **を整数部とする。**
符号付き整数型->long _ _accum	**小数部は** 0 **とする。**
符号なし整数型-> _ _fixed	**変換後のすべての** bit **を** 0 **とする。**
符号なし整数型->long _ _fixed	
符号なし整数型-> _ _accum	**値の下位** 9bit **を整数部とする。**
符号なし整数型->long _ _accum	**小数部は** 0 **とする。**
固定小数点 ->**浮動小数点**	**変換後の型で表現可能な値は、元の値と同じ。** **表現できない場合は、最も近い値に丸める。**
浮動小数点 ->**固定小数点**	**小数部分は固定小数点** -> **浮動小数点変換と同じ。** **整数部分は浮動小数点** -> **整数変換と同じ。** **整数部分が固定小数点の表現可能範囲内の場合は、** **そのままの値を保つようにする。** **範囲外の場合はあふれた部分の最下位** bit **を符号** bit **とする。また、変換後の型に飽和処理指定があっても飽和処理は行わない。**

■注意事項

（1） (long)_ _fixed ⇔ 整数型
(long)_ _fixed型で表現できる整数は、０と－１のみとなります。
このため、この変換を行うことで情報落ちが発生してしまいます。

（2） (long)_ _accum ⇔ 整数型
(long)_ _accum型は、－256～255の間の整数型を表現することができ、この範囲の整数型であれば、変換後も情報を保持します。
ただし、符号なし整数型への変換の場合、負の値はオーバフローとなりますので注意が必要です。
整数型のみを必要とする演算が続く場合は、整数型に変換した方が、性能向上になる場合もあります。

（3） ビットパターンコピー
ビットパターンをコピーする場合、代入演算子などで行いますと型変換されてしまい、期待どおりの結果を得られません。この場合は、組み込み関数(long_as_lfixed, lfixed_as_long)を使用してください。

4. よくある問い合わせ

4.1 C/C++言語記述で DSP 命令は生成されますか？

■質問

コンパイラオプションで、"cpu=sh2dsp"、または"cpu=sh3dsp"、または"cpu=sh4aldsp"を選択すれば、
C 言語あるいは C++言語から直接 DSP 命令が出力されますか？

■回答

SHC コンパイラバージョン 7 以前のコンパイラを御使用の場合、C 言語あるいは C++言語から DSP 命令を生成することはできません。
DSP 命令を使用するためには、アセンブリ言語で記述するか、DSP ライブラリを使用してください。
SHC コンパイラ Ver.8 以降を御使用の場合、DSP-C 言語仕様に従った記述を行うことで、DSP 命令を生成させることができます。
また、コンパイラの"repeat"オプションを選択した場合、DSP 拡張リピートループを使用した命令を生成する場合があります。
DSP 拡張リピートループは LDRC 命令をサポートした CPU のみ使用することができます。

DSP ライブラリについては 2 章を、DSP-C 言語については 3 章をご参照ください。

4.2 DSP ライブラリの FFT 関数で実行可能な最大サンプリング数は幾つでしょうか？

■質問

DSP ライブラリの FFT 関数で実行可能な最大サンプリング数は幾つでしょうか？

■回答

FFT 関数では、ヒープメモリと X/Y メモリを必要とします。
ヒープメモリのサイズはユーザでコントロール可能ですが、X/Y メモリのサイズはディバイスに依存します。
その為、X/Y メモリのサイズから FFT 関数の最大サンプリング数が求まります。

X/Y メモリのサイズがそれぞれ 8k バイトの場合、最大サンプリング数と使用ヒープサイズは以下の通りです。

表 4.1 FFT ライブラリ関数 - 最大サンプリング数、使用ヒープサイズ

	最大サンプリング数	使用ヒープサイズ
FftComplex	2048	17334
FftReal （入力データを X/Y メモリ以外に配置）	4096	18358
FftReal （入力データを X/Y メモリに配置）	2048	17334
IfftComplex	2048	17334
IfftReal	2048	19382
FftInComplex	4096	18358
FftInReal	4096	18358
IfftInComplex	4096	18358
IfftInReal	4096	18358

ホームページとサポート窓口<website and support,ws>

ルネサステクノロジホームページ

http://japan.renesas.com/

お問合せ先

http://japan.renesas.com/inquiry

csc@renesas.com

改訂記録<revision history,rh>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2007.6.1	—	初版発行

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任は負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。