

YellowSoft

浮動小数演算プロセッサ ラブラシアン

# ***L placian***

---

プログラマーズマニュアル

お使いになる前に必ずお読み下さい

## ソフトウェアの使用条件と保障

### 著作権

本ソフトウェアは著作権法により保護されています。有限会社イエローソフトが本ソフトウェアの著作権を保有します。

したがって、本ソフトウェアを弊社に無断で、売却、譲渡、賃貸またはその他いかなる方法であっても第三者に使用させることはできません。

### 使用の制限

お客様は、本ソフトウェアを1台のコンピュータに限りインストールすることができます。また、バックアップを目的とする複製のみ許可します。

### 保障

弊社は、本ソフトウェアのディスクおよびマニュアルに物理的欠陥がある場合、ご購入日から30日以内であれば無料で交換いたします。

### 免責

弊社は、前項に定める場合を除き、本ソフトウェアの使用、あるいは使用結果に対していかなる責任も負いません。

弊社は、本ソフトウェアによって生成される2次的ソフトウェアに関して、いかなる制限も与えません。またいかなる保障もいたしません。

# もくじ

第 1 章	はじめに-----	4
第 2 章	YCH8 での使用-----	6
第 3 章	他社製 C コンパイラでの使用-----	11
第 4 章	割り込み関数内での浮動小数演算の使用-----	14
第 5 章	浮動小数演算例外処理-----	19
第 6 章	関数リファレンス-----	23

# 第1章 はじめに

## 1.1 概要

ラブラシアンはイエローソフトの開発した IEEE754 準拠浮動小数演算プロセッサです。H8S2367CPU ボードに搭載され、イエローソフトの C コンパイラ YCH8 と共に使用すると浮動小数演算の処理速度は飛躍的に向上します。例えばサイン関数の計算は約 100 倍になります。

他社製コンパイラユーザの方も使用できるライブラリも用意しました。しかし YCH8 の場合はラブラシアン専用のコードをインライン出力するため、最高のパフォーマンスを得ることができます。

## 1.2 何が速くなるか

ラブラシアンを使うことによって、倍精度、単精度の四則演算や各種変換の基本演算、および ANSI-C で定義された数学関数が速くなります。それ以外にテーブルデータを一括して演算、変換するルーチンもあります。

## 1.3 どの程度速くなるか

弊社 YCH8 での使用において実計測した結果を以下に示します。この数値は同じ演算でもオブジェクトの値によって若干ことなる場合があります。

四則演算で概ね 10 倍、数学関数で概ね 100 倍くらい速くなります。

単位[ns]

	演算	FPU 無し	FPU 有り	倍率
倍精度 基本演算	加算	71156	6937	10.3
	乗算	60687	6937	8.7
	除算	140968	7687	18.3
	比較	15062	6437	2.3
	float->double	20781	4562	4.6
	long->double	26281	4562	5.8
	double->float	39125	4437	8.8
	double->long	33906	4437	7.6
倍精度 数学関数	sin	1996875	19781	100.9
	cos	2082593	20656	100.8
	tan	4226406	21531	196.3
	sqrt	974906	16281	59.9
	exp	1610312	21968	73.3
	sinh	3437843	20656	166.4
	cosh	3435968	22843	150.4
	tanh	3524437	24593	143.3

	asin	13117437	39468	332.4
	acos	13188125	40343	326.9
	atan	11628906	25468	456.6
	atan2	11795750	29656	397.8
	log	1605500	25906	62.0
	log10	1797312	27218	66.0
	pow	343062	15218	22.5
単精度 基本演算	加算	17562	4937	3.6
	乗算	17062	4937	3.5
	除算	17843	5375	3.3
	比較	12250	4937	2.5
	long->float	15906	3625	4.4
	float->long	21281	3625	5.9
単精度 数学関数	sin		19375	
	cos		19812	
	tan		21125	
	sqrt		16312	
	exp		22875	
	sinh		18937	
	cosh		23312	
	tanh		25062	
	asin		34687	
	acos		36000	
	atan		23750	
	atan2		26750	
	log		22000	
	log10		22875	
	pow		18000	
拡張命令	2 乗	60812	8656	7.0
	積和	130718	14843	8.8
テーブル演算	倍精度積和	12639218	142343	88.8
	単精度積和	3492906	124375	28.1
	倍精度整数変換	2846312	48531	58.6
	単精度整数変換	1807625	49125	36.8

## 第 2 章 YCH8 での使用

### 2.1 必要なシステム

ラブラシアンを YCH8 で使用するためには YellowIDE Ver.6.46 以上が必要です。Ver.6.45 以下のバージョンをご使用の場合は Ver.6.46 以上にバージョンアップして下さい。バージョンアップは弊社ホームページでダウンロードにより行えます。

### 2.2 H8S2367CPU ボードの使い方を覚える

ラブラシアンは弊社 H8S2367CPU ボードに搭載されています。したがって、まずは H8S2357 の通常の CPU ボードとしての使い方を覚えて下さい。そのためには「はじめの一步」を参照して下さい。「はじめの一步」は開発セットに含まれている他、YellowIDE Ver.6.46 以上のマニュアルフォルダにあります。または弊社ホームページよりダウンロードできます。

### 2.3 使用できる演算

YCH8 を使うことによって以下の浮動小数演算が高速に実行できます。

#### 基本演算

加算、減算、乗算、除算、比較、整数 浮動小数変換、単精度 倍精度変換

これらの基本演算は YCH8 がインライン展開します。したがって、ユーザは特に意識することなく通常の C 言語のプログラムを記述すればラブラシアンが高速実行します。例えば、

```
double d1, d2, d3;  
d3 = d1 + d2;
```

と記述すればこの加算はラブラシアンを使って行われます。

## 数学関数

ANSI-C の規格で定義された数学関数がすべてラプラシアンによって計算されます。さらに倍精度だけではなく、単精度の数学関数も用意しました。

その一覧を以下に示します。

倍精度	単精度
sin	fsin
cos	fcos
tan	ftan
sqrt	fsqrt
exp	fexp
sinh	fsinh
cosh	fcosh
tanh	ftanh
asin	fsin
acos	facos
atan	fatan
atan2	fatan2
log	flog
log10	flog10
pow	fpow

これらの関数の使い方は通常の ANSI-C 数学関数とまったく同じです。

## 拡張演算

基本演算の拡張として以下の 2 命令を追加しました。(倍精度のみ)

2 乗演算	FpuDMul2(x)	x の 2 乗を返します
積和演算	FpuDMulAdd(x, y, z)	$x \times y + z$ を返します

### テーブル演算

テーブルデータに対して一度に演算をします。演算はすべてラプリアンが行いますので高速です。しかも割り込みと合わせて使用すれば演算処理中、CPU に負荷がかかりません。倍精度、単精度両方とも用意しています。

### テーブル積和演算      FpuDMacN      FpuFmacN

テーブルデータの積和演算を行います。

### テーブル変換命令      FpuDTconv16      FpuFTconv16

テーブル上の 16 ビット整数データを浮動小数に変換します。16 ビット整数データをシフトさせてから変換することもできますので H8 内蔵の AD 変換取り込みデータを高速に浮動小数に変換することができます。

### 多項式の計算（倍精度のみ）      FpuDPoly

多項式の係数テーブルを使って関数の計算をします。これを使えば ANSI-C 以外の数学関数を自作することができます。

これらの関数の使い方は関数リファレンスをご覧ください。

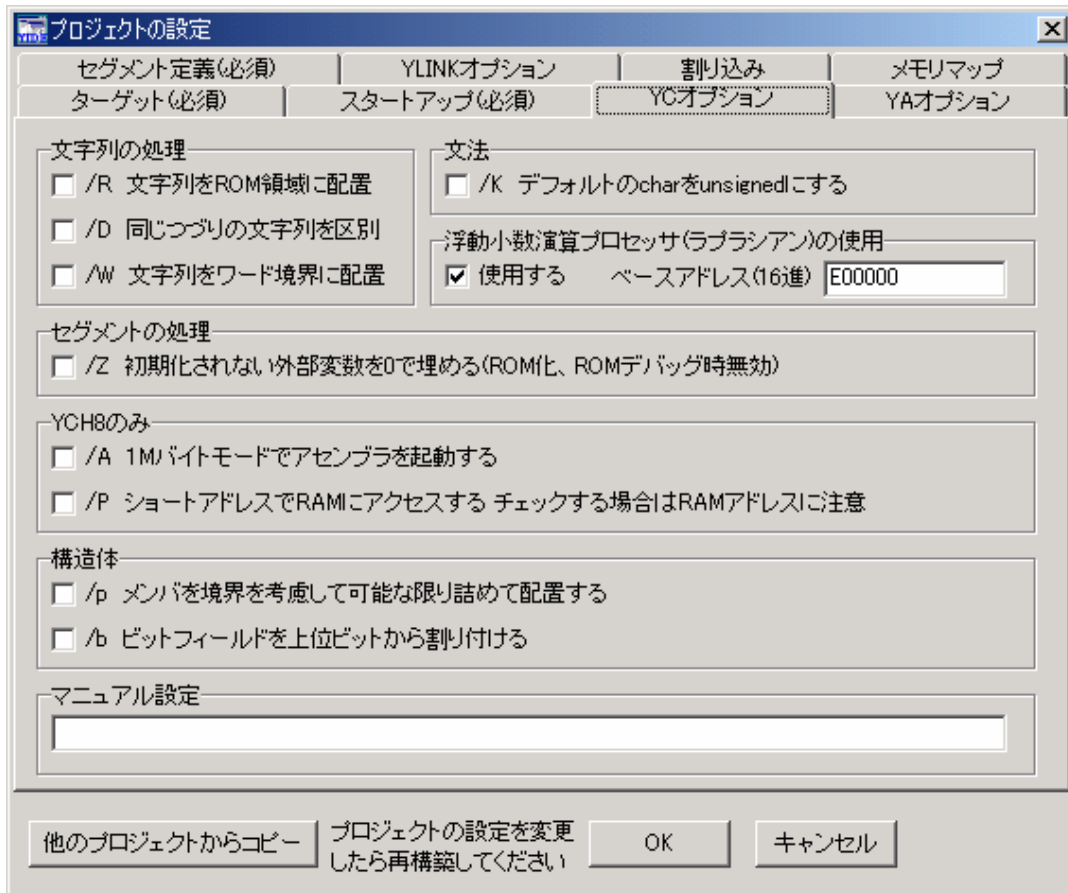
## 2.4 プログラム方法

ラプラシアンを使う場合のプログラミング方法について説明します。重要なのは以下の2点です。

プロジェクトの設定でオプション指定する

プロジェクトの設定画面を開き「YC オプション」のページを開く

「浮動小数演算プロセッサ(ラプラシアン)の使用」で「使用する」をチェックして下さい。またベースアドレスに「E00000」と入力します。



ヘッダファイルのインクルードと初期化関数の呼び出し

浮動小数演算を使う C ソースファイルでは fpu.h というファイルをインクルードして下さい。

また main 関数の最初で FpuInit(0xE00000)を呼び出して下さい。

```
#include <fpu.h>

void main(void)
{
    FpuInit(0xE00000);

    ...
}
```

FpuInit の引数 0xE00000 はラプラシアンが接続されているメモリ空間のアドレスです。

なお、ラプラシアンが接続されているメモリ空間(CS7 空間)のバスの初期設定も行わなければなりません、これはスタートアップルーチン(CS2367.ASM)で記述済みです。ユーザ独自のスタートアップルーチンを使う場合はバスの設定も行って下さい。

以上でラプラシアンを使うための設定およびプログラムの初期化は終わりです。これだけで浮動小数演算が速くなります。

例

```
double d1, d2, d3;

d1 = d2 + d3;           // ラプラシアンが使われ高速になる!!
d1 = sin(d2);          // ラプラシアンが使われ高速になる!!
```

なお、割り込み関数内での浮動小数演算の使用と、拡張関数、テーブル演算関数の使用方法に関しては後の章をご参照下さい。

## 第3章 他社製Cコンパイラでの使用

他社製のCコンパイラでラブラシアンを使う場合は、基本的な四則演算を含めてすべて関数呼び出し形式になります。弊社Cコンパイラに比べて関数呼び出しのオーバーヘッドがあるため、性能は落ちます。特に基本的な四則演算やもともと処理速度が遅くない変換処理などはラブラシアンを使うメリットは僅かになります。しかし、数学関数の演算やテーブル演算などは処理が重いため関数呼び出しのオーバーヘッドは無視できるに等しく、ラブラシアンを使う意義は多いにあります。

### 3.1 必要なファイルのダウンロード

ラブラシアンを使うための関数を集めたCソースファイルとヘッダファイルを弊社ホームページよりダウンロードして下さい。

ソースファイル fpu.c

ヘッダファイル cfpu.h

コンパイルの仕方などは、各Cコンパイラのマニュアルを参照して下さい。

### 3.2 プログラム方法

ヘッダファイルのインクルード

ラブラシン関数を呼び出すためにcfpu.hをインクルードします。

例

```
#include "cfpu.h"
```

## 初期化関数の呼び出し

ラプラシアンを使用する前に、CPU ボードを初期化する関数 CpuBoardInit、およびラプラシンを初期化する関数 FpuInit 呼び出して下さい。

## 例

```
#include "cfpu.h"

void main(void)
{
    CpuBoardInit();
    FpuInit(0xE00000);
    .
    .
    .
}
```

CpuBoardInit 関数は必ず、FpuInit 関数の前に呼び出して下さい。

CpuBoardInit 関数は、main 関数の先頭やスタートアップ関数など、CPU 起動後、一番最初に処理される場所に記述して下さい。

スタックは最初、内蔵 RAM に設定して下さい。なぜなら、CpuBoardInit 処理が終了するまでは外部 RAM が機能していないからです。CpuBoardInit 処理後は、スタックは外部 RAM でも大丈夫です。

FpuInit の引数 0xE00000 は、CS7 メモリ空間の開始アドレスです。

## 参考

CpuBoardInit 関数では以下の処理をしています。

- 1 クロックの設定 システムクロックを原振 16MHz の 2 倍に設定 = 32MHz
- 2 ラプラシアン起動の確認 WAIT 端子が 0 になったことを確認後、1 になるのを確認
- 3 ポートの設定 CS1(RAM)、CS7 (ラプラシアン)、LWR、D0-D15、A0-A18 を有効
- 4 バスの設定 16 ビットバス、CS1(RAM)を 2 ステートアクセス、CS7 (ラプラシアン)を 3 ステートアクセス、ウエイト 0
- 5 シリアル CH0、CH1 の設定 38,400bps 8bit パリティなし、ストップビット 1

### 3.3 プログラムの記述

ラプラシアンを使用し、浮動小数演算を高速に実行するには専用の関数を呼び出す必要があります。例えば加算、sin 関数の計算なら以下ようになります。

例

```
double d1, d2, d3;  
d1 = FpuDAdd(d2, d3);           //d1 = d2 + d3;  
d1 = FpuDSin(d2);              //d1 = sin(d2);
```

加算を次のように記述した場合はラプラシアンは使われません。この場合はコンパイラが出力した CPU 命令によって処理されます。

```
d1 = d2 + d3;
```

テーブル演算をバックグラウンドで処理している場合は、ラプラシアンは使用できませんので、上記のような記述を意図的にします。

## 第 4 章 割り込み関数内での浮動小数演算の使用

メインのルーチンで浮動小数演算を使用し、かつ割り込み関数内でも浮動小数演算を使う場合は、割り込みの発生によって、ラプシアンがメインと割り込みの両方のルーチンからアクセスされる恐れがあります。

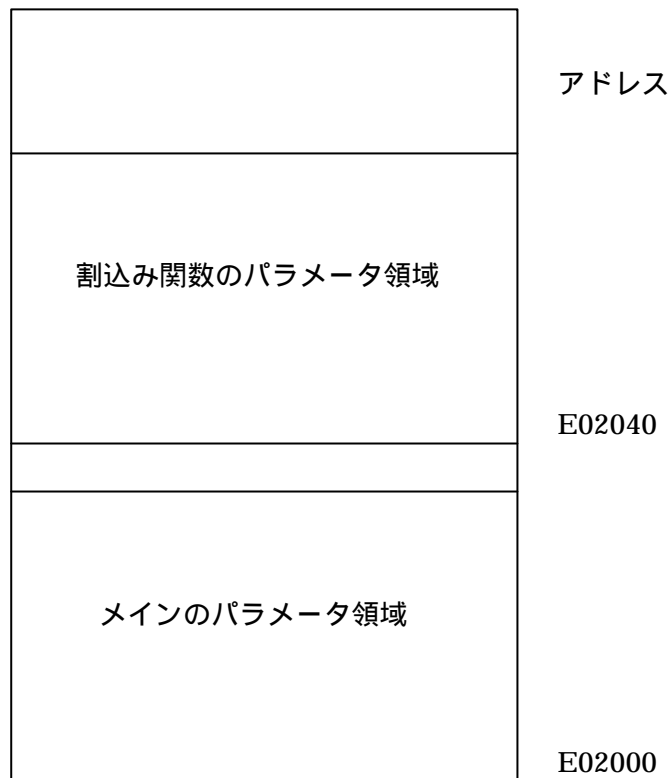
ラプシアンは内部の RAM 領域を使って引数データの受け取り、および演算結果の格納をしています。この領域を「パラメータ領域」と呼びます。

	アドレス
第 6 引数	E02028
第 5 引数	E02020
第 4 引数	E02018
第 3 引数	E02010
第 2 引数	E02008
第 1 引数および戻り値	E02000

今、ラプシアンが一つの演算を終わって、結果を上記のパラメータ領域に格納したとします。CPU がこの演算結果をレジスタにロードする前に割り込みが入り、割り込みルーチン内でラプシアンが次の演算を始めたとします。そうすると演算結果が上書きされてしまうことは容易に想像できるでしょう。

これを避けるために、割り込み関数内では、パラメータ領域を別の領域に移してやらなければなりません。また割り込み関数の最後では元に戻します。

パラメータ領域はラプシアン内部の RAM にあり、RAM の大きさは 4K バイトあります。この 4K バイトのどの領域を使っても構いません。パラメータ領域の最大は 48 バイトですが区切りが良いように 64 バイト RAM 領域をずらしてやればデータが上書きされることはありません。(次ページの図参照)



RAM 領域の先頭は 0xE02000 番地ですから、割り込み関数内では 0xE02040 番地からをパラメータ領域に使用することになります。

割り込み関数の入り口では FpuIntEnter 関数を呼び出します。第 1 引数には、割り込みに入る前のパラメータ領域のアドレスを保存するための変数を確保し、そのアドレス渡して下さい。第二引数には割り込み関数内のパラメータ領域のアドレスの下位 16 ビットを渡します。

関数の出口では FpuIntExit 関数を呼び出します。第 1 引数には、FpuIntEnter の第 1 引数で指定した変数のアドレスを指定します。

```

void interrupt int_func(void)
{
    unsigned short buf;
    FpuIntEnter(&buf, 0x2040);    パラメータ領域アドレスの下位 16 ビットを記述する
    。
    。 割り込み処理
    。
    FpuIntExit(&buf);
}

```

上図の例では変数 buf に割り込みに入る前のパラメータ領域のアドレスが保存されます。そして新しいアドレス 0xE02040 番地になります。

これで割り込みによってメインルーチンでの演算結果が上書きされる恐れがなくなります。

## 多重割り込み

多重割り込みを禁止している場合は、すべての割り込み関数で 0xE02040 番地のパラメータ領域を使用します。つまり、どの割り込み関数でも `FpuIntEnter(&buf, 0x2040);` を呼び出します。しかし、多重割り込みを許可している場合はすべての割り込み関数でパラメータ領域のアドレスを 0x40 番地ずらします。

そこで 2 番目の割り込み関数内では RAM 領域をさらにずらし、0xE02080 番地にします。

```
void interrupt int_func1(void)
{
    unsigned short buf;
    FpuIntEnter(&buf, 0x2040);
    。
    。 割り込み処理
    。
    FpuIntExit(&buf);
}

void interrupt int_func2(void)
{
    unsigned short buf;
    FpuIntEnter(&buf, 0x2080);
    。
    。 割り込み処理
    。
    FpuIntExit(&buf);
}
```

繰り返しになりますが、`FpuIntEnter` や `FpuIntExit` の呼び出しは、その割り込み関数内で浮動小数演算を使う場合だけ記述します。浮動少数演算を使わない場合は、なんら特別なことは必要ありません。

### FpuIntEnter の記述場所

FpuIntEnter は通常、割り込み関数の先頭に記述します。しかし、FpuIntEnter 関数はラプシアンが演算中の場合は演算が終わるまで待つようになっています。

したがって、割り込み関数内で早急に処理しなければならないことは FpuIntEnter の前で処理して下さい。もちろん FpuIntEnter を呼び出すまでは浮動小数演算は使用できません。

```
void interrupt int_func1(void)
{
    unsigned short buf;
    。
    。    早急に処理すべきことを記述
    。    (ただし、浮動小数演算は使用できない)
    FpuIntEnter(&buf, 0x2040);
    。
    。    割り込み処理
    。
    FpuIntExit(&buf);
}
```

テーブル演算をバックグラウンドで実行させている場合で、データ量が多い場合は長い時間 FpuIntEnter によって待たされる可能性がありますので注意して下さい。

## 浮動小数演算使用の判定

割り込み関数内で浮動小数演算を使用するかどうかの判定は注意が必要です。特に YCH8 の場合はインライン展開されますので、一見使用していないと見えても使用している場合もあります。

次の場合は、浮動少数演算が使用されます。つまりラブラシアンが使用されます。

### 1 double、float の変数を定義して演算している

例

```
double d1, d2, d3;  
d1 = d2 + d3;
```

これは分かりやすい例です。浮動小数演算を使用していることは明らかです。

### 2 printf 関数の使用

printf 関数で %e、%f、%g などを使って double、float の数値を表示させる場合は浮動小数演算を使用していることとなります。scanf についても同様です。

%d など整数を表示させる場合や文字列の表示は浮動小数演算を使用しません。

### 3 その他の ANSI 関数

ANSI 関数で浮動少数演算を使用するものは以下の通りです。

sin など math.h で定義された数学関数

atof

strtod

他社製 C コンパイラの場合は、ラブラシンの使用はすべて関数呼出形式ですから、それらの関数を呼び出ししなければ、ラブラシアンを使用したことにはなりません。つまり上記 d1=d2+d3 や、printf、sin など呼び出してもラブラシンが使われるわけではありません。それはコンパイラが生成した CPU の命令によって実行されます。

## 第5章 浮動小数演算例外処理

### 5.1 浮動小数演算例外

浮動小数演算に伴う様々な例外を捕捉することができます。これらの例外は IEE754 で規定された例外で以下のものがあります。

無効演算	INVALID_OPCODE
0 による除算	ZERO_DIV
オーバーフロー	OVER_FLOW
アンダーフロー	UNDER_FLOW
精度落ち	PRECISION_LOST

これらの例外が発生したかどうかを調べることができます。

そのためには、ラブラシアンステータスフラグを調べます。ステータスフラグに関する関数は以下のものがあります。

`unsigned short FpuStatusClear(void)`

ステータスフラグをクリアします。クリアする前の値を返します。

`unsigned short FpuStatusRead(void)`

ステータスフラグの値を返します。

この2つの関数を使って、例外が発生したかどうか調べることができます。

例

```
unsigned short st;
FpuStatusClear();           //ステータスクリア
。
浮動小数演算実行
。
st = FpuStatusRead();       //ステータス読み込み st=FpuStatusClear()でも OK
if (st & INVALID_OPCODE)   puts("無効演算が行われました");
else if (st & ZERO_DIV)    puts("0 による除算が行われました");
else if (st & OVER_FLOW)   puts("オーバーフローがありました");
else if (st & UNDER_FLOW) puts("アンダーフローがありました");
else if (st & PRECISION_LOST) puts("精度落ちがありました");
```

精度落ちは浮動小数演算においては頻繁に発生します。したがってチェックする必要はないでしょう。

## 5.2 割り込みによる例外の捕捉

例外が発生した段階で CPU に対して割り込みを発生させることができます。

ラプラインの初期状態ではこれらの割り込みは発生しないようになっています。割り込みを発生させるには FpuIntMask 関数を呼び出し、引数に例外の識別シンボルを指定します。複数指定する場合は OR 演算でつなげます。

以下の例は無効演算と 0 による除算、オーバーフローの 3 つの例外による割り込みを許可する例です。

例

```
main()
{
    FpuInit(0xE00000);
    FpuStatusClear();
    FpuIntMask(INVALID_OPCODE | ZERO_DIV | OVER_FLOW);
}
```

なお、精度落ち例外ですが、精度落ちは浮動小数演算においては頻繁に発生するので特別な理由がない限り許可しないで下さい。FpuIntMask の呼び出しの前に FpuStatusClear を呼び出します。これにより、既に発生した例外によってセットされているフラグをクリアします。

### 例外を受ける

CPU 側では例外の信号は IRQ6 端子に接続されていますので IRQ6 で割り込みを発生させます。

IRQ6 割り込みを発生させる初期化処理の例を以下に示します。

```
#define IER    *((volatile unsigned short *)0xffff32)
#define ISR    *((volatile unsigned short *)0xffff34)
#define ISCRL  *((volatile unsigned short *)0xfffe1c)

//IRQ6 のセット
IER |= 0x0040;    //IRQ6 ON
ISCRL |= 0x1000; //立ち下がりエッジ

WriteEXR(0);    //割り込み許可
```

また、YellowIDE を使う場合はプロジェクトの設定の「割り込み」でベクタの登録も行って下さい。ベクタ番号は 22 です。(次ページの図を参照)



### 5.3 IRQ6 割り込み関数での処理

割り込み関数内では必ず FpuIntClear 関数を呼び出して下さい。この関数の呼び出しによってラプシアン割り込みフラグがクリアされます。同時にステータスフラグもクリアされます。この関数は FpuStatusClear 関数と似ていますが、ステータスフラグだけでなく、割り込みフラグも同時にクリアする関数です。

また、すべての浮動小数演算例外で同じ IRQ6 割り込みが発生しますので、割り込み関数内で区別する必要があります。そのためには FpuIntClear の戻り値を利用します。例外を区別する例を以下に示します。

```
void interrupt irq6(void)
{
    unsigned short st;
    st = FpuIntClear();           //割り込みフラグ、およびステータスフラグのクリア
    if (st & INVALID_OPCODE) {
        puts("無効演算が行われました");
    }
    else if (st & ZERO_DIV) {
        puts("0 による除算が行われました");
    }
    else if (st & OVER_FLOW) {
        puts("オーバーフローがありました");
    }
}
```

例外を区別しない場合でも FpuIntClear() の呼び出しは必要です。

```
void interrupt irq6(void)
{
    FpuIntClear();           //割り込みフラグ、ステータスフラグのクリア
    puts("何らかの浮動小数演算例外が発生しました");
}
```

## 第6章 関数リファレンス

### 6.1 基本関数（他社製コンパイラのみ）

イエローソフトの C コンパイラ YCH8 の場合はインライン展開されるためこれらの関数の呼び出しは不要です。

#### 倍精度

double FpuDAdd(double x, double y)	加算 $x + y$ を返します
double FpuDSub(double x, double y)	減算 $x - y$ を返します
double FpuDMul(double x, double y)	乗算 $x * y$ を返します
double FpuDDiv(double x, double y)	除算 $x / y$ を返します
unsigned short FpuDCmp(double x, double y)	比較 $x == y$ のとき 0、 $x < y$ のとき 1、 $x > y$ のとき 2 を返します
long FpuDToi(double x)	倍精度を 32 ビット符号付き整数に変換して返します
double FpuSiTod(signed long x)	32 ビット符号付き整数を倍精度に変換して返します
double FpuUiTod(unsigned long x)	32 ビット符号無し整数を倍精度に変換して返します
float FpuDTof(double x)	倍精度を単精度に変換して返します
double FpuFTod(float x)	単精度を倍精度に変換して返します

#### 単精度

float FpuFAdd(float x, float y)	加算 $x + y$ を返します
float FpuFSub(float x, float y)	減算 $x - y$ を返します
float FpuFMul(float x, float y)	乗算 $x * y$ を返します
float FpuFDiv(float x, float y)	除算 $x / y$ を返します
unsigned short FpuFCmp(float x, float y)	比較 $x == y$ のとき 0、 $x < y$ のとき 1、 $x > y$ のとき 2 を返します
long FpuFToi(float x)	単精度を 32 ビット符号付き整数に変換して返します
float FpuSiTof(signed long x)	32 ビット符号付き整数を単精度に変換して返します
float FpuUiTof(unsigned long x)	32 ビット符号無し整数を単精度に変換して返します

## 6.2 拡張関数

倍精度のみ

double FpuDPow2(double x)	2 乗 $x*x$ を返します
double FpuDMulAdd(double x,double y,double z)	積和 $x*y+z$ を返します

## 6.3 数学関数（他社製 C コンパイラのみ）

イエローソフトの C コンパイラ YCH8 の場合は ANSI-C の数学関数がラブラシアン使用になりますので、これらの関数は使用せず、ANSI-C の数学関数を使用して下さい。

引数、戻り値の型はすべて double です

倍精度関数名	単精度関数名	対応する ANSI-C 関数
FpuDSin(x)	FpuFSin(x)	sin(x)
FpuDCos(x)	FpuFCos(x)	cos(x)
FpuDTan(x)	FpuFTan(x)	tan(x)
FpuDAsin(x)	FpuFAsin(x)	asin(x)
FpuDAcos(x)	FpuFAcos(x)	acos(x)
FpuDAtan(x)	FpuFAtan(x)	atan(x)
FpuDAtan2(x,y)	FpuFAtan2(x,y)	atan2(x)
FpuDSqrt(x)	FpuFSqrt(x)	sqrt(x)
FpuDExp(x)	FpuFExp(x)	exp(x)
FpuDSinh(x)	FpuFSinh(x)	sinh(x)
FpuDCosh(x)	FpuFCosh(x)	cosh(x)
FpuDTanh(x)	FpuFTanh(x)	tanh(x)
FpuDLog(x)	FpuFLog(x)	log(x)
FpuDLog10(x)	FpuFLog10(x)	log10(x)
FpuDPow(x,y)	FpuFPow(x,y)	pow(x,y)

## 6.4 テーブル演算関数

### 6.4.1 テーブル積和

倍精度

**double FpuDMacN(double init, double \*adr1, double \*adr2, short n, short n)**

単精度

**float FpuFMacN(float init, float \*adr1, float \*adr2, short n, short n)**

引数   init    初期値  
      adr1    第 1 テーブルアドレス  
      adr2    第 2 テーブルアドレス  
      n       テーブルの要素数  
      start   第 2 テーブルの開始位置

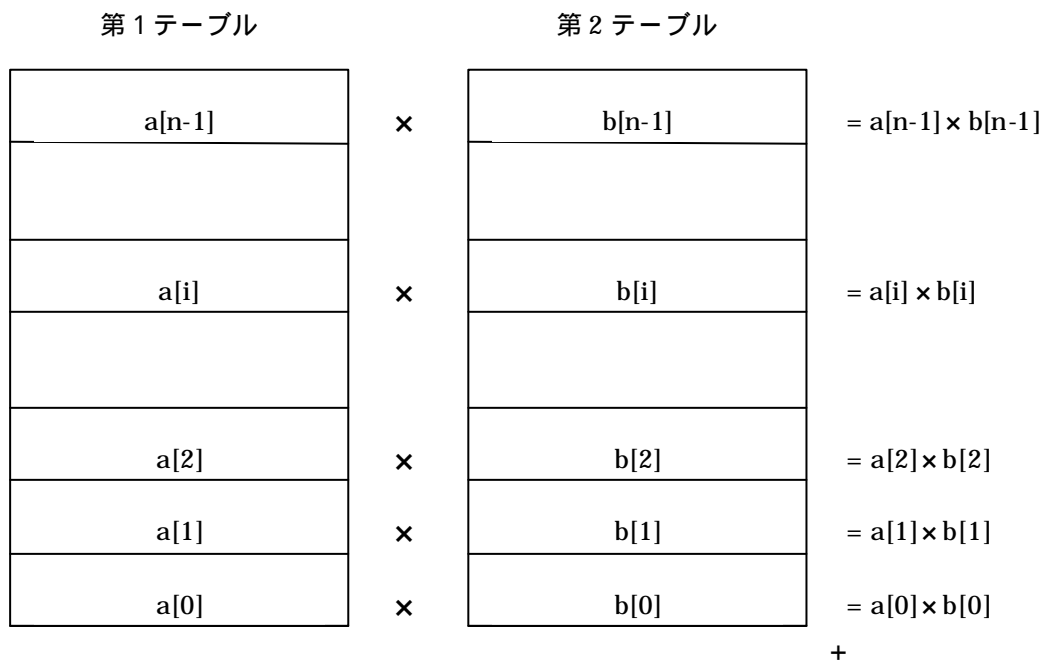
戻り値 演算結果

#### 説明

第 1 テーブルのデータと第 2 テーブルのデータを乗算し、次々と足し込んでいきます。第 1 テーブルのデータを  $a[i]$ 、第 2 テーブルのデータを  $b[i]$  とすれば

$$\text{init} + a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + \dots + a[n-1]*b[n-1]$$

を計算して返します。



なお、第 2 テーブルの計算開始位置はテーブル先頭である必要はありません。その場合はテーブルの最後まで計算するとテーブル先頭に戻ります。第 2 テーブルの開始位置は start で指定します。最初の要素の場合は start=0、例えば 10 番目の要素から計算する場合は start=9 を指定します。

### テーブル領域

データテーブルはラプラシアン内部の RAM にユーザが作成して下さい。ラプラシアン内部の RAM のアドレスは 0xE02000 番地から 0xE02FFF 番地までの 4K バイトです。ただし、先頭付近はラプラシアンと CPU のデータの受け渡しのパラメータ領域です。したがって最低 64 バイト以上空けた領域にします。データ量に余裕があるなら E02100 番地からテーブルを作成すると良いでしょう。ただし、割り込み内で浮動少数演算を使う場合はパラメータ領域が増えますのでその分を考慮しなくてはなりません。「第 4 章 割り込み関数内での浮動小数演算の使用」をご覧ください。

	アドレス
第 2 テーブル	E02800
第 1 テーブル	E02100
割り込み関数パラメータ領域	E02040
メインパラメータ領域	E02000

## 使用例

```
double *table1_adr = (double *)0xE02100;    //第 1 テーブルのアドレス
double *table2_adr = (double *)0xE02800;    //第 2 テーブルのアドレス

void table_muladd(void)
{
    double d;

    //テーブルの作成
    table1_adr[0] = 1.0;    table2_adr[0] = 2.0;
    table1_adr[1] = 2.0;    table2_adr[1] = 5.0;
    table1_adr[2] = -4.0;   table2_adr[2] = 5.0;
    table1_adr[3] = 0.0;    table2_adr[3] = 6.0;
    table1_adr[4] = 2.0;    table2_adr[4] = -2.0;
    table1_adr[5] = 5.0;    table2_adr[5] = 6.0;
    table1_adr[6] = 4.0;    table2_adr[6] = -7.0;

    //計算
    FpuDMacN(1.0, table1_adr, table2_adr, 7, 0);
    printf("結果=%f¥n", d); // -9 を表示
}
```

## 6.4.2 テーブル変換

倍精度

```
void FpuDTconv16(short *adr1, short inc1, double *adr2, short inc2, short n, short shift);
```

単精度

```
void FpuFTconv16(short *adr1, short inc1, float *adr2, short inc2, short n, short shift);
```

引数    adr1    第 1 テーブルアドレス  
         inc1    第 1 テーブル増分  
         adr2    第 2 テーブルアドレス  
         inc2    第 2 テーブル増分  
         n        テーブルの要素数  
         shift   シフト量

戻り値   なし

### 説明

第 1 テーブルの 16 ビット整数データを浮動小数に変換して第 2 テーブルに格納します。第 1 テーブルのアドレスと第 2 テーブルのアドレスが重なっていても構いません。しかしこの場合は完全に一致している場合は問題ありませんが、そうでない場合は第 1 テーブルのデータが変換前に上書きされる可能性がありますので注意して下さい。変換はアドレスの小さい方から行われます。

テーブルデータは、完全に詰まっていなくても構いません。つまりとびとびのデータでも構いません。その場合は増分を inc1、inc2 で指定します。

シフト量(shift)が 0 でない場合は、整数データを右側にシフトした後、浮動少数に変換します。これは AD コンバータの入力データなど、左側に 6 ビットシフトしたデータを扱う場合などに有効です。

第 1 テーブル		第 2 テーブル	
354	E02108	354.000	E02820
563	E02106	563.000	E02818
332	E02104	332.000	E02810
677	E02102	677.000	E02808
456	E02100	456.000	E02800

## テーブル領域

データテーブルはラプラシアン内部の RAM にユーザが作成して下さい。ラプラシアン内部の RAM のアドレスは 0xE02000 番地から 0xE02FFF 番地までの 4K バイトです。ただし、先頭付近はラプラシアンと CPU のデータの受け渡しのパラメータ領域です。したがって最低 64 バイト以上空けた領域にします。データ量に余裕があるなら 0xE02100 番地からテーブルを作成すると良いでしょう。ただし、割り込み内で浮動少数演算を使う場合はパラメータ領域が増えますのでその分を考慮しなくてはなりません。「第 4 章 割り込み関数内での浮動小数演算の使用」をご覧ください。

	アドレス
第 2 テーブル	E02800
第 1 テーブル	E02100
割り込み関数パラメータ領域	E02040
メインパラメータ領域	E02000

## 使用例

前ページの図を変換する例を以下に示します。

第 1 テーブルは 16 ビットデータですから増分は+2 です。第 2 テーブルは double のデータですから増分は+8 になります。

```
void table_conv(void)
{
    FpuDTconv16(0xE02100, 2, 0xE02800, 8, 5, 0);
}
```

### 6.4.3 多項式の計算

---

倍精度のみ

**double FpuDPoly(double x, double \*adr1, short n)**

引数    x            計算する値  
       adr1        多項式の係数テーブルアドレス  
       n            係数テーブルの要素数

戻り値  計算結果

#### 説明

一般に関数  $y=f(x)$  は  $x=0$  の近傍で次のような多項式に展開できます。(テーラー展開)

$$y = a_0 \times x^0 + a_1 \times x^1 + a_2 \times x^2 \cdots$$

ここで  $x^0$  は  $x$  の 0 乗を意味します。

この係数  $a_0, a_1, a_2, \dots$  をテーブルにしたものから、関数の値を計算するのが FpuDPoly 関数です。係数テーブルは次数の高い方から並べます。

例として FpuDPoly 関数を使って、sin 関数を作ってみます。

$y=\sin(x)$  は以下のようにテーラー展開できます。

$$y = a_0 \times x + a_1 \times x^3 + a_2 \times x^5 \dots$$

sin 関数は奇関数のため  $x$  の次数が奇数だけになります。そこであらためて  $x^2=X$  と置き換えてやります。そうすると

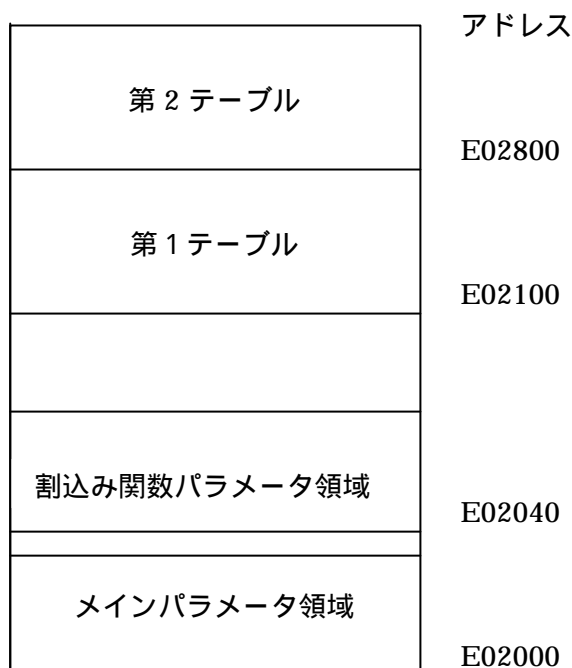
$$y = x \times (a_0 \times X^0 + a_1 \times X^1 + a_2 \times X^2 \dots)$$

このように変形することによって  $X$  の次数が  $0, 1, 2, \dots$  となりますので括弧の中は FpuDPoly 関数で計算出来ます。sin 関数は以下ようになります。

```
double MySin(double x)
{
    return x * FpuDpoly(x*x, table_adr, 8);
}
```

## 係数テーブルの作成

係数テーブルはラプラシアン内部の RAM にユーザが作成して下さい。ラプラシアン内部の RAM のアドレスは 0xE02000 番地から 0xE02FFF 番地までの 4K バイトです。ただし、先頭付近はラプラシアンと CPU のデータの受け渡しのパラメータ領域です。したがって最低 64 バイト以上空けた領域にします。データ量に余裕があるなら 0xE02100 番地からテーブルを作成すると良いでしょう。ただし、割り込み内で浮動少数演算を使う場合はパラメータ領域が増えますのでその分を考慮しなくてはなりません。「第 4 章 割り込み関数内での浮動小数演算の使用」をご覧ください。



sin 関数の係数テーブル作成例を以下に示します。

```
double *table_adr = (double *)0xE02100;
const double sin_table[] = {
    -0.000000000000764723,
    0.000000000160592578,
    -0.000000025052108383,
    0.000002755731921890,
    -0.000198412698412699,
    0.0083333333333333372,
    -0.166666666666666667,
    1.0,
};
void SinTableSet(void) {
    int i;
    for (i = 0; i < 8; i++)
        table1_adr[i] = sin_table[i];
}
```

SinTableSet()の呼出しによってラプラシン内部の RAM に係数テーブルが作成されます。係数データは次数の高い方から並べて下さい。

### 多項式の計算の注意点

一般にテーラー展開が有効なのは  $x$  が 0 付近の値の場合のみです。例えば  $\sin$  の計算では  $x$  の値がにまで大きくなると誤差が顕著になります。これを回避するには  $\sin$  関数の周期性を利用して、 $x$  の値を  $-\pi/4 \sim +\pi/4$  の範囲に切りつめてやる必要があります。詳しくは数学の専門書などをご覧下さい。

## 6.4.4 割り込みによるテーブル演算

---

テーブル演算のうち、積和、変換はデータ量が多くなると時間がかかります。通常のテーブル演算命令は計算が終了するまでリターンしません。したがって、計算が終了するまで待たされることになります。割り込み版のテーブル演算命令は、計算終了まで待たず、即リターンするようになっています。計算終了は割り込みによって知らされます。

割り込み版テーブル演算命令を使用することにより、時間のかかるテーブル演算をバックグラウンドで実行させることができ、CPUの資源を他の処理に振り分けることができます。

しかし、割り込み版テーブル演算では計算が終了するまでラプリアンは占有状態にありますから、その間、他の浮動小数演算を実行することができません。これは十分な注意が必要です。間違えて、浮動小数演算を使用してしまったら誤った答えを返すでしょう。特にタイマー割り込みなど一定間隔で浮動小数演算をしている場合などは注意します。

割り込み版のテーブル演算命令には次のものがあります。

倍精度テーブル積和

**void FpuDMacNInt(double init, double \*adr1, double \*adr2, short n, double \*start)**

単精度テーブル積和

**void FpuFMacNInt(float init, float \*adr1, float \*adr2, short n, float \*start)**

倍精度テーブル変換

**void FpuDTconv16Int(short \*adr1, short inc1, double \*adr2, short inc2, short n, short shift)**

単精度テーブル変換

**void FpuFTconv16Int(short \*adr1, short inc1, float \*adr2, short inc2, short n, short shift)**

通常のテーブル演算関数の関数名に Int がつきます。引数はまったく同じですが、戻り値はありません。

計算が終了すると IRQ6 の割り込みが発生します。積和演算の場合は以下の関数を呼び出せば答えを取り出すことができます。

倍精度

**double FpuDLoad(void)**

単精度

**float FpuFLoad(void)**

IRQ6 割り込み関数の中でこれらの関数を呼び出しても良いし、割り込み関数から戻った後に呼び出しても構いません。いずれにせよ積和演算の計算結果を返します。

テーブル変換の場合は、計算結果はそのままメモリ上に書かれるため、これらの関数を使って答えを

得る必要はありませんが、割り込み発生後にこれらの関数の呼び出しが必要です。なぜなら、これらの関数は割り込み終了処理も含まれているからです。

### IRQ6 割り込みの初期設定

IRQ 割り込みの初期設定の例を以下に示します。

```
#define IER    (*((volatile unsigned short *)0xffff32))
#define ISR    (*((volatile unsigned short *)0xffff34))
#define ISCRL  (*((volatile unsigned short *)0xfffe1c))

//IRQ6 のセット
IER |= 0x0040;          //IRQ6 ON
ISCRL |= 0x1000;       //立ち下がりエッジ

WriteEXR(0);
```

YellowIDE を使う場合はプロジェクトの設定の「割り込み」のページで割り込みベクタの登録を行って下さい。IRQ6 のベクタ番号は 22 です。

## 割り込み版テーブル積和の例

```
#define IER    (*((volatile unsigned short *)0xffff32))
#define ISR    (*((volatile unsigned short *)0xffff34))
#define ISCR   (*((volatile unsigned short *)0xfffe1c))
double *table1_adr = (double *) (0xE02100);
int fpu_flag = 0;
void interrupt irq6(void)
{
    FpuIntClear();
    fpu_flag = 1;
}
main()
{
    int i;
    double d, e;

    FpuInit(0xE00000);
    //IRQ6 のセット
    IER |= 0x0040;          //IRQ6 ON
    ISCR |= 0x1000;        //立ち下がりエッジ
    WriteEXR(0);

    //DMacN のテスト
    puts("FpuDMacN");
    table1_adr[0] = 1.; table2_adr[0] = 2.;    //    2
    table1_adr[1] = 2.; table2_adr[1] = 5.;    //   10
    table1_adr[2] = -4.; table2_adr[2] = 5.;    //  -20
    table1_adr[3] = 0.; table2_adr[3] = 6.;    //    0
    table1_adr[4] = 2.; table2_adr[4] = -2.;    //   -4
    table1_adr[5] = 5.; table2_adr[5] = 6.;    //   30
    table1_adr[6] = 4.; table2_adr[6] = -7.;    // -28    -> -10
    FpuDMacNInt(1.0, table1_adr, table2_adr, 7, 0);
    fpu_flag = 0;
    while (fpu_flag == 0) {
        ;
    }
    d = FpuDLoad();
    printf("FpuFMacN = %f¥n", d);    // -9
}
```

この例では、計算が終了するまで while でループしています。IRQ6 割り込みが発生すると fpu\_flag が 1 になりますのでループを抜けます。実際のプログラムではこの while ループの間に他の処理をさせることができます。ただし、割り込みが発生するまで（計算が終了するまで）他の浮動小数演算は使用できません。

IRQ6 割り込み関数内では FpuIntClear 関数を必ず呼び出して下さい。これによりラプリアンの割り込みフラグがクリアされます。

積和の計算結果は関数 FpuDLoad によって得ることができます。

### 割り込み版テーブル変換の例

積和の例と違う部分だけ掲載します。他はすべて同じです。

```
FpuDTconv16Int(itable1_adr, 2, table2_adr, 8, 100, -1);
fpu_flag = 0;
while (fpu_flag == 0) {
    ;
}
FpuDLoad();
```

積和の例と同じように割り込みが発生するまで while でループしています。割り込みが発生するとループを抜けます。

FpuDLoad 関数は必ず呼び出して下さい。これにより割り込み終了処理が行われます。

## テーブル演算計算中に他の浮動小数演算を使用したい場合

割り込み版のテーブル演算計算中は、つまり割り込みが発生するまでは他の浮動少数演算を使用することはできません。しかし、どうしても使用したい場合は、ラプラシアンではなくコンパイラの生成する CPU 命令を使って浮動小数演算を使用することができます。もちろんラプラシアンを使用しませんので計算速度は遅いです。

他社製 C コンパイラの場合は、ラプラシアンのライブラリ関数を使わず、通常の浮動小数演算命令を記述すれば、(例えば  $d1=d2+d3$ ) ラプラシアンは使用されません。

イエローソフトの C コンパイラの場合は、インライン展開でラプラシアン命令を出力してしまいますので、ラプラシアン命令を部分的に出力させないようにします。そのためには以下の疑似命令で関数を囲みます。

```
#pragma no_fpu
void func(void)
{
    ...
}
#pragma no_fpu_end
```

これによりこの関数内では、ラプラシアン命令は出力されません。

なお、ラプラシアン命令を出力させない単位は関数単位であることに注意して下さい。つまり関数内部で分けることはできません。

また、この疑似命令で囲った関数であっても、この関数が他の関数を呼び出し、その関数で浮動小数演算を使った場合はラプラシアン命令が使われることになります。したがって、その場合はすべての関数をこの疑似命令で囲む必要があります。

また、この疑似命令で囲った関数であっても  $\sin()$  など ANSI の数学関数や `printf` で浮動少数を表示させる場合もラプラシアン命令が使用されることになります。

```
#pragma no_fpu
void func(void)
{
    y = sin(x);                // x ラプラシアンが使用されてしまう。
    ...
    printf("answer = %f¥n", y); // x ラプラシアンが使用されてしまう。
}
#pragma no_fpu_end
```

## 6.5 システム関数

### 6.5.1 CPU ボードの初期化(他社製 C コンパイラのみ)

---

#### **void CpuBoardInit(void)**

引数 なし

戻り値 なし

#### 説明

ラプシアン搭載 CPU ボードを初期化します。CpuBoardInit 関数は、main 関数の先頭やスタートアップ関数など、CPU 起動後、一番最初に処理される場所に記述して下さい。

FpuInit 関数の前に記述して下さい。

イエローソフトの C コンパイラの場合はこの関数の呼び出しは不要です。同じ処理をスタートアップルーチンで行っているからです。

CpuBoardInit 関数では以下の処理をしています。

- 1 クロックの設定 システムクロックを原振 16MHz の 2 倍に設定 = 32MHz
- 2 ラプシアン起動の確認 WAIT 端子が 0 になったことを確認後、1 になるのを確認
- 3 ポートの設定 CS1(RAM)、CS7 (ラプシアン)、LWR、D0-D15、A0-A18 を有効
- 4 バスの設定 16 ビットバス、CS1(RAM)を 2 ステートアクセス、CS7 (ラプシアン)を 3 ステートアクセス、ウエイト 0
- 5 シリアル CH0、CH1 の設定 38,400bps 8bit パリティなし、ストップビット 1

### 6.5.2 ラプシアンの初期化

---

#### **void FpuInit(unsigned long adr)**

引数 adr ラプシンのアドレス

戻り値 なし

#### 説明

ラプシアンを初期化します。main 関数の先頭で一回だけ呼び出します。引数にはラプシアンのアドレスを渡します。通常は 0xE00000 です。

例

```
void main(void) {
    FpuInit(0xE00000);
    ...
}
```



### 6.5.3 浮動小数演算例外割り込みの許可

---

#### **void FpuIntMask(unsigned short mask)**

引数 許可する例外の値

戻り値 なし

#### 説明

浮動少数演算例外で発生させたい割り込みを指定します。各例外のシンボルは以下のものがあります。

無効演算	INVALID_OPCODE
0 による除算	ZERO_DIV
オーバーフロー	OVER_FLOW
アンダーフロー	UNDER_FLOW
精度落ち	PRECISION_LOST

複数ある場合は OR 演算でつなげて下さい。

具体的な使用方法は「第 5 章 浮動小数演算例外の処理」を参照下さい。

#### 例

```
main()
{
    FpuInit(0xE00000);
    FpuStatusClear();
    FpuIntMask(INVALID_OPCODE | ZERO_DIV | OVER_FLOW);
}
```

### 6.5.4 割り込みフラグのクリアとステータスフラグのクリア

---

#### **unsigned short FpuIntClear(void)**

引数 なし

戻り値 クリアする前のステータスフラグの値

#### 説明

浮動小数演算例外による割り込み、および割り込み版テーブル演算の計算終了による割り込み関数内でこの関数を呼び出して下さい。これにより、ラプリアンの割り込みフラグとステータスフラグがクリアされます。

この関数はクリアする前のステータスフラグの値を返しますので、それを調べることによって、どの種類の例外は発生したか調べることができます。ステータスフラグに関しては「6.5.6 ステータ

スフラグのリード」を参照下さい。

### 6.5.5 ステータスフラグのクリア

---

#### **unsigned short FpuStatusClear(void)**

引数 なし

戻り値 クリアする前のステータスフラグの値

#### 説明

ラブラシアンステータスフラグをクリアします。クリアする前のステータスフラグの値を返します。ステータスフラグに関しては「6.5.6 ステータスフラグのリード」を参照下さい。

### 6.5.6 ステータスフラグのリード

---

#### **unsigned short FpuStatusRead(void)**

引数 なし

戻り値 ステータスフラグの値

#### 説明

ステータスフラグの値を返します。

ステータスフラグは、浮動少数演算例外、およびラブラシアンのすべての命令の計算が終了した場合に対応するビットが立ちます。

ビット	説明	シンボル
0	無効演算	INVALID_OPCODE
1	0による除算	ZERO_DIV
2	オーバーフロー	OVER_FLOW
3	アンダーフロー	UNDER_FLOW
4	精度落ち	PRECISION_LOST
5	計算終了	COMMAND_EXIT

各ビットがセットされるタイミングは計算が終了した時点です。計算終了のフラグは必ず立ちます。

### 6.5.7 割り込み関数内で浮動小数演算を使用する場合の前処理

---

#### **void FpuIntEnter(unsigned short \*work, unsigned short bank)**

引数    work    ワークエリアのアドレス  
       bank    バンクアドレス

戻り値   なし

#### 説明

メインのルーチンで、浮動小数演算を使用し、割り込み関数内でも浮動小数演算を使用したい場合に、割り込み関数の先頭でこの関数を呼び出します。具体的な使い方は「第4章 割り込み関数内での浮動小数演算の使用」を参照下さい。

### 6.5.8 割り込み関数内で浮動小数演算を使用する場合の後処理

---

#### **void FpuIntExit(unsigned short \*work)**

引数    work    ワークエリアのアドレス

戻り値   なし

#### 説明

メインのルーチンで、浮動小数演算を使用し、割り込み関数内でも浮動小数演算を使用したい場合に、割り込み関数の出口でこの関数を呼び出します。具体的な使い方は「第4章 割り込み関数内での浮動小数演算の使用」を参照下さい。

### 6.5.9 ラプラシアン計算中の判定

---

#### **int FpuBusy(void)**

引数    なし

戻り値   ラプラシアンが計算中の場合は 1 を返す。それ以外の場合は 0 を返す。

#### 説明

ラプラシアンが計算中かどうかを判定します。計算中なら 1 を返します。割り込み版テーブル演算などで計算終了待ち状態にあるときに、誤って浮動小数演算を実行させてしまわないようにするためにこの関数は利用できます。