

プロファイラについて

Ver. 0.4.0 2010-01-15

目次

1	Profilerの仕組み.....	2
1.1	profile機能.....	2
1.2	コンパイル時の指定	4
1.3	pragmaによる切り替え	6
1.3.1	pragma を入れる場所.....	6
2	TWL-SDKのprofiler.....	7
2.1	関数コールトレース.....	7
2.2	関数コスト計測	7
3	関数コールトレース	8
3.1	トレース記録のしくみ.....	8
3.2	保存される情報	9
3.3	関数コールトレースの2つのモード.....	10
3.4	プログラムへの導入	11
3.5	OS_DumpCallTrace() による表示例.....	13
3.5.1	スタックモードの場合.....	13
3.5.2	ログモードの場合.....	14
3.6	リンク時の指定	15
3.7	スレッド上での動作.....	15
3.8	コスト	15
4	関数コスト計測	16
4.1	コスト計測のしくみ	16
4.2	保存される情報	17
4.3	統計バッファへの変換.....	18
4.4	プログラムへの導入	19
4.5	OS_DumpStatistics() による表示例.....	22
4.6	リンク時の指定	22
4.7	スレッド上での動作.....	22
4.8	コスト	22
5	TWL-SDK以外のProfiler	23
5.1	リンク時の指定.....	23

1 Profilerの仕組み

1.1 profile機能

フリースケール社の C コンパイラ `mwccarm.exe` には `profile` 機能に対応出来る仕組みが用意されています。これは、関数の入り口と出口に特定の関数呼び出しのコードを自動的に挿入するというものです。そして、呼び出される関数のなかで呼び出しに関する記録や統計を取ることでデバッグ等に役立つ `profile` 情報を得ることが出来ます。

`mwccarm.exe` では、`-profile` というオプションをつけてコンパイルすることでこの機能を有効にすることが出来ます。

実際にどのように `profile` のためのコードが入るかをみてみます。

```
u32 test( u32 a )
{
    return a + 3;
}
```

この関数をコンパイルすると通常は以下のようなコードのオブジェクトを出力します。

```
test:
    add    r0, r0, #3          // 3 を足す
    bx     lr
```

単に引数である `r0` に 3 を足しているだけです。(戻り値も `r0` に格納されることになっています。)

次に `profile` 機能をオンにしてコンパイルした場合をみてみます。以下のように、関数の入り口と出口に、それぞれ `__PROFILE_ENTRY` と `__PROFILE_EXIT` の呼び出しが入ります。また、そのためのスタック操作作業などいくつかのコードが入ります。

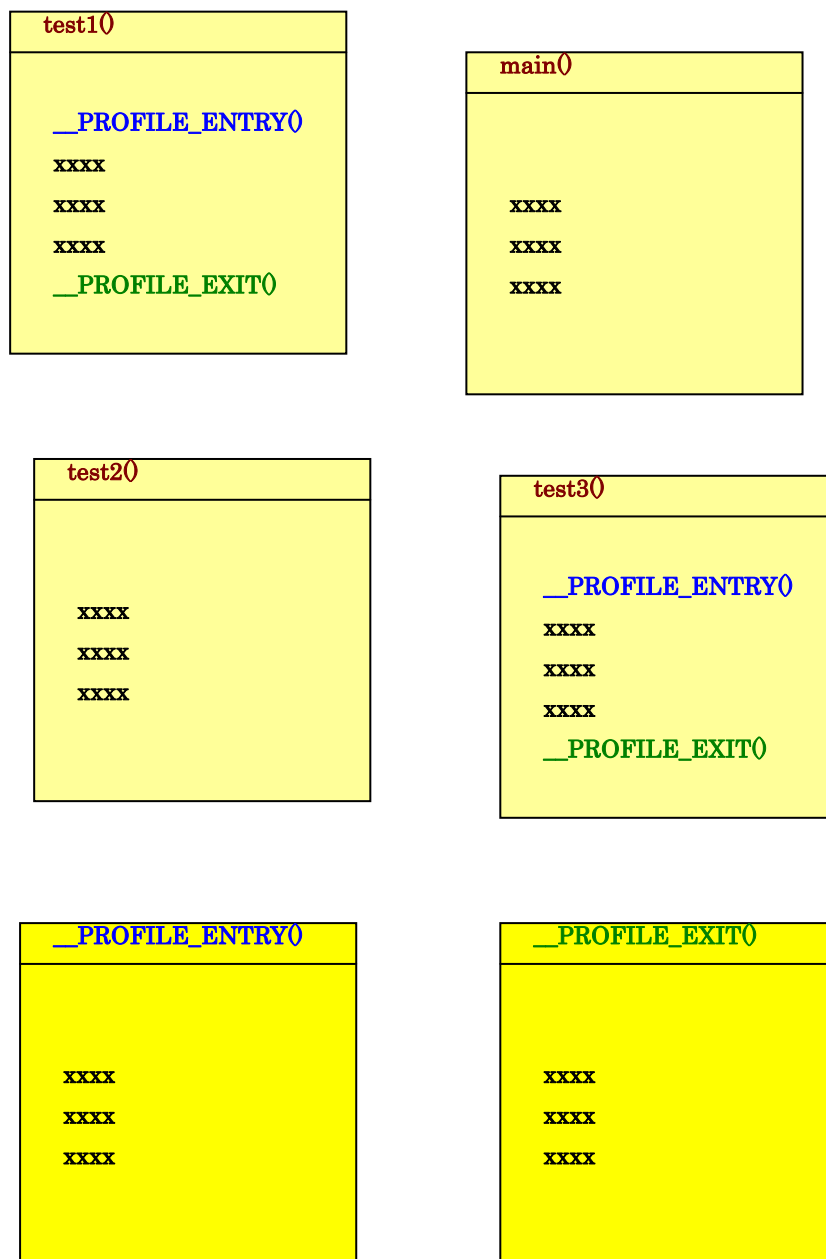
```
test:
    stmfd   sp!, {r0,lr}
    ldr     r0, [pc, #32]      // "test" という文字列へのポインタを r0 に代入
    bl      __PROFILE_ENTRY    // __PROFILE_ENTRY の呼び出し
    ldmfd   sp!, {r0,lr}

    add     r0, r0, #3          // 3 を足す

    sub     sp, sp, #4
    stmfd   sp!, {lr}
    bl      __PROFILE_EXIT     // __PROFILE_EXIT の呼び出し
    ldmfd   sp!, {lr}
    add     sp, sp, #4
    bx      lr
    :
    dcd     xxxx                // "test" という文字列へのポインタ
    :
    xxxx: 74 65 73 74 00        // "test" という文字列
```

`__PROFILE_ENTRY` と `__PROFILE_EXIT` 自体はアプリケーション側で作成しなければなりません。TWL-SDK では、`os_callTrace.c` と `os_functionCost.c` にこれらが定義されており、必要に応じてリンクすることができます。

リンクオブジェクト中に、`__PROFILE_ENTRY` や `__PROFILE_EXIT` を呼び出す関数と呼び出さない関数が混在していても構いません。呼び出さないものは `profile` 対象とならないだけです。なお、コンパイル時には関数単位で呼び出しを入れるかどうかを判断するので、片方しか呼ばない関数は無理やり作らない限り存在しません。



`__PROFILE` 関数が入っているオブジェクトと入っていないオブジェクトが混在することもあります。
(`__PROFILE` 関数自体には `__PROFILE` 関数の呼び出しは入りません)

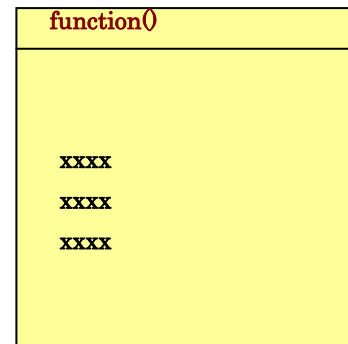
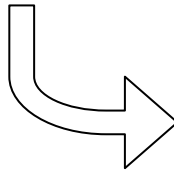
1.2 コンパイル時の指定

TWL-SDK では `TWL_PROFILE` または `NITRO_PROFILE` を定義して `make` すると、C ソースのコンパイル時に `-profile` オプションが付くようになります。 `-profile` をつけてコンパイルしたソース中の関数ではオブジェクトの関数の先頭と末尾に、`__PROFILE_ENTRY`, `__PROFILE_EXIT` への呼び出しが入ります。

先ほど「`TWL_PROFILE` または `NITRO_PROFILE`」と説明しましたが、`NITRO-TWL` では `NITRO_PROFILE` のみが有効だったため、互換性のためにどちらの定義でもよいようにしているためです。`TWL-SDK` で `NITRO ROM` を作成するときに `TWL_PROFILE` を定義してもよいですし、`TWL LIMITED ROM` を作成する際に `NITRO_PROFILE` と指定しても `TWL_PROFILE` と指定しても等価です。

単に `make` とすると

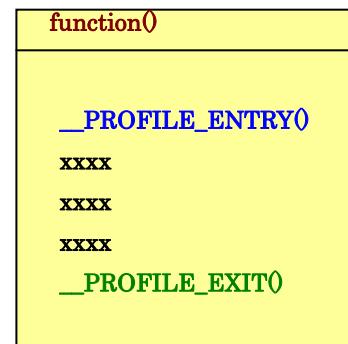
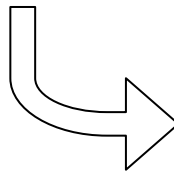
`mwccarm ... test.c`



`make TWL_PROFILE=TRUE` または `make NITRO_PROFILE=TRUE`

とすると

`mwccarm -profile ... test.c`



Makefile の中に書いておいても OK

Makefile

```

:
TWL_PROFILE = TRUE
:
```

1.3 pragmaによる切り替え

Cソースの中で一時的に profile 機能を切り替えるときには、`#pragma` を使用します。

`#pragma profile on` でオンになります。

`#pragma profile off` でオフになります。

`#pragma profile reset` でオンオフ切り替え以前の状態に戻ります。

(例)

```
void test1( void )
{
    :
}

void test2( void )
{
    :
}

#pragma profile off
void test3( void )
{
    :
}

#pragma profile reset

void test4( void )
{
    :
}
```

このソースを `-profile` ありでコンパイルすると、`test10`、`test20`、`test40` が profile 機能オンになります。(もちろん、`-profile` なしならばすべての関数が profile 機能オフとなります。)

1.3.1 pragma を入れる場所

関数が終わるまでに `on` になっていれば、その関数自体は profile 機能がオンとなります。逆に関数の終りの時点で `off` ならば、その関数は profile 機能はオフとなります。通常は関数の外で設定の方がわかりやすいでしょう。

(例)

```
#pragma profile off
void test1( void )
{
    xxxxx();
    xxxxx();
    xxxxx();
    #pragma profile on
}

void test2( void )
{
    xxxxx();
    xxxxx();
    xxxxx();
    #pragma profile off
}
```

この関数は profile on

この関数は profile off

2 TWL-SDKのprofiler

`__PROFILE_ENTRY()` と `__PROFILE_EXIT()` の呼び出しを入れたオブジェクトに対し、適切なそれらの関数 (PROFILE 関数と呼びます) を準備してやることで、TWL-SDK ではデバッグに役立つ以下の機構を提供します。

- ・ 関数コールトレース (OS_CallTrace)
- ・ 関数コスト計測 (OS_FunctionCost)

これらの機能は OS ライブラリとは別のライブラリとして構築されています。具体的には OS ライブラリが `libos.a` (または `libos.thumb.a`) であるのに対し、関数コールトレースライブラリは `libos.CALLTRACE.a` (または `libos.CALLTRACE.thumb.a`)、関数コスト計測 `libos.FUNCTIONCOST.a` (または `libos.FUNCTIONCOST.thumb.a`) となっています。

2.1 関数コールトレース

PROFILE 関数で呼び出された関数を指定のバッファに記録していく機構で、2つのモードがあります。

1 つは `__PROFILE_ENTRY()` で関数の呼び出しを記録しておき、`__PROFILE_EXIT()` でその記録を削除するスタックモードです。ある時点の記録を調べることで、そこがどの関数から(どのような呼び出しを経て)呼び出されているかを知ることが出来ます。

もう1つは、`__PROFILE_ENTRY()` で関数の呼び出しを記録し、`__PROFILE_EXIT()` では何もしないログモードです。記録用のバッファを繰り返し使いますので、記録の最新部分を常に保持しています。これは、どの関数が呼び出されたか(呼び出されていたか)を表示することが出来ます。

このプロファイル機能を有効にするためには、`make` のオプションに `TWL_PROFILE_TYPE=CALLTRACE` (または `NITRO_PROFILE_TYPE=CALLTRACE`) を指定する必要があります。(Makefile の中に指定しても構いません。)

2.2 関数コスト計測

PROFILE 関数の ENTRY 部と EXIT 部で時間計測を行い、その差分で関数の滞在時間を調べることができる機構です。

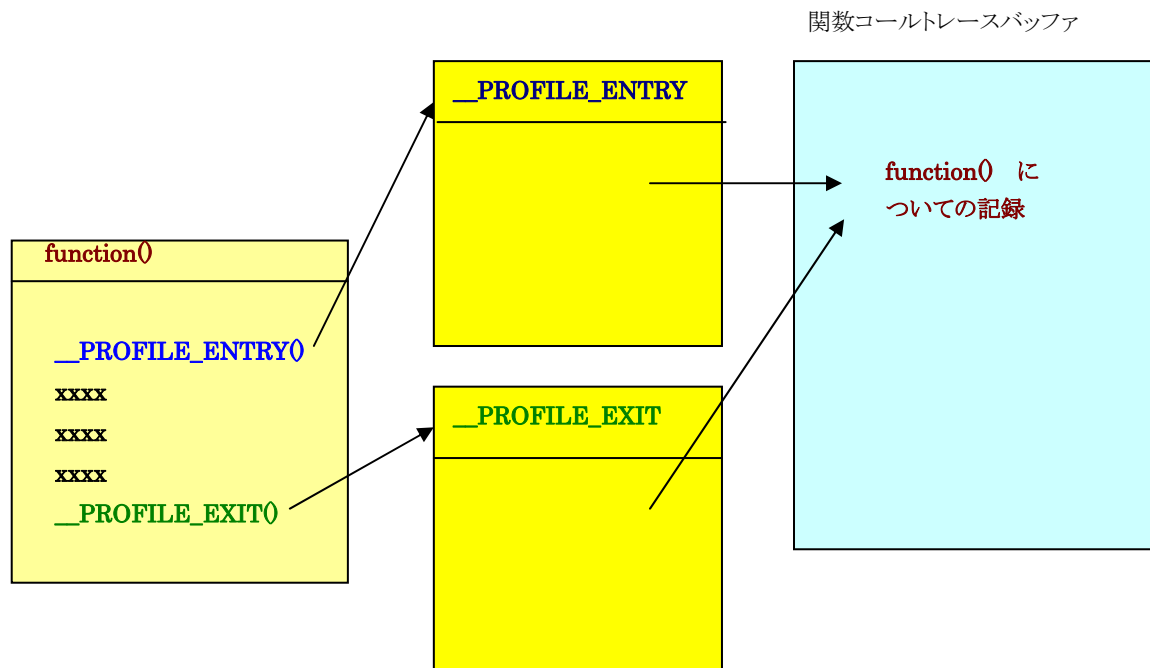
スレッドシステムを使用している場合、スレッドが切り替わって別スレッドが実行されている間の時間を差し引きますので、正確にある関数のコストを比較することが出来ます。また、呼び出し回数も記録しますので、呼び出される頻度を計測するのにも役立ちます。

このプロファイル機能を有効にするためには、`make` のオプションに `TWL_PROFILE_TYPE=FUNCTIONCOST` (または `NITRO_PROFILE_TYPE=FUNCTIONCOST`) を指定する必要があります。(Makefile の中に指定しても構いません。)

3 関数コールトレース

3.1 トレース記録のしくみ

関数コールトレースは以下のように動作します。



`__PROFILE_ENTRY()`

関数コールトレースバッファに、「関数が呼び出された」という記録を行います。具体的には、関数名文字列へのポインタ、関数からの戻りアドレス、引数(オプション)をあわせて記録します。

`__PROFILE_EXIT()`

(スタックモードの場合)

関数コールトレースバッファに、最も最近書き込まれた、「関数が呼び出された」という記録を削除します。

(ログモードの場合)

何も行いません。

3.2 保存される情報

関数コールトレースでは、以下の情報を保存します。

- 関数名文字列へのポインタ
- 関数を呼び出した時点の `lr` レジスタの値
- 関数を呼び出した時点の `r0` レジスタの値 (オプション)
- 関数を呼び出した時点の `r1` レジスタの値 (オプション)
- 関数を呼び出した時点の `r2` レジスタの値 (オプション)
- 関数を呼び出した時点の `r3` レジスタの値 (オプション)

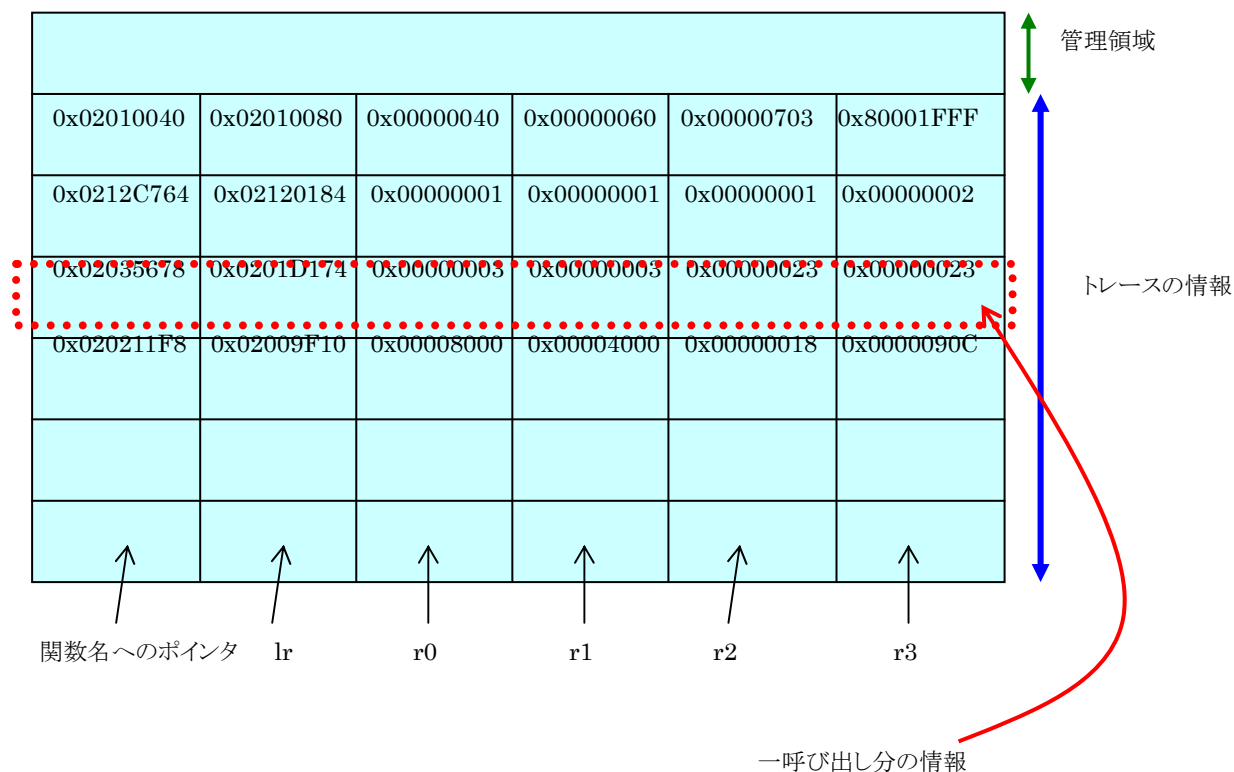
`lr` レジスタは、関数から戻るべきアドレスが格納されています。逆にいえば、`lr` レジスタの値が判れば、その関数がどのアドレスから呼び出されたかを知ることが出来ます。

`r0`～`r3` レジスタは、引数を伴う関数の、引数の値の引渡しに使用されます。従って、どのような引数でその関数が呼び出されたかを知ることが出来ます。ただし引数の引渡しに使用されないレジスタの値には余り意味はありません。

`r0`～`r3` の保存についてはオプションになっています。これらを保存する場合、1つについて 4 バイトの保存領域が必要になります。すべてのトレース情報についてその分の情報が必要になりますのでバッファを確保する際のサイズに注意が必要です。

ユーザが用意したバッファは以下のように使用されます。

関数コールトレースバッファ



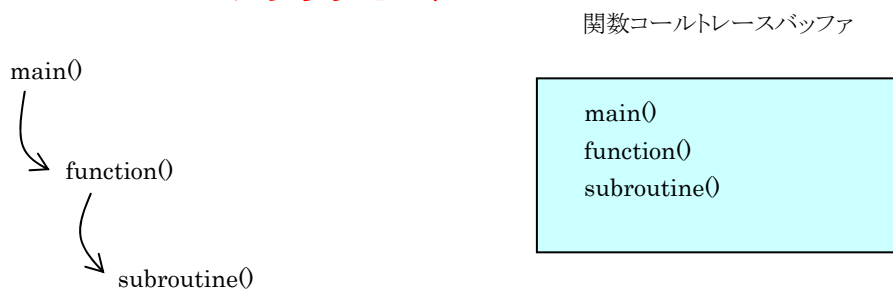
この図では、**r0～r3** が保存されていますので、一呼び出しあたり、24 バイトの情報領域が必要になることがわかります。もしも **r0～r3** を保存しないのなら、一呼び出しで必要となるバッファサイズは 8 バイトです。

「管理領域」には、バッファのどの部分が現在使用されているかや、バッファの上限がどこかなどの情報が格納されています。

3.3 関数コールトレースの2つのモード

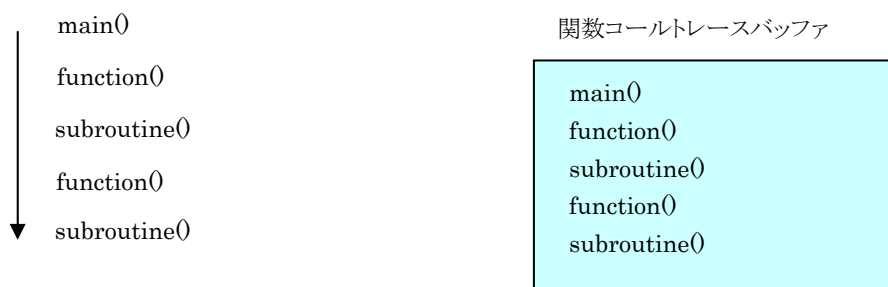
関数コールトレースには2つのモードが存在します。スタックモードとログモードですが、スタックモードでは `__PROFILE_ENTRY()` で情報を保存し、`__PROFILE_EXIT()` で情報を破棄します。ログモードでは `__PROFILE()` での情報破棄はしません。また、情報を保存する際に、領域を使いまわして古い情報を捨てていきます。バッファに記録された情報から、それぞれ次のようなことがわかります。

スタックモード



関数コールトレースバッファを調べれば、ある時点での関数呼び出しについての情報を得ることが出来ます。例えば上の図では、**main()**が **function()**を呼び出し、**function()** が **subroutine()**を呼び出した、ということがわかります。

ログモード



関数コールトレースバッファを調べれば、そこまでに呼び出された関数の情報を得ることが出来ます。例えば上の図では、**main()**、**function()**、**subroutine()**、**function()**、**subroutine()**、と入ったことがわかります。

3.4 プログラムへの導入

基本的にはプログラムの最初に関数コールトレースバッファの初期化宣言を行うことで、関数コールトレースの記録が開始されます。スタックモードの場合、その性質上、初期化を行う関数が最も呼び出しの上位(呼び出しの階層構造上、どこからも呼び出されない)であることが望ましいです。ログモードの場合は余りそうした考慮は必要ありません。

```
// 関数コールトレース初期化
void OS_InitCallTrace( void* buf, u32 size, OSCallTraceMode mode );

    buf    関数コールトレースバッファ
    size    バッファのサイズ
    mode    スタックモードかログモードか
```

関数コールトレースバッファには前述の通り、そのバッファを管理するための情報と、実際のトレースの情報とが格納されます。`mode` は `OSCallTraceMode` 型で、`OS_CALLTRACE_STACK` (スタックモード)、`OS_CALLTRACE_LOG` (ログモード) の何れかを指定します。

あるバッファサイズから、そのバッファ中に何呼び出し分の情報を格納することが出来るかは以下の関数で求めることが出来ます。

```
// バッファサイズから格納出来るトレースの情報数を求める
int OS_CalcCallTraceLines( u32 size )

    size    バッファのサイズ
    戻り値   確保可能なライン数(トレースの情報数)
```

また、ある呼び出し分の関数バッファを宣言するために、バッファサイズをどれだけ確保すればよいかは以下の関数で求めることが出来ます。

```
// 格納できるトレースの情報数からバッファサイズを求める
u32 OS_CalcCallTraceBufferSize( int lines );

    lines    ライン数(トレースの情報数)
    戻り値   必要なバッファのサイズ
```

情報を表示するのは以下の関数です。表示内容については後述します。

```
// 関数コールトレース表示
u32 OS_DumpCallTraceBufferSize( void );
```

一時的に情報の保存を止めたり、それを戻したり等は次の関数で操作可能です。これは、`__PROFILE_ENTRY()` や `__PROFILE_EXIT()` が呼び出されても何もしないで戻るようにするというものです。従って、スタックモードを使用している場合、`__PROFILE` 関数の機能を停止するタイミングによってはバッファ内の情報が破壊されてしまいます。

```
// 関数コールトレース許可／禁止／復帰
BOOL OS_EnableCallTrace( void );
BOOL OS_DisableCallTrace( void );
BOOL OS_RestoreCallTrace( BOOL enable );

enable 許可するか(TRUE)、禁止するか(FALSE)
戻り値 その関数が呼ばれる以前の状態。許可(TRUE)／禁止(FALSE)
```

ログモードで使用している際に、バッファの内容を全て消去したい場合、以下の関数を呼びます。
(スタックモードでも使用できますが、その仕組みを十分理解した上で使用してください。)

```
// 関数コールトレースバッファのクリア
void OS_ClearCallTraceBuffer ( void );
```

以下は実際にプログラム中に記述した例です。
スタックモードの場合：

```
#define TRACEBUFSIZE 0x300
u32 traceBuffer[ TRACEBUFSIZE / sizeof(u32) ];

void NitroMain( void )
{
    OS_Init();

    //---- init callTrace (STACK mode)
    OS_InitCallTrace( &traceBuffer, TRACEBUFSIZE, OS_CALLTRACE_STACK );
    :
}

void function()
{
    //---- display callTrace
    OS_DumpCallTrace(); // この時点での関数呼び出しの状態を表示
}
```

ログモードの場合：

```
#define TRACEBUFSIZE 0x300
u32 traceBuffer[ TRACEBUFSIZE / sizeof(u32) ];

void NitroMain( void )
{
    OS_Init();

    //---- init callTrace (LOG mode)
    OS_InitCallTrace( &traceBuffer, TRACEBUFSIZE, OS_CALLTRACE_LOG );
    : // ここのログを取りたい

    //---- display callTrace
    OS_DumpCallTrace();
}
```

3.5 OS_DumpCallTrace() による表示例

3.5.1 スタックモードの場合

OS_DumpCallTrace() で、以下のような表示を得ることが出来ます。

```
OS_DumpCallTrace: lr=0200434c
test3: lr=02004390, r0=00000103, r1=00000080, r2=00000080, r3=2000001f
test2: lr=020043c4, r0=00000101, r1=00000080, r2=00000080, r3=2000001f
test1: lr=02004254, r0=00000100, r1=00000080, r2=00000080, r3=2000001f
```

この結果から、OS_DumpCallTrace()を呼び出した場所は、OS_DumpCallTrace() を呼び出したときの lr レジスタの位置の直前なので、0x0200434c の前だということがわかります。また、test1()が test2()を呼び出し、test2()が test3()を呼び出していることもわかります。また、例えば test3() から戻る位置は 0x2004390 の前ということがわかります。更に、test3() を呼び出した時、r0 は 0x103、r1 は 0x80、r2 は 0x80、r3 は 0x2000001f だったことが示されていますので、test3()が引数を取る関数ならば、それらを当てはめていけば関数を呼び出した時の引数を知ることが出来ます。その他の関数でも同様に解析することが可能です。

(注) 上で「test1() が test2() を呼び出した」などの記述をしましたが、これらは、この実行ファイルがすべてのオブジェクトで profile 機能を有効にしてコンパイルされていることを前提にしています。従って、例えば、test1() が profile 機能が有効でない test4()を呼び出し、test4()が profile 機能が有効となっている test2()を呼び出した場合、見た目に test1()のすぐ上の行が test2() と表示されることになります。

上の表示は、以下のプログラムから出力されたものです。

```
int test1( int a){ return test2( a + 1); }
int test2( int a){ return test3( a + 2); }
int test3( int a){ OS_DumpCallTrace(); return a + 4; }

void NitroMain( void )
{
    OS_Init();
    :
    OS_InitCallTrace( &buffer, BUFFERSIZE, OS_CALLTRACE_STACK );
    (void) test1( 0x100 );
    :
}
```

3.5.2 ログモードの場合

OS_DumpCallTrace() で、以下のような表示を得ることが出来ます。

```
test3: lr=020043a0, r0=00000103, r1=00000080, r2=00000080, r3=2000001f
test2: lr=020043d4, r0=00000101, r1=00000080, r2=00000080, r3=2000001f
test1: lr=0200423c, r0=00000100, r1=00000080, r2=00000080, r3=2000001f
test3: lr=020043a0, r0=00000203, r1=00000080, r2=00000080, r3=2000001f
test2: lr=020043d4, r0=00000201, r1=00000080, r2=00000080, r3=2000001f
test1: lr=02004244, r0=00000200, r1=00000080, r2=00000080, r3=2000001f
```

先に表示したもの(この結果の上の行)ほど新しい情報なので、この結果から、**profile** 機能が有効になっている関数のうち、**test1**、**test2**、**test3**、**test1**、**test2**、**test3** の順序で呼び出しが行われたということを知ることが出来ます。呼び出し時の戻りアドレスや引数などの情報は、その時点の **lr** レジスタや **r0**～**r4** レジスタから知ることが出来ます。

test1、**test2**、**test3** の表示をみると、**test2** と **test3** がインデントして表示されています。これは、**test2** は **test1** の **__PROFILE_EXIT()** が来る前に呼び出されたものであり、**test3** は **test2** の **__PROFILE_EXIT()** が来る前に呼び出されたことから行っているもので、関数間の呼び出し関係の目安にすることが出来ます。

上の表示は、以下のプログラムから出力されたものです。

```
int test1( int a ){ return test2( a + 1 ); }
int test2( int a ){ return test3( a + 2 ); }
int test3( int a ){ return a + 4; }

void NitroMain( void )
{
    OS_Init();
    :
    OS_InitCallTrace( &buffer, BUFFERSIZE, OS_CALLTRACE_LOG );
    (void) test1( 100 );
    (void) test1( 100 );
    OS_DumpCallTrace();
}
```

3.6 リンク時の指定

関数コールトレースの機能を有効にするためには `make` のオプションに `TWL_PROFILE_TYPE=CALLTRACE` (または `NITRO_PROFILE_TYPE=CALLTRACE`) を指定する必要があります。この指定で、リンク時に `libos.CALLTRACE.a` (または `libos.CALLTRACE.thumb.a`) が含まれるようになります。`Makefile` の中に記述しても構いません。

3.7 スレッド上での動作

関数コールトレースの情報は、スレッドシステムを用いている場合はスレッドごとに独立しています。従って、あるバッファを `OS_InitCallTrace0` で初期化宣言した場合、それを行ったスレッド以外のスレッドがそのバッファにトレースの情報を付加していく事はありません。また、`OS_EnableCallTrace0` 等の状態設定もスレッドごとに独立しています。

同じバッファを異なるスレッドで `OS_InitCallTrace0` 宣言することは避けてください。

3.8 コスト

関数呼び出しの度に関数コールの情報をバッファに保存する処理が入るので、普通に動作させる以上に関数呼び出しのコストがかかります。関数の入り口と出口の `__PROFILE_ENTRY/EXIT` を必ず通過しなければならないので、コンパイル時の最適化についても何も制約がないときに比べると期待は出来ません。また、関数名へのポインタをバッファに保存する都合上、関数名文字列がメモリ上に置かれることになり、容量のコストもかかります。

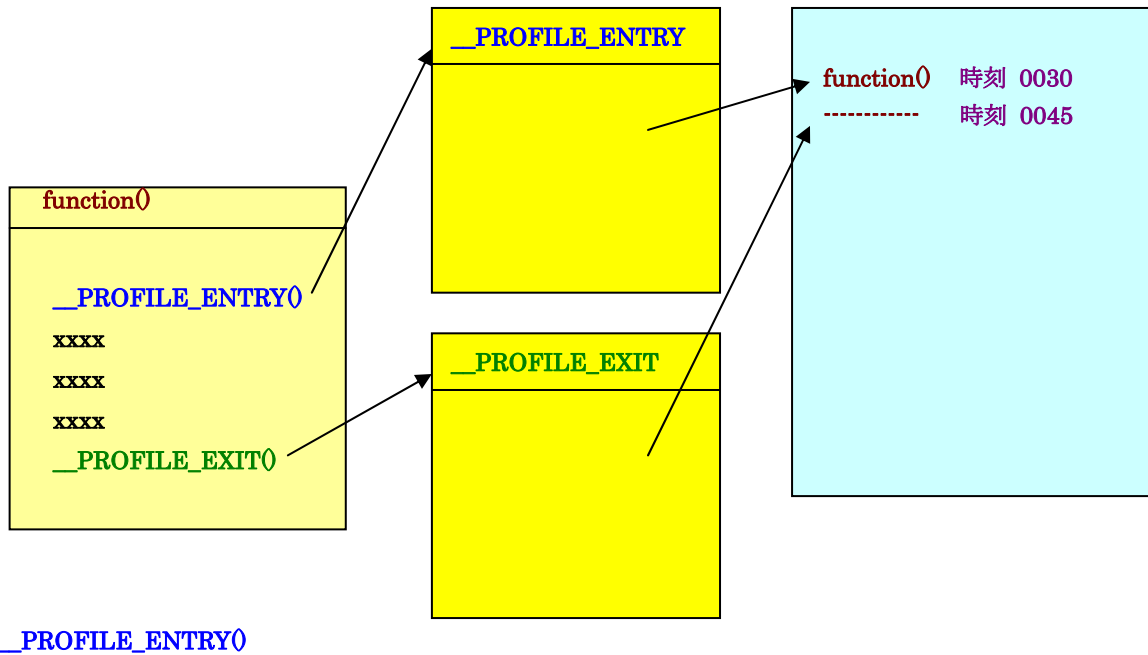
動作コストについては、スレッドの有無や保存する情報やモードなどで変わってきますが、`__PROFILE_ENTRY0` で 60～70 命令、`__PROFILE_EXIT0` で 20～40 命令余計に通過します。

4 関数コスト計測

4.1 コスト計測のしくみ

関数コスト計測では、以下のように「関数コスト計測バッファ」と「関数コスト統計バッファ」の2つを用います。

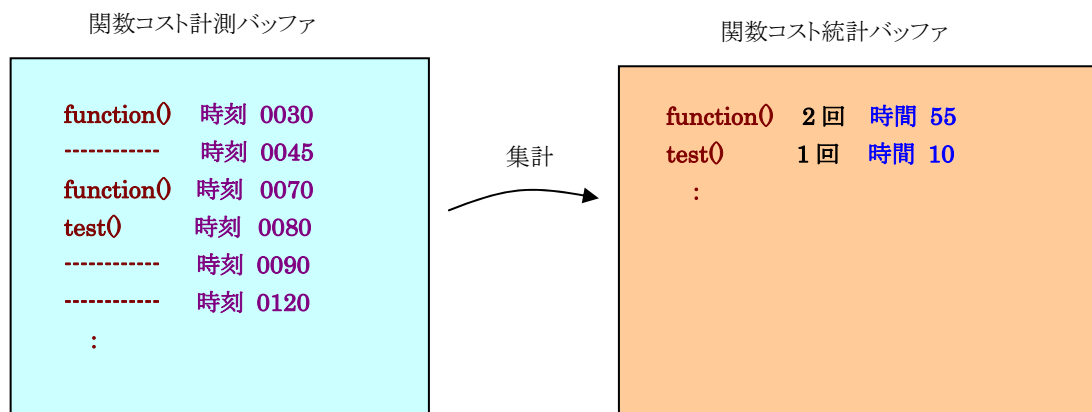
関数コスト計測バッファ



ユーザが指定したコスト計測バッファに、関数名文字列へのポインタと現在の時刻を記録します。

`__PROFILE_EXIT()`

`__PROFILE_EXIT()` が書き込んだというタグと、現在の時刻を記録します。



4.2 保存される情報

関数コスト計測では以下の情報を記録します。

__PROFILE_ENTRY では、

- 関数名文字列へのポインタ
- 現在の時刻。OS_GetTickLo0 の値

__PROFILE_EXIT では、

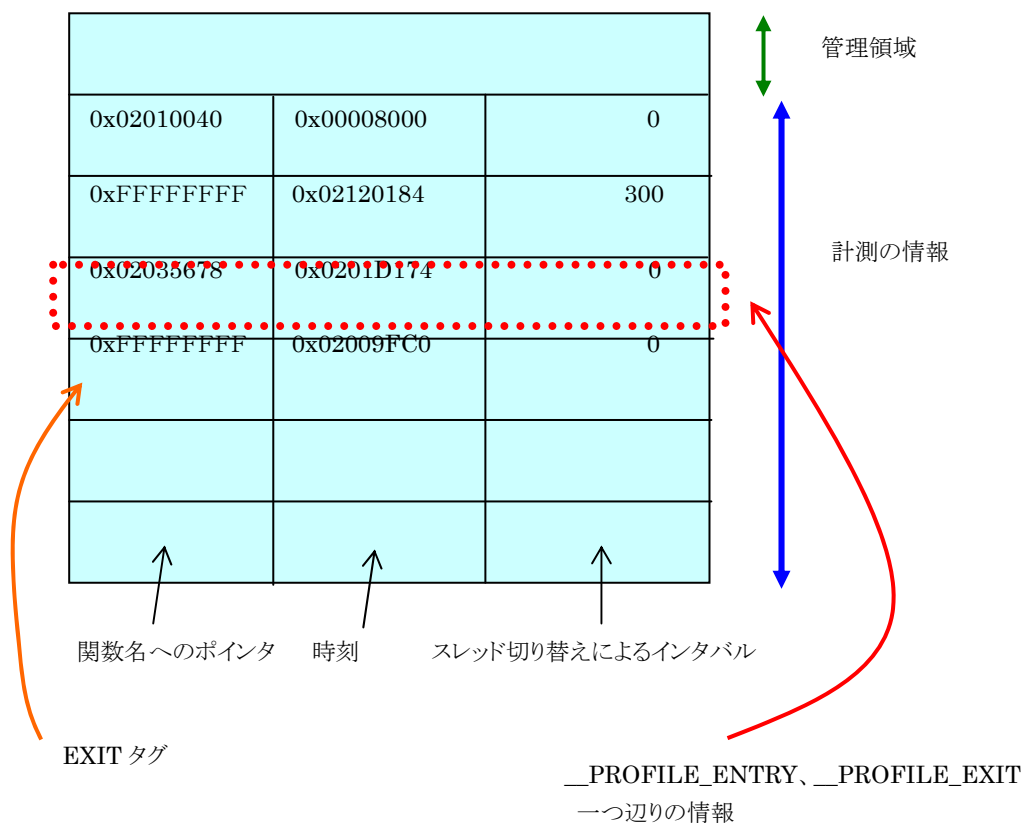
- __PROFILE_ENTRY でポインタを保存していた領域に特別な値(EXIT タグ値と呼びます)
- 現在の時刻。OS_GetTickLo0 の値。
- 必要ならスレッド切り替えによるインタバル (オプション)

ここでいう現在の時刻は OS_GetTickLo0 で取得出来る値です。OS の Tick 機能自体は 64bit の値を持っていますが、関数呼び出しに際してはその下位半分のみを調べることで十分と考えられるので 32bit 値で管理します。

関数名文字列へのポインタには、__PROFILE_ENTRY では文字列へのポインタを、__PROFILE_EXIT では、ポインタと区別できる特別な値(EXIT タグと呼びます)を確保します。

スレッド切り替えによるインタバルは、スレッドが切り替わり、別の関数に制御が渡っている間を差し引きするための値です。

関数コスト計測バッファ



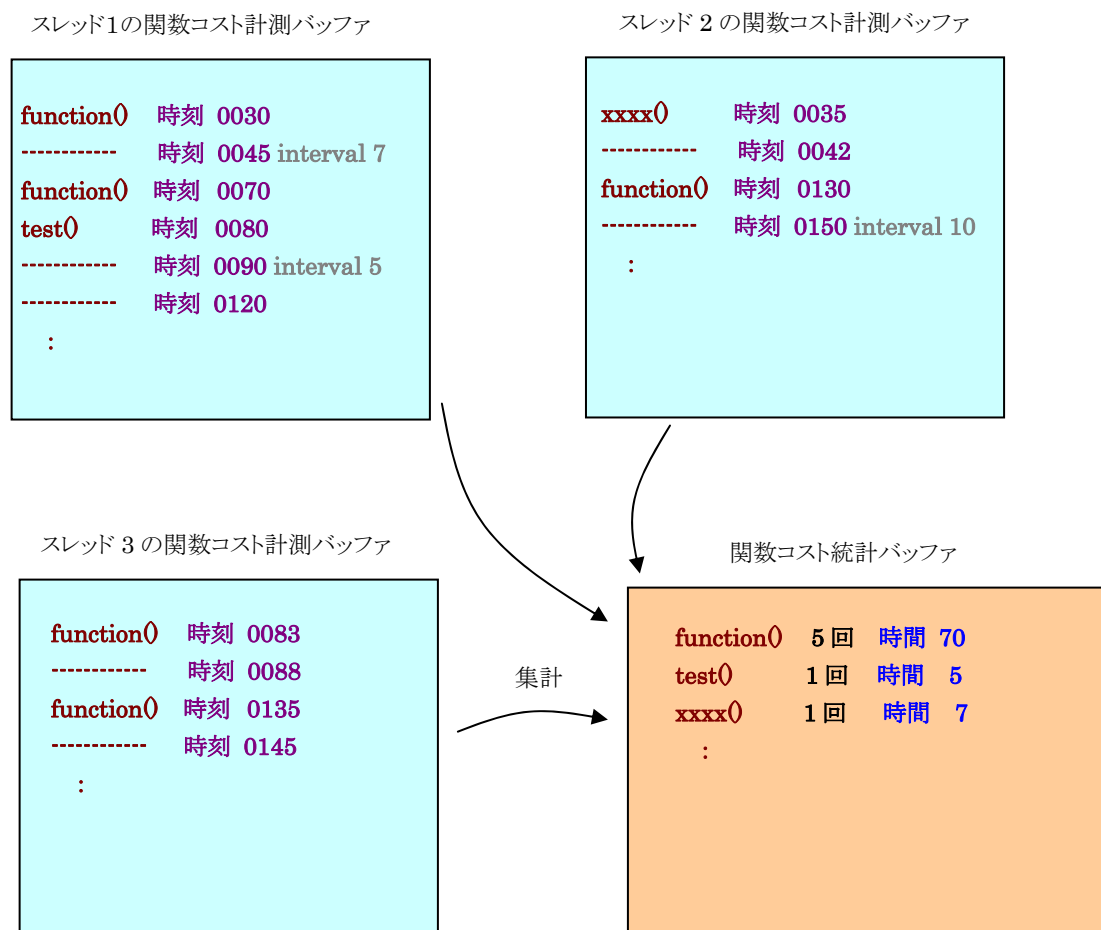
「管理領域」には、バッファのどの部分が現在使用されているかや、バッファの上限がどこか、またスレッド切り替えによるインタバル計測のためのカウンタ値などの情報が格納されています。

4.3 統計バッファへの変換

関数コストの計測データだけでは、コストの情報を得ることは困難です。この計測データを統計バッファデータとして集計してやる必要があります。

集計は、関数の呼び出し情報と関数の EXIT タグとを結びつけて、呼び出し回数と関数内での滞留時間を加算していきます。スレッドが切り替わることによって別スレッドに制御が移ったときには、その間の時間はインタバル値として EXIT タグの行に記録されますので、その分を考慮して計算を行います。

集計は明示的に行う必要があります。集計を行うと、関数コスト計測バッファの内容がクリアされます。関数コスト計測バッファが溢れないうちに集計バッファに反映させることを繰り返すことで、長時間の計測を正確に行うことが出来ます。また、集計バッファは複数のスレッドで共通に使用することも可能です。(ただし、集計中に別のスレッドで集計を行うことは避けてください。)



統計バッファに複数の計測結果を反映させることも出来る

4.4 プログラムへの導入

関数コスト計測バッファの初期化宣言を行うことで、関数コスト計測ための記録が開始されます。

なお、コスト計測には OS の Tick システムを使用しますので OS_InitTick0 を先に呼ぶ必要があります。

```
// 関数コスト計測初期化
void OS_InitFunctionCost( void* buf, u32 size );

    buf    関数コスト計測バッファ
    size   バッファのサイズ(byte)
```

バッファは前述の通り、そのバッファを管理するための情報と、実際の時刻情報とが格納されます。

あるバッファサイズから、そのバッファ中に何呼び出し分の情報を格納することが出来るかは以下の関数で求めることが出来ます。

```
// バッファサイズから格納出来る情報数を求める
int OS_CalcFunctionCostLines( u32 size )

    size   バッファのサイズ(byte)
    戻り値  確保可能なライン数(コスト計測の情報数)
```

また、ある呼び出し分のバッファを宣言するために、バッファサイズをどれだけ確保すればよいかは以下の関数で求めることが出来ます。

```
// 格納できる計測情報数からバッファサイズを求める
u32 OS_CalcFunctionCostBufferSize( int lines );

    lines   ライン数(コスト計測の情報数)
    戻り値   必要なバッファのサイズ(byte)
```

コストの統計を取るためのバッファは以下の関数で初期化しておく必要があります。

```
// 関数コスト統計バッファの初期化
void OS_InitStatistics( void* statBuf, u32 size );

    statBuf バッファ
    size     バッファのサイズ(byte)
```

そして、この関数コスト統計バッファに対し、以下の関数で関数コスト計測バッファの内容を反映していきます。

```
// 関数コスト集計
OS_CalcStatistics( void* statBuf );

    StatBuf  統計バッファ
```

OS_CalcStatistics0 を呼ぶと、現在の関数コスト計測バッファの内容はクリアされます。

集計結果を表示するのは以下の関数です。表示内容については後述します。

```
// 関数コスト集計表示
OS_DumpStatistics( void* statBuf);

StatBuf 統計バッファ
```

一時的に情報の保存を止めたり、それを戻したり等は次の関数で操作可能です。これは、__PROFILE_ENTRY() や __PROFILE_EXIT() が呼び出されても何もしないで戻るようにするというものです。__PROFILE_ENTRY()で記録される情報と、__PROFILE_EXIT() で記録される情報の一方のみがバッファに書き込まれるような切り替えを行うとコスト計測のデータが崩れることになりますので注意が必要です。

```
// 関数コスト計測許可／禁止／復帰
BOOL OS_EnableFunctionCost( void );
BOOL OS_DisableFunctionCost( void );
BOOL OS_RestoreFunctionCost( BOOL enable );

enable 許可するか(TRUE)、禁止するか(FALSE)
戻り値 その関数が呼ばれる以前の状態。許可(TRUE)／禁止(FALSE)
```

明示的に関数コスト計測バッファの内容をクリアしたい場合には以下の関数を呼びます。

```
// 関数コスト計測バッファのクリア
void OS_ClearFunctionCostBuffer ( void );
```

以下は実際にプログラム中に記述した例です。

```
#define COSTSIZE 0x3000
#define STATSIZE 0x300

u32 CostBuffer[ COSTSIZE / sizeof(u32) ]
u32 StatBuffer[ STATSIZE / sizeof(u32) ];

void NitroMain( void )
{
    OS_Init();
    OS_InitTick();

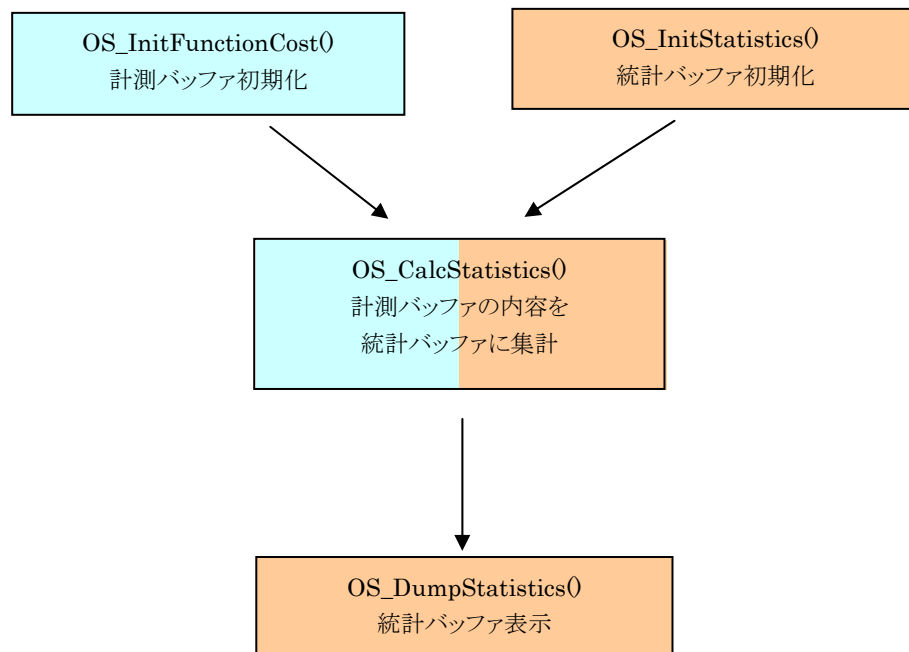
    //---- init functionCost
    OS_InitFunctionCost( &CostBuffer, COSTSIZE );
    OS_InitStatistics( &StatBuffer, STATSIZE );    // この初期化は計測後でもよい

    : // ここが計測したい箇所

    //---- calculate cost
    OS_CalcStatistics( &StatBuffer );

    //---- display functionCost
    OS_DumpStatistics( &StatBuffer );

    :
}
```



4.5 OS_DumpStatistics() による表示例

OS_DumpStatistics() で以下のような表示を得ることが出来ます。

```
test1: count 1, cost 25
test2: count 3, cost 185
test3: count 4, cost 130
```

この結果から、test1()は1回の呼び出しがあり、その関数を実行するための時間は25だったことがわかります。(この時間の単位はOSのTickシステムで用いているものと同じです。)

同様に、test2()は3回の呼び出しがあり、合計時間は185、test3()は4回の呼び出しがあり、合計時間は130だということがわかります。

4.6 リンク時の指定

関数コスト計測の機能を有効にするためには make のオプションに TWL_PROFILE_TYPE=FUNCTIONCOST (または NITRO_PROFILE_TYPE=FUNCTIONCOST) を指定する必要があります。この指定で、リンク時に libos.FUNCTIONCOST.a (または libos.FUNCTIONCOST.thumb.a) が含まれるようになります。Makefile の中に記述しても構いません。

4.7 スレッド上での動作

関数コスト計測の情報は、スレッドシステムを用いている場合はスレッドごとに独立しています。従って、あるバッファを OS_InitFunctionCost() で初期化宣言した場合、それを行ったスレッド以外のスレッドがそのバッファに計測時間の情報を付加していく事はありません。また、OS_EnableFunctionCost() 等の状態設定もスレッドごとに独立しています。

同じ計測バッファを異なるスレッドで OS_InitFunctionCost() 宣言することは避けてください。

4.8 コスト

関数コールの度に時間情報をバッファに保存する処理が入るので、普通に動作させる以上に関数呼び出しのコストがかかります。関数の入り口と出口の __PROFILE_ENTRY/EXIT を必ず通過しなければならないので、コンパイル時の最適化についても何も制約がないときに比べると期待は出来ません。また、関数名へのポインタをバッファに保存する都合上、関数名文字列がメモリ上に置かれることになり、容量のコストもかかります。

動作コストについては、スレッドの有無などで変わってきますが、__PROFILE_ENTRY() で 25～35 命令、__PROFILE_EXIT() で 20～30 命令余計に通過します。また、スレッドの切り替わりの際にインタバル計算を行いますので、ここで 30～40 命令ほど余計にかかります。なお、時間の取得については 32bit のタイマの値を IO レジスタから読むだけです。でそれほどコストはかかりません。

5 TWL-SDK以外のProfiler

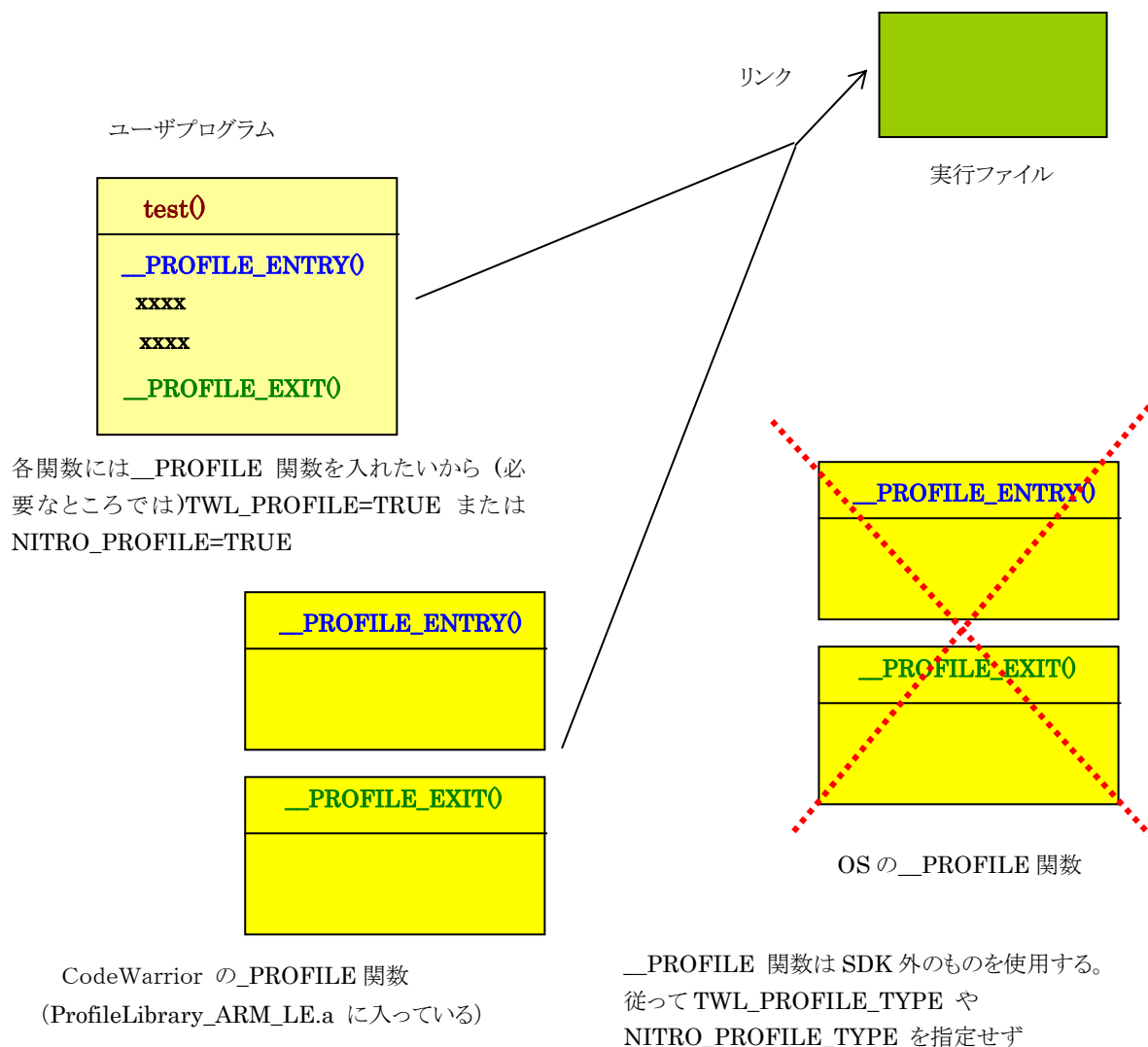
TWL-SDK の OS で用意した Profiler 以外にも、`__PROFILE_ENTRY()` と `__PROFILE_EXIT()` を用意することで異なる Profiler を用いることが出来ます。

例えば CodeWarrior の Example として提供されている Profiler を使用する場合、`__PROFILE_ENTRY()` と `__PROFILE_EXIT()` はその中で定義されているので、OS で用意されているものは定義しないようにする必要があります。

5.1 リンク時の指定

OS のコンパイル時に `TWL_PROFILE_TYPE()` (または `NITRO_PROFILE_TYPE()`) を `CALLTRACE` または `FUNCTIONCOST` 以外する必要があります。(つまり何も指定しなくて構いません。) これによって、`libos.CALLTRACE.a` や `libos.FUNCTIONCOST.a` といったプロファイルライブラリをリンクしないようになります。

但し、各関数の先頭と末尾に `__PROFILE` 関数を挿入するためには `TWL_PROFILE=TRUE` または `NITRO_PROFILE=TRUE` を行う必要があることに注意して下さい。



改定履歴

2010/01/15	0.4.0	誤記 OS_DispatchStatistics()を OS_DispatchStatistics() に修正
2008/09/26	0.3.0	TWL の考慮
2004/08/11	0.2.0	3.4 で、「スタックモード」を「トレースモード」と間違っている個所を修正
2004/	0.1.0	初版