

About Profiler

Version 0.4.0

**The content of this document is highly confidential
and should be handled accordingly.**

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

1	How Profiler Works	4
1.1	The Profile Feature	4
1.2	Compile-Time Options	6
1.3	Switching with <code>pragma</code>	7
1.3.1	Where to Place <code>pragmas</code>	8
2	TWL-SDK profiler	9
2.1	Function Call Tracing	9
2.2	Function Cost Measurement	9
3	Function Call Tracing	10
3.1	How Trace Recording Works	10
3.2	Saved Information	11
3.3	Two Modes of Function Call Tracing	11
3.4	Implementing in the Program	12
3.5	Display Example with <code>OS_DumpCallTrace()</code>	14
3.5.1	In Stack Mode	14
3.5.2	In Log Mode	15
3.6	Settings When Linking	16
3.7	Threaded Operations	16
3.8	Cost	16
4	Function Cost Measurement	17
4.1	How Cost Measurement Works	17
4.2	Saved Information	18
4.3	Conversion to Statistics Buffer	18
4.4	Implementing in the Program	20
4.5	Display Example with <code>OS_DumpStatistics()</code>	22
4.6	Settings When Linking	22
4.7	Threaded Operations	22
4.8	Cost	23
5	Profilers Other Than TWL-SDK	24
5.1	Settings When Linking	24

1 How Profiler Works

1.1 The Profile Feature

The profile feature automatically inserts calls to profiling functions at the entry and exit points of functions. By recording information and statistics about the call from within the function, you can obtain profile information, which is especially useful for things like debugging. The Freescale Semiconductor C compiler `mwccarm.exe` is designed to support the profile feature. To enable this feature, add the option `-profile` to `mwccarm.exe` and compile.

Let's look at an example of how profiler code is added to functions.

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

When this function is compiled, an object with the following code is normally output.

```
test:
    add    r0, r0, #3          // Add 3
    bx     lr
```

The function adds 3 to the argument `r0`. (The return value is also stored in `r0`.)

If this code is compiled with the profile feature ON, a call to `__PROFILE_ENTRY` is added to the entry point of the function and a call to `__PROFILE_EXIT` is added to the exit point. The feature also adds other lines of code that are needed for profiling, such as code that handles stack operations.

```
test:
    stmfd    sp!, {r0,lr}
    ldr      r0, [pc, #32]      // Assign the pointer to the character string "test"
to r0
    bl      __PROFILE_ENTRY    // __PROFILE_ENTRY Call
    ldmfd    sp!, {r0,lr}

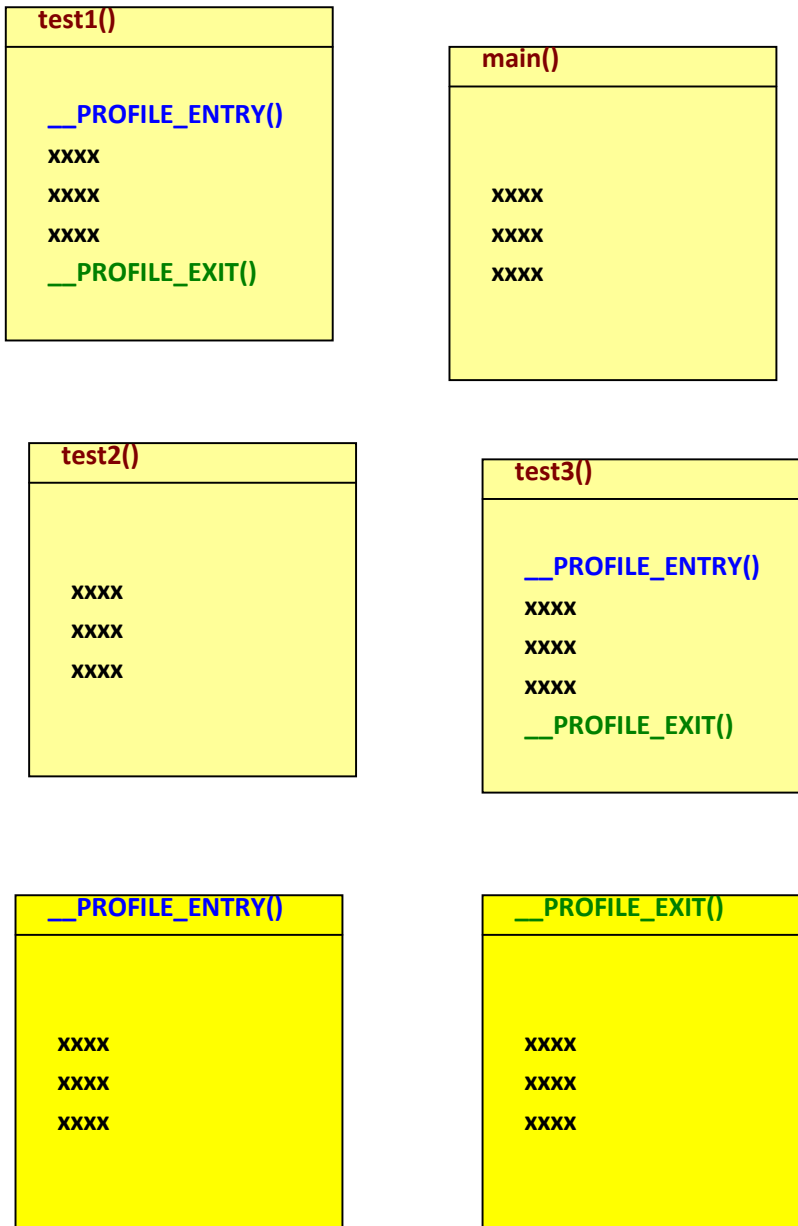
    add      r0, r0, #3         // Add 3

    sub      sp, sp, #4
    stmfd    sp!, {lr}
    bl      __PROFILE_EXIT     // __PROFILE_EXIT Call
    ldmfd    sp!, {lr}
    add      sp, sp, #4
    bx      lr

    :
dcd      xxxx                 // pointer to the string "test"
    :
xxxx: 74 65 73 74 00          // string "test"
```

`__PROFILE_ENTRY` and `__PROFILE_EXIT` only call the profiling functions. The actual functions must be created in the application. For TWL-SDK, `__PROFILE_ENTRY` and `__PROFILE_EXIT` are defined in `os_callTrace.c` and `os_functionCost.c`, so if necessary you can link them.

Functions that call `__PROFILE_ENTRY` and `__PROFILE_EXIT` and functions that do not call `__PROFILE_ENTRY` and `__PROFILE_EXIT` can exist in the linked objects. Functions that don't make these calls will simply not be profiled. The decision to add these calls is made per function at compile time. Normally, a function would have calls to both `__PROFILE_ENTRY` and `__PROFILE_EXIT`, and would not contain only one of these calls.



Objects that have `_PROFILE` functions and objects that do not have `_PROFILE` functions can be mixed. (The `_PROFILE` function itself does not have any calls to the `_PROFILE` functions.)

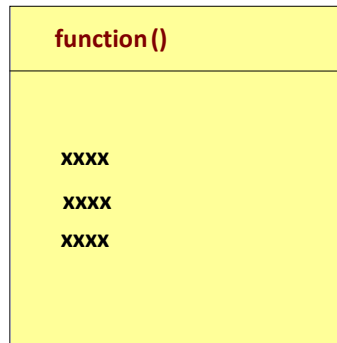
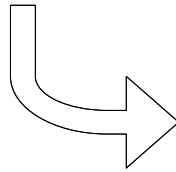
1.2 Compile-Time Options

With TWL-SDK, if you define either the `TWL_PROFILE` or `NITRO_PROFILE` build switches when you run `make`, the `-profile` option will be used when the C source code is compiled. If the `-profile` option is used, the compiler adds calls for `__PROFILE_ENTRY` and `__PROFILE_EXIT` at the entry and exit points of functions in the resulting object code.

As we mentioned, in TWL-SDK, both the `TWL_PROFILE` and `NITRO_PROFILE` build switches are valid, but in NITRO-SDK, only the `NITRO_PROFILE` switch is valid. For compatibility reasons, we allow either build switch to be defined in TWL-SDK. When creating a NITRO ROM using TWL-SDK, it's perfectly fine to define `TWL_PROFILE`. Likewise, when creating a TWL LIMITED ROM, defining either `NITRO_PROFILE` or `TWL_PROFILE` will have the same effect.

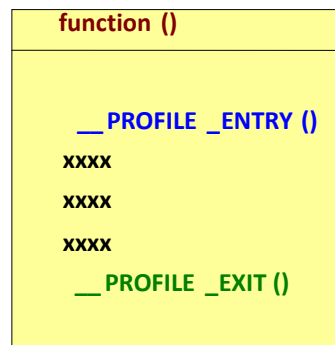
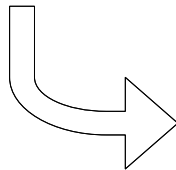
If a simple `make` is executed

`mwccarm ... test.c`



If `make TWL_PROFILE=TRUE` or `make NITRO_PROFILE=TRUE` is executed

`mwccarm -profile ... test.c`



It is okay to include this in the makefile.

Makefile

```
:  
TWL_PROFILE = TRUE  
:
```

1.3 Switching with `pragma`

To temporarily switch the profile feature ON or OFF in the C source code, use the `#pragma` directive.

`#pragma profile on` turns it ON.

`#pragma profile off` turns it OFF.

`#pragma profile reset` returns it to the original status before switching to ON or OFF.

```
(Example)

void test1( void )
{
    :
}

void test2( void )
{
    :
}

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

void test4( void )
{
    :
}
```

If this source code is compiled using `-profile`, the profile feature for `test1()`, `test2()`, and `test4()` are ON (if `-profile` is not used, the profile feature will be OFF for all functions).

1.3.1 Where to Place pragmas

If the “`#pragma profile off`” directive is added to the end of a function, the profiler feature will be enabled for that function. If the “`#pragma profile on`” is added to the end of a function, the profiler feature will be disabled for that function. Normally this pragma should be placed outside the function for readability.

```
(Example)

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

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

profile off

profile on

profile on

profile off

Profiling is enabled for this function.

Profiling is disabled for this function

2 TWL-SDK profiler

If you add calls to `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` to an object, you can use the following mechanisms for debugging with TWL-SDK:

- Function call trace (`OS_CallTrace`)
- Function cost measurement (`OS_FunctionCost`)

These features are not provided by the OS library, `libos.a` (or `libos.thumb.a`). The function call trace library is `libos.CALLTRACE.a` (or `libos.CALLTRACE.thumb.a`), and the function cost measurement is `libos.FUNCTIONCOST.a` (or `libos.FUNCTIONCOST.thumb.a`).

2.1 Function Call Tracing

Function call tracing provides two ways to record profiling information to a buffer.

- In stack mode, `__PROFILE_ENTRY()` records the invocation of the function and `__PROFILE_EXIT()` deletes the record of the invocation. By checking the records at any point in time, you can find out the function that called the current function and view the sequence of calls up to that point.
- In log mode, `__PROFILE_ENTRY()` records the invocation of the function and `__PROFILE_EXIT()` does nothing. The same recording buffer is used each time. When the buffer is full, the newest records overwrite the oldest records. This allows the display of the most recently called functions (and the function that was in the middle of being called.)

To enable this profile feature, you must specify `TWL_PROFILE_TYPE=CALLTRACE` (or `NITRO_PROFILE_TYPE=CALLTRACE`) as a make option. (You can also specify it in the Makefile.)

2.2 Function Cost Measurement

Function cost measurement records the times that `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` functions were called and uses the difference between the two times to determine the time spent in the function.

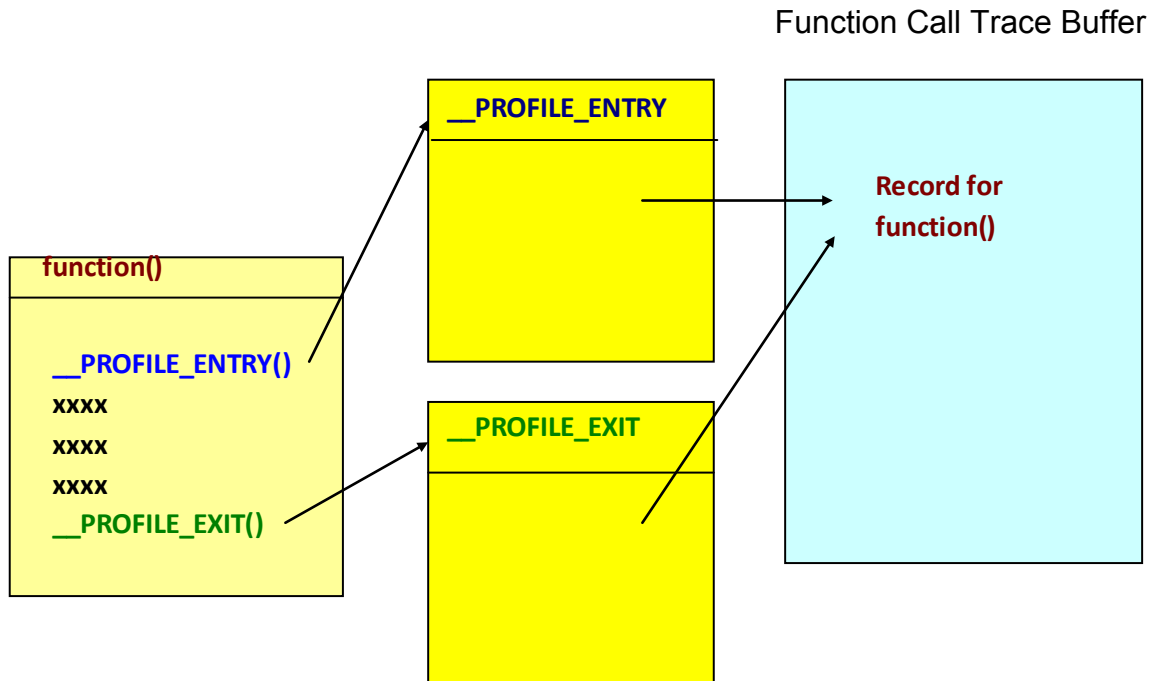
If you are using the thread system, the mechanism subtracts the time required to switch execution from one thread to another. This allows you to accurately compare the costs of functions. In addition, the number of invocations is recorded so it is useful for measuring the frequency of calls.

To enable this profile feature, you must specify `TWL_PROFILE_TYPE=FUNCTIONCOST` (or `NITRO_PROFILE_TYPE=FUNCTIONCOST`) as a make option. (You can also specify it in the Makefile.)

3 Function Call Tracing

3.1 How Trace Recording Works

The function call trace works in the following way.



`__PROFILE_ENTRY()`

Records that “function was called” in the function call trace buffer. Specifically, the pointer to the function name character string, return address from the function, and argument (options) are recorded together.

`__PROFILE_EXIT()`

(In `stack mode`) — Deletes the most recent record of invocation in the function call trace buffer.

(In `log mode`) — Does nothing.

3.2 Saved Information

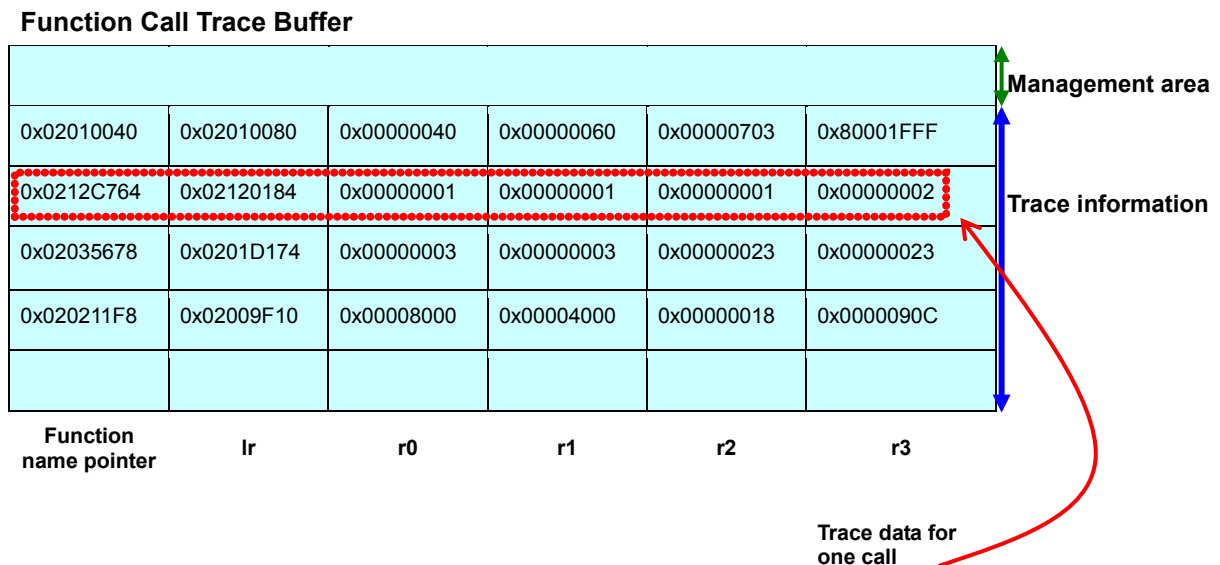
Function call tracing saves the following information:

- Pointer to the function name string
- Value of register `lr` at the time the function was called
- Value of register `r0` at the time the function was called (optional)
- Value of register `r1` at the time the function was called (optional)
- Value of register `r2` at the time the function was called (optional)
- Value of register `r3` at the time the function was called (optional)

Register `lr` stores the address to which control returns after it leaves the function. In other words, if the value of register `lr` is known, you can determine the address from which the function was called.

Registers `r0` – `r3` are used to pass argument values for functions that have arguments. This allows you to see what arguments were specified when the function was called. However, the values of registers that were not used to pass arguments are not meaningful. Saving the values of registers `r0` – `r3` is optional. These require a dedicated 4-byte area for each register. Keep these memory requirements in mind when allocating your buffer.

The buffer is used in the following manner.



In the preceding diagram, registers `r0` – `r3` are saved so each invocation requires a buffer of 24 bytes. If registers `r0` – `r3` do not need to be saved, each invocation requires only 8 bytes.

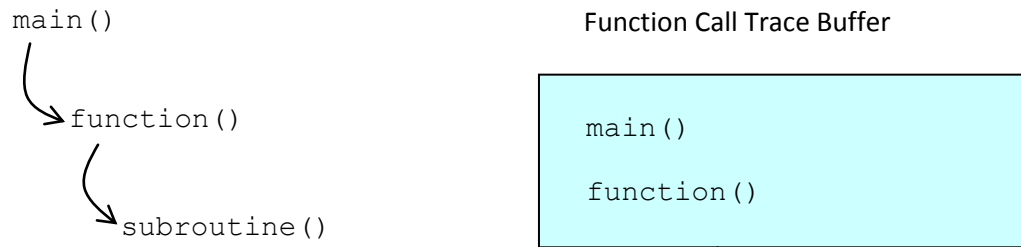
The control area stores the area of the buffer currently in use, the upper limit of the buffer, and other information.

3.3 Two Modes of Function Call Tracing

Function call tracing can be used in two modes: stack mode and log mode. In stack mode, `__PROFILE_ENTRY()` saves information and `__PROFILE_EXIT()` discards information. In log mode, `__PROFILE()` does not discard information. Because the same buffer is used repeatedly, the oldest information discarded as needed.

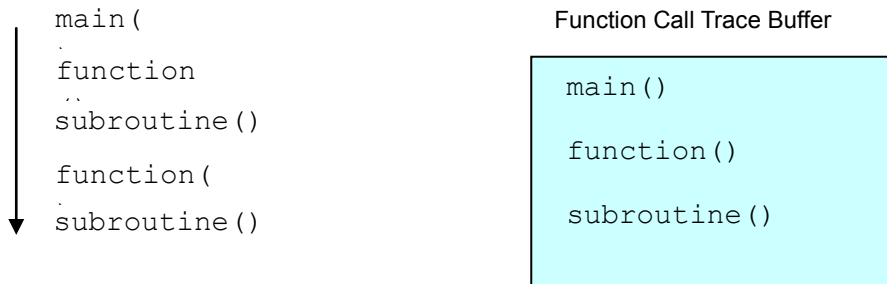
The buffer stores the following information:

Stack Mode



If you check the function call trace buffer, you can view information about function calls at that point in time. In the diagram above, you can see that `main()` called `function()`, and `function()` called `subroutine()`.

Log Mode



If you check the function call trace buffer, you can view the information for functions called up to that point. In the diagram above, you can see that `main()`, `function()`, `subroutine()`, `function()`, and `subroutine()` were called.

3.4 Implementing in the Program

Call tracing begins with the declaration and initialization of the call trace buffer at the beginning of the program. In stack mode, the function that performs initialization should be called at the highest level in the call hierarchy and not called from within other functions. This consideration is not necessary in log mode.

```
// Function call trace initialization
void OS_InitCallTrace( void* buf, u32 size, OSCallTraceMode mode );
```

<code>buf</code>	Function call trace buffer
<code>size</code>	Buffer size
<code>mode</code>	stack mode or log mode

As described previously, the function call trace buffer stores the information for managing the buffer as well as the actual trace information. Specify the mode to use. The mode parameter is of type `OSCallTraceMode` and should have a value of either `OS_CALLTRACE_STACK` (for stack mode) or

OS_CALLTRACE_LOG (for log mode).

If you know the size of the buffer and want to know how many calls (lines) it can store, use the following function.

```
// Calculate the number of calls using the size of the buffer
int OS_CalcCallTraceLines( u32 size )
```

size	Buffer size
Return Value	Number of lines that can be stored (number of trace information sets)

If you know the number of lines you want the buffer to hold and want to know the minimum buffer size required, use the following function.

```
// Calculate the buffer size based on the number of lines to store
u32 OS_CalcCallTraceBufferSize( int lines );
```

lines	Number of lines in the buffer (number of trace information sets)
Return Value	Required size of the buffer

The following function is used to display the contents of the trace buffer. The displayed content is described later in this document.

```
// Function call trace display
u32 OS_DumpCallTrace( void );
```

You can temporarily disable or restore call tracing by using the following functions. Call tracing remains disabled even if `__PROFILE_ENTRY()` or `__PROFILE_EXIT()` is called. If you are using stack mode, note that disabling the profiling functions at the wrong time could corrupt the information inside the buffer.

```
// Function call trace enable/disable/restore
BOOL OS_EnableCallTrace( void );
BOOL OS_DisableCallTrace( void );
BOOL OS_RestoreCallTrace( BOOL enable );
```

enable	Enable (TRUE) or Disable (FALSE)
Return Value	Status prior to this function call. Enable (TRUE)/ Disable (FALSE)

To clear the contents of the buffer in log mode, use the following function. (You can also use this function in stack mode. However, it is strongly recommended that you develop a full understanding of the way in which this function operates before using it in stack mode.)

```
// Function call trace buffer clear
void OS_ClearCallTraceBuffer ( void );
```

The following are actual in-program examples.

In stack mode:

```
#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(); // Displays function call traces at this point
}
```

In log mode:

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

void NitroMain( void )
{
    OS_Init();

    //---- init callTrace (LOG mode)
    OS_InitCallTrace( &traceBuffer, TRACEBUFSIZE, OS_CALLTRACE_LOG );

    : // Code to be logged

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

3.5 Display Example with OS_DumpCallTrace()

3.5.1 In Stack Mode

The following is an example of the output from a `OS_DumpCallTrace()` function call.

```
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
```

The results indicate that `OS_DumpCallTrace()` was called immediately above the position stored in the `lr` register, so we know that it was called immediately above `0x0200434c`. We can also see that `test1()` calls `test2()` and `test2()` calls `test3()`. From `test3()` control should return to the position above `0x2004390`.

The example also shows that when `test3()` is called, `r0` was `0x103`, `r1` was `0x80`, `r2` was `0x80`, and `r3` was `0x2000001f`. If `test3()` is a function that uses arguments, you can use these values to

determine the values of the arguments at the time the function was called. The same analysis is possible with other functions.

Note that in the example above, we said that “test1() called test2()”, but this assumes that profiling was enabled for all objects in the executable file. If test1() called test4(), which did not have profiling enabled, and then test4() called test2(), which did have profiling enabled, the call tracing information would look exactly like the snippet above, with test2() directly above test1(), and test4() would not be displayed at all.

The trace information above was generated from the program below.

```
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 In Log Mode

The following is an example of the output from a OS_DumpCallTrace() function call.

```
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
```

The function calls are displayed in reverse chronological order, with the most recently called functions at the top of the trace. Of the functions that have profiling enabled, the call sequence was: test1, test2, test3, test1, test2, and test3. The address to return to, arguments, and other information can be determined from registers lr and r0 through r3.

Looking at the display of test1, test2, and test3, you can see that test2 and test3 are indented. This means that test2 was called before the __PROFILE_EXIT() of test1, and test3 was called before the __PROFILE_EXIT() of test2. This provides information about the relationships between the calls.

The display above was output by the following program.

```
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 Settings When Linking

To enable the function call trace feature, use the `TWL_PROFILE_TYPE=CALLTRACE` (or `NITRO_PROFILE_TYPE=CALLTRACE`) make option. This setting tells the linker to link to `libos.CALLTRACE.a` (or `libos.CALLTRACE.thumb.a`). You can also set this option in the makefile.

3.7 Threaded Operations

If the thread system is being used, the function call trace information runs independently for each thread. When you initialize a buffer by using `OS_InitCallTrace()`, only the thread that made the call can write trace information to that buffer. Status settings made using `OS_EnableCallTrace()` and other functions are also independent for each thread.

Avoid declaring the same buffer with a different thread using `OS_InitCallTrace()`.

3.8 Cost

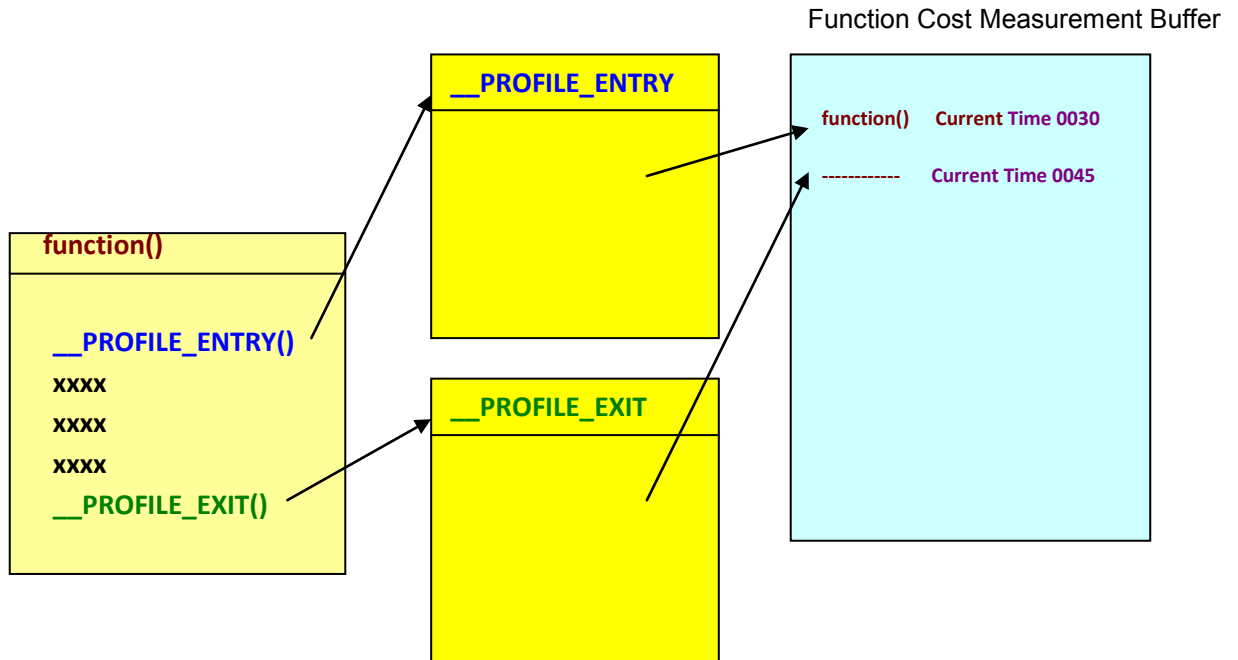
Calling a function that has profiling enabled costs more than it normally would because a process saves function call information to the buffer each time the function is called. Because the `__PROFILE_ENTRY` and `__PROFILE_EXIT` calls in profiled functions must be processed, you cannot expect to obtain the degree of compiler optimization that you would without these restrictions. Furthermore, function name strings are saved in memory when pointers to function names are saved to the buffer, resulting in additional memory usage.

The operational cost depends on factors, such as whether threading is used, the nature of the information saved, and the selected mode. A call of `__PROFILE_ENTRY()` results in the processing of an additional 60 to 70 instructions; a call of `__PROFILE_EXIT()` results in the processing of an additional 20 to 40 instructions.

4 Function Cost Measurement

4.1 How Cost Measurement Works

Two buffers are used with function cost measurement. As shown below, they are the function cost measurement buffer and function cost statistics buffer.

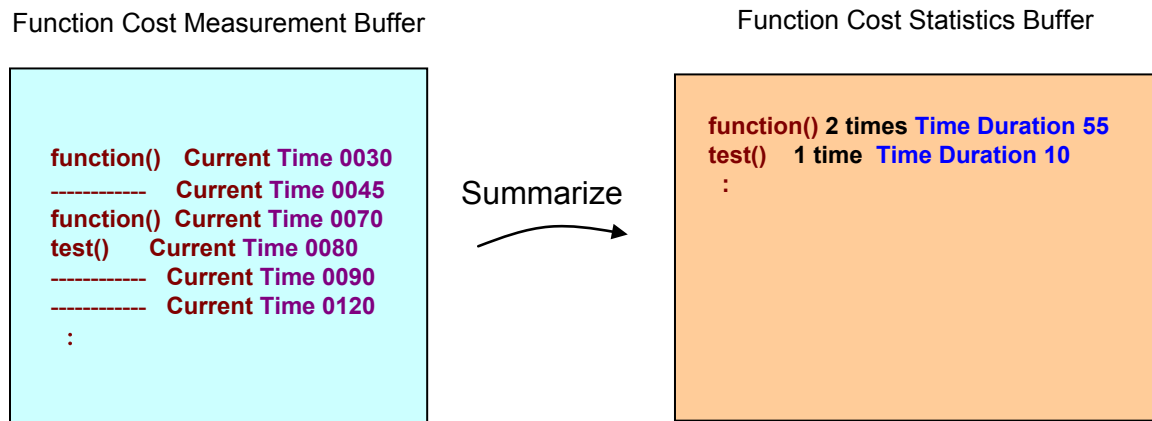


`__PROFILE_ENTRY()`

A pointer to the function name string and the current time are saved in the cost measurement buffer the user specified.

`__PROFILE_EXIT()`

This records the tag written by `__PROFILE_EXIT()` and the current time.



4.2 Saved Information

The following information is recorded by function cost measurement.

With `__PROFILE_ENTRY`:

- A pointer to function name string
- The current time (the return value of `OS_GetTickLo()`)

With `__PROFILE_EXIT`:

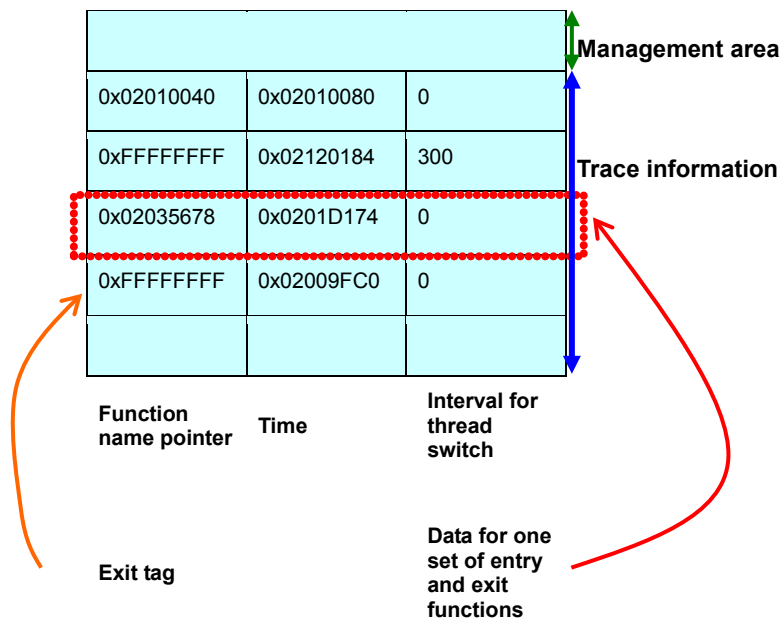
- A special value, called an “exit tag,” is saved to the area where `__PROFILE_ENTRY` saved the pointer.
- The current time (the return value of `OS_GetTickLo()`)
- Interval needed for thread switching if required (optional)

The current time is the value obtained using `OS_GetTickLo()`. The tick feature of the OS uses 64-bit values, but profiler uses 32-bit values, as the lower-order 32 bits are sufficient for the purposes of profiling.

`__PROFILE_ENTRY` stores a pointer to the function name string. `__PROFILE_EXIT` replaces this pointer with a special value, called “an exit tag”.

The “thread switching interval” information is used to subtract the amount of time it takes to change threads (including the time spent in other functions).

Function Cost Measurement Buffer



The management area stores information, such as current buffer usage, the upper limit of the buffer, and the counter value used to measure the thread switching interval.

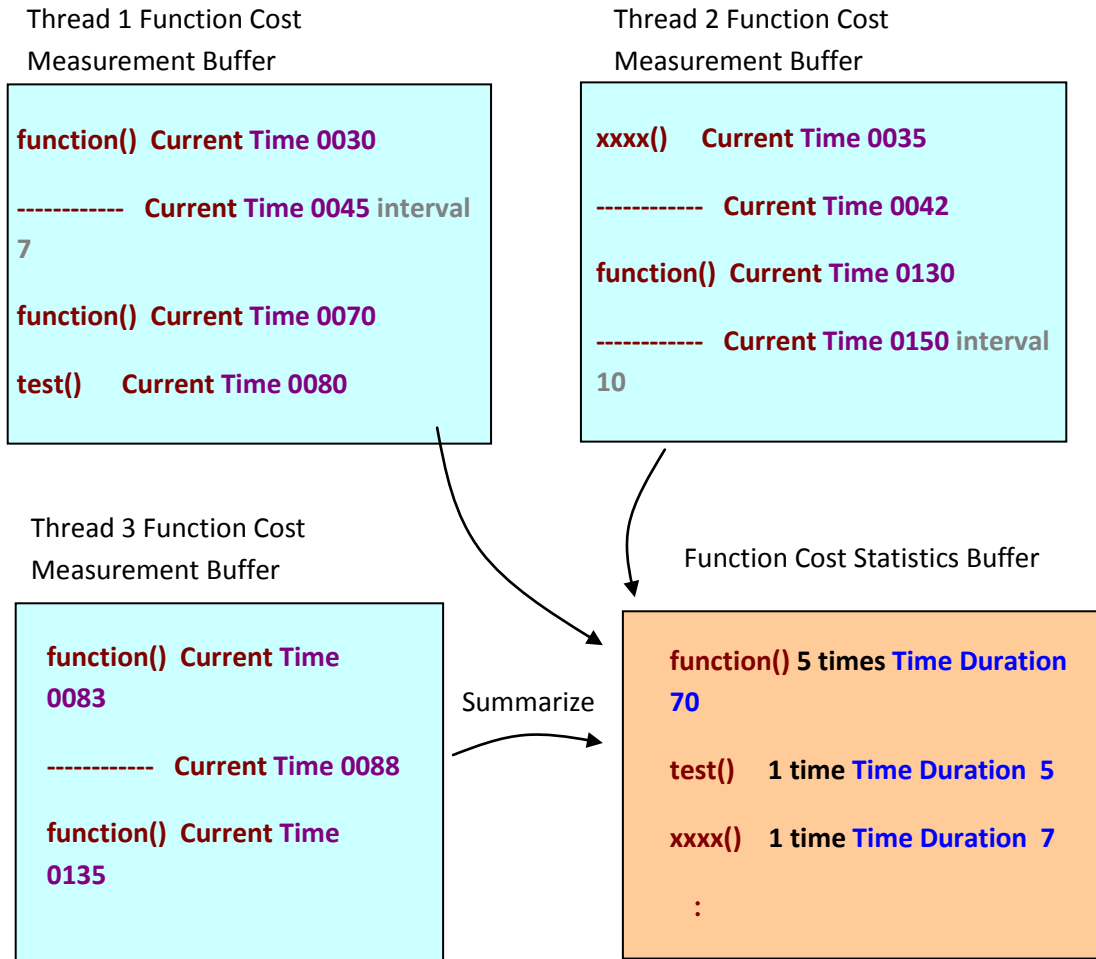
4.3 Conversion to Statistics Buffer

It is difficult to obtain cost information only with the function cost measurement. The measurement data needs to be converted to statistics buffer data.

The summary links the call information for each function to the exit tag, then totals the number of calls and the time spent in each function. When the control is transferred to a separate thread due to a

switch in threads, the amount of time until it returns to the original thread is recorded as the interval value of the exit tag. Calculations take this into consideration.

Summarizations must be carried out explicitly. When summarized, the contents of the function cost measurement buffer are cleared. Repeatedly storing these results in the summarization buffer (before the function cost measurement buffer overflows) helps to ensure accurate measurement for long processes. The same summarization buffer can be shared among multiple threads. However, avoid creating a separate thread to perform summarization.



The results of multiple measurements can be written to the statistics buffer.

4.4 Implementing in the Program

Cost measurement begins recording to the buffer as soon as the buffer is initialized. The tick system of the operating system is used for cost measurement so you must call `OS_InitTick()` before initializing any cost measurement buffers.

```
// Function cost measurement initialization
void OS_InitFunctionCost( void* buf, u32 size );

    buf    Function cost measurement buffer
    size    Buffer size (byte)
```

As described previously, the information for managing the buffer and the actual time information are stored. If you know the size of a buffer and want to know how many calls it can store, use the following function.

```
// Calculate the number of calls based on the size of the buffer.
int OS_CalcFunctionCostLines( u32 size )

    size        Buffer size (bytes)
    Return Value Number of calls that can be stored (number of sets of call information for cost
                  measurement)
```

Use the following function to determine the minimum size required for your buffer based on the number of calls you want it to contain.

```
// Calculate the buffer size based on the number of
// information sets (calls) that can be stored.

u32 OS_CalcFunctionCostBufferSize( int lines );

    lines        Number of lines in the buffer
    Return Value Required size of your buffer (in bytes)
```

Initialize the cost statistics buffer with the following function.

```
// Function cost statistics buffer initialization
void OS_InitStatistics( void* statBuf, u32 size );

    statBuf    Buffer
    size        Buffer size (bytes)
```

The following function stores the values from the cost measurement buffer in the cost statistics buffer.

```
// Summarize function cost
OS_CalcStatistics( void* statBuf );

    statBuf    Statistics buffer
```

The current contents of the function cost measurement buffer are cleared when `OS_CalcStatistics()` is called.

The following function displays the summarization results. The output of this function is described later in this document.

```
// Function cost summarization display
OS_DumpStatistics( void* statBuf );

statBuf    Statistics buffer
```

You can temporarily stop or resume recording of profiling data by using the following functions. Recording of information remains disabled even if `__PROFILE_ENTRY()` or `__PROFILE_EXIT()` are called. If you use these functions in an inconsistent way so that only the information recorded with `__PROFILE_ENTRY()` or the information recorded with `__PROFILE_EXIT()` is written to the buffer, the cost measurement data may be invalid. It is strongly suggested that you pay particular attention when using these functions.

```
// Function cost measurement enable/disable/restore
BOOL OS_EnableFunctionCost( void );
BOOL OS_DisableFunctionCost( void );
BOOL OS_RestoreFunctionCost( BOOL enable );

enable        Enable (TRUE), Disable (FALSE)
Return Value   Status prior to function call. Enable (TRUE)/disable (FALSE)
```

If you want to explicitly clear the contents of the function cost measurement buffer, call the following function.

```
// Function cost measurement buffer clear
void OS_ClearFunctionCostBuffer ( void );
```

The following is an in-program example.

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

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

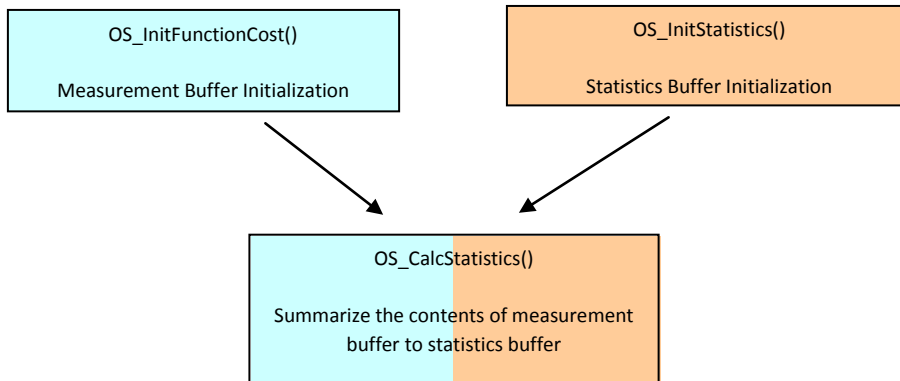
void NitroMain( void )
{
    OS_Init();
    OS_InitTick();

    //---- init functionCost
    OS_InitFunctionCost( &CostBuffer, COSTSIZE );
    OS_InitStatistics( &StatBuffer, STATSIZE ); // This initialization can be done after
measurement

    : // This is the code to be profiled
    //---- calculate cost
    OS_CalcStatistics( &StatBuffer );
```

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

:
}
```



4.5 Display Example with OS_DumpStatistics()

A following is an example of the output of an `OS_DumpStatistics()` function call.

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

In the example, there was one call of `test1()` with an elapsed time (duration) of 25. (The units are those used in the Tick system of the OS.)

There were three calls of `test2()` with a total duration of 185. For `test3()`, there were four calls with a total duration 130.

4.6 Settings When Linking

To enable the function cost measurement feature, use the `TWL_PROFILE_TYPE=FUNCTIONCOST` (or `NITRO_PROFILE_TYPE=FUNCTIONCOST`) option. Due to this setting, the linker will link to `libos.FUNCTIONCOST.a` (or `libos.FUNCTIONCOST.thumb.a`). This can also be specified in the makefile.

4.7 Threaded Operations

If the thread system is being used, function cost measurement information is maintained independently for each thread. Therefore, when a buffer is initialized in `OS_InitFunctionCost()` only the initializing thread can write to that buffer. Status settings for functions like `OS_EnableFunctionCost()` are also independently maintained for each thread.

Avoid initializing a single measurement buffer from different threads using `OS_InitFunctionCost()`.

4.8 Cost

Because a process saves time information to the buffer every time the function is called, function calls using profiler cost more than the normal operation. Since every function must include the `__PROFILE_ENTRY/ __PROFILE_EXIT` calls, you cannot expect the code to be optimized during compilation to the degree that it would be without these restrictions. Furthermore, to save the pointer to the function name in the buffer, the function name string must be saved to memory, which causes additional memory usage.

The operational cost depends on factors, such as whether threading is used. A call of `__PROFILE_ENTRY()` results in the processing of an additional 25 to 35 instructions; a call of `__PROFILE_EXIT()` results in the processing of an additional 20 to 30 instructions. Also, interval calculations are done when thread are switched, which require an additional 30 – 40 instructions.

The time is obtained by reading the 32-bit timer value from the IO register so the cost of obtaining the time is not significant.

5 Profilers Other Than TWL-SDK

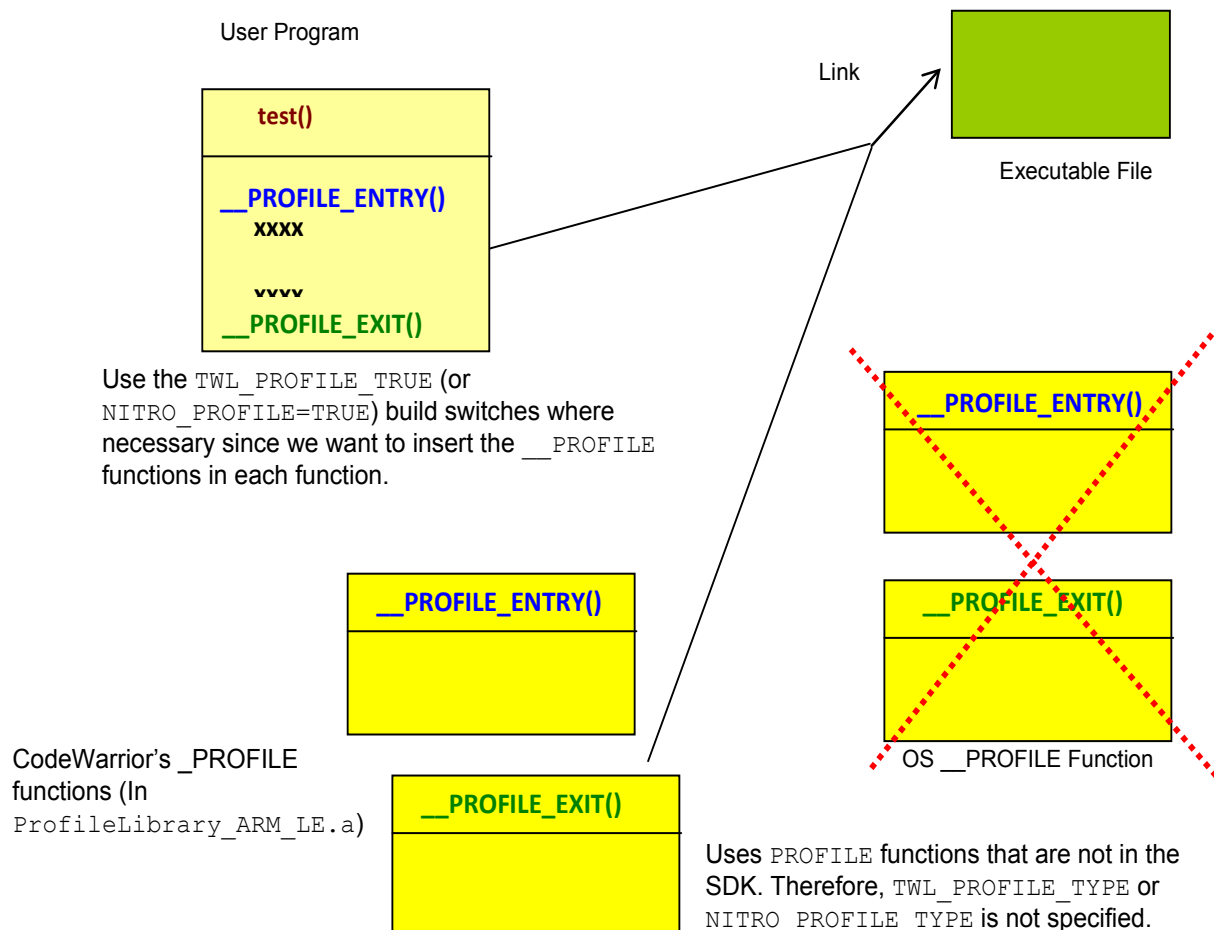
Preparing `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` allows you to use a profiler other than the one provided in the TWL-SDK OS.

For example, if you use the profiler provided by CodeWarrior, `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` are defined within it. Do not define the ones provided by the OS.

5.1 Settings When Linking

You must set `TWL_PROFILE_TYPE` (or `NITRO_PROFILE_TYPE`) to something other than 'CALLTRACE' or 'FUNCTIONCOST' during compilation of the operating system. (In other words, you don't have to specify anything.). This will prevent linking to the `libos.CALLTRACE.a` and `libos.FUNCTIONCOST.a` profile libraries.

Note that `TWL_PROFILE=TRUE` or `NITRO_PROFILE=TRUE` must be specified in order to insert the `__PROFILE` functions at the entry and exit points of each function.



Revision History

Version	Revision Date	Description
0.4.0	2010/01/15	Corrected typo (Changed OS_DispatchStatistics to OS_DumpStatistics).
0.3.0	2008/09/26	Revised document to reflect TWL.
0.2.0	2004/08/11	Revised the error in section 3.4 where "stack mode" was "trace mode."
0.1.0	2004	Initial version.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

© 2004-2010 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.