

Profiler

Version 0.3.0 2008/09/26

Table of Contents

1	Profiler Mechanism.....	2
1.1	Profile Feature	3
1.2	Specifying During a Compile	5
1.3	Switching with <code>pragma</code>	6
1.3.1	Where to Place Pragma	7
2	Nitro-SDK profiler	8
2.1	Function Call Trace	8
2.2	Function Cost Measurement	8
3	Function Call Trace	9
3.1	Mechanism of Trace Recording	9
3.2	Saved Information	10
3.3	Two Modes of Function Call Trace.....	11
3.4	Implementing in the Program	12
3.5	Display Example with Dump	15
3.5.1	In Stack Mode	15
3.5.2	In Log Mode	16
3.6	Specification When Linking	17
3.7	Operation on Thread	17
3.8	Cost	17
4	Function Cost Measurement	18
4.1	Cost Measurement Mechanism	18
4.2	Saved Information	19
4.3	Conversion to Statistics Buffer	20
4.4	Implementing in the Program	21
4.5	Display Example with Dump	24
4.6	Specification When Linking	24
4.7	Operation on Thread	24
4.8	Cost	24
5	Other Profilers (non-Nitro-SDK).....	25
5.1	Specification When Linking	25

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo 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.

1 Profiler Mechanism

1.1 Profile Feature

`mwccarm.exe`, the C compiler from Freescale Semiconductor, is set up to support the profile feature. This feature automatically inserts the code for a specific function call in the entry and exit of the function. By taking records and statistics about the call from within the function, you can acquire profile information; which is especially useful for things like debugging. The profile feature can be enabled by adding the option `-profile` in `mwccarm.exe` and compiling.

The profiler typically created lines of code like the following example.

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

If this function is compiled, usually an object with the following code is output.

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

3 is simply added to the argument `r0`. (Return value is also stored in `r0`.)

Next, the case in which compiling with the profile feature ON is shown. `__PROFILE_ENTRY` and `__PROFILE_EXIT` are called during entry and exit respectively. The following is an example of code created by this feature for a stack operation.

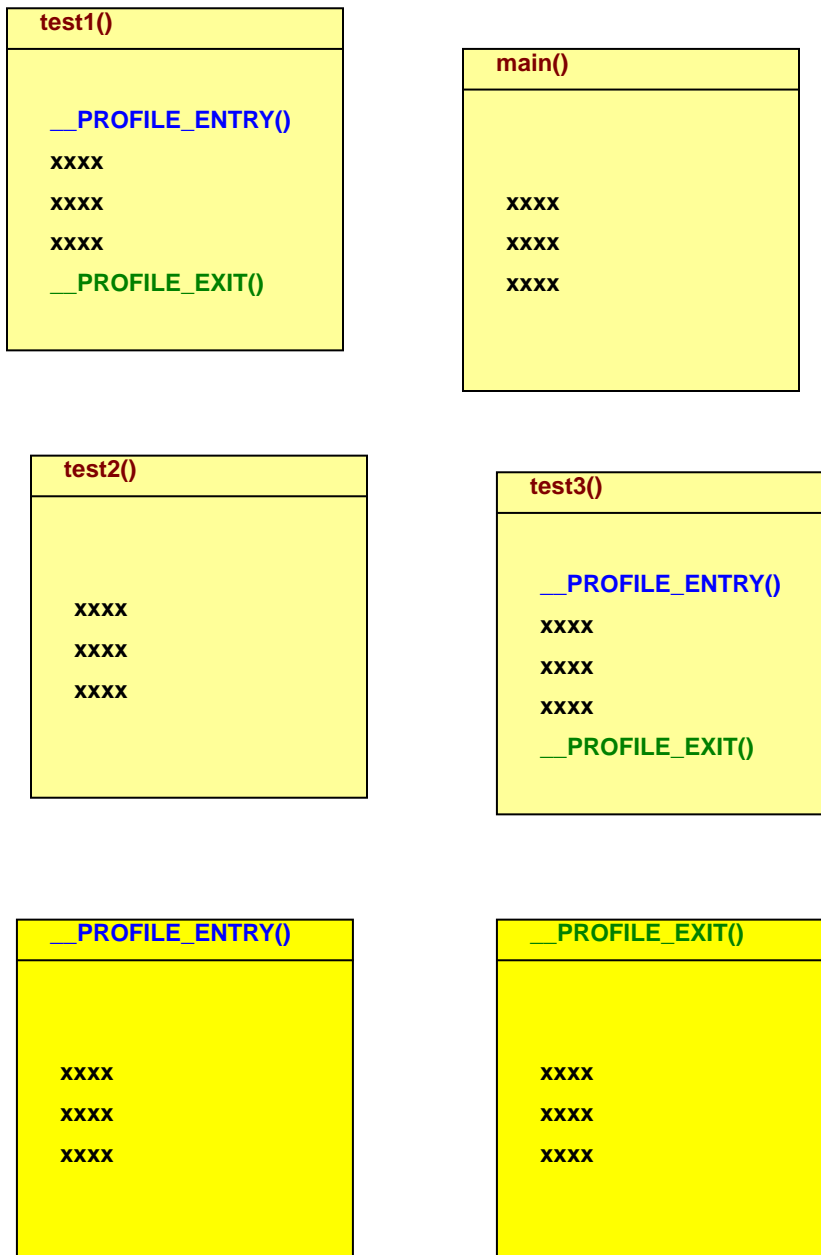
```
test:
    stmfd   sp!, {r0,lr}
    ldr     r0,[pc,#32]        // Assign the pointer to the character string "test"
to r0
    bl      __PROFILE_ENTRY    // __PROFILE_ENTRY Call
    ldmfd   sp!, {r0,lr}

    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 character string "test"
:
    xxxx:  74 65 73 74 00     // character string "test"
```

`__PROFILE_ENTRY` and `__PROFILE_EXIT` have only the code for calling functions, and the entity of the function must be created in the application. For TWL-SDK, they are defined in `os_callTrace.c` and `os_functionCost.c` so that you can link them into your code if necessary.

Functions that call `__PROFILE_ENTRY` and `__PROFILE_EXIT` and functions that do not call `__PROFILE_ENTRY` and `__PROFILE_EXIT` can exist in the link object. Functions that don't make these calls will simply not be profiled. Since the compiler enters profiling calls function by function, there are no function that calls only `__PROFILE_ENTRY` or `__PROFILE_EXIT` unless such function is created deliberately.



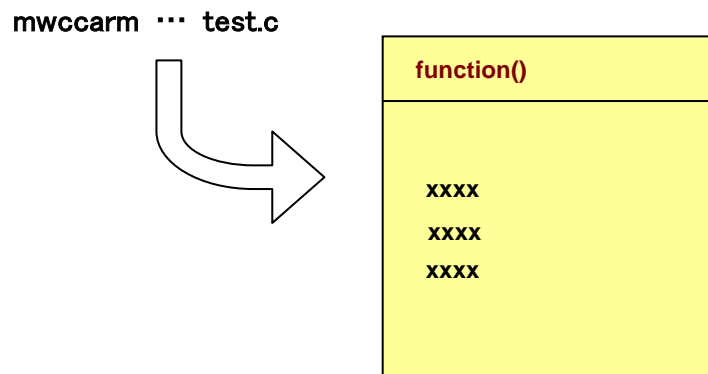
The object that has the `__PROFILE` function and the object that do not have the `__PROFILE` function can be mixed. (The `__PROFILE` function itself does not have the calls to the `__PROFILE` function.)

1.2 Compile-Time Specifications

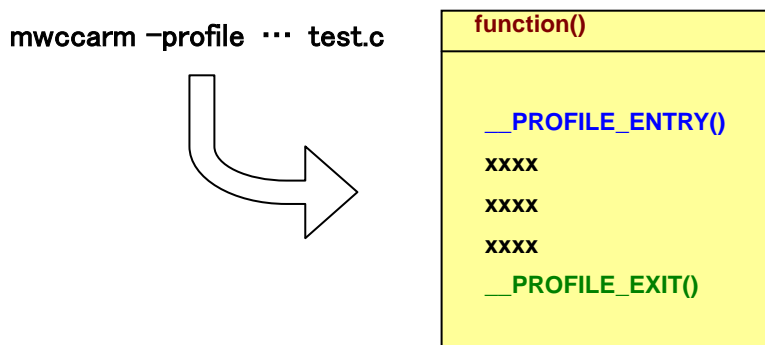
With TWL-SDK, if you define either the `TWL_PROFILE` or `NITRO_PROFILE` build switches when you run `make`, the `-profile` option will be added when the C source is compiled. For functions in source compiled with the `-profile` option, the calls for `__PROFILE_ENTRY` and `__PROFILE_EXIT` are entered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

We used the phrase “`TWL_PROFILE` or `NITRO_PROFILE`” in the description above. Because only the `NITRO_PROFILE` build switch is valid for NITRO-SDK, we have made provisions for the sake of compatibility so that either build switch can be defined. 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



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



It is okay to include this in the makefile.

Makefile

```
        :  
        TWL_PROFILE = TRUE  
        :
```

1.3 Switching with `pragma`

When temporarily switching the profile feature in the C source use `#pragma`.

`#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)

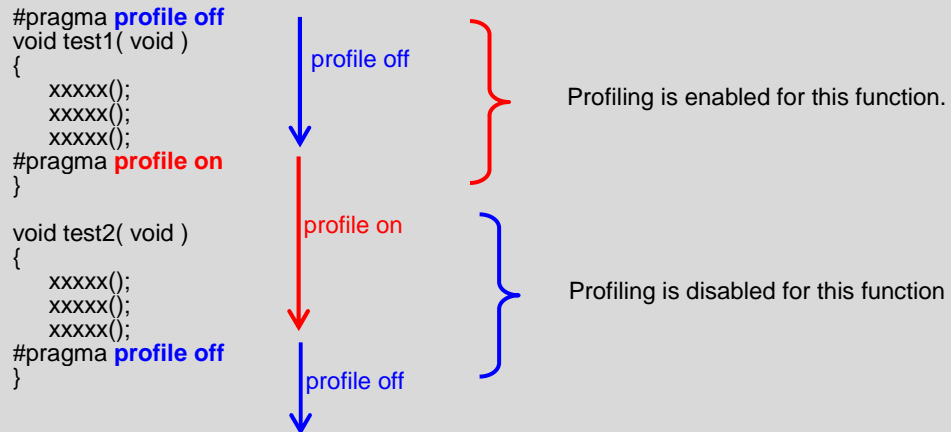
```
void test1( void )  
{  
    :  
}  
  
void test2( void )  
{  
    :  
}  
  
#pragma profile off  
void test3( void )  
{  
    :  
}  
#pragma profile reset  
  
void test4( void )  
{  
    :  
}
```

If this source is compiled with `-profile`, the profile feature for `test1()`, `test2()`, and `test4()` are ON. (without `-profile`, the profile feature will be OFF with all functions.)

1.3.1 Where to Place Pragmas

If the profile pragma is turned on before a given function ends, the profiler feature will be enabled for that function. If the profile pragma is turned off at the moment it ends, the profiler feature will be disabled for that function. Normally this pragma should be set outside the function so that it is easier to follow.

(Example)



2 TWL-SDK profiler

By setting up PROFILE functions for objects with calls for `__PROFILE_ENTRY()` and `__PROFILE_EXIT()`, the following mechanisms for debugging with TWL-SDK are available:

- Function Call Trace (`OS_CallTrace`)
- Function Cost Measurement (`OS_FunctionCost`)

These features are built separately from the OS library. More specifically, the OS library is `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 Trace

There are two modes for the mechanism that records the results of a PROFILE function to a specified memory location.

One is a stack mode that records the call of the function with `__PROFILE_ENTRY()` and deletes the record with `__PROFILE_EXIT()`. By checking the record at a certain point you can find out what function wrote to the record at that point (what type of call was used).

The other is a log mode that records the call of the function with `__PROFILE_ENTRY()` and does nothing with `__PROFILE_EXIT()`. The buffer for recording is used repeatedly so the most recent record is always maintained. This allows display of the function that was called (was being called when the record was written).

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

2.2 Function Cost Measurement

This mechanism measures the time in the ENTRY and EXIT areas of the PROFILE function and checks the duration of the function based on the difference between the two.

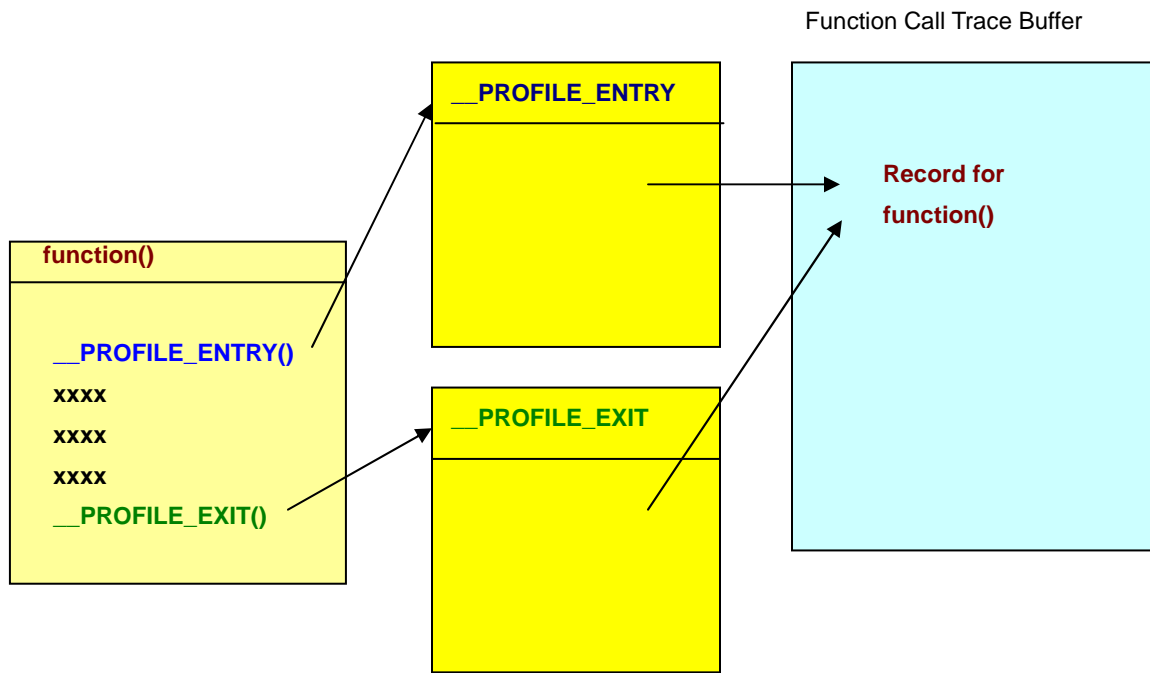
If you are using a thread system, the time while the thread is switched and another thread is running is subtracted from the duration. This allows you to compare the cost of a particular function. In addition, the number of calls is recorded so it is useful for measuring the frequency of calls.

To enable this profile feature, you must specify `TWL_PROFILE_TYPE=FUNCTIONCOST` (or `NITRO_PROFILE_TYPE=FUNCTIONCOST`) in the make option. (It can be specified in the Makefile also.)

3 Function Call Trace

3.1 Mechanism of Trace Recording

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()`

(Stack Mode) — Deletes the record that states the “function was called” most recently written to the function call trace buffer.

(Log Mode) — Performs no processes.

3.2 Saved Information

The following information is saved with the function call trace:

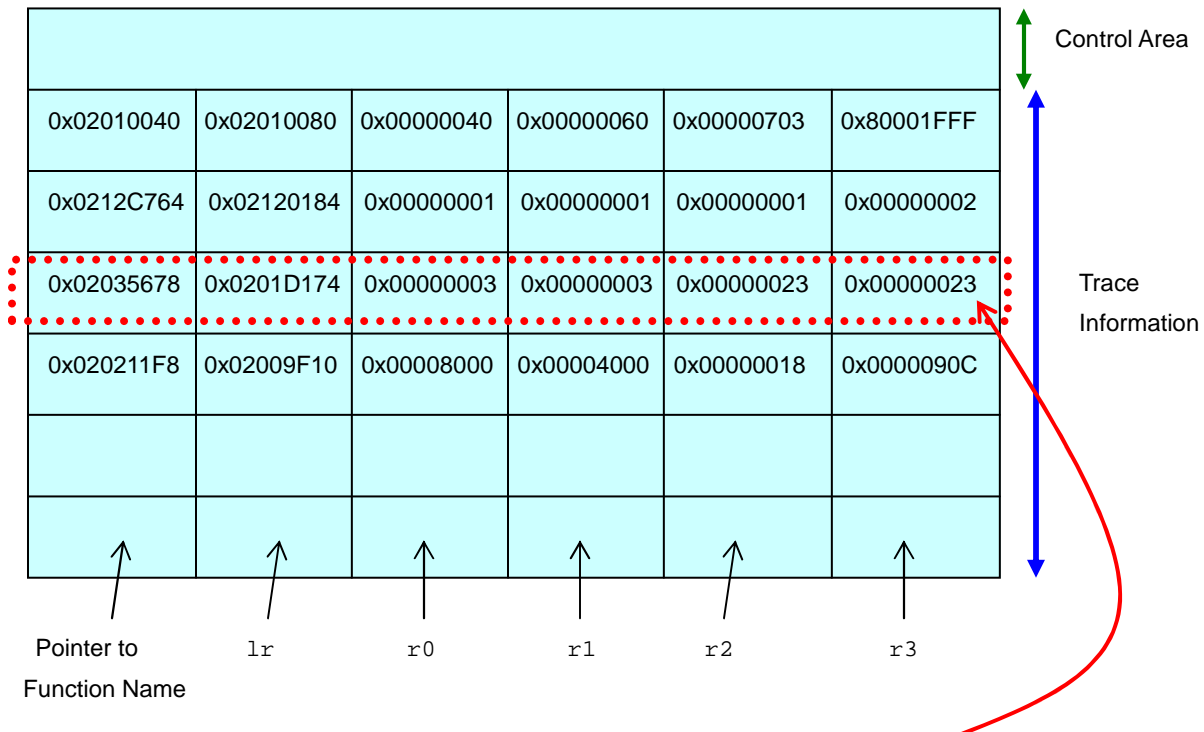
- Pointer to the function name character string
- Value of `lr` Register at the point from which the function was called
- Value of `r0` Register at the point from which the function was called (optional)
- Value of `r1` Register at the point from which the function was called (optional)
- Value of `r2` Register at the point from which the function was called (optional)
- Value of `r3` Register at the point from which the function was called (optional)

The memory address from which the function was called is stored in the `lr` register. In other words, if the value of the `lr` register is known, you can use that value to determine the address from which the function was called.

The `r0 - r3` registers are used for the passing of the values of argument for the function that has arguments. This allows you to see what type of argument was specified when the function is called. However, the values of registers not used in passing arguments do not have much meaning. Saving the values of `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.

Function Call Trace Buffer



In the preceding diagram, $r0 - r3$ are saved so a 24-byte information region is required for one call. If $r0 - r3$ do not need to be saved, the buffer size required for one call is 8 bytes.

Information such as the area of the buffer currently in use and the location of the upper limit is stored in the Control Area.

3.3 Two Modes of Function Call Trace

There are two modes for the function call trace, stack mode and log mode. In stack mode, information is saved by `__PROFILE_ENTRY()` and deleted by `__PROFILE_EXIT()`. In log mode, `__PROFILE()` does not delete the information. Also, the same region is used for saving information and the old information is deleted.

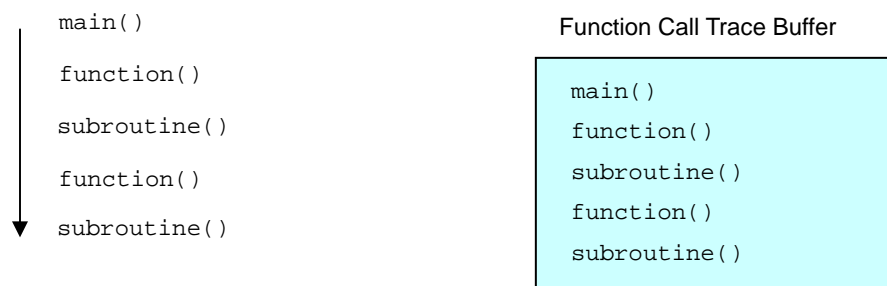
The buffer stores the following information:

Stack Mode



If you check the function call trace buffer, you can view the information about a function call at a particular point. 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

The call trace begins with the initialization of the call trace buffer at the beginning of the program. In stack mode, the functions carrying out initialization must be at the highest level (not called from within other functions). This consideration is not particularly necessary in log mode.

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

    buf    Function call trace buffer
    size    Buffer size
    mode    stack mode or log mode
```

As described previously, the function call trace buffer stores the information necessary for controlling the buffer and the actual trace information. The mode is specified by `OSCallTraceMode` with a value of either `OS_CALLTRACE_STACK` (stack mode) or `OS_CALLTRACE_LOG` (log mode).

If you know the size of a `CallTrace` buffer and want to know how many lines it can store, use the following function.

```
// Calculate the number of trace information sets based on the size of the buffer.
int OS_CalcCallTraceLines( u32 size )

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

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

```
// Calculate the buffer size based on the number of
// trace information sets that can be stored.
u32 OS_CalcCallTraceBufferSize( int lines );

    lines        Number lines in the buffer (number of trace information sets)
    Return Value required size of your buffer
```

The following function is used for displaying the contents of a trace buffer. The displayed content is described later in this document.

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

You can temporarily stop recording information or restore the setting by using the following functions. Recording of information remains disabled even if `__PROFILE_ENTRY()` or `__PROFILE_EXIT()` are called. If you are using stack mode, the information inside the buffer may be invalid or corrupt depending on when the `__PROFILE` function is stopped.

```
// 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 status of function call 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 );

    : // Location 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
```

In this example, the `lr=` value shows that `OS_DumpCallTrace()` was called immediately before `0x0200434c`. You can also see that `test1()` calls `test2()` and `test2()` calls `test3()`. The position returning from `test3()` is before `0x2004390`.

The example also shows that when `test3()` is called, `r0` is `0x103`, `r1` is `0x80`, `r2` is `0x80`, and `r3` is `0x2000001f`. Therefore, if `test3()` is a function that uses an argument, you can apply these values and figure out the arguments when the functions are called. The same analysis is possible with other functions.

(Note) In the example above, descriptions like “`test1()` called `test2()`” are based on the premise that this executable file has the profile feature enabled for all the objects and is compiled. So, if `test1()` calls `test4()` when `test4()` does not have an enabled profile feature. Then, `test4()` calls `test2()` which does have an enabled profile feature. The result will be what you see in the example—`test2()` above `test1()` with no `test4()` displayed at all.

The display above was output 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

```

Since the newest information is displayed first, you can see that for the functions that have an active profile feature, the calling order is test1, test2, test3, test1, test2, and test3. Return address, argument and other information can be determined from the `lr` register or `r0 – r3` registers at that point.

Looking at the display of test1, test2, and test3 you can see that test2 and test3 are indented. This happens because test2 was called before the `__PROFILE_EXIT()` of test1, and test3 was called before the `__PROFILE_EXIT()` of test2.

The display above was output from the program below.

```

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 Specification When Linking

To enable the function call trace feature, `TWL_PROFILE_TYPE=CALLTRACE` (or `NITRO_PROFILE_TYPE=CALLTRACE`) must be specified for the make option. Due to this requirement, `libos.CALLTRACE.a` (or `libos.CALLTRACE.thumb.a`) is included when linking. This can also be described in the Makefile.

3.7 Operation on Thread

If a thread system is being used, the function call trace information runs independently for each thread. Therefore, initialization of a particular buffer declared with `OS_InitCallTrace()` only records information from the thread in which it was generated. Status settings for functions like `OS_EnableCallTrace()` are also independent for each thread.

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

3.8 Cost

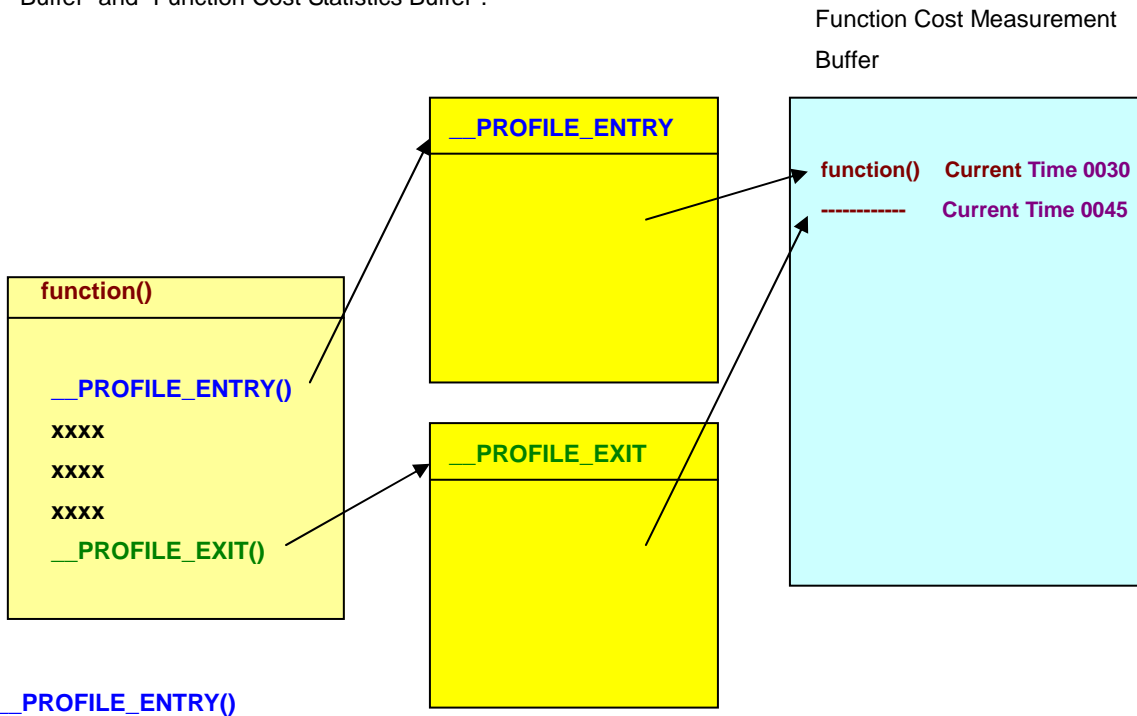
Because the function calls are saved in the buffer, function calls cost more than the normal operation. Since every function must include the `__PROFILE_ENTRY/EXIT` calls, the optimization during compile is not as much as expected, compared to a situation where there is no restrictions. Further, to save the pointer to the function name in the buffer, the function name string is placed on the memory, which causes additional memory usage.

The operational cost varies based on factors such as: whether there is a thread or not, the information saved, and the mode. With `__PROFILE_ENTRY()` extra 60 – 70 instructions are passed, and with `__PROFILE_EXIT()` an extra 20 – 40 instructions.

4 Function Cost Measurement

4.1 Cost Measurement Mechanism

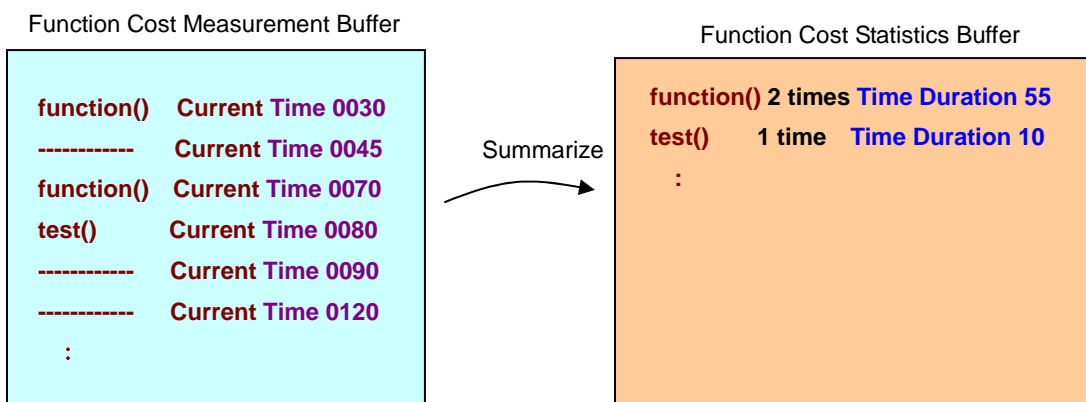
Two buffers are used with function cost measurement. As shown below, they are the “Function Cost Measurement Buffer” and “Function Cost Statistics Buffer”.



This records the pointer to the function name character string and the current time to 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 with the function cost measurement.

With `__PROFILE_ENTRY`:

- Pointer to function name character string
- Current time, value of `OS_GetTickLo()`

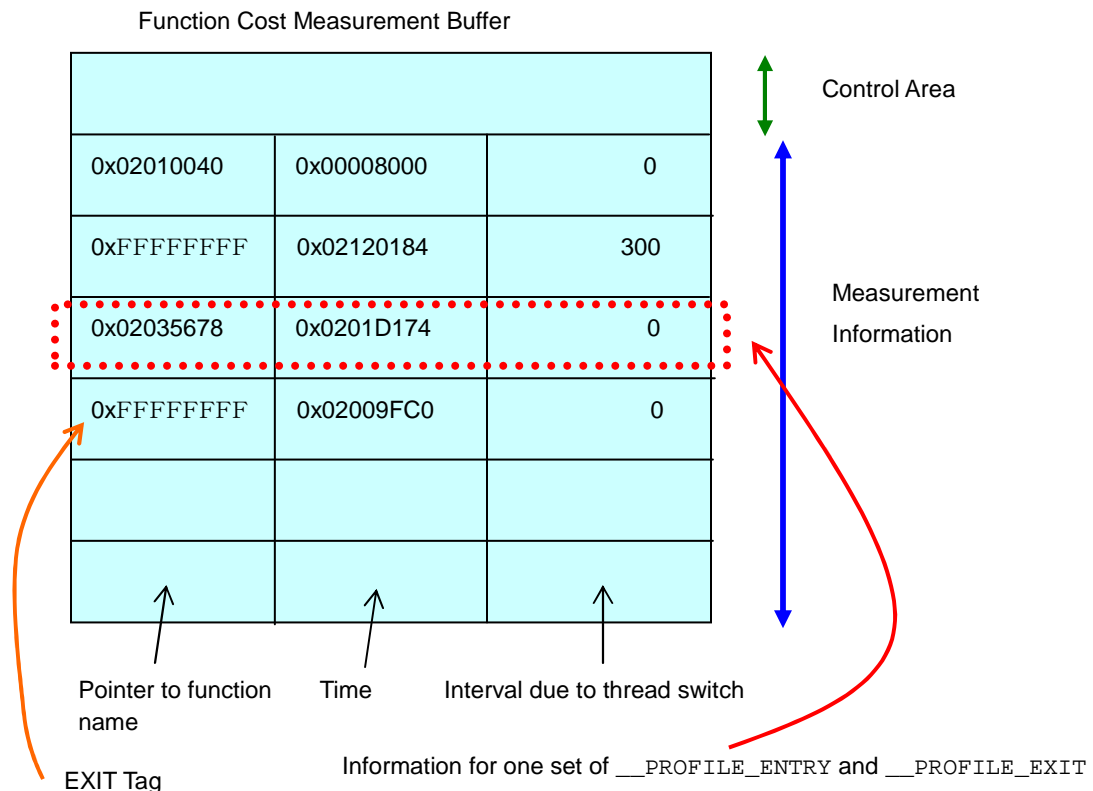
With `__PROFILE_EXIT`:

- Special value for area where pointer was saved with `__PROFILE_ENTRY` (called the EXIT tag value.).
- Current time, value of `OS_GetTickLo()`
- Interval due to thread switch if required (optional)

The current time is a value that can be obtained with `OS_GetTickLo()`. The Tick feature of the OS has a 64-bit value, but it is sufficient to only check the lower half when calling a function so it is managed as a 32-bit value.

The special value (called the EXIT tag) for distinguishing the pointer to the character string of `__PROFILE_ENTRY` with `__PROFILE_EXIT` is secured in the pointer to the function name character string.

The amount of time it takes to change threads (including the time spent in any other thread) is deducted from the total time elapsed from `__PROFILE_ENTRY` to `__PROFILE_EXIT`.



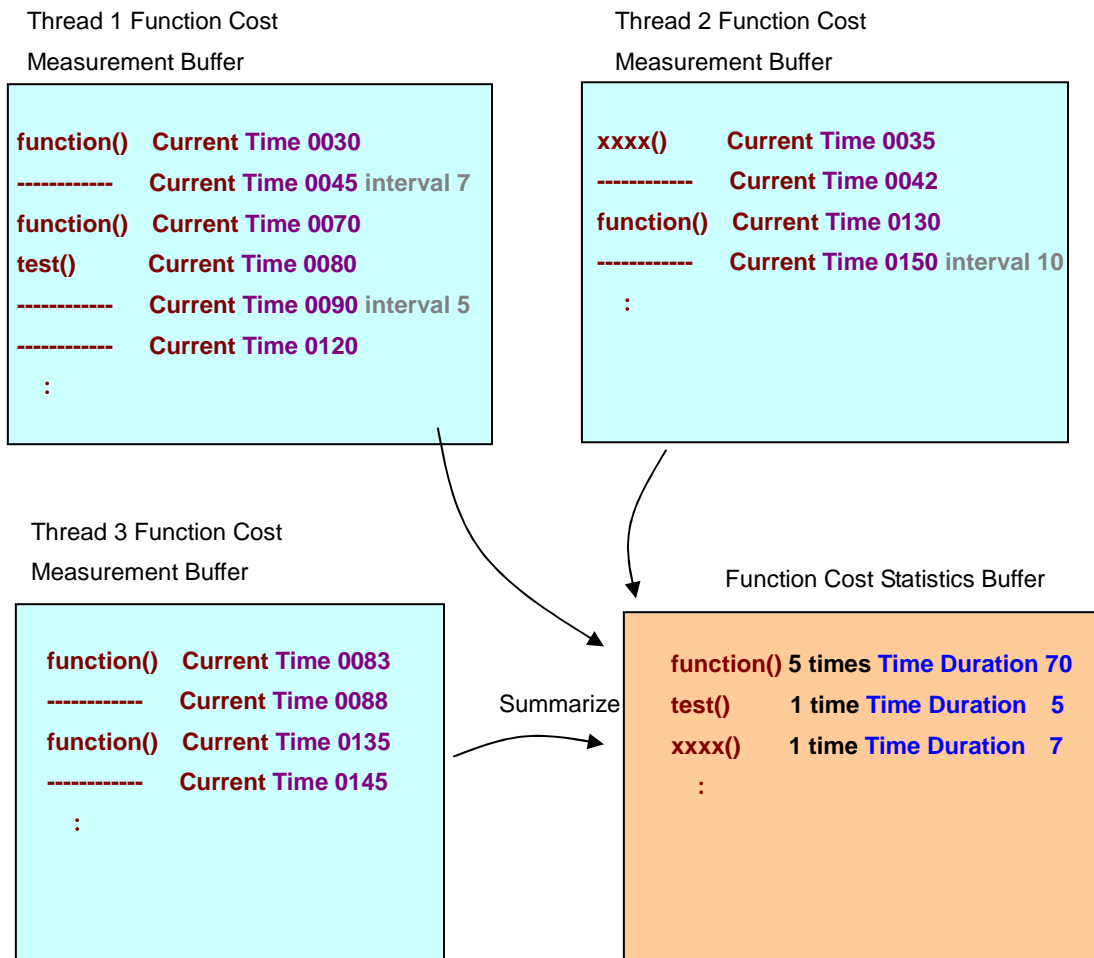
Types of information stored in the “Control Area” include: the part of the buffer currently being used, location of the upper limit of the buffer, the counter value for the duration of a thread switch, etc.

4.3 Conversion to Statistics Buffer

It is difficult to obtain cost information only with the function cost measurement data since the measurement data needs to be summarized as statistics buffer data.

The summary relates the call information of the function and the EXIT tag, then totals the number of calls and the time spent in the 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 an interval line of the EXIT tag. Calculations are carried out to 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. Since the same summarization buffer can be shared with multiple threads, avoid summarizing a separate thread while summarizing on another thread.



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 a function is initialized. The Tick system of the OS 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 controlling the buffer and the actual time information are stored. If you know the size of a buffer and want to know how many lines it can store, use the following function.

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

    size        Buffer size (bytes)
    Return Value Number of lines that can be secured (number of information sets for cost measurement)
```

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

```
// Calculate the buffer size based on the number of
// measurement information that can be stored.

u32 OS_CalcFunctionCostBufferSize( int lines );

    lines        Number of lines in the buffer
    Return Value required size of you 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 value of 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_DispStatistics( void* statBuf );
```

statBuf Statistics buffer

You can temporarily stop recording profiling data or restore the setting by using the following functions. Recording of information remains disabled even if `__PROFILE_ENTRY()` or `__PROFILE_EXIT()` are called. If a thread switch takes place 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();

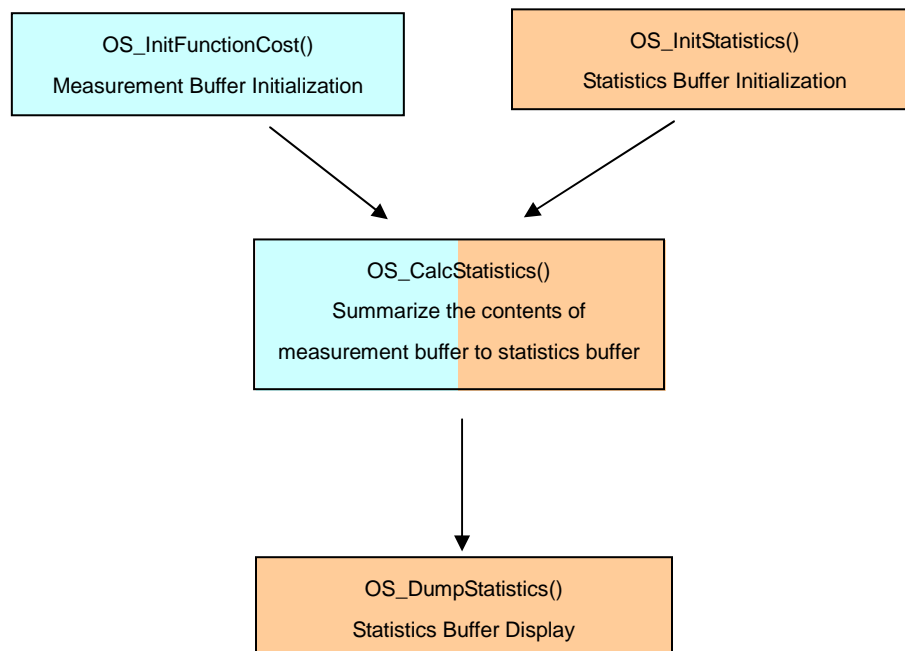
    //---- init functionCost
    OS_InitFunctionCost( &CostBuffer, COSTSIZE );
    OS_InitStatistics( &StatBuffer, STATSIZE ); // This initialization can be
done after measurement

    : // This is the location to be measured

    //---- 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 for `test1()` with an elapsed time (duration) of 25. (The units of this value are the same as those used in the Tick system of the OS.)

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

4.6 Specification When Linking

To enable the function cost measurement feature, `TWL_PROFILE_TYPE=FUNCTIONCOST` (or `NITRO_PROFILE_TYPE=FUNCTIONCOST`) must be specified for the make option. Due to this setting, `libos.FUNCTIONCOST.a` (or `libos.FUNCTIONCOST.thumb.a`) is included when linking. This can also be described in the makefile.

4.7 Operation on Thread

If a thread system is being used, the function cost measurement information runs independently for each thread. Therefore, initialization of a particular buffer declared with `OS_InitFunctionCost()` only records information from the thread in which it was generated. Status settings for functions like `OS_EnableFunctionCost()` are also independent for each thread.

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

4.8 Cost

Because time information is saved in the buffer every time the function is called, function calls cost more than the normal operation. Since every function must include the `__PROFILE_ENTRY/EXIT` calls, the optimization during compile is not as much as expected, compared to a situation where there is no restrictions. Further, to save the pointer to the function name in the buffer, the function name string is placed on the memory, which causes additional memory usage.

The operational cost changes based on factors such as if there is a thread or not. With `__PROFILE_ENTRY()` an extra 25 – 35 instructions, and with `__PROFILE_EXIT()` an extra 20 – 30 instructions are passed. Also, interval calculation is done when the thread is switched so an extra 30 – 40 instructions are required. 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 Other Profilers (other than TWL-SDK)

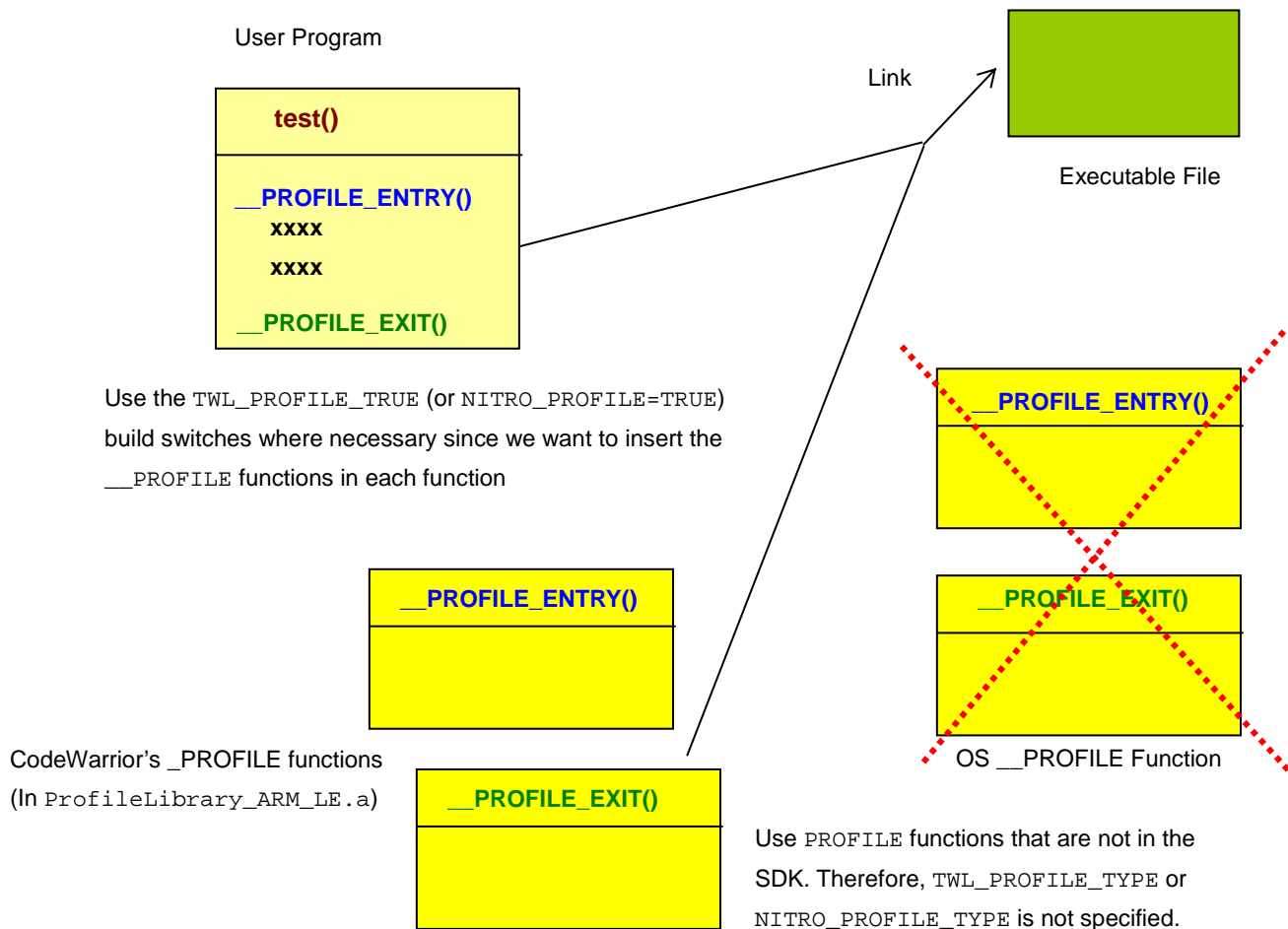
Preparing `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` allows you to use a profiler other than the one provided in the Nitro-SDK OS.

For example, if you use the profiler offered as the CodeWarrior Example, `__PROFILE_ENTRY()` and `__PROFILE_EXIT()` are defined within it so the ones provided in the OS should not be defined..

5.1 Specification When Linking

`TWL_PROFILE_TYPE` (or `NITRO_PROFILE_TYPE`) must be specified for elements other than `CALLTRACE` or `FUNCTIONCOST` during an OS compile (In other words, nothing needs to be specified). Due to this, profile libraries such as `libos.CALLTRACE.a` or `libos.FUNCTIONCOST.a` will not be linked.

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

2008/09/26, 0.3.0, Revised document to reflect TWL.

2004/08/11, 0.2.0, Revised the error in 3.4 where “stack mode” was “trace mode”

2004/, 0.1.0, Initial version.