# Determining the stack usage of applications

AN 316, January 2025, V 1.3

feedback@keil.com


## Abstract

Determining the required stack sizes for a software project is a crucial part of the development process. The developer aims to create a stable application, while not wasting resources. This application note explains methods that help finding the optimal setting while looking specifically on the stack load caused by interrupt service routines (ISRs) in RTOS applications running on an Arm Cortex-M based processor.


## Contents


Abstract ........ 1
Introduction ........ 2
Usage of Stack Memory ........ 2
  Stack usage of Interrupt Service Routines ........ 3
    Memory requirement for automatic register stacking ........ 3
  Stack usage of the RTX5 Kernel ........ 4
Analysis of Stack Usage ........ 5
  Static analysis ........ 5
  Dynamic analysis ........ 6
    Thread stack watermarking ........ 6
    Main stack watermarking ........ 7
Calculate and configure stack usage ........ 10
  Thread stacks ........ 10
  Main stack ........ 10
Example ........ 11
  Thread stack usage ........ 11
    Dynamic stack analysis ........ 11
    Static analysis ........ 12
    Configure thread stacks ........ 12
  Main stack usage ........ 12
    Static analysis ........ 12
    Calculate main stack size ........ 13
Summary ........ 13
References ........ 13

## Introduction

Stacks are memory regions where data is added or removed in a last-in-first-out (LIFO) manner. In an RTOS, each thread has a separate memory region for its stack. During function execution, data may be added on top of the stack; when the function exits, it removes that data from the stack.

In a Cortex-M processor system, two stack memory regions need to be considered:

- The *system stack* is used before the RTOS kernel starts and by interrupt service routines (ISRs). It is addressed via the Main Stack Pointer (MSP).
- The *thread stack(s)* are used by running RTOS threads and are addressed via the Process Stack Pointer (PSP).

As the memory region for stack is constrained in size, allocating more memory on the stack than is available, can result in a program crash or stack overflow. In embedded systems, the timing of external program events influences the program flow and a stack size issue may create infrequent, sporadic program errors. It is therefore critical to understand the stack memory requirements of an application.

For calculating (and therefore optimizing) the required stack memory size, the following methods may be used:

- *Static analysis* (using call tree analysis) is performed at build time (by a linker for example).
- *Dynamic analysis* (using stack watermarking) is performed at run-time (in a debug session for example).

## Usage of Stack Memory

In an embedded application, the stack memory is typically used in the following constructs:

- On function calls to save register content (such as the link register (LR) for the return address)
- Local function variables are stored on the stack when no CPU registers are available.
- For interrupt service execution, the register frames are store on the stack.

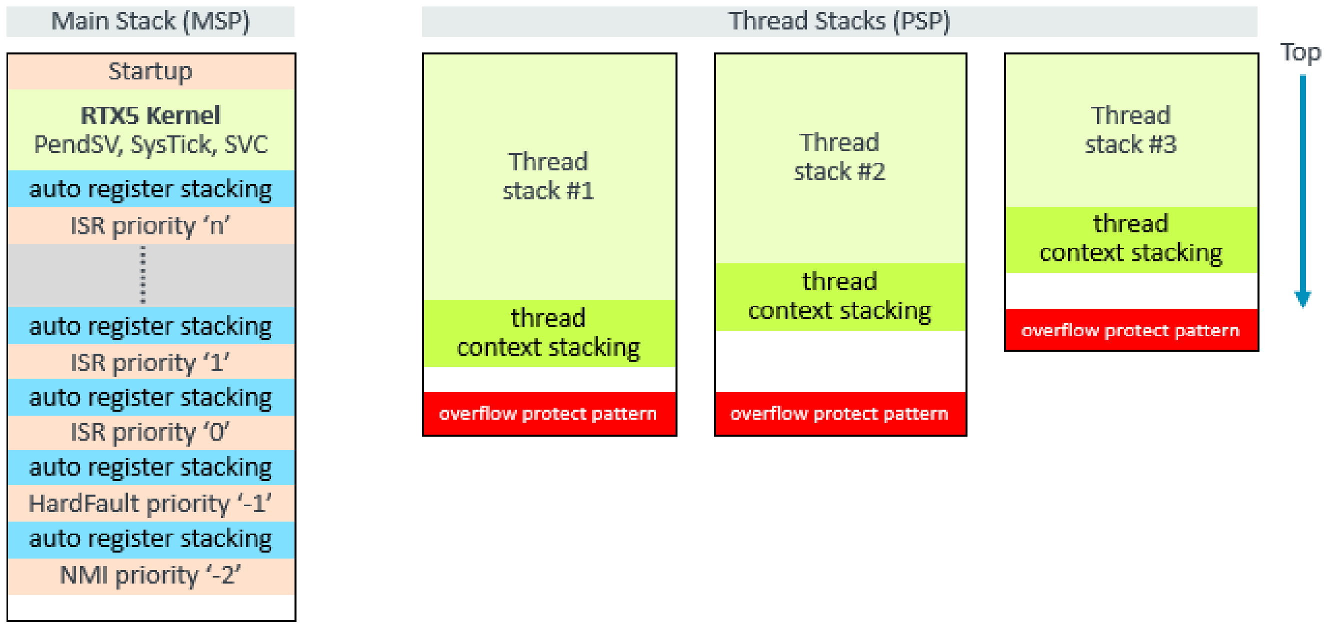
The application programmer may influence the stack memory usage with for following techniques:

- For arrays, allocate space from memory pools instead of local function variables.
- Reduce the potential interrupt nesting by choosing the right number of interrupt priority levels.
- Simplify the function call nesting. However, as this impacts the program readability, there is a balance. Also, modern compiler optimizations perform automatic function in-lining and therefore function call nesting is less important.

The picture below shows the stack usage of an embedded application that is using an RTOS kernel. ISRs use the *main stack*, a thread uses the *thread stack* whereby each thread has its own stack space that is managed by the RTOS kernel. Each *thread stack* should reserve additional memory that is required for “thread context switching”. The memory required for “thread context switching” depends on the usage of the floating-point unit (FPU):

- without FPU: 64 bytes (to save R0..R12, LR, PC, xPSR)
- with FPU: 200 bytes (to save S0..S31, FPSCR, R0..R12, LR, PC, xPSR)

Optionally, an RTOS stores an “overflow protect pattern” (which is a fixed value) at the stack bottom which is used by the kernel to check for stack overflows.




Note that RTX5 itself executes in handler mode and uses the *main stack* for kernel functions. This is different from other RTOS kernels (i.e. FreeRTOS), where the kernel functions use the *thread stack* and therefore require additional memory space on each individual *thread stack*.

## *Stack usage of Interrupt Service Routines*

Interrupt service routines run when an exception has occurred and use the *main stack*. They are triggered by a peripheral, hardware fault, or by software with the Service Call (SVC) instruction. For interrupt service routines, the processor does automatic register stacking on the current active stack: when thread stack is active, PSP is used, otherwise MSP.

### Memory requirement for automatic register stacking

The memory required for automatic register stacking depends on the actual stack alignment and the usage of the floating-point registers of the program code that is interrupted. The usage of the floating-point registers is indicated by the processor in CONTROL register - FPCA bit (bit 2):

- When CONTROL – bit 2 = 0: automatic register stacking uses 32 bytes (+ 4 bytes aligner)
- When CONTROL – bit 2 = 1: automatic register stacking uses 104 bytes (+ 4 bytes aligner)

**NOTES:**

- For Cortex-M processors without hardware FPU (Cortex-M0/M0+/M3/M23) always use 32 bytes for automatic register stacking.
- For Cortex-M processors with hardware FPU, it might be complex to analyze the floating-point register usage of the various threads and ISRs. In this case, always use 104 bytes for automatic register stacking.

Interrupt service routines can be nested due to preemption of interrupts or exceptions. Cortex-M processors have the following configurations that influence the maximum nesting:

- Each interrupt source has a priority register, whereby lower values indicate higher priority.
- The AIRCR (Application Interrupt and Reset Control Register) contains a PRIGROUP field that defines the split of the priority register into a group priority and sub-priority within the group. Only a lower group priority value can preempt code execution.
- Some exceptions have a fixed priority which is typically higher than other interrupt sources.

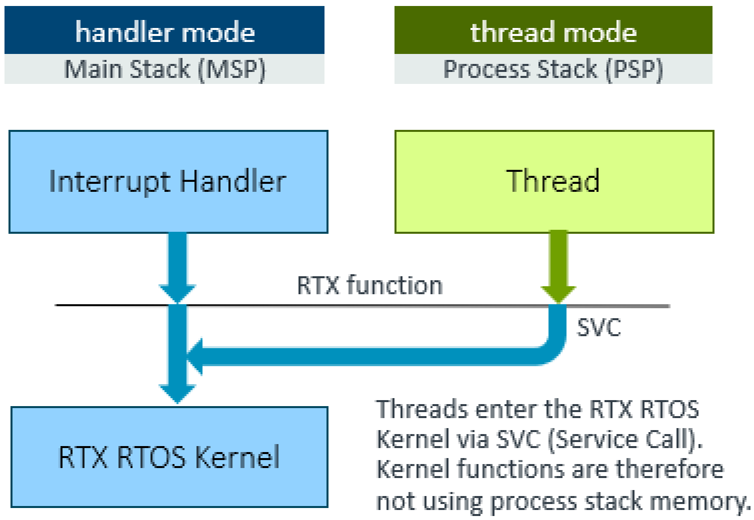
To consider the interrupt nesting the maximum depth of the stack loads must be added.

**NOTE:**

Consider reducing the maximum interrupt nesting by reducing the potential group priority levels with the AIRCR->PRIGROUP field (refer also to the CMSIS function NVIC_SetPriorityGrouping). Note that the group priority level must be configured before starting the RTX5 kernel with the osKernelStart() function.

### *Stack usage of the RTX5 Kernel*

The RTX5 Kernel is always executed in handler mode. This differs from several other RTOS kernels where the kernel functions itself use the *thread stack* and therefore each thread must consider this extra stack load.




The RTX5 Kernel uses the following interrupt service routines:

- SVC for most of the RTX functions
- SysTick for the RTX5 Kernel tick
- PendSV for RTX function calls from other interrupt service routines.

The priorities of SVC, SysTick, and PendSV are different, but these ISRs are never nested and therefore the user must only consider the maximum stack load of one path (the highest stack usage of SVC, SysTick, or PendSW).

The stack requirements of the RTX5 Kernel depend on the compiler and the optimization level. As RTX5 supports event annotations and this configuration impacts also the stack requirement. For technical details, refer to the CMSIS documentation under “CMSIS-RTOSv2 – RTXv5 Implementation – Technical Data – Stack Requirements”. For this application note we use the information for Arm Compiler ARMCC v5.06 with -O0. The stack requirements for the SVC/SysTick/PendSV is:

- 176 bytes when not using the Event Recorder
- 360 bytes when using the Event Recorder

**NOTE:**

Refer to the CMSIS documentation under “CMSIS-RTOSv2 – RTXv5 Implementation – Technical Data – Stack Requirements”, as the stack requirements might be different.

## Analysis of Stack Usage

There are two different methods to analyze the required memory size of a stack:

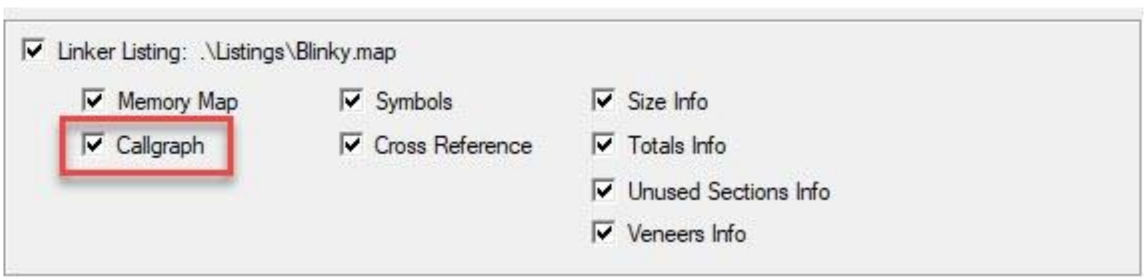
- **Static analysis** does not require to execute the program. It counts the stack requirements of each individual function and requires knowledge of the program flow. The program flow of complex applications may be hard to track as function pointer values might be not known. Static analysis is typical the best method for the *main stack* as under test conditions the worst-case ISR nesting will rarely occur.
- **Dynamic analysis** requires the program to be executed with all possible conditions. Typically, it is examined using a debugger that watches the memory stack usage. Dynamic analysis is the preferred method for the *thread stack* as it delivers the real stack memory requirement (static analysis may delivery significant higher values due to worst case assumptions of the program flow that do not occur during real-world execution).

### *Static analysis*

*Static analysis* uses the program flow (or call tree) to track the stack memory usage for every function and the related call tree. As it does not require to execute the program, it is the best method for evaluating the stack requirement. However, *static analysis* has restrictions when function pointers or assembly code is used, as it may be impossible to track the exact control flow and hence calculate the stack usage.

*Static analysis* can be performed by the Arm linker (armlink) with the `--callgraph` option. Refer to the "Linker User Guide, Linker command-line Options, --callgraph" (www.keil.com/support/man/docs/armclang_link/armclang_link_pge1362075422709.htm).

In µVision enable **Callgraph** under **Project – Options for Target – Listing**:




This generates an HTML file (in the folder of the output *.axf file) that contains the call tree along with stack usage information.

A snippet of an example listing is shown below. The function 'phaseA' (defined in blinky.o) has a maximum stack depth (Max  Depth) of 264 bytes when executing the listed call chain:

```
phaseA (Thumb, 84 bytes, Stack size 24 bytes, blinky.o(.text))

[Stack]
  - Max Depth = 264
  - Call Chain = phaseA ⇒ __hardfp_sin ⇒ __ieee754_rem_pio2 ⇒ __aeabi_dsub ⇒ __aeabi_dadd ⇒
_double_epilogue ⇒ _double_round
```

**IMPORTANT:**

The linker call graph report ("Max Depth" value) does not contain the additional memory space that is required for "thread context switching" or "automatic register stacking".

## *Dynamic analysis*

*Dynamic analysis* requires that the application executes the program paths that cause the maximum stack usage in a debug session. In practice, this method relies typically on a fixed memory pattern value that is checked by the debugger and is therefore called also stack watermarking.

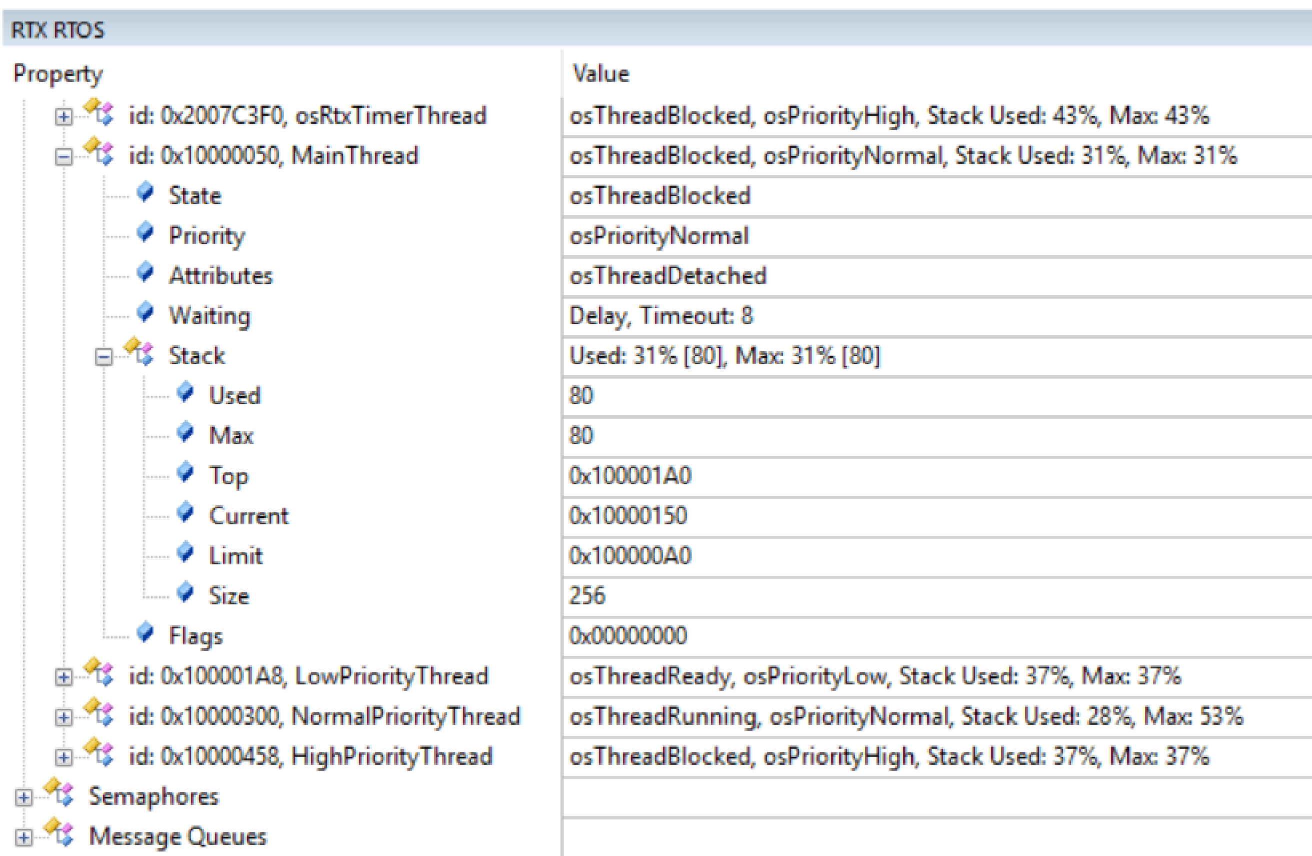
**IMPORTANT:**

With dynamic analysis it is hard to capture the maximum usage of the main stack, as the interrupt nesting depends also on the timing of interrupt events. The worst-case scenario will rarely occur.

However, the method can reliably evaluate the memory requirement of a thread stack (addressed by PSP), when complete execution of the thread functionality is ensured.

### Thread stack watermarking

For RTX, stack watermarking can be enabled in the RTOS configuration file RTX_Config.h. In a debug session, the current maximum stack usage per thread is then shown in the Component Viewer window for RTX (access via **View – Watch Windows – RTX RTOS – Threads**). This measurement covers also register stacking of interrupts and exceptions that occur during thread execution. However, you should add on top of the additional space for automatic register stacking as the timing of interrupts may have not occurred at the maximum depth of the function nesting.



### Main stack watermarking

Stack watermarking may be used also for the main stack (MSP). In embedded systems, interrupt execution depends on the timing of external program events and therefore it is almost impossible to capture the maximum interrupt nesting using stack watermarking.

If stack watermarking should be used for the "main stack", it can be added as described below. The example project **AN316.uvprojx** that is part of this application note contains the relevant files already.

**Step 1:** Add the following assembler module **FillSystemStack.S** to the project. It initializes the correct main stack area with constant values.

```
    .syntax  unified

    // specify the symbols referencing the stack area
    .equ StackBase, Image$$ARM_LIB_STACK$$ZI$$Base
    .equ StackLimit, Image$$ARM_LIB_STACK$$ZI$$Limit
    //.equ StackBase, STACK$$Base
    //.equ StackLimit, STACK$$Limit

    .global StackBase
    .global StackLimit

    .thumb
    .section .text.FillSystemStack, "ax", %progbits
    .global  FillSystemStack
    .p2align 2
    .type    FillSystemStack, %function
    .thumb_func
FillSystemStack:
    .fnstart
    .cfi_sections .debug_frame
    .cfi_startproc
    ldr      r0,=StackBase
    mov      r1, sp
    ldr      r2,=0xCDCDCDCD
    ldr      r3,=0xABABABAB
    str      r3,[r0]
    b        Loop_Check

Loop:
    str      r2,[r0]

Loop_Check:
    adds     r0,r0,#0x04
    cmp      r0,r1
    bne      Loop
    bx       lr

    .size FillSystemStack, .-FillSystemStack
    .cfi_endproc
    .cantunwind
    .fnend

    .end
```

Step 2: Configure FillSystemStack.S so that symbols StackBase and StackLimit are set to the correct stack range values. Two options using Arm "linker defined symbols" for stack configured in the scatter file or the standard CMSIS assembler startup files (that define the main stack memory region) are already given and can be commented/uncommented. If the project uses other symbols, they need to be used accordingly.

**Step 3:** Call the function `FillSystemStack()`. It initializes the main stack with a fixed value (0xCDCDCDCD) and an overflow protection value (0xABABABAB) at the bottom of the stack. This function `FillSystemStack()` should be called at the beginning of the `main()` function as it writes the required memory pattern for the debugger to perform stack watermark analysis.

**NOTE:**

If the function shall be called early in the reset handler, make sure the Arm runtime library startup code does not initialize the stack area with zeros. Normally this will happen, as the stack area is ZI data. And ZI initialization would erase the patterns written by `FillSystemStack()` again.
To prevent ZI data from getting zero initialized, the **UNINIT** region attribute needs to be used, as in this example for the stack configured in the scatter file:

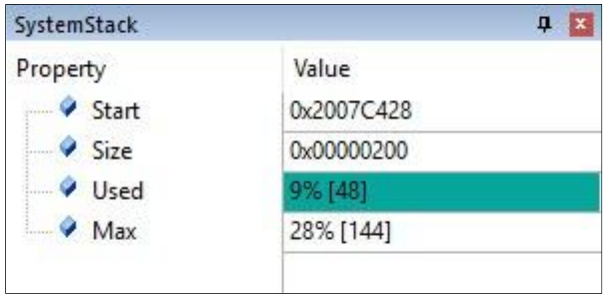
```
  ARM_LIB_STACK __STACK_TOP UNINIT EMPTY -__STACK_SIZE
  {   ; Reserve empty region for stack
  }
```

**Step 4:** For the µVision debugger the [SCVD](#) file **SystemStack.svcd** implements the related viewer:

```
<?xml version="1.0" encoding="utf-8"?>
<component_viewer schemaVersion="0.1" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="Component_Viewer.xsd">
  <component name="MyExample" version="1.0.0"/>
   <objects>
    <object name="SystemStack">
      <var  name="StackStart" type="int32_t" value="0"  />
      <var  name="StackLimit" type="int32_t" value="0"  />
      <var  name="StackSize"  type="int32_t" value="0"  />
      <var  name="StackMax"   type="int32_t" value="0"  />
      <var  name="StackUsed"  type="int32_t" value="0"  />
      <calc cond="__Symbol_exists (&quot;Image$$ARM_LIB_STACK$$ZI$$Base&quot;)">
        <!-- symbols for stack configured in scatter file -->
        StackStart = __FindSymbol ("Image$$ARM_LIB_STACK$$ZI$$Base");
        StackLimit = __FindSymbol ("Image$$ARM_LIB_STACK$$ZI$$Limit");
      </calc>
      <calc cond="__Symbol_exists (&quot;STACK$$Base&quot;)">
        <!-- symbols for stack configured in Keil assembler startup file -->
        StackStart = __FindSymbol ("STACK$$Base");
        StackLimit = __FindSymbol ("STACK$$Limit");
      </calc>
      <calc>
        StackSize  = StackLimit - StackStart;
        StackMax   = __CalcMemUsed(StackStart, StackSize, 0xCDCDCDCD, 0xABABABAB);
        StackUsed  = StackLimit - __GetRegVal("MSP");
      </calc>
      <calc cond="StackUsed&gt;(StackMax&amp;0xFFFF)">
        <!-- in case stack was allocated, but not used -->
        StackMax = ((StackUsed*100/StackSize)&lt;&lt;20)|StackUsed;
      </calc>
      <out name="SystemStack">
        <item property="Start" value="%x[StackStart]"/>
        <item property="Size"  value="%x[StackSize]" />
        <item>
          <print cond="__Running == 0"
            property="Used"
            value="%d[StackUsed*100/StackSize]%% [%d[StackUsed]]" />
          <print cond="__Running == 1" property="Used" value="unknown" />
        </item>
        <item alert="(StackMax >> 31)" property="Max"
            value="%d[(StackMax>>20) &amp; 0xFF]%% [%d[StackMax &amp; 0xFFFFF]]" />
      </out>
    </object>
  </objects>
</component_viewer>
```

**Step 5:** If the application uses other symbols than the SCVD file is already checking for, it needs to be adjusted accordingly. The **StackStart** and **StackLimit** variables need to refer to the same stack area related symbols that are already used in the **FillSystemStack.S** assembler module.

**Step 6:** Add the **SystemStack.svcd** file to the debug session via **Project – Options for Target – Debug – Manage Component Viewer Description Files**. In a running debug session, open this view via **View – Watch Windows – SystemStack**:




After the call to `FillSystemStack()`, the maximum system stack usage is reported.

## Calculate and configure stack usage

### *Thread stacks*

The worst-case *thread stack* usage can be determined using the “Thread stack watermarking” method described on page 6. To get the maximum stack usage, ensure that all functions of a thread are executed. Then calculate the required stack memory with the following steps:

1. Open the RTX Component Viewer (**View – Watch Windows – RTX RTOS – Threads**) to get the maximum stack usage of a thread.
2. Thread switches may have unpredictable timing. Add therefore additional memory for “thread context switching”: 64 Bytes for processors without FPU), 200 Bytes when using FPU.
3. Round up the stack usage to a multiple of 8 to consider alignment requirements.

The thread stack can be allocated from a memory pool or provided as static memory to osThreadNew().

As an alternative you may use “Static analysis” provided by the linker to supply the stack usage for step 1. However, keep in mind that “Static analysis” might be tricky for applications that use function pointers.

For example, for “Static analysis” of the RTX Timer Thread stack usage must also consider the timer callback functions. When the function A, B, C are called as timer callbacks and the call graph shows for function A=16, B=8, and C=32 bytes stack usage, the maximum (32) must be added to the stack depth of osRtxTimerThread.

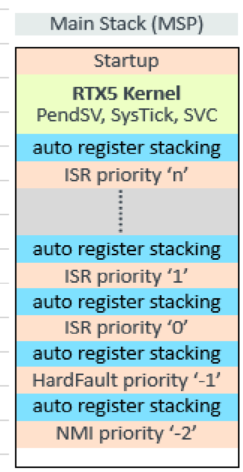
### *Main stack*

The amount of *main stack* can be calculated by adding the memory requirements of the startup code and each potential ISR routine, considering the various group priority levels. To calculate the *main stack* usage, the Excel spread sheet ***MSP_Calculation.xlsx*** is part of this application note. Using the instructions, it is easy to calculate the total memory requirements for the *main stack.* As an alternative, the numbers can be added manually, whereby the “Memory requirement for automatic register stacking” should be considered.

Calculate **main stack** memory requirements using the following instructions:

1. Insert all interrupt sources of your application software
2. Enter the group priority of the interrupt sources
3. Use static analysis provided by the linker to enter max. stack requirement for each ISR
4. If an ISR is using floating registers, enter '1' under "Uses FP regs"
5. Enter the stack usage for SVC (RTX) from the information provided with the RTX RTOS kernel
6. For each "Group Priority", consider only the ISR with the max. stack usage

| **Source** | **Group Priority** | **Max Stack** | **Uses FP regs** | **Total Stack** |
|---|---|---|---|---|
| Startup code | | | | 32 |
| RTX5 Kernel | | 360 | 0 | 392 |
| IRQ0 | 1 | 16 | 1 | 120 |
| IRQ1 | 0 | 60 | 0 | 92 |
| IRQ2 | 0 | 40 | 0 | |
| HardFault | -1 | 0 | 0 | 32 |
| NMI | -2 | 0 | 0 | 0 |
| **Total (bytes)** | | | | **668** |




The size of the main stack is (depending on the method to configure the stack) configured in the scatter file or startup file of the related microcontroller device.

## Example

To demonstrate stack usage calculation, a code example running on an NXP LPC1768 with an Arm Cortex-M3 core is provided. The project **AN316.uvprojx** can be downloaded with this application note and either runs in simulation or on real hardware using the MCB1700 development board.

### *Thread stack usage*

#### Dynamic stack analysis

The "Thread stack watermarking" method delivers the threads the following results:

| | |
|---|---|
| `osRtxIdleThread` | 64 bytes |
| `osRtxTimerThread` | 96 bytes |
| `MainThread` | 72 bytes |
| `LowPriorityThread` | 80 bytes |
| `NormalPriorityThread` | 128 bytes |
| `HighPriorityThread` | 80 bytes |

The dynamic stack analysis also includes the stack space required for the "thread stack switch". However, as the timing of embedded applications depend also on external inputs, it is not guaranteed that it happened at the maximum stack load. Therefore, reserve additional space to allow the worst-case execution of the "thread stack switch" (which is 64 bytes in our example).

The user thread with the maximum stack load is the `NormalPriorityThread` and its worst-case stack requirement is in our example: 128 + 64 = 192 bytes.

The maximum stack load for the timer thread is: 96 + 64 = 160 bytes

As the idle thread is empty, we use the number that we obtained with static stack analysis below.

#### Static analysis

The callgraph file **.\Objects\test.html** contains the stack load calculated with "Static analysis". The values for the threads are:

| | |
|---|---|
| `osRtxIdleThread` | 0 bytes |
| `osRtxTimerThread` | 88 bytes |
| `MainThread` | 8 bytes |
| `LowPriorityThread` | 48 bytes |
| `NormalPriorityThread` | 80 bytes |
| `HighPriorityThread` | 48 bytes |

The static stack analysis does not include the stack space required for the "thread stack switch". The numbers are therefore somewhat lower. For dynamic stack analysis it is important that all potential paths are executed. If in doubt, you should validate the results also with static stack analysis.

The user thread with the maximum stack load is the `NormalPriorityThread` and its worst-case stack requirement is in our example: 80 + 64 = 144 bytes.

The maximum stack load for the timer thread is: 88 + 64 = 152 bytes

As the osRtxIdleThread is empty the number obtained with static stack analysis is correct and the stack space required is therefore just: 0 + 64 bytes.

### Configure thread stacks

For configuring thread stack usage, consider also the "overflow protect pattern" and the required 8-byte alignment. The following uses the worst-case user thread stack requirement obtained with the "Thread stack watermarking" method. The minimum configuration settings for the application in RTX_Config.h are therefore:

| | |
|---|---|
| `OS_IDLE_THREAD_STACK_SIZE` | 64 + 4 + 4 = 72 bytes |
| `OS_TIMER_THREAD_STACK_SIZE` | 160 + 4 + 4 = 168 bytes |

For the user threads, the default stack size is used. The settings are therefore:

| | |
|---|---|
| `OS_STACK_SIZE` | 192 + 4 + 4 = 200 bytes |

**NOTE:**

The memory requirements could be further reduced by specifying a user stack space for the various threads with the osThreadNew function call. This allows to optimize memory for constrained systems.

## *Main stack usage*

The application example itself uses interrupt grouping 4 and enables three interrupts:

- `EINT3_IRQn` with group priority 0 – sub-priority 0
- `TIMER0_IRQn` with group priority 0 – sub-priority 1
- `TIMER1_IRQn` with group priority 1 – sub-priority 0

### Static analysis

The callgraph file **.\Objects\test.html** contains the stack load information.

The stack memory for "Startup" is the "Stack size" reported for 'main' (8 bytes) + 32 Bytes for osKernelStart. In our example it is therefore 40 bytes.

**`main`** (Thumb, 288 bytes, Stack size 8 bytes, main.o(.text.main))

For the RTX5 Kernel, the ARM Compiler V6.10 value with -O1 value is used: 152 bytes without Event Recorder.

The stack size for each ISR routine is also part of that callgraph file (use the information "Max Depth" or when not present the Stack size value to get the memory requirements in bytes). Round-up to a value that is a multiple of 8 to consider alignment.

**TIMER1_IRQHandler** (Thumb, 32 bytes, Stack size 120 bytes, main.o(.text.TIMER1_IRQHandler))

[Stack]
  Max Depth = 120
  Call Chain = TIMER1_IRQHandler

The ISR stack requirement values obtained from the callgraph file **.\Objects\test.html** are:

- `EINT3_IRQHandler`: 0 bytes
- `TIMER0_IRQHandler`: 12 rounded up to 16 bytes (for alignment)
- `TIMER1_IRQHandler`: 120 bytes

### Calculate main stack size

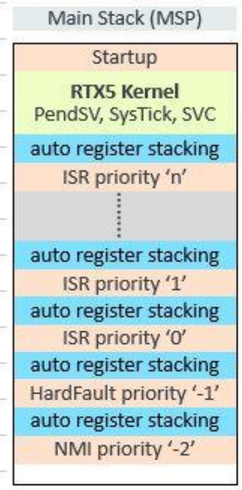
The values are then entered into the Excel spread sheet ***MSP_Calculation_test.xlsx*** as shown below. This calculates the total stack memory required for the main stack to 456 bytes.

The default configuration of the startup code is 512 bytes, therefore this setting may be reduced if memory is a critical resource.

Calculate **main stack** memory requirements using the following instructions:

1. Insert all interrupt sources of your application software
2. Enter the group priority of the interrupt sources
3. Use static analysis provided by the linker to enter max. stack requirement for each ISR
4. If an ISR is using floating registers, enter '1' under "Uses FP regs"
5. Enter the stack usage for RTX5 Kernel from the information provided with the RTX RTOS kernel
6. For each "Group Priority", consider only the ISR with the max. stack usage

| **Source** | **Group Priority** | **Max Stack** | **Uses FP regs** | **Total Stack** |
|---|---|---|---|---|
| Startup code | | | | 40 |
| RTX5 Kernel | | 152 | 0 | 184 |
| TIMER1_IRQ | 1 | 120 | 0 | 152 |
| TIMER0_IRQ | 0 | 16 | 0 | 48 |
| EXT3_IRQ | 0 | 0 | 0 | |
| HardFault | -1 | 0 | 0 | 32 |
| NMI | -2 | 0 | 0 | 0 |
| **Total (bytes)** | | | | **456** |




## Summary

Verifying the stack requirements is an essential task before releasing an embedded application. Stack overflows may occur infrequently but typically cause a shut-down of the operation. The instructions described here should be therefore part of every verification and validation process.

This application note provides detailed step-by-step instructions for calculating the stack requirements of an RTX5 based applications. It also contains helpful procedures (such as an Excel sheet) that help during the process. While the process is exemplified using RTX5, the information also applies to other Cortex-M based systems, regardless whether using a real-time operating system or not.

## References


- Arm Compiler –Linker Command-line Options contains information about the `--callgraph` option.
- Cortex-M4(F) Lazy Stacking and Context Switching explains the "Stack usage of Interrupt Service Routines".
- Cortex-M Devices Generic User Guides provide generic information about a processor and the various hardware stacks.
- Arm Blog: How much stack memory do I need for my Arm Cortex-M applications?