FreeRTOS - Stack Overflow

The stack size for a task in RTOS usually is fixed, therefore there is a chance that the stack will overflow if there is a nest of many calls. When stack overflows, the data is written beyond the stack bound which leads to corrupted data in another task or memory area. It only causes problem when the overwritten data is accessed, therefore, the problem does not occur immediately and is hard to detect correctly.

#arm #stm32 #memory #stack

Last update: 2021-08-10 16:50:49

Table of Content

1. Lab 0: Task Stack Overflow

Each task maintains its own stack. This is where function calls store parameters, and functions store its local variable.

Let's create a new project that has only one task and this task continues printing a counter value to an UART port.

1.1. Precondition

Let's create a new FreeRTOS application using STM32CubeMX with below settings:

1. Time base Source for HAL is moved to a general timer, such as TIM6 or TIM10, TIM11
2. An UART port is enabled, such as UART1 TX on the pin **PA9**
3. An LED output which is used as the heartbeat of the system
4. FreeRTOS is enabled with configs:
    a. CMSIS V2 is selected
    b. Only need the default task

1.2. Redirect `printf` to UART

The simplest way is to overwrite the low-level `_write()` function. Refer to UART Redirection for a full version, but it is not necessary for this lab.

```
int _write(int file, char *ptr, int len) {
    // block write to UART if it is not ready
    while(HAL_UART_GetState(&huart1) != HAL_UART_STATE_READY);

    HAL_StatusTypeDef hstatus =
        HAL_UART_Transmit(&huart1, (uint8_t*) ptr, len, HAL_MAX_DELAY);

    return (hstatus == HAL_OK ? len : 0);
}
```

1.3. The default task

In this default task, there is `counter` variable which will be counted up and printed out in the task main loop. The LED will be toggled every 1 second to indicate that the default task is running. Let's implement this in a simple way:

```
void StartDefaultTask(void *argument)
{
  uint32_t counter = 0xbeefbeef;
  /* Infinite loop */
```

```
  for(;;)
  {
    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    printf("Counting, %lu %lu %lu %lu. No problem?\r\n", counter++, counter++,
counter++, counter++);
    osDelay(1000);
  }
}
```

1.4. Compile and Run

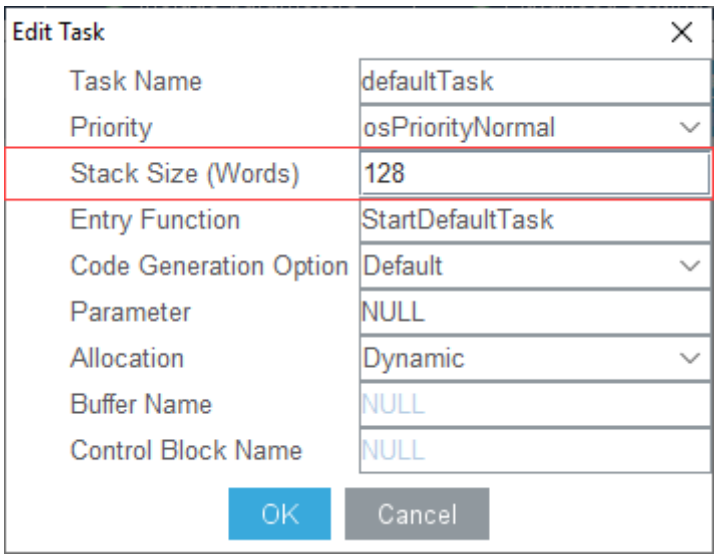
This simple project can be compiled and run successfully.

```
Counting, 3203383023 3203383024 3203383025 3203383026. No problem?
Counting, 3203383027 3203383028 3203383029 3203383030. No problem?
Counting, 3203383031 3203383032 3203383033 3203383034. No problem?
Counting, 3203383035 3203383036 3203383037 3203383038. No problem?
Counting, 3203383039 3203383040 3203383041 3203383042. No problem?
Counting, 3203383043 3203383044 3203383045 3203383046. No problem?
Counting, 3203383047 3203383048 3203383049 3203383050. No problem?
```

Messages are printed out successfully

1.5. Check the task stack

Be default, the minimum stack size of a FreeRTOS task is 128 Words (128 x 4 = 512 Bytes).

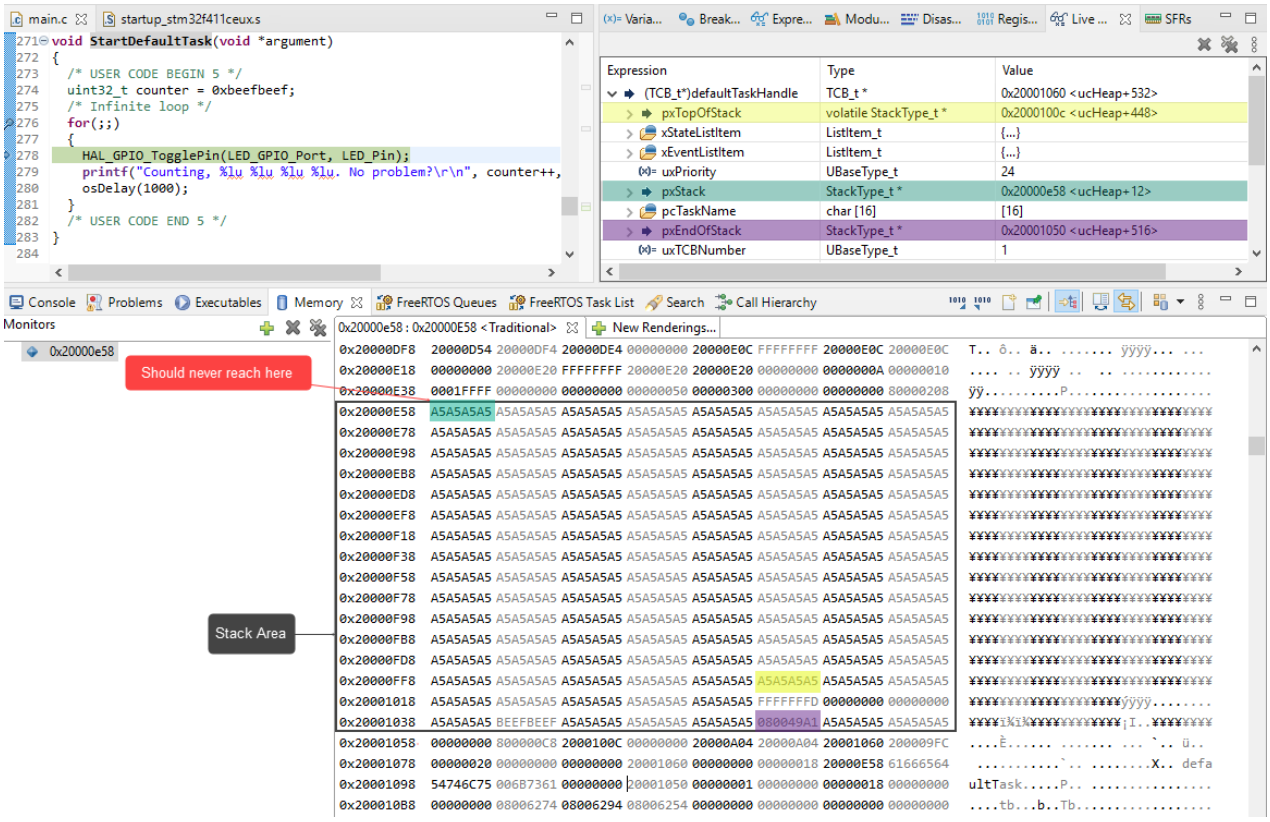


The settings of the default task

Set a breakpoint at the beginning of the main loop in the function `StartDefaultTask()` and read the structure of the default task control block by casting the pointer `defaultTaskHandle` to type of `TCB_t*` pointer in Live Expression view.

Note the below things:

- The start address of the Stack is stored in `pxStack`
- The end address of the Stack is stored in `pxEndOfStack`
- The whole stack is filled in a pattern `0xA5A5A5A5`
- The stack will be consumed from the end of the start back to the start address. The stack should never reach the start address to avoid stack overflow.



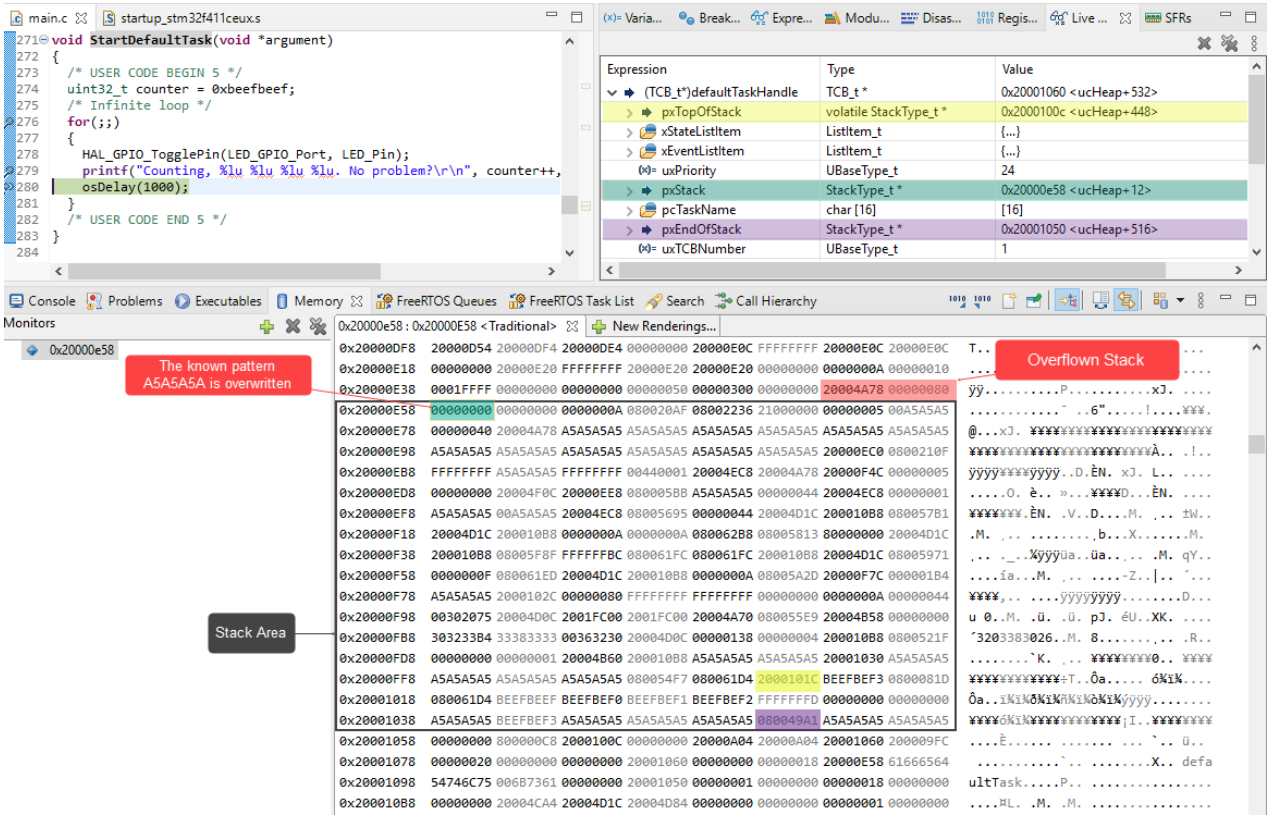
*Memory view **before** stack overflow*

> The task is initialized by the function `prvInitialiseNewTask()` which stores the task name, set priority, container lists, and finally, it prepares the task's stack. The top of the stack is initialized based on how stack size is defined. Usually, task stack is fixed, it means stack size is not growing, determined by `portSTACK_GROWTH = -1`.
>
> The top the stack will be changed, because the stack will be filled with some task information used for the task switching. The current top of the tack when a function is running is saved in the `PSP` register.

1.6. Step over the print function

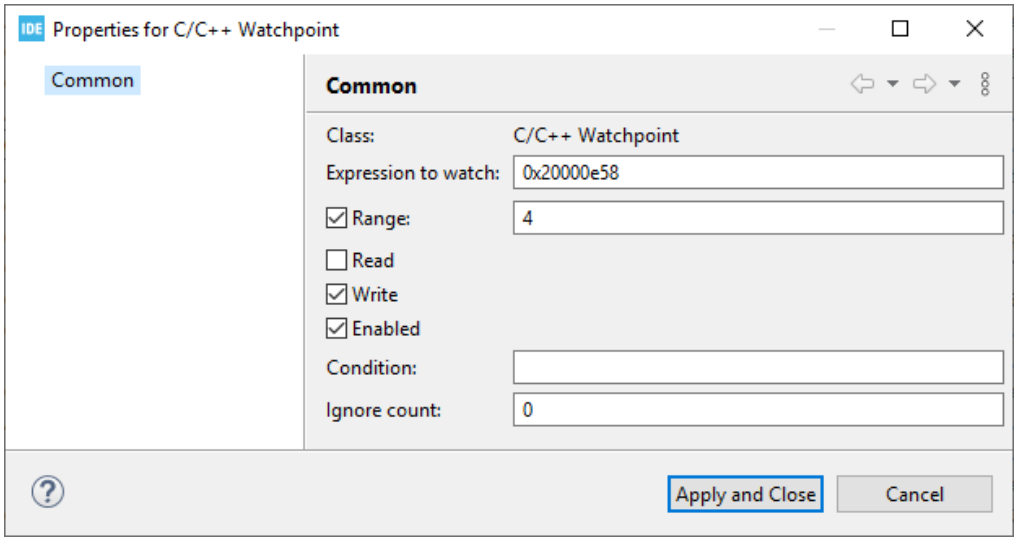
Set a breakpoint right after the `printf()` function to check the stack after the message is printout to UART. Now, look at the value stored at the start address of the stack. It's changed!

Moreover, 8 bytes beyond the stack boundary are also changed. Stack Overflow does really occur.



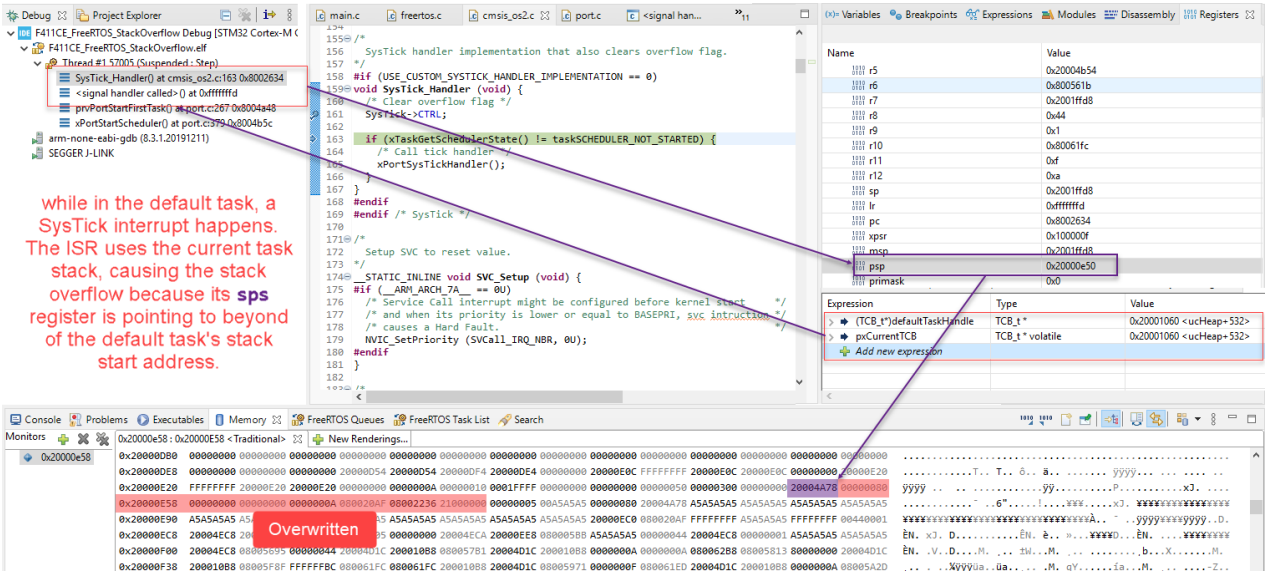
*Memory view **after** stack overflow*

How did it happen? An address can be added as a watch point, to check if that address is read, written. Select the start address of the stack and add its address to the watch list:



Adding a watch point by monitoring what writes into this memory

The printing function takes more than 1 ms to complete. The top of the stack of the default tasks is at `0x20000E50`, and a SysTick fires, the `SysTick_Handler()` ISR is called, causing it uses the current stack. The ISR consumes the stack and overwrites the stack boundary, going beyond the stack start address. It is stack overflow.



Stack overflow occurs when SysTick ISR is called

The good thing in this lab is there is no important bytes outside the default task's stack. After SysTick is handled, system still runs normally, without any problem.

2. Catching Stack Overflow

Stack overflow is a very common cause of application instability. FreeRTOS therefore provides two optional mechanisms that can be used to assist in the detection and correction of just such an occurrence. The option used is configured using the `configCHECK_FOR_STACK_OVERFLOW` configuration constant.

Note that these options are only available on architectures where the memory map is not segmented. Also, some processors could generate a fault or exception in response to a stack corruption before the FreeRTOS kernel overflow check can occur. The application must provide a stack overflow hook function:

```
void vApplicationStackOverflowHook( TaskHandle_t xTask,
                                    signed char *pcTaskName );
```

The `xTask` and `pcTaskName` parameters pass to the hook function the handle and name of the offending task respectively. Note however, depending on the severity of the overflow, these parameters could themselves be corrupted, in which case the `pxCurrentTCB` variable can be inspected directly.

⚠ Stack overflow checking introduces a context switch overhead so its use is only recommended during the development or testing phases.

In the function `vTaskSwitchContext()`, before switching to another task, the current task stack will be checked. The macro `taskCHECK_FOR_STACK_OVERFLOW()` will be called based on the stack overflow detection configuration.

2.1. Stack Overflow Detection — Method 1

This simple method check the Top of Stack is still in the stack range. The stack overflow hook function is called if the stack pointer contain a value that is outside the valid stack range. This method is quick but not guaranteed to catch all stack overflows. Set `configCHECK_FOR_STACK_OVERFLOW` to `1` to use this method.

```
#if( ( configCHECK_FOR_STACK_OVERFLOW == 1 ) && ( portSTACK_GROWTH < 0 ) )
  define taskCHECK_FOR_STACK_OVERFLOW()                                  \
    /* Is the currently saved stack pointer within the stack limit? */   \
    if( pxCurrentTCB->pxTopOfStack <= pxCurrentTCB->pxStack )            \
    {                                                                    \
        vApplicationStackOverflowHook( (TaskHandle_t) pxCurrentTCB,      \
                                       pxCurrentTCB->pcTaskName ); \
    }                                                                    \
#endif /* configCHECK_FOR_STACK_OVERFLOW == 1 */
```

2.2. Stack Overflow Detection — Method 2

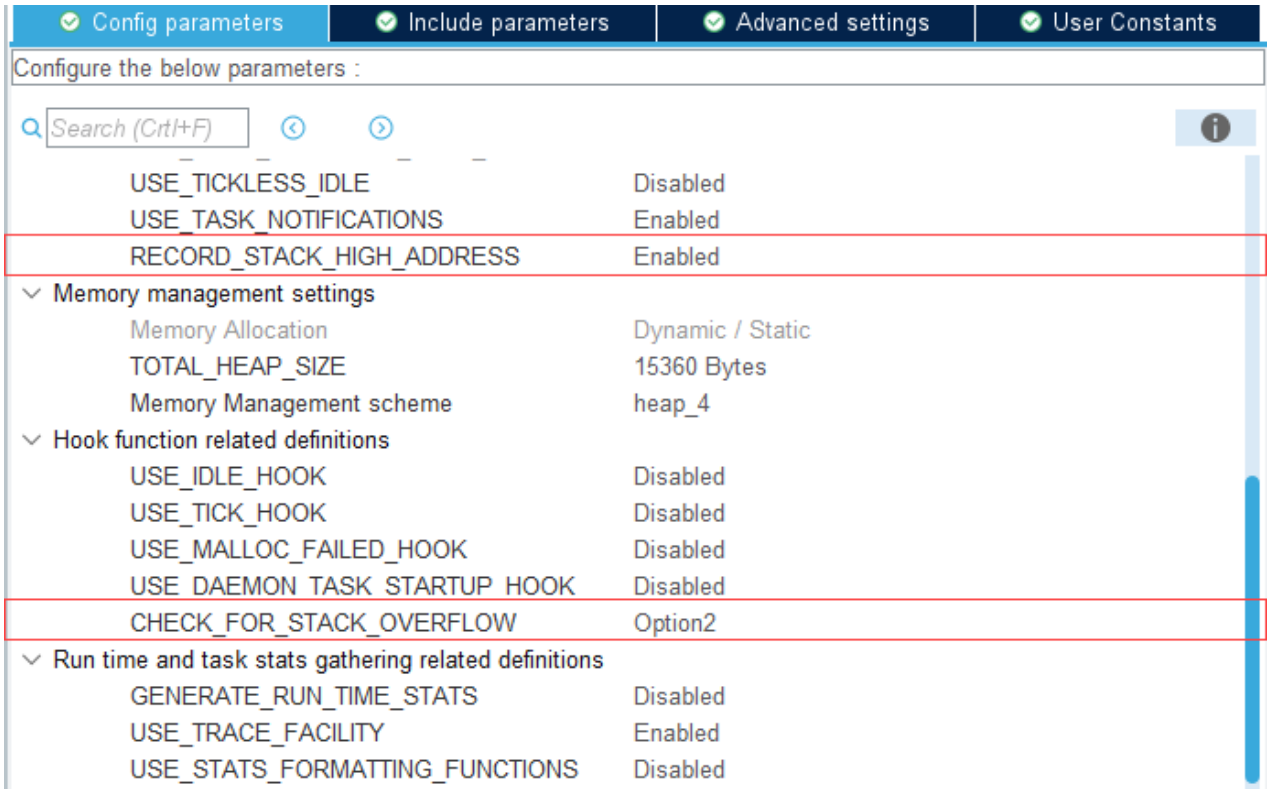
When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the FreeRTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value. This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.

```
#if( ( configCHECK_FOR_STACK_OVERFLOW > 1 ) && ( portSTACK_GROWTH < 0 ) )
  #define taskCHECK_FOR_STACK_OVERFLOW()                                 \
  {                                                                      \
    const uint32_t* const pulStack = (uint32_t*) pxCurrentTCB->pxStack; \
    const uint32_t ulCheckValue = (uint32_t  0xa5a5a5a5;                 \
    /* check if boundary bytes are changed? */                           \
    if( ( pulStack[ 0 ] != ulCheckValue ) ||                             \
        ( pulStack[ 1 ] != ulCheckValue ) ||                             \
        ( pulStack[ 2 ] != ulCheckValue ) ||                             \
        ( pulStack[ 3 ] != ulCheckValue ) )                              \
    {                                                                    \
      vApplicationStackOverflowHook((TaskHandle_t) pxCurrentTCB,         \
                                    pxCurrentTCB->pcTaskName );          \
    }                                                                    \
```

```
  }
#endif /* #if( configCHECK_FOR_STACK_OVERFLOW > 1 ) */
```

2.3. Trap the Stack Overflow

Firstly, enable a method to detect Stack Overflow by setting the `configCHECK_FOR_STACK_OVERFLOW` config.



Enable Stack Overflow detection

In the hook (callback) function, it can call to a soft breakpoint in Debug mode to stop debugger in the Stack Overflow hook function. In a release version, it is good idea to visually notify the system state by toggling a LED with a defined pattern.

```
void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName)
{
#ifdef CATCH_STACK_OVERFLOW
#ifdef DEBUG
  __BKPT();
#else
  for(;;) {
    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    HAL_Delay(100);
  }
#endif
#endif
}
```

If a stack overflow happens, use the `pcTaskName` parameter to note the task name, and examine the task information from the `xTask` handler.

Solutions

In the embedded world, especially in high reliability code (automotive, aircraft, space), it needs to do extensive code reviews and checking, following:

- Disallow recursion and cycles — enforced by policy and testing
- Keep code and stack far apart (code in flash, stack in RAM, and never the twain shall meet)
- Place guard bands around the stack — empty area of memory that is filled with a magic number (usually a software interrupt instruction, but there are many options here), and hundreds or thousands of times a second, do look at the guard bands to make sure they haven't been overwritten.
- Use memory protection (i.e., no execute on the stack, no read or write just outside the stack)
- Interrupts don't call secondary functions — they set flags, copy data, and let the application take care of processing it (otherwise it might get 8 deep in function call tree, have an interrupt, and then go out another few functions inside the interrupt, causing the blowout). It has several call trees — one for the main processes, and one for each interrupt.