FreeRTOS - Newlib Reentrant

Reentrancy is an attribute of a piece of code and basically means it can be re-entered by another execution flow, for example by an interrupt or by another task or thread. GNU ARM Embedded Toolchain distributions include a non-polluting reduced-size runtime library called newlib. FreeRTOS has not fully supported reentrant for newlib.

#arm #stm32 #memory #reentrant

Last update: 2021-08-06 13:25:05

Table of Content

1. Reentrancy

Reentrancy is an attribute of a piece of code and basically means it can be re-entered by another execution flow, for example by an interrupt or by another task or thread.

Generally speaking, a function produces output data based on some input data (though both are optional, in general). Shared data could be accessed by any function at any time. If data can be changed by any function (and none keep track of those changes), there is no guarantee to those that share a datum that that datum is the same as at any time before.

Data has a characteristic called scope, which describes where in a program the data may be used. Data scope is either global (outside the scope of any function and with an indefinite extent) or local (created each time a function is called and destroyed upon exit).

Local data is not shared by any routines, re-entering or not; therefore, it does not affect re-entrance. Global data is defined outside functions and can be accessed by more than one function, either in the form of global variables (data shared between all functions), or as static variables (data shared by all invocations of the same function).

Reentrancy is distinct from, but closely related to, thread-safety. A function can be thread-safe and still not reentrant.

1.1. Rules for reentrancy

1. Reentrant code may not hold any static or global non-constant data.
2. Reentrant code may not modify itself.
3. Reentrant code may not call non-reentrant computer programs or routines.

1.2. Examples

Two functions below are reentrant:

```
int f(int i)
{
    return i + 2;
}

int g(int i)
{
    return f(i) + 2;
}
```

However, if `f()` depends on non-constant global variable, both functions become non-reentrant, such as:

```
int v = 1;

int f(int i)
{
    v += i;
    return v;
}

int g(int i)
{
    return f(i) + 2;
}
```

Some functions are thread-safe, but not reentrant, such as below function. `function()` can be called by different threads without any problem. But, if the function is used in a reentrant interrupt handler and a second interrupt arises inside the function, the second routine will hang forever.

```
int function()
{
    mutex_lock();

    // function body

    mutex_unlock();
}
```

2. Newlib implementation

GNU ARM libs use `newlib` to provide standard implementation of C libraries. However, to reduce the code size and make it independent to hardware, there is a lightweight version `newlib-nano` used in MCUs.

The `newlib` library maps standard C functions to a specific implementation environment through a chain of functions, for example:

1. `write()` invokes `_write_r()` with the current reentrancy context (e.g. thread/task-unique errno);
2. `_write_r()` invokes `_write()` and copies errno appropriately;
3. `_write()` must be provided by something. By default,

The `newlib-nano` library does not provide an implementation of low-level system calls which are used by C standard libs, such as `_write()` or `_read()`.

To make the application compilable, a new library named `nosys` (enabled with `-specs=nosys.specs` to the gcc linker command line) should be added. This library just provide

an simple implementation of low-level system calls which mostly return a by-pass value.STM32CubeMX, with `nosys`, will generate `syscalls.c` and `sysmem.c` to provide low-level implementation for `newlib-nano` interface:

Function and data object definitions:

```
char ** environ;
int _chown (const char * path, uid_t owner, gid_t group);
int_execve (const char * filename, char * const argv[], char * const envp[]);
pid_t _fork (void);
pid_t _getpid (void);
int _gettimeofday (struct timeval * tv, struct timezone * tz);
int _kill (pid_t pid, int sig);
int _link (const char * oldpath, const char * newpath);
ssize_t _readlink (const char * path, char * buf, size_t bufsiz);
int _stat (const char * path, struct stat * buf);
int _symlink (const char * oldpath, const char * newpath);
clock_t _times (struct tms *buf);
int _unlink (const char * pathname);
pid_t _wait (int * status);
void _exit (int status);
```

File Descriptor Operations:

```
int _close (int fd);
int _fstat (int fd, struct stat * buf);
int _isatty (int fd);
off_t _lseek (int fd, off_t offset, int whence);
int _open (const char * pathname, int flags);
ssize_t _read (int fd, void * buf, size_t count);
ssize_t _write (int fd, const void * buf, size_t count);
```

Heap Management:

```
void * _sbrk (ptrdiff_t increment);
```

3. Newlib reentrant

The `newlib` library does support reentrancy, but for `newlib-nano`, the reentrant attribute depends on how its interfaces are implemented.

The most concerned functions of reentrant support are `malloc()` and `free()` which directly are related to dynamic memory management. If these functions are not reentrant, the

information of memory layout will be messed up if there are multiple calls to `malloc()` or `free()` at a time.

`newlib` maintains information it needs to support each separate context (thread/task/ISR) in a *reentrancy structure*. This includes things like a thread-specific *errno*, thread-specific `pointers to allocated buffers`, etc. The active reentrancy structure is pointed at by global pointer `_impure_ptr`, which initially points to a statically allocated structure instance.

`newlib` requires below things to complete its reentrancy:

1. Switching context. Multiple reentrancy structures (one per context) must be created, initialized, cleaned and pointing upon `_impure_ptr` to the correct context each time the context is switching
2. Concurrency protection. For example of using `malloc()`, it should be `lock()` and `unlock()` in that function to make it thread-safe first

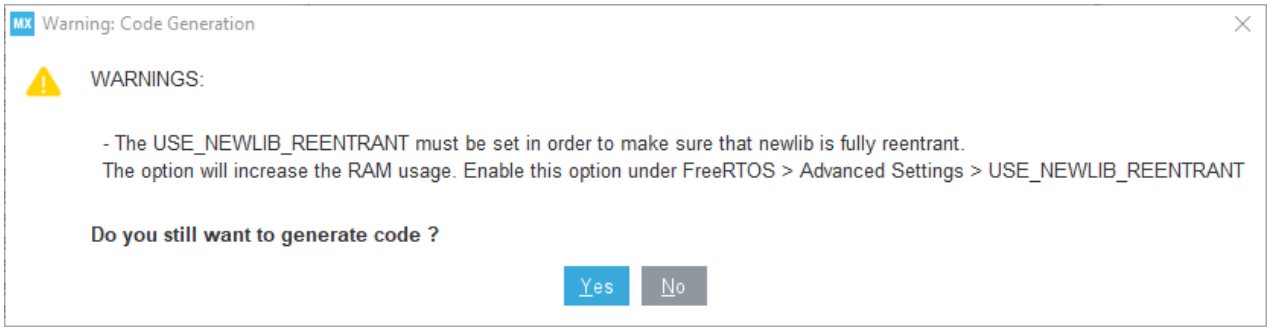
4. FreeRTOS supports Newlib reentrant

4.1. Switching context

FreeRTOS provides support for newlib's context management. In `FreeRTOSconfig.h`, add:

```
/* The following flag must be enabled only when using newlib */
#define configUSE_NEWLIB_REENTRANT           1
```

Be default, STM32 projects generated by STM32CubeIDE use `newlib-nano`. Whenever FreeRTOS is enabled, IDE will prompt to enable **Newlib Reentrant** attribute:



A prompt asking to enable newlib reentrant

With this option `configUSE_NEWLIB_REENTRANT = 1`, FreeRTOS does the following (in `task.c`):

- For each task, allocate and initialize a newlib reentrancy structure in the task control block
- Each task switch, set `_impure_ptr` to point to the newly active task's reentrancy structure

- On task destruction, clean up the reentrancy structure (help newlib free any associated memory)

4.2. Concurrency protection

There is one more thing to fully support newlib reentrant: FreeRTOS Memory Management.

FreeRTOS internally uses its own memory management scheme with different heap management implementations in `heapxx.c`, such as `heap_1`, or `heap_4`.

If an application only uses FreeRTOS-provided memory management APIs such as `pvPortMalloc()` and `vPortFree()`, this application is safe for newlib rerentrant, because FreeRTOS suspends the task-switching and interrupts during memory management.

However, many third party libraries do use the standard C `malloc()` and `free()` functions. For those cases, the Concurrency protection is not guaranteed. That is the reason that Dave Nadler implemented a new heap scheme for newlib in FreeRTOS. Details in https://nadler.com/embedded/newlibAndFreeRTOS.html.

> ⚠ **FreeRTOS in STM32CubeMX**
>
> The FreeRTOS version shipped in STM32CubeMX does not fully resolve Memory Management for newlib. Dave Nadler provides a version for STM32 at heap_useNewlib_ST.c. The usage will be covered in a below section.

5. Lab 0: Reentrant `printf` function

This example project demonstrates an issue when using `printf()` function without reentrant enabled for newlib in FreeRTOS. In that case, the data printed out is interrupted by different tasks.

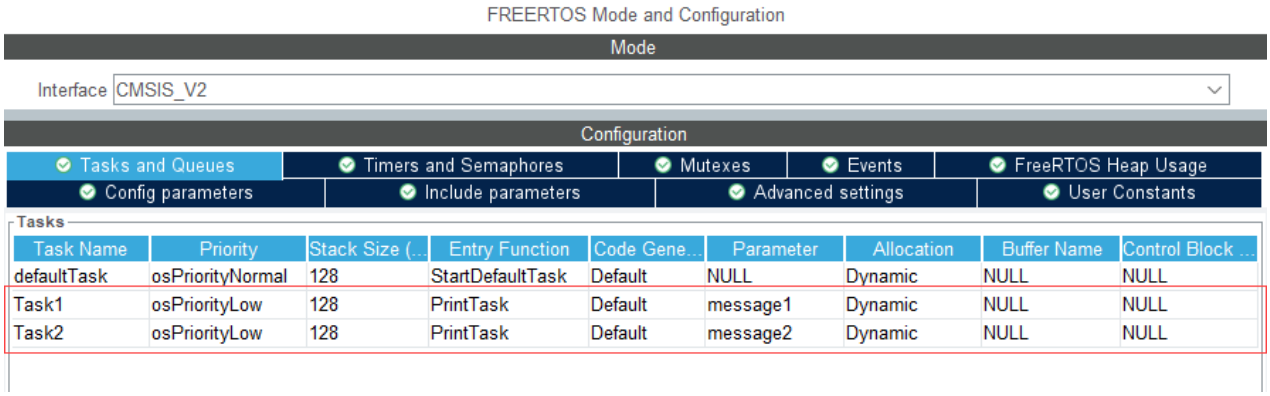
5.1. Precondition

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

1. Timebase 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. FreeRTOS is enabled with configs:
    a. CMSIS V2 is selected
    b. Memory Management Scheme is either `heap_3` or `heap_4`
    c. Newlib setting is `USE_NEWLIB_REENTRANT = Disabled`

5.2. Create 2 printing tasks

Add 2 tasks: `Task1` and `Task2` which call to a same funtion `PrintTask` but with different input messages `message1` and `message2`. Note that two tasks have the same priority.



Add 2 printing tasks

5.3. Define messages and print out

Two different messages will be prepared. To make the issue happens, the length of messages will be chosen to be long enough, such as 128 bytes (comparing to 128 * 4 bytes of the task stack size).

```
char* message1 =
"................................................................................

char* message2 =
"++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The function `PrintTask()` will print out a message along with the task name, the Reentrant config, a increasing counter to see new messages clearly.

```
#include "FreeRTOSConfig.h"

void PrintTask(void *argument)
{
  char* name = pcTaskGetName(NULL);
  char* message = (char*)argument;
  char counter = 0;
  /* Infinite loop */
  for(;;)
  {
    printf("%d %s: %03d %s\r\n",
        configUSE_NEWLIB_REENTRANT, name, counter++, message);
    osDelay(500);
  }
}
```

A final step is to redirect printed data to an UART port, such as USART1 using the low-level `_write()` function:

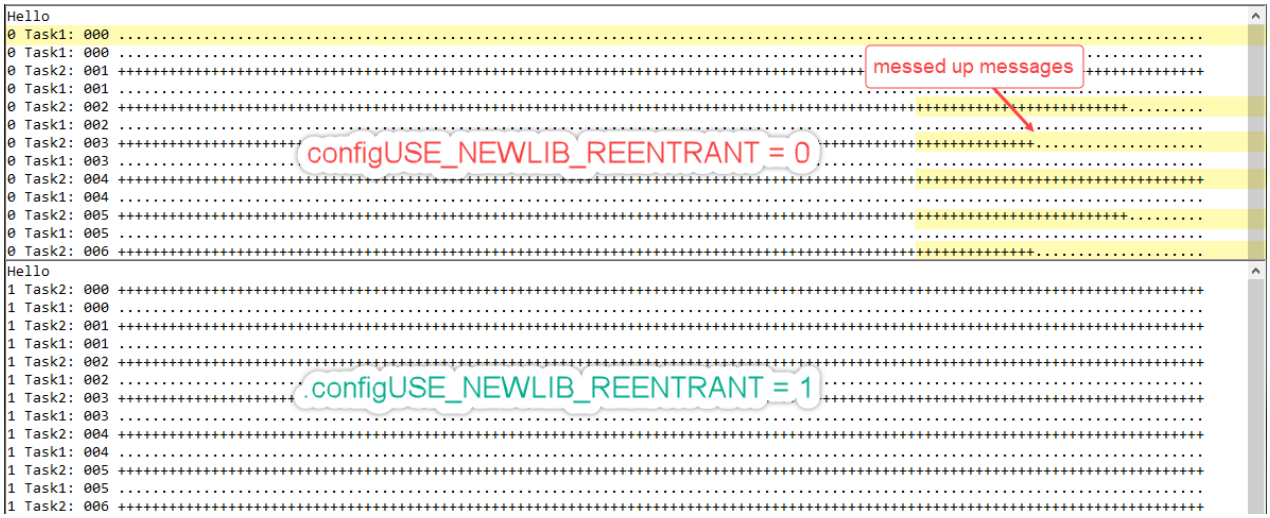
```
int _write(int file, char *ptr, int len) {
    // block write to UART if it is not ready
    while(HAL_UART_GetState(&huart1) != HAL_UART_STATE_READY);

    HAL_StatusTypeDef hstatus;
    hstatus = HAL_UART_Transmit(&huart1, (uint8_t*) ptr, len,
                                HAL_MAX_DELAY);
    if (hstatus == HAL_OK) {
        return len;
    } else {
        return 0;
    }
}
```

5.4. Compile and Run

Build the project and run a target board, the output will be messed up as it can been seen that characters in the `messages1` is printed in the line of the `messages2` .

5.5. Turn on Newlib reentrant

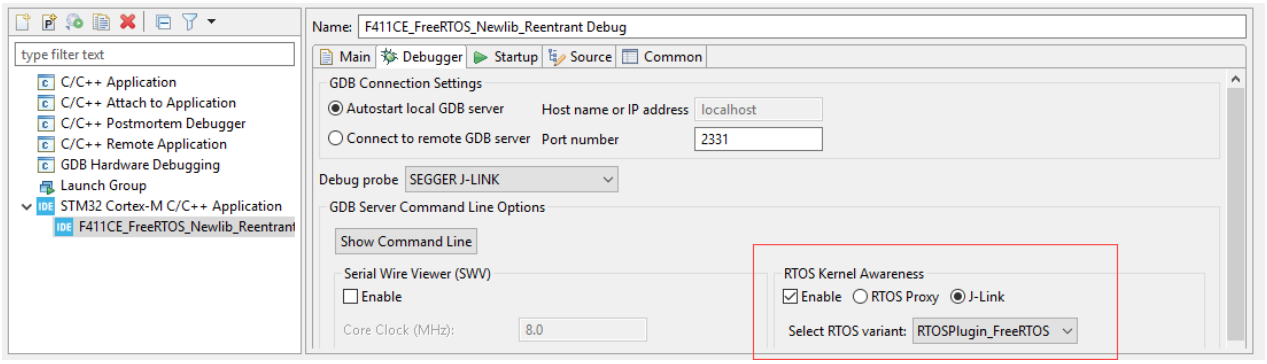
If set the `configUSE_NEWLIB_REENTRANT = 1` , the issue will not happen!



Different outputs on different reentrant settings

5.6. Debug and Analyze

Enable RTOS Kernel Awareness feature to seen task information and call stack when system is suspended.



Select RTOS kernel awareness feature

5.6.1. Non-reentrant

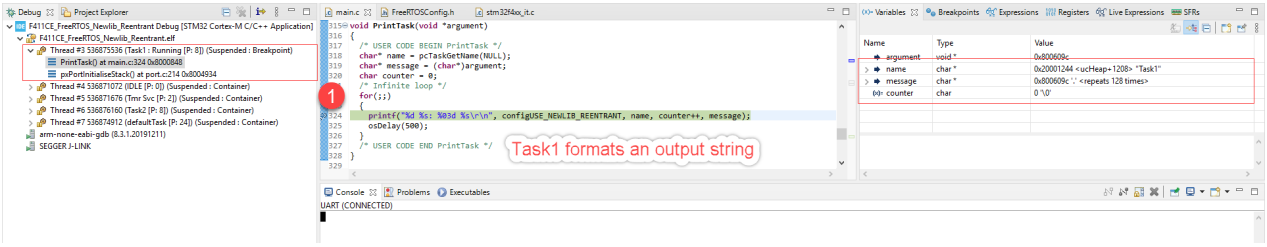
Step 1: Task 1 formats its output string.

Step 2: Task 1 starts printing the formatted string saved at the address pointed by the `ptr` pointer.

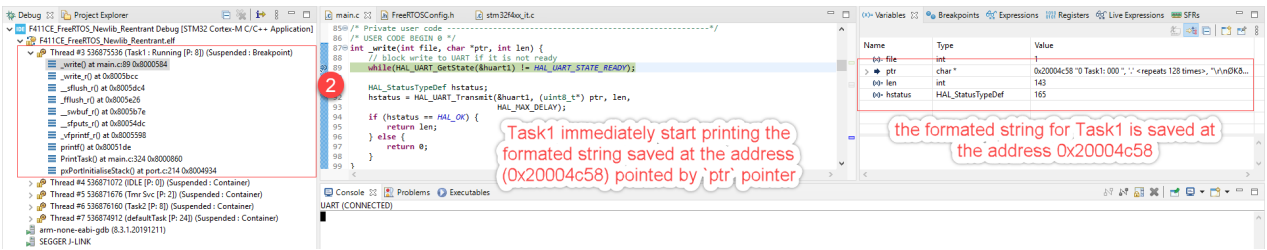
Step 3: Task 1 is suspended, Task 2 formats its output string. Due to non-reentrant, `printf` saves the formatted string at a fixed location - the same location of formatted string in Task 1, causing an overwrite.

Step 4: Task 2 cannot print as it has to wait for a ready flag.

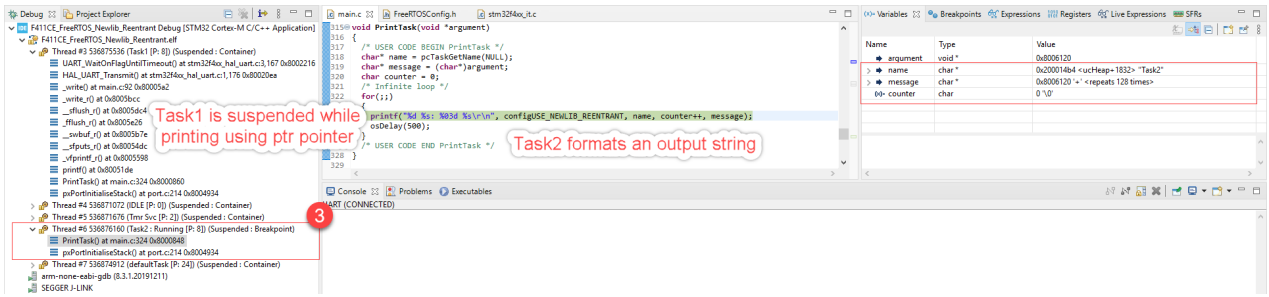
Step 5: Task 1 is reactivated after a SysTick interruption, and Task 1 continues printing but the content of the formatted string is overwritten when Task2 formats its string.



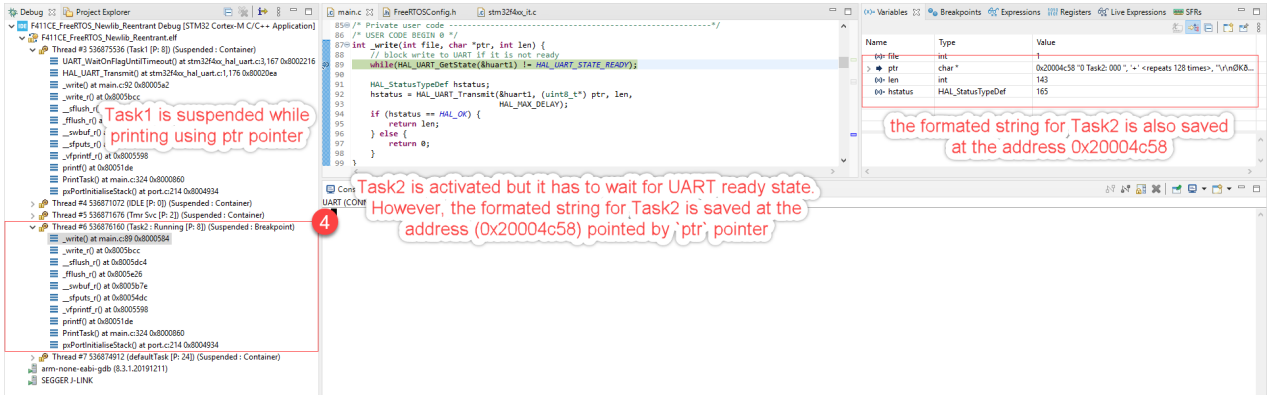
Non-reentrant debug step 1



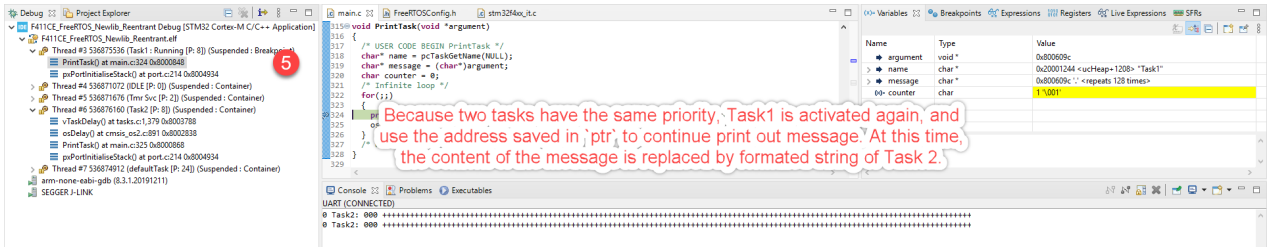
Non-reentrant debug step 2



Non-reentrant debug step 3



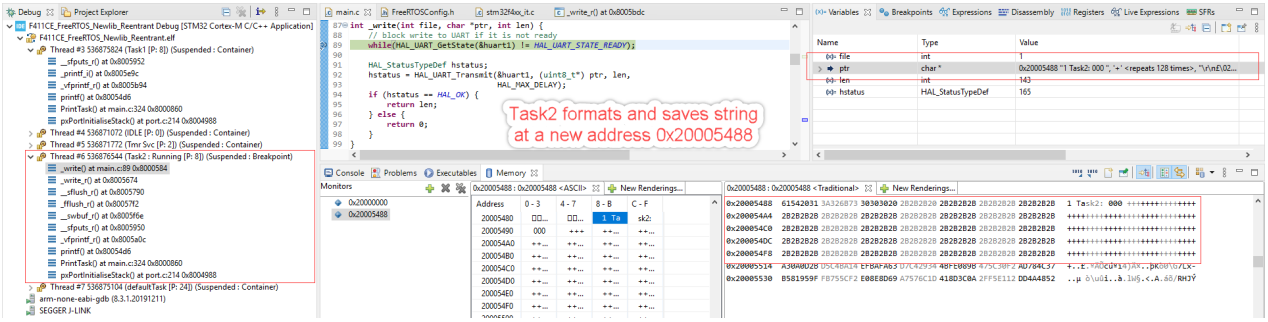
Non-reentrant debug step 4



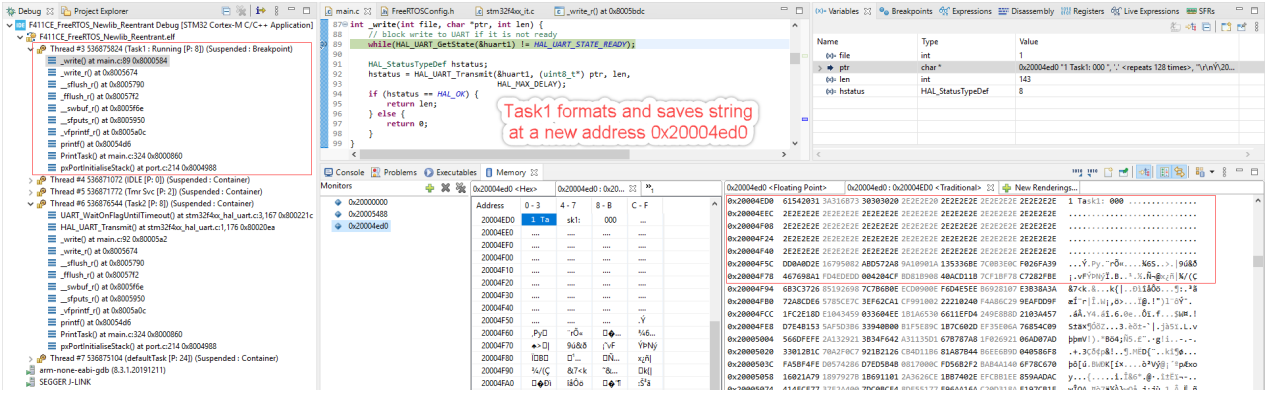
Non-reentrant debug step 5

5.6.2. Reentrant

When reentrant is enabled, each task has its own reentrant struct and uses its own stack for newlib functions. As a result, there is no shared memory location used by both tasks, that avoid overwriting data.



Task2 printf stack



Task1 printf stack

6. Integrate newlib memory scheme

As mentioned above, Dave Nadler provides a version for STM32 at heap_useNewlib_ST.c. It is not officially supported by ST.

A method to ensure thread-safe for `malloc()` and `free()` is to wrap newlib malloc-functions to use FreeRTOS's porting memory management functions. However, FreeRTOS heap implementations do not support `realloc()`.

The `heap_usNewlib_ST` scheme choose another method to solve malloc-functions' thread-safe. This memory scheme implements thread-safe for `malloc()` and `free()` in newlib, and then overwrites FreeRTOS's memory function to use newlib's functions.

Here are step to integrate `heap_usNewlib_ST` into STM32 project:

1. Exclude `sysmem.c` file from build. This file provides an implementation of `_sbrk()` which is used by `malloc()`
2. Exclude FreeRTOS heap manegement such as `heap_4.c` which implements `pvPortMalloc()` and `vPortFree()`
3. Include `heap_useNewlib_ST.c` to project.
4. Define 2 configs below to support ISR stack check

```
#define configISR_STACK_SIZE_WORDS (128) // DRN in WORDS, must be valid
constant for assembler
#define configSUPPORT_ISR_STACK_CHECK  1   // DRN initialize and check ISR
stack
```

5. Set reentrant support using `#define configUSE_NEWLIB_REENTRANT 1`

7. References

- FreeRTOS Memory Management: https://www.freertos.org/a00111.html
- Newlib interface: http://pabigot.github.io/bspacm/newlib.html
- Newlib FreeRTOS Memory Management: https://nadler.com/embedded/newlibAndFreeRTOS.html
- Thread-safe in C library: https://developer.arm.com/documentation/dui0492/i/the-c-and-c---libraries/thread-safe-c-library-functions