Analyze Memory Access Issues

AN 327, Spring 2020, V 1.0

keil-feedback@arm.com

Abstract

Debugging memory problems is not an easy task. Using sophisticated debug technologies that are part of Arm Keil MDK can help to get to a working result quicker. Such features as Logic Analyzer, SWO Trace, and the Call Stack + Locals window help to analyze complex problems that occur while running large software stacks.

This application note shows an example that uses WiFi to communicate with a server. During wireless communication, an error occurs that cannot be explained easily. We show how the error can be debugged using built-in functionality in Arm Keil MDK.

Contents

Prerequisites

The application runs on NXP's LPC54018 IoT Module (OM40007). We are using ULINKplus for debugging, but any member of the ULINK family is up to the task. The debug adapter is connected to the on-module 10-pin Arm Cortex-M debug port (J7).

Used software packs are:

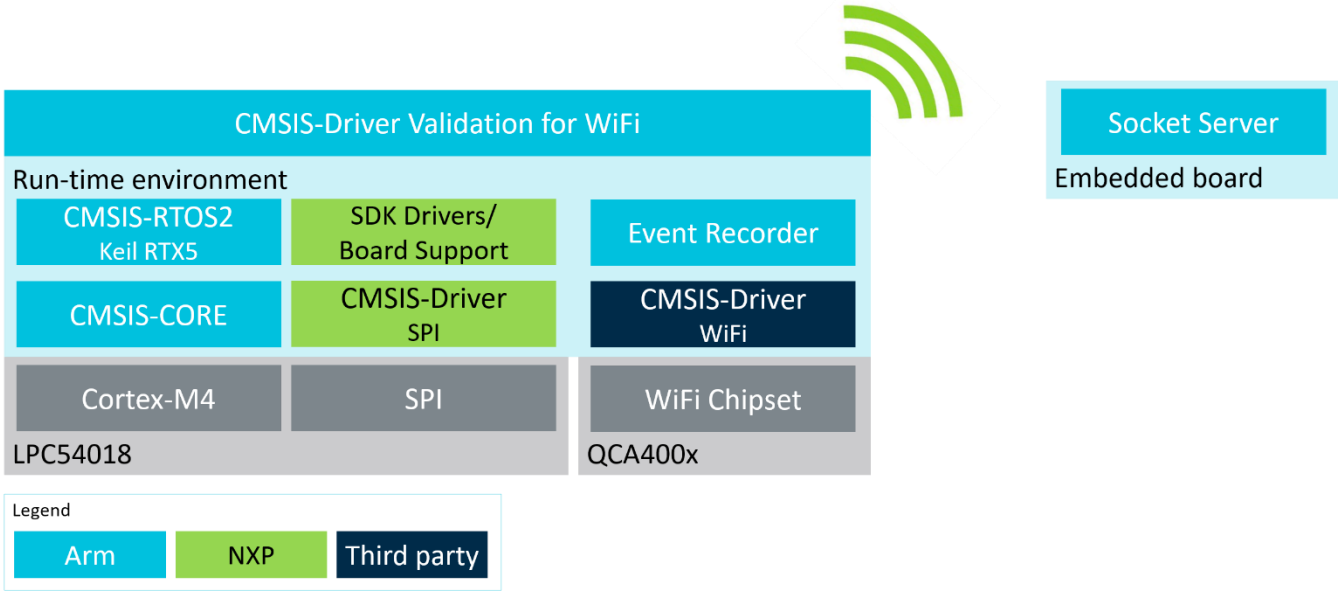
- ARM::CMSIS.5.6.0
- ARM::CMSIS-Driver_Validation.1.4.0
- Keil::ARM_Compiler.1.6.2
- MDK-Packs::QCA400x_Host_Driver_SDK.1.1.0
- MDK-Packs::QCA400x_WiFi_Driver.1.1.0
- NXP::LPC54018-IoT-Module_BSP.12.0.0
- NXP::LPC54018_DFP.12.0.0

It is assumed that you have basic knowledge of the C language and that you are familiar with Arm Keil MDK.

Note: the WiFi driver of the module already contains the correct source code, so if you want to see the failure, you need to change it back to the initial fault (see **Abstract.txt** in the µVision project).

Introduction

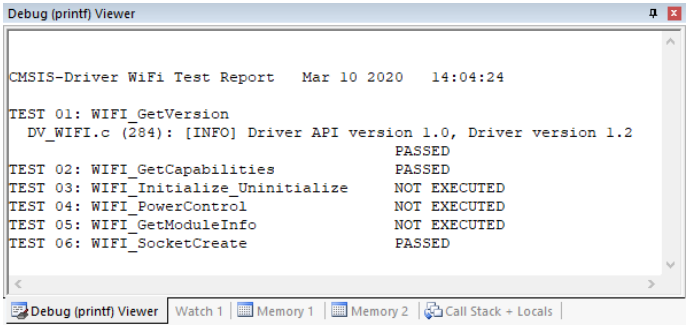
Our example application runs a CMSIS-Driver WiFi test and communicates with a socket server using an external WiFi module that is attached via SPI to the underlying microcontroller device. Shortly after the application connects to the wireless access point, the communication stops. Simple run-stop debugging does not help in this case so that more sophisticated MDK debug techniques are required to find the root cause of the issue.

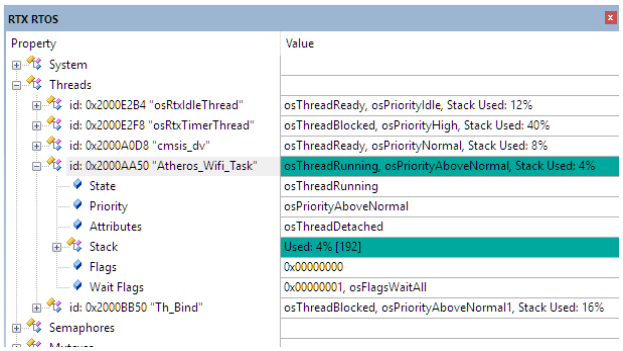


Analysis

The application used here is available for download in a ZIP file on www.keil.com/appnotes/docs/apnt_327.asp. Unzip it, open the project in µVision, check the content of the Abstract.txt file, build, and start running on the target hardware.

Running the application, debug output is shown in the **Debug (printf) Viewer** window. The WiFi test starts to run, but then freezes after the `WIFI_SocketCreate` test:





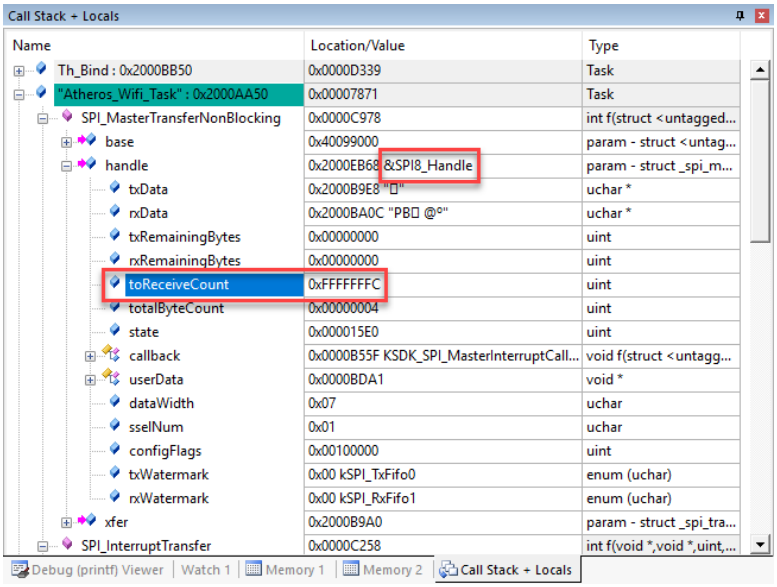
We stop the program and after some debugging find out that the execution is locked in the thread **Atheros_Wifi_Task** which implements the communication between the microcontroller and the WiFi module. We use the RTX RTOS window to determine how threads are switching. Further debugging allows us to narrow down the issue to the SPI communication.

Step 1: Check SPI Communication between the LPC54018 and the WiFi module

The module's SPI driver is based on Qualcomm's SDK and is implemented in the file `cust_spi_hcd.c`. The SPI transfer is done in the `Custom_Bus_InOutToken` function at line 201. While the application is still running, set a breakpoint here. Program execution stops shortly afterwards at this location. **Step Over (F10)** the SPI transfer function. It returns an error, which is unexpected.

Run (F5) the application again. When it hits the breakpoint the next time, **Step (F11)** until you reach line 650 in the file fsl_spi.c. Open the **Call Stack + Locals** window and check **"Atheros_Wifi_Task" → SPI_MasterTransferNonBlocking → handle**, which holds the internal state of the SPI driver.

Notice that the value of the `toReceiveCount` variable (which indicates the remaining data bytes to receive) is 0xFFFFFFFC, although the expected value is 0 (all data received):



Reviewing the code in fsl_spi.c and focusing on the lines where the `toReceiveCount` variable is used or modified, there is no explanation how the value of the `toReceiveCount` variable could have changed to 0xFFFFFFFC.

Step 2: Examine SPI variables (dynamic)

The variable `toReceiveCount` is held in a structure of the type `cmsis_spi_handle_t` (see line 259 in fsl_spi.c). The **Call Stack + Locals** window shows that the SPI8 peripheral is used (in fsl_spi_cmsis.c) and therefore the `SPI8_Handle` structure is declared to be of the type `cmsis_spi_handle_t`. The offset of the `toReceiveCount` variable is 16.

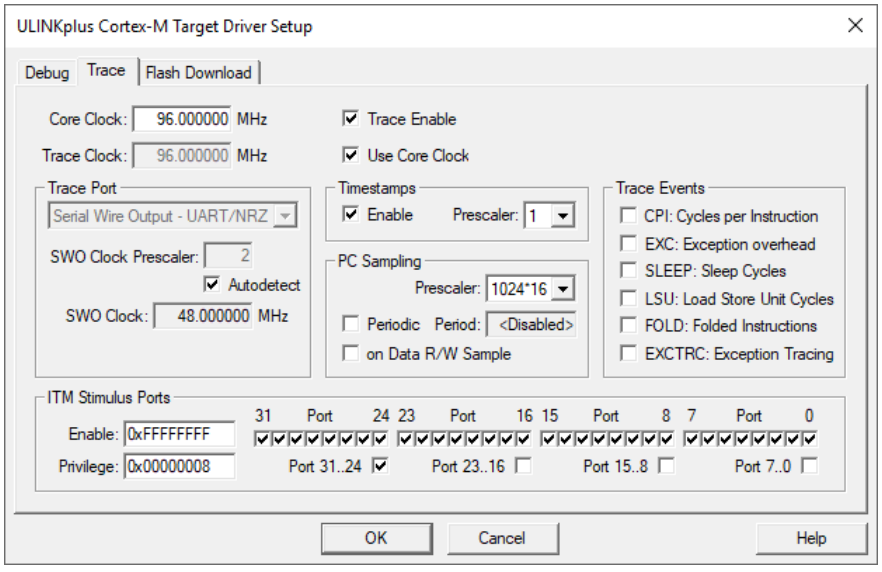
```
typedef union _cmsis_spi_handle
{
    spi_master_handle_t masterHandle;
    spi_slave_handle_t slaveHandle;
} cmsis_spi_handle_t;

 /*! @brief SPI transfer handle structure */
struct _spi_master_handle
{
    uint8_t *volatile txData;           /*!< Transfer buffer */
    uint8_t *volatile rxData;           /*!< Receive buffer */
    volatile uint32_t txRemainingBytes; /*!< TX Data [in bytes] */
    volatile uint32_t rxRemainingBytes; /*!< RX Data [in bytes] */
    volatile uint32_t toReceiveCount;   /*!< RX Data remaining in bytes */
    uint32_t totalByteCount;            /*!< A number of transfer bytes */
    volatile uint32_t state;            /*!< SPI internal state */
    spi_master_callback_t callback;     /*!< SPI callback */
    void *userData;                     /*!< Callback parameter */
    uint8_t dataWidth;                  /*!< Width of the data [1 to 16] */
    uint8_t sselNum;                    /*!< Slave select number */
    uint32_t configFlags; /*!< Additional option to control transfer */
    spi_txfifo_watermark_t txWatermark; /*!< txFIFO watermark */
    spi_rxfifo_watermark_t rxWatermark; /*!< rxFIFO watermark */
};
```

The memory location of the `SPI8_Handle` structure can be found in the memory map (just double-click the project's target "**Debug**" in the **Project** window to open the map file):

```
SPI8_Handle              0x2000eb68    Data          48  fsl_spi_cmsis.o(.bss)
```

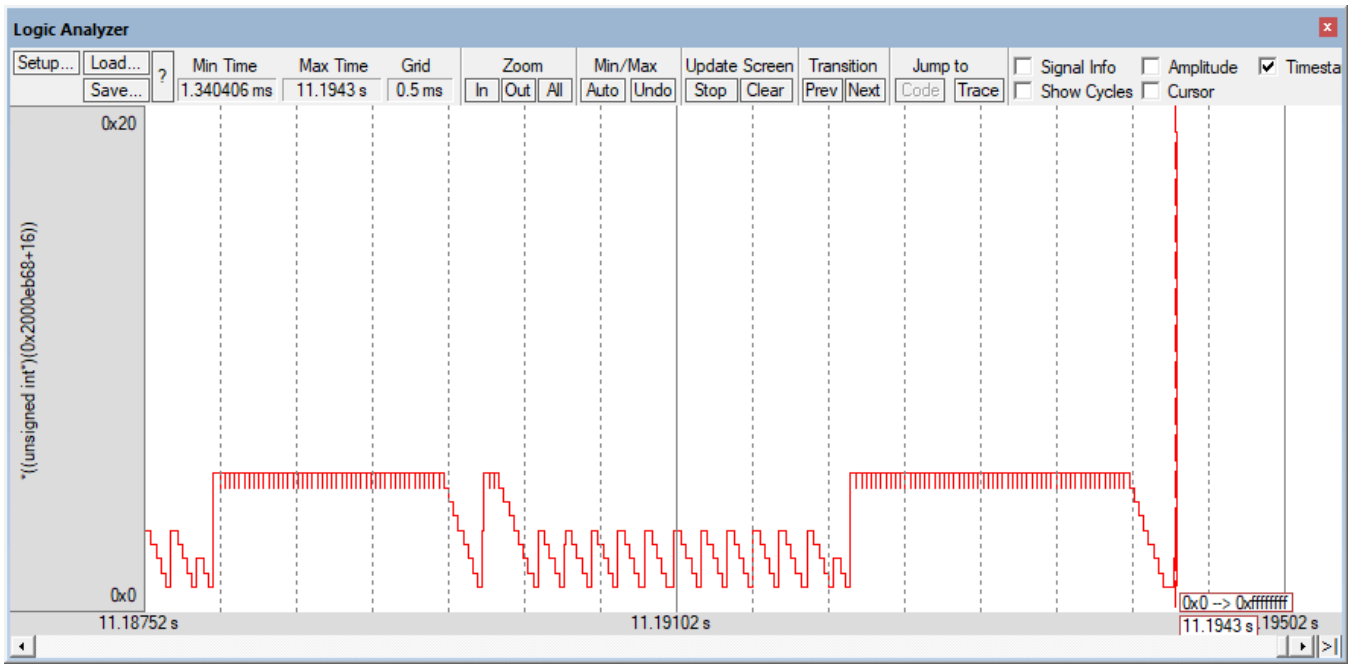
The symbol `toReceiveCount` cannot be accessed directly, therefore we use the `SPI8_Handler` address and the offset of `toReceiveCount` to determine the `toReceiveCount` variable location in memory: 0x2000eb68 + 16. To observe how the `toReceiveCount` variable changes over time we use the **Logic Analyzer** feature in µVision.

The Logic Analyzer uses trace data and for that the SWO trace needs to be configured as shown on the figure.



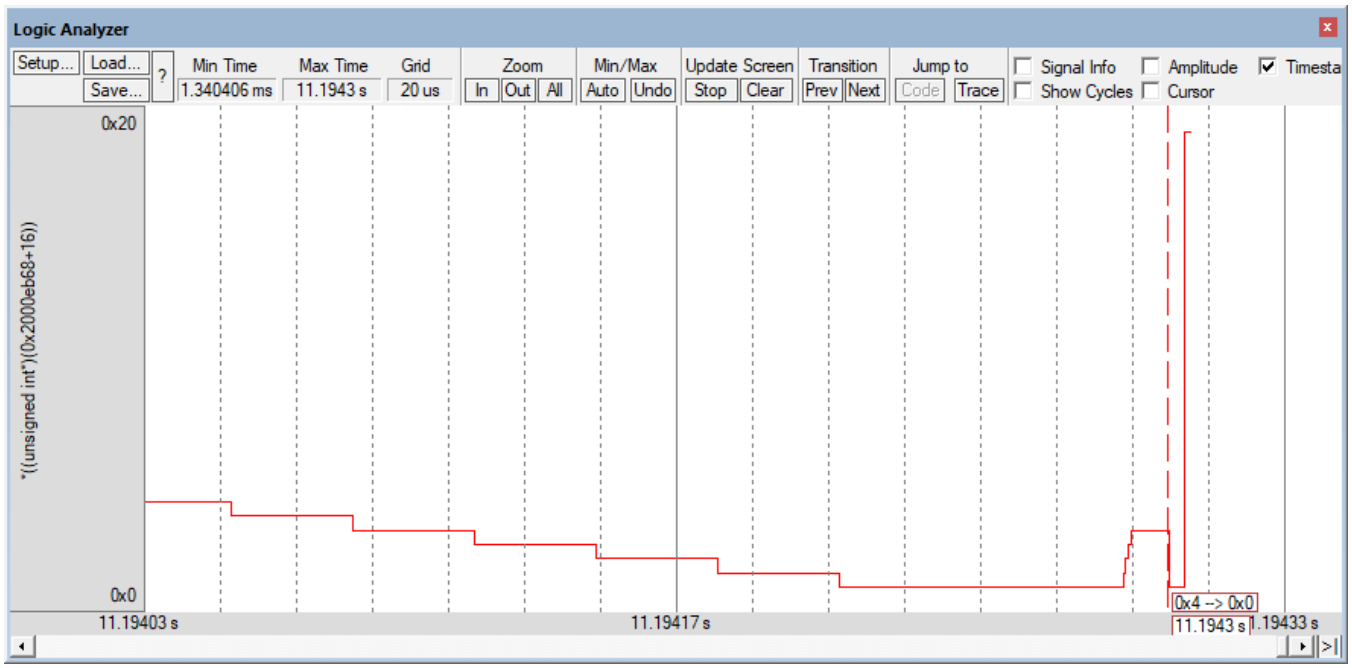
To add a memory location to the **Logic Analyzer** it needs to be typecast. In our case it is done as **((unsigned int*)(0x2000eb68+16))*.

Running the application, notice that as soon as WiFi module stops communicating with the server, the `toReceiveCount` variable stops changing. The following picture shows the unexpected change of the `toReceiveCount` variable from value 0 to value 0xFFFFFFFF:



In **Step 1**, we reviewed the source code and saw that the change from value 0 to value 0xFFFFFFFF is not possible. Subsequently, the value of the `toReceiveCount` variable is then decreased by 1 to value 0xFFFFFFFC.

The next picture shows the last change of `toReceiveCount` variable before it is changed from value 0 to value 0xFFFFFFFF:

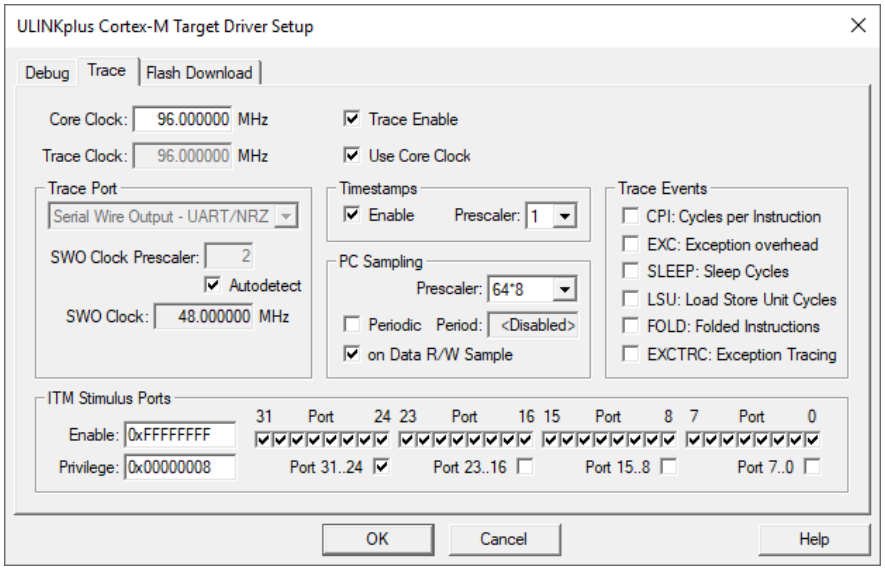


The value of the `toReceiveCount` variable drops from 4 to 0. It is expected that value decreases by 1 as in the previous samples. This indicates that the `toReceiveCount` variable might be unintentionally overwritten with the value 0 by another part of the application. Before the next step, **exit the debug session (CTRL+F5)**.

In many cases, using an access breakpoint can help to identify the location in code where memory gets unexpectedly overwritten. However, in our example this approach is not practically applicable. We need to find the code that changes `toReceiveCount` from 4 to 0. But both values are frequently assigned to the variable during normal operation as well. Hence, setting a read or write access breakpoints will frequently stop the execution. In communication test scenario such ours this can lead to timeouts on the server side and different program behavior.

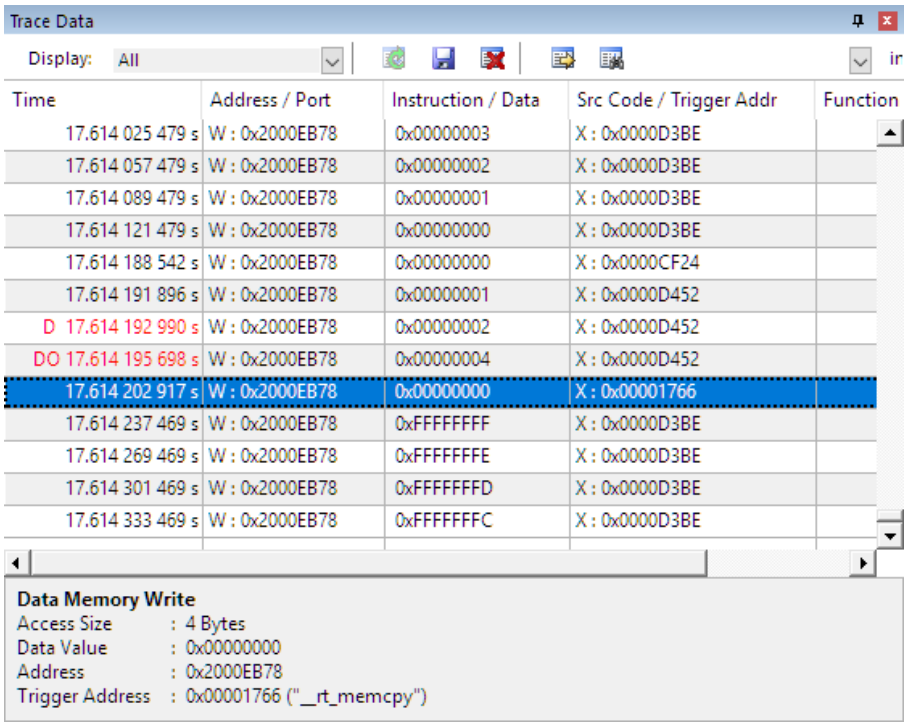
Step 3: Use SWO Trace

The LPC54018 device offers serial-wire output trace (SWO) that enables useful debugging features in µVision. The picture shows the setup dialog for the SWO trace.



The **PC Sampling on Data R/W Sample** option allows us to get PC samples for each read/write instruction. In our case, for each read/write access to the `toReceiveCount` variable already present in the Logic Analyzer. The **PC Sampling Prescaler** must be set to the lowest possible value, which does not yet cause a Trace Data Overflow (this depends on many factors, you might need to try this out before it works reliably).

Run the application again and stop when the WiFi module stops communicating. Open the **Trace Data** window (View – Trace – Trace Data) and observe the trace capture:



We can see the same behavior as previously observed in the Logic Analyzer: `toReceiveCount` value is changed from 4 to 0 and then to 0xFFFFFFFF. The Trigger Address value is decoded and shows that this is done in "__rt_memcpy" function.

Reviewing the source code, there was no sign of the `memcpy()` function being intentionally used to modify the `SPI8_Handle` structure. This is a strong indication that the function `memcpy()` overwrites the `toReceiveCount` variable.

Step 4: Patching the memcpy() function

The `memcpy()` function is called from many places in the application. We need to find out from where `memcpy()` is called when the overwrite happens. Unfortunately, the `memcpy()` function is built into the standard C library, therefore debugging code cannot be added to it.

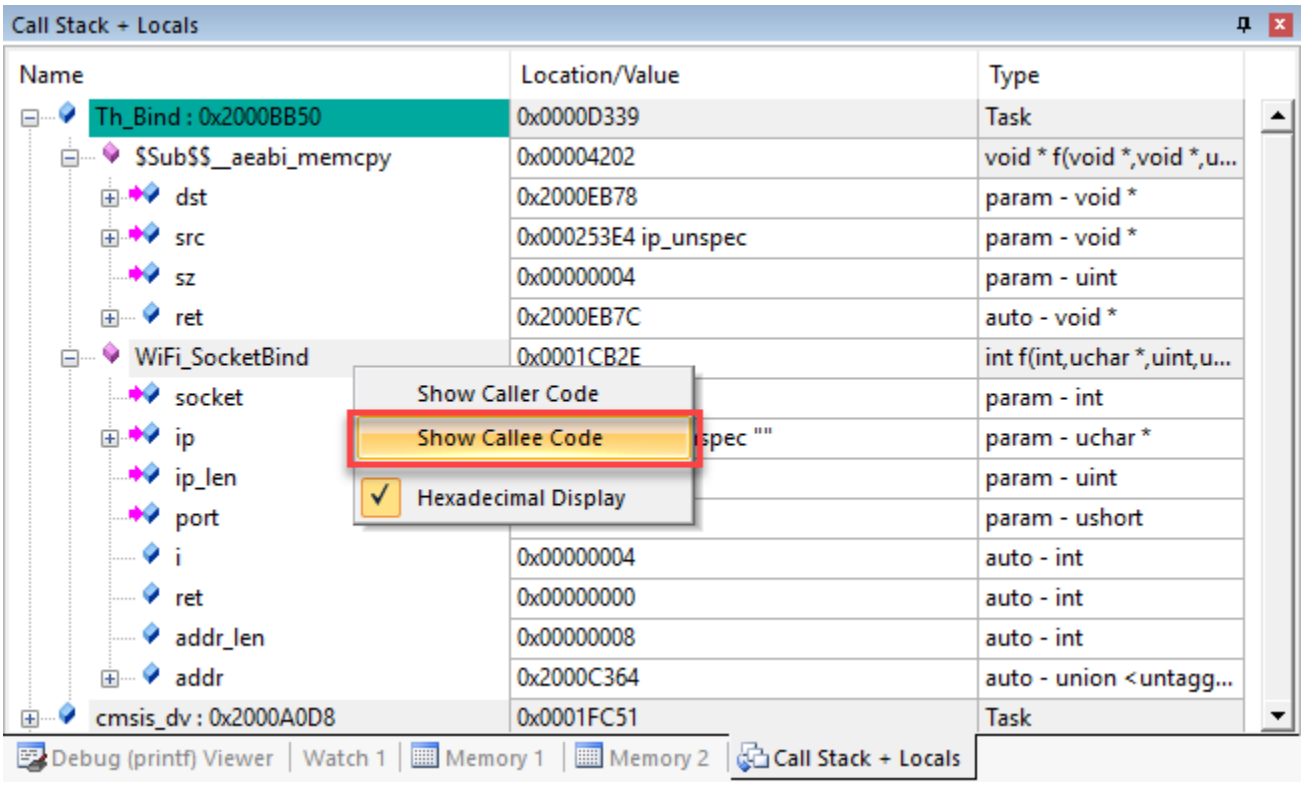
Luckily, there is a mechanism available that lets you patch existing symbol definitions. Use `$Super$$` and `$Sub$$` to do this as explained in the Linker User Guide.

The following source code shows how `$Super$$` and `$Sub$$` are used to insert additional code to `memcpy()` after the call to the legacy function `memcpy()`.

```
#include <string.h>
#include <stdint.h>
#include "cmsis_compiler.h"

#define toReceiveCount_adr (0x2000EB78)
#define toReceiveCount_ptr ((uint32_t *)toReceiveCount_adr)

extern void * $Super$$__aeabi_memcpy(void * dst, void * src, size_t sz);

/* this function is called instead of the original __aeabi_memcpy() */
void * $Sub$$__aeabi_memcpy(void * dst, void * src, size_t sz)
{
  void * ret;

  // call the original __aeabi_memcpy
  ret = $Super$$__aeabi_memcpy(dst, src, sz);
  if ((*toReceiveCount_ptr == 0U)                    &&
      ((toReceiveCount_adr >= (uint32_t)dst)         &&
      ((toReceiveCount_adr < ((uint32_t)dst + sz))))) {
    __NOP();
  }
  return ret;
}
```

In this code, we first call the original `memcpy` function and then we check if the `memcpy` destination memory overlaps with the `toReceiveCount` variable.

Run the application and place a breakpoint on `__NOP()` inside the `if statement`, to catch the faulty call to memcpy. When the application stops at the breakpoint, the **Call Stack** looks like this:



The **Call Stack + Locals** window shows that `memcpy()` is called from the function `WiFi_SocketBind`. Right-click on the function and select **Show Callee Code**. This takes you to the following code fragment shows the problematic call to `memcpy()`:

```
    socket_arr[socket].local_port = port;
    memcpy((void *)socket_arr[i].local_ip, (void *)ip, ip_len);
```

The element index of the array `socket_arr` is out of bounds. The local variable `i` is incorrectly used for the element index. From the **Call Stack + Locals** window is evident that `i` has a value of 4.

`socket_arr` is an array of four structures `socket_t`. Therefore, the destination parameter of `memcpy()` points to memory outside the array `socket_arr`.

The memory map shows that the structure `SPI8_Handle` is positioned directly after the array `socket_arr`. Thus, `memcpy()` actually overwrites the `toReceiveCount` variable:

```
    socket_arr          0x2000eaa8   Data          192  wifi_qca400x.o(.bss)
    .bss                0x2000eb68   Section        48  fsl_spi_cmsis.o(.bss)
    SPI8_Handle         0x2000eb68   Data           48  fsl_spi_cmsis.o(.bss)
```

Solution

The variable `socket` should be used instead of `i`. The following code shows the correct implementation:

```
    socket_arr[socket].local_port = port;
    memcpy((void *)socket_arr[socket].local_ip, (void *)ip, ip_len);
```

Conclusion

Memory overwrites can be tricky to catch and hard to spot. µVision's advanced debug features (SWO trace, different viewer windows, and Logic Analyzer) helped to find the root cause of this non-trivial problem.