Analyze Memory Access Issues

AN 327, Spring 2020, V 1.0

arm KEIL

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

1
1
2
2
3
4
6
7
8
8

Prerequisites

The application runs on <u>NXP's LPC54018 IoT Module (OM40007)</u>. We are using <u>ULINKplus</u> for debugging, but any member of the <u>ULINK</u> 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 μ Vision 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 <u>www.keil.com/appnotes/docs/apnt_327.asp</u>. Unzip it, open the project in μ Vision, 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 OxFFFFFFC, although the expected value is O (all data received):

Call Stack + Locals		д	X
Name	Location/Value	Туре	
✔ Th_Bind : 0x2000BB50	0x0000D339	Task	
· → · · · · · · · · · · · · · · · · · ·	0x00007871	Task	
SPI_MasterTransferNonBlocking	0x0000C978	int f(struct < untagged	
🗉 🁐 base	0x40099000	param - struct <untag< td=""><td></td></untag<>	
🖃 🂖 handle	0x2000EB68 &SPI8_Handle	param - struct _spi_m	
🐓 txData	0x2000B9E8 "0"	uchar *	
🔗 rxData	0x2000BA0C "PBD @°"	uchar *	
🖤 🐓 txRemainingBytes	0x0000000	uint	
xRemainingBytes	0x00000000	uint	
🔮 toReceiveCount	0xFFFFFFC	uint	
✓ totalByteCount	0x00000004	uint	
🛶 🔗 state	0x000015E0	uint	
🕣 🔧 callback	0x0000B55F KSDK_SPI_MasterInterruptCall	void f(struct < untagg	
🕣 🔧 userData	0x0000BDA1	void *	
🛹 🔗 dataWidth	0x07	uchar	
🔗 sselNum	0x01	uchar	
🔗 configFlags	0x00100000	uint	
🐓 txWatermark	0x00 kSPI_TxFifo0	enum (uchar)	
🔍 🔗 rxWatermark	0x00 kSPI_RxFifo1	enum (uchar)	
🕀 🏘 xfer	0x2000B9A0	param - struct _spi_tra	
SPI_InterruptTransfer	0x0000C258	int f(void *,void *,uint,	-
Debug (printf) Viewer Watch 1 III Memo	ory 1 🔲 Memory 2 🖓 Call Stack + Locals		

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 OxFFFFFFC.

Step 2: Examine SPI variables (dynamic)

The variable <code>toReceiveCount</code> is held in a structure of the type <code>cmsis_spi_handle_t</code> (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 <code>SPI8_Handle</code> structure is declared to be of the type <code>cmsis_spi_handle_t</code>. The offset of the <code>toReceiveCount</code> 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 */</pre>
    uint8 t *volatile rxData;
                                            /*!< Receive buffer */</pre>
    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;
volatile uint32_t state;
    uint32 t totalByteCount;
                                           /*!< A number of transfer bytes */</pre>
                                           /*!< SPI internal state */
    spi_master_callback_t callback; /*!< SPI internal sta
spi_master_callback_t callback; /*!< SPI callback */</pre>
    void *userData;
                                            /*!< Callback parameter */</pre>
    uint8 t dataWidth;
                                            /*!< Width of the data [1 to 16] */</pre>
    uint8 t sselNum;
                                            /*!< Slave select number */</pre>
    uint32 t configFlags; /*!< Additional option to control transfer */
    spi txfifo watermark t txWatermark; /*!< txFIFO watermark */
    spi_rxfifo_watermark_t rxWatermark; /*!< rxFIFO watermark */</pre>
};
```

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.

ULINKplus Cortex-M Target Driver Setup	\times
Debug Trace Rash Download Core Clock: 96.000000 MHz Image: Trace Enable Trace Clock: 96.000000 MHz Image: Use Core Clock Trace Port Image: Trace Enable Image: Trace Events SWO Clock Prescaler: 2 Image: Trace Events Image: Wire Output - UART/NRZ Image: Trace Events Image: CPI: Cycles per Instruction SWO Clock Prescaler: 2 Image: Trace Port Prescaler: Image: Trace Port Image: CPI: Cycles per Instruction Image: Prescaler: 1024*16 Image: Prescal	
ITM Stimulus Ports 31 Port 24 23 Port 16 15 Port 8 7 Port 0 Enable: [0xFFFFFFFF] [v]v]v]v]v]v]v]v]v]v]v]v]v]v]v]v]v]v]v]	

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 serialwire output trace (SWO) that enables useful debugging features in μ Vision. 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

ULINKplus Cortex-M Target Driver Setup	Х
Debug Trace Rash Download	
Core Clock: 96.000000 MHz Image: Trace Enable Trace Clock: 96.000000 MHz Image: Use Core Clock Trace Port Image: Use Core Clock Image: Use Core Clock Swo Clock Prescaler: 2 Image: Clock Prescaler: Image: Clock Prescaler: Swo Clock: 48.000000 MHz Prescaler: Image: Clock Prescale: Swo Clock: 48.000000 MHz Prescale: Image: Clock Prescale: Image: Prescale: Image: Clock Prescale: Image: Clock Prescale: Image: Clock Prescale: Image: Clock Prescale: Image: Prescale: Image: Clock Prescale: Ima	3
ITM Stimulus Ports 31 Port 24 23 Port 16 15 Port 8 7 Port 0 Enable: 0xFFFFFFFF Implementation Implement	

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:

Trace Data				д 🗙
Display: All	\sim	18 🛃 💽 🖻		√ ir
Time	Address / Port	Instruction / Data	Src Code / Trigger Addr	Function
17.614 025 479 s	W: 0x2000EB78	0x0000003	X:0x0000D3BE	
17.614 057 479 s	W: 0x2000EB78	0x0000002	X:0x0000D3BE	
17.614 089 479 s	W: 0x2000EB78	0x00000001	X:0x0000D3BE	
17.614 121 479 s	W: 0x2000EB78	0x0000000	X:0x0000D3BE	
17.614 188 542 s	W: 0x2000EB78	0x0000000	X:0x0000CF24	
17.614 191 896 s	W: 0x2000EB78	0x0000001	X:0x0000D452	
D 17.614 192 990 s	W: 0x2000EB78	0x0000002	X:0x0000D452	
DO 17.614 195 698 s	W: 0x2000EB78	0x00000004	X:0x0000D452	
17.614 202 917 s	W: 0x2000EB78	0x0000000	X:0x00001766	
17.614 237 469 s	W: 0x2000EB78	0xFFFFFFFF	X:0x0000D3BE	
17.614 269 469 s	W: 0x2000EB78	0xFFFFFFE	X:0x0000D3BE	
17.614 301 469 s	W: 0x2000EB78	0xFFFFFFD	X:0x0000D3BE	
17.614 333 469 s	W: 0x2000EB78	0xFFFFFFC	X:0x0000D3BE	-
•	1	1		
Data Memory Write Access Size : 4 Data Value : 0 Address : 0 Trigger Address : 0	Bytes ‹00000000 ‹2000EB78 ‹00001766 ("rt_mem	сру")		

AN327 – Analyze Memory Access Issues

We can see the same behavior as previously observed in the Logic Analyzer: toReceiveCount value is changed from 4 to 0 and then to 0xFFFFFFF. 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 <code>memcpy()</code> function being intentionally used to modify the <code>SPI8_Handle</code> structure. This is a strong indication that the function <code>memcpy()</code> overwrites the <code>toReceiveCount</code> 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 supers 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)
                                                     88
      ((toReceiveCount adr >= (uint32 t)dst)
                                                     82
      ((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:

Call Stack + Locals 4 X			
Name	Location/Value	Туре	
	0x0000D339	Task	
🖃 🔮 \$Sub\$\$_aeabi_memcpy	0x00004202	void * f(void *,void *,u	
🕀 👐 dst	0x2000EB78	param - void *	
🕀 🗰 🗰 src	0x000253E4 ip_unspec	param - void *	
+ sz	0x00000004	param - uint	
🕀 🔶 ret	0x2000EB7C	auto - void *	
□···· ♦ WiFi_SocketBind	0x0001CB2E	int f(int,uchar *,uint,u	
socket Show Ca	ller Code	param - int	
🕀 🗰 ip Show Ca	illee Code ispec ""	param - uchar *	
ip_len	imal Display	param - uint	
port rexadec		param - ushort	
🗭 i	0x00000004	auto - int	
🖉 🔶 ret	0x0000000	auto - int	
addr_len	0x0000008	auto - int	
🗈 🔍 🔗 addr	0x2000C364	auto - union <untagg< td=""><td></td></untagg<>	
	0x0001FC51	Task	•
📴 Debug (printf) Viewer Watch 1 🛄 Memo	ory 1 🛄 Memory 2 🚰 Call Stack + Locals		

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, memopy() 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. μ Vision's advanced debug features (SWO trace, different viewer windows, and Logic Analyzer) helped to find the root cause of this non-trivial problem.