

メモリアクセスの問題解析方法

株式会社 DTS インサイト プロダクトソリューション事業部 営業部 営業技術課

本ドキュメントは Arm の提供する" Analyze Memory Access Issues AN 327, Spring 2020, V 1.0" の内容に基づき 作成されたものです。内容につきましては全て上記ドキュメントをマスターといたしておりますので、ご使用の 際には必ず上記ドキュメントを参照の上、本ドキュメントは参考資料として用いる形をお取りくださいますよう お願い申し上げます。



目次

1.	概要.		3
-	L.1.	前提条件	3
2.	イン	トロダクション	4
3.	解析.		4
	3.1.	ステップ 1: LPC54018 と WiFi モジュール間の SPI 通信を確認する	5
	3.2.	ステップ 2: SPI 変数を調べる (動的)	6
	3.3.	ステップ 3: SWO トレースを使用する	8
	3.4.	ステップ 4: memcpy() 関数にパッチをあてる	10
4.	解決領	策	11
5.	まとめ	ø	11



1. 概要

メモリの問題をデバッグするのは簡単な作業ではありません。 Arm Keil MDK の一部である高度なデバッグテクノロジを使用することでより迅速に動作するようにできます。 ロジックアナライザ、SWO トレース、Call Stack + Locals ウィンドウなどの機能は大規模なソフトウェアスタックの実行中に発生する複雑な問題の分析に役立ちます。

本アプリケーション ノートでは、WiFi を使用してサーバと通信するケースを例示しています。 無線通信の最中では、簡単には説明できないエラーが発生することがあります。

Arm Keil MDK の持つ機能を使用してエラーをデバッグする方法を示します。

1.1. 前提条件

この例でのアプリケーションは、<u>NXP LPC54018 IoT モジュール (OM40007)</u>上で実行されます。 デバッグで は <u>ULINKplus</u> を使用していますが、ULINK ファミリに含まれるどの製品でもこの操作を実行できます。デバッ グアダプタはモジュール上の 10 ピン Arm Cortex-M デバッグ ポート(J7) に接続されます。

使用している software pack は以下の通りです:

- 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

C 言語の基本的な知識があり、Arm Keil MDK の使用に慣れていることを前提としています。

注: モジュールの WiFi ドライバにはあらかじめ正しいソースコードが含まれているため、障害を確認したい場合は初期の障害を含んた状態に戻す必要があります。(その場合、uVision プロジェクト内の Abstract.txt を参照してください)



2. イントロダクション

このサンプルアプリケーションは、CMSIS-Driver WiFi テストを実行し、SPI 経由で基盤となるマイクロコント ローラデバイスに接続された外部 WiFi モジュールを使用してソケット サーバと通信します。アプリケーション が無線アクセスポイントに接続した直後に通信が停止します。このような場合、単純に、実行と停止を用いた手 法のデバッグは役に立たないため、問題の根本的な原因を見つけるには、より高度な MDK デバッグテクニック を要します。



3. 解析

ここで使用するアプリケーションは、<u>https://developer.arm.com/documentation/kan327/latest/</u>から ZIP ファイルでダウンロードできます。

そちらを解凍し、μVision でプロジェクトを開き、Abstract.txt ファイルの内容を確認したあと、ビルドを行い、 ターゲットハードウェア上で実行を開始します。

アプリケーションを実行すると、デバッグ出力が **Debug (printf) Viewer** ウィンドウに表示されます。WiFi テストの実行が開始しますが、WIFI_SocketCreate テストの後にフリーズします:

Debug (printf) Viewer		Ф 🔀	RTX RTOS	
CMSIS-Driver WiFi Test Report Mar 10 2020 14:04:24 TEST 01: WIFI_GetVersion DV_WIFI.c (284): [INFO] Driver API version 1.0, Driver version 1. PASSED TEST 02: WIFI_GetCapabilities PASSED TEST 03: WIFI_Initialize_Uninitialize NOT EXECUTED TEST 04: WIFI_PowerControl NOT EXECUTED TEST 05: WIFI_GetModuleInfo NOT EXECUTED TEST 05: WIFI_GetCapate PASSED		₽ <mark> 2</mark>	RTX RTOS Property	Value osThreadReady, osPriority/Idle, Stack Used: 12% osThreadBiocked, osPriority/High, Stack Used: 40% osThreadBearing, osPriorityNormal, Stack Used: 6% osThreadBearing, osPriorityNooveNormal, Stack Used: 45% osThreadBearing, osPriorityNooveNormal, Stack Used: 45%
		~	Plags	0x00000000
Debug (printf) Viewer Watch 1 Memory 1 Mem	nory 2 🚰 Call Stack + Locals		₩ Wait Hags	0x00000001, osthegsWertAll osThreadBlocked, osPriorityAboveNormal1, Stack Useck 16%

Our insight, your Value

プログラムを停止しデバッグを行うと、マイクロコントローラと WiFi モジュール間の通信を実装するスレッド Atheros_Wifi_Task で実行がロックされていることがわかります。 RTX RTOS ウィンドウを使用して、スレ ッドがどのように切り替わっているかを確認します。さらにデバッグを行うと、SPI 通信に問題があることが絞 り込めます。

3.1. ステップ 1: LPC54018 と WiFi モジュール間の SPI 通信を確認する

モジュールの SPI ドライバは Qualcomm の SDK に基づいており、ファイル cust_spi_hcd.c に実装されて います。SPI 転送は、201 行目の Custom_Bus_InOutToken 関数で行われています。アプリケーションの実行 中に、ここにブレークポイントを設定します。

プログラムの実行は、その後すぐにこの場所で停止します。SPI 転送関数を Step Over (F10)します。すると、 予期しないエラーが返されます。

Run (F5) を選択してアプリケーションを再度実行します。次にブレークポイントに到達したら、ファイル fs1_spi.c の 650 行目まで Step (F11) 実行をおこないます。

Call Stack + Locals ウィンドウを開き、"Atheros_Wifi_Task" \rightarrow SPI_MasterTransferNonBlocking \rightarrow handle をチェックします。これは SPI ドライバの内部状態を保持しています。

期待値は 0 (全データ受信)ですが、toReceiveCount 変数の値(受信する残りのデータ バイトを示します)が 0xFFFFFFC であることに注意してください。

e	Location/Value	Туре
Th_Bind : 0x2000BB50	0x0000D339	Task
"Atheros_Wifi_Task" : 0x2000AA50	0x00007871	Task
SPI_MasterTransferNonBlocking	0x0000C978	int f(struct < untagged
😟 🚧 base	0x40099000	param - struct < untag
📄 👐 handle	0x2000EB68 & SPI8_Handle	param - struct _spi_m
🛶 🔗 txData	0x2000B9E8 "0"	uchar *
🗭 rxData	0x2000BA0C "PBD @o"	uchar *
- 🖉 txRemainingBytes	0x00000000	uint
rxRemainingBytes	0x00000000	uint
- 🖉 toReceiveCount	0xFFFFFFFC	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,

fsl_spi.cのコードを確認し、toReceiveCount 変数が使用または変更されている行に注目すると、 toReceiveCount 変数の値がどのようにして 0xFFFFFFC に変更されたのかがわかりません。



3.2. ステップ 2: SPI 変数を調べる (動的)

変数 toReceiveCount は、cmsis_spi_handle_t 型の構造体に保持されています。(fsl_spi.c の 259 行目を参照) Call Stack + Locals ウィンドウでは、SPI8 ペリフェラルが (fsl_spi_cmsis.c で) 使用されており、そのため SPI8_Handle 構造体が cmsis_spi_handle_t 型として宣言されていることが示されています。

toReceiveCount 変数のオフセットは 16 です。

```
typedef union cmsis spi handle
ſ
    spi master handle t masterHandle;
    spi slave handle t slaveHandle;
} cmsis spi handle t;
/*! @brief SPI transfer handle structure */
struct spi master handle
ł
   uint8 t *volatile txData;
                                         /*!< Transfer buffer */
   uint8 t *volatile rxData;
                                         /*!< Receive buffer */
   volatile uint32_t txRemainingBytes; /*!< TX Data [in bytes] */</pre>
   volatile uint32_t rxRemainingBytes; /*!< RX Data [in bytes] */</pre>
    volatile uint32_t toReceiveCount; /*!< RX Data remaining in bytes */</pre>
                                         /*!< A number of transfer bytes */</pre>
    uint32 t totalByteCount;
    volatile uint32 t state;
                                         /*!< SPI internal state */
    spi master callback t callback;
                                         /*!< SPI callback */
    void *userData;
                                         /*!< Callback parameter */
                                         /*!< Width of the data [1 to 16] */
    uint8 t dataWidth;
                                         /*!< Slave select number */
    uint8 t sselNum;
    uint32 t configFlags; /*!< Additional option to control transfer */
    spi txfifo watermark t txWatermark; /*!< txFIFO watermark */</pre>
    spi rxfifo watermark t rxWatermark; /*!< rxFIFO watermark */</pre>
};
```

SPI8_Handle 構造体のメモリ位置はメモリマップで確認できます。(マップファイルを開くには、Project ウィンドウでプロジェクトのターゲット"Debug"をダブルクリックするだけです):

SPI8 Handle 0x2000eb68 Data 48 fsl spi cmsis.o(.bss)

シンボル toReceiveCount には直接アクセスで	ULINKplus Cortex-M Target Driver Setup
きないため、SPI8_Handler アドレスと	Debug Trace Rash Download
toReceiveCount のオフセットを使用して、	Core Clock: 96.000000 MHz Trace Clock: 96.000000 MHz Use Core Clock
メモリ内の toReceiveCount 変数の場所	Trace Port
(0x2000eb68 + 16) を確定します。	SWO Clock Prescaler: 2 PC Sampling CLC Exception overhead SIEEP: Siece Curles
toReceiveCount 変数が時間の経過とともに	Image: Work of the second s
どのように変化するかを観察するには、uVision	On Data R/W Sample EXCTRC: Exception Tracin
の Logic Analyzer 機能を使用します。	ITM Stimulus Ports 31 Port 24 23 Port 16 15 Port 8 7 Port 0 Enable: McFFFFFFF Improvinging interpretation interpretation interpretation interpretation interpretation Improvinging interpretation
_ogic Analyzer はトレースデータを使用するた	Privilege: 0x00000008 Port 3124 🔽 Port 2316 🗂 Port 158 🗂 Port 70 🗂
め、図に示すように SWO トレースをコンフィギ	OK Cancel Help
ュレーションする必要があります。	

Our insight, your Value

Logic Analyzer にメモリの場所を追加するには、型をキャストする必要があります。

この例では、*((unsigned int*)(0x2000eb68+16)) とします。

アプリケーションを実行すると、WiFi モジュールがサーバとの通信を停止するとすぐに、toReceiveCount 変数の変化がとまることがわかります。次の図は、toReceiveCount 変数の値が 0 から 0xFFFFFFF へ予期せず変わる現象を示しています。



ステップ1で確認したところ、値0から値0xFFFFFFFへの変更はソースコード上不可能であることがわかりました。その後、toReceiveCount変数の値が1減らされ、値0xFFFFFFFCになります。

次の図は、toReceiveCount 変数が値 0 から値 0xFFFFFFFF に変更される前の最後の変更を示しています。



toReceiveCount 変数の値は 4 から 0 に減少します。前のサンプル同様、値が 1 ずつ減少することが予測されます。これは、toReceiveCount 変数がアプリケーションの別の部分によって意図せずに値 0 で上書きされた可能性があることを示しています。

次の手順に進む前に、exit the debug session (CTRL+F5) を選択してデバッグセッションを終了します。 多くの場合、アクセスブレークポイントを使用すると、コード内でメモリが予期せず上書きされる場所を特定す るのに役立ちます。ただし、この例では、実際のところこのアプローチは適用できません。 toReceiveCount を4から0に変更するコードを見つける必要があります。

ただし、通常の操作中にも両方の値が変数に割り当てられることが多くあります。 したがって、read および write アクセスについてブレークポイントを設定すると、実行が頻繁に停止します。 今回のような通信テストの シナリオでは、これによりサーバ側でタイムアウトが発生し、プログラムの動作が変わってしまう可能性があり ます。

3.3. ステップ 3: SWO トレースを使用する

LPC54018 デバイスは、uVision 上で便利な デバッグ機能となる serial-wire output (SWO)トレースを実装しています。 この図は、 SWO トレースの設定ダイアログを示していま す。

PC Sampling on Data R/W Sample

オプションを使用すると、read/write の命令 ごとに PC サンプルを取得できます。

今回のケースでは、toReceiveCount 変数 への read/write ごとの情報はすでに Logic Analyzer で参照できるようになっています。

PC Sampling の Prescaler の値は、Trace

- 1							
Debug Trace Rash Downl Core Clock: 96.000000 Trace Clock: 96.000000 Trace Port Senal Wire Output - UART SWO Clock Prescaler: SWO Clock: 48.000	MHz MHz /NRZ / Autodetect 200 MHz	✓ Trace En ✓ Use Core ✓ Timestamps - ✓ Enable PC Sampling Pr Pr Periodic ✓ on Data H	able Clock Prescaler: 1 escaler: 64*8 Period: 058 R/W Sample		ace Events CPI: Cycle EXC: Exce SLEEP: S LSU: Load FOLD: Fol EXCTRC:	es per Instru eption overf leep Cycles d Store Unit ded Instruc Exception	ction head Cycles tions Tracing
ITM Stimulus Ports Enable: 0xFFFFFFF Privilege: 0x0000008	31 F I기기기 Port	Port 24 23 기기기기기 기기 31.24 17	Port 16	15 Port	87 777 777 8	Port	0 ସ 7 7 0

Data Overflow がまだ発生していない状態の可能な限り低い値に設定する必要があります(これは多くの要因に 依存します。確実に動作させるためにあらかじめテストを行ってみる必要があるかもしれません) アプリケーションを再度実行し、WiFi モジュールが通信をやめてしまったら実行を止めます。**Trace Data** ウ ィンドウを(View → Trace → Trace Data メニューから) 開き、キャプチャされたトレースデータを確認しま す:



Display: All	~	1		 ✓ i
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	0x0000001	X : 0x0000D3BE	
17.614 121 479 s	W: 0x2000EB78	0x00000000	X : 0x0000D3BE	
17.614 188 542 s	W: 0x2000EB78	0x00000000	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	0x00000000	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	-
(
Data Memory Write Access Size : 4 Data Value : 0x Address : 0x Trigger Address : 0x	Bytes .00000000 .2000EB78 .00001766 ("rt_me	mcpy")		

Logic Analyzer で以前に観測されたものと同じ次の動作が確認できます:

toReceiveCount 値が 4 から 0 に変化し、その後 0xFFFFFFFF に変化します。

Trigger Addressの値がデコードされ、これが"___rt_memcpy" 関数で行われたことがわかります。

ソースコードを確認すると、memcpy() 関数が SPI8_Handle 構造体を変更するために意図的に使われた形跡は ありませんでした。これは、memcpy() 関数が toReceiveCount 変数を上書きしていることを強く示していま す。 Our insight, your Value

3.4. ステップ 4: memcpy() 関数にパッチをあてる

memcpy() 関数は、アプリケーション内のさまざまな場所から呼び出されます。上書きが発生したときにどこから memcpy() が呼び出されるかを調べる必要があります。残念ながら、memcpy() 関数は標準 C ライブラリに 組み込まれているため、デバッグコードを追加することはできません。

幸い、既存のシンボル定義にパッチを適用できるメカニズムを利用できます。

Linker User Guide で説明されているように、これには\$Super\$\$ と\$Sub\$\$ を使用します。

以下のソースコードは、旧来の関数 memcpy()の呼び出し後に\$Super\$\$ と\$Sub\$\$ を使用して memcpy() に 追加のコードを挿入する方法を示しています。

```
#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)
1
  void * ret;
  // call the original __aeabi_memcpy
ret = $Super$$__aeabi_memcpy(dst, src, sz);
  if ((*toReceiveCount_ptr == 0U)
                                                         22
      ((toReceiveCount_adr >= (uint32_t)dst)
                                                         22
      ((toReceiveCount_adr < ((uint32_t)dst + sz))))) {
      NOP();
  }
  return ret;
}
```

このコードでは、最初に元の memcpy 関数を呼び出し、次に memcpy がコピーを行う先となるメモリが toReceiveCount 変数と重複しているかどうかをチェックします。

アプリケーションを実行し、if ステートメント内の ___NOP() にブレークポイントを設定して、memcpy への 誤った呼び出しを捕捉します。アプリケーションがブレークポイントで停止すると、Call Stack ウィンドウ内 の表示は次のようになります:



			4	•
Name	Location/Valu	ue	Туре	
□	0x0000D339		Task	-
📄 🕺 \$Sub\$\$_aeabi_memo	сру 0х00004202		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 Caller Code		param - int	
🕀 🚧 ip	Show Callee Code	spec ""	param - uchar *	
👐 ip_len			param - uint	
🛶 port	Hexadecimal Display		param - ushort	
💊 i	0x00000004		auto - int	
💮 🔗 ret	0x00000000		auto - int	
🔍 🔗 addr_len	0x00000008		auto - int	
🗄 🤗 addr	0x2000C364		auto - union <untagg< td=""><td></td></untagg<>	
cmsis_dv : 0x2000A0D8	0x0001FC51		Task	

Call Stack + Locals ウィンドウでは、memcpy()が関数 WiFi_SocketBind から呼び出されていることが表示されます。関数を右クリックし、Show Callee Code を選択します。 これにより、次のようなコード部分が表示され、問題のある memcpy()の呼び出しが示されます:

socket_arr[socket].local_port = port; memcpy((void *)socket arr[i].local ip, (void *)ip, ip len);

配列 socket_arr の要素のインデックスが範囲外になっています。ローカル変数 i が要素のインデックスに誤って使用されています。Call Stack + Locals ウィンドウ上で、i の値が 4 であることがわかります。

socket_arr は、4 つの構造体 socket_t の配列です。したがって、memcpy() のコピー先となる引数は、配列 socket arr 外のメモリを指します。

メモリマップは、構造体 SPI8_Handle が配列 socket_arr の直後に配置されていることを示しています。したがって、事実上 memcpy () によって toReceiveCount 変数が上書きされてしまいます。

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)

4. 解決策

iの代わりに変数 socket を使用する必要があります。次のコードは、正しい実装を示しています。:

5. まとめ

メモリの上書きの問題は複雑で、発見するのが難しい場合があります。uVision の高度なデバッグ機能 (SWO トレース、さまざまなビューアウィンドウおよび Logic Analyzer) は、この重要な問題の根本となる原因を見つけるのに役立ちます。