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

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、NXP LPC54018 IoT モジュール (OM40007) 上で実行されます。 デバッグでは ULINKplus 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：

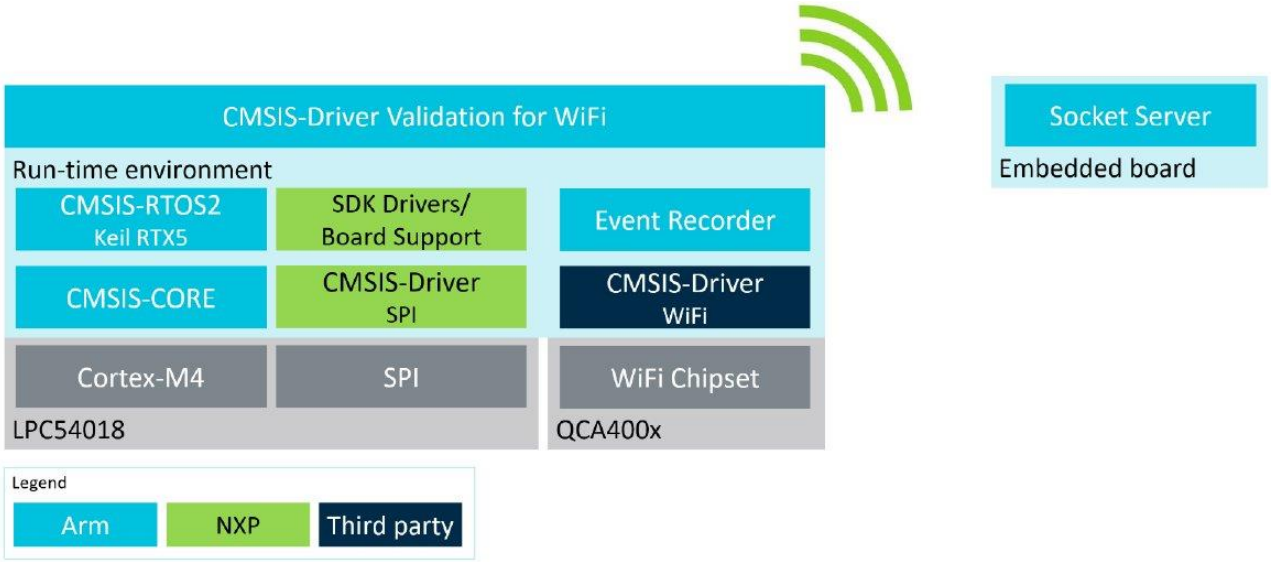
- ARM::CMSIS.5.6.0
- ARM::CMSIS-Driver_Validation.1.4.0
- Keil::ARM_Compiler.1.6.2
- MDK-Packs::QCA400x_Host_Driver_SDK.1.1.0
- MDK-Packs::QCA400x_WiFi_Driver.1.1.0
- NXP::LPC54018-IoT-Module_BSP.12.0.0
- NXP::LPC54018_DFP.12.0.0

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だ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。

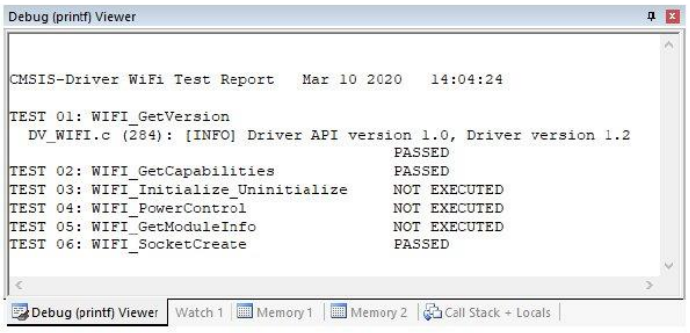


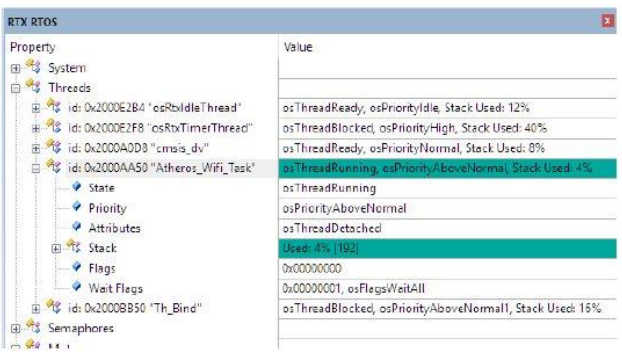
3. 解析

ここで使用するアプリケーションは、https://developer.arm.com/documentation/kan327/latest/ 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：





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

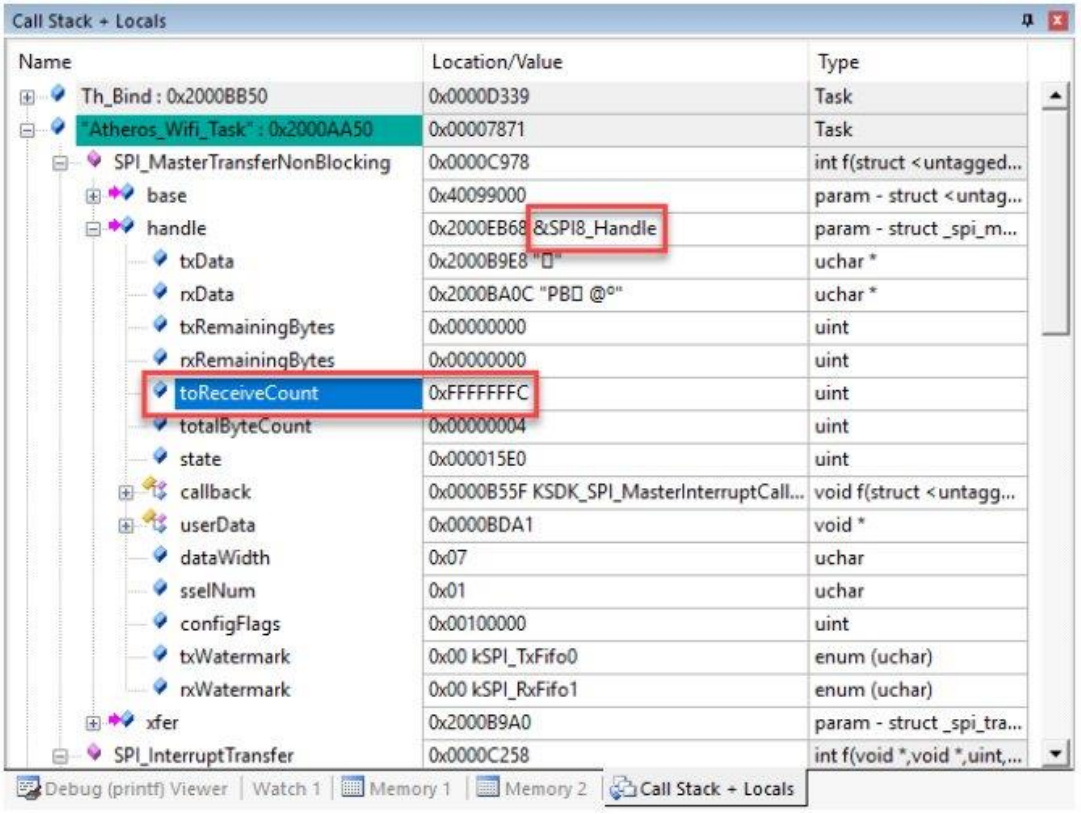
モジュールの SPI ドライバは Qualcomm の SDK に基づいており、ファイル **`cust_spi_hcd.c`** に実装されています。SPI 転送は、201 行目の **`Custom_Bus_InOutToken`** 関数で行われています。アプリケーションの実行中に、ここにブレークポイントを設定します。

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 **`fsl_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" → SPI_MasterTransferNonBlocking → 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 0xFFFFFFFC であることに注意してください。



`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 0xFFFFFFFC 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)

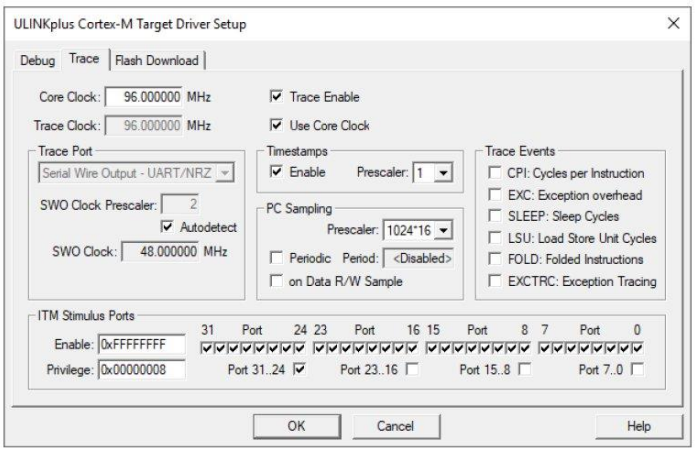
変数 toReceiveCount は、cmsis_spi_handle_t 型の構造体に保持されています。(fsl_spi.c の 259 行目を参照) **Call Stack + Locals** ウィンドウでは、SPI8 ペリフェラルが (fsl_spi_cmsis.c で) 使用されており、そのため **SPI8_Handle** 構造体が **cmsis_spi_handle_t** 型として宣言されていることが示されています。**toReceiveCount** 変数のオフセットは 16 です。

```
typedef union _cmsis_spi_handle
{
    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 */
};
```

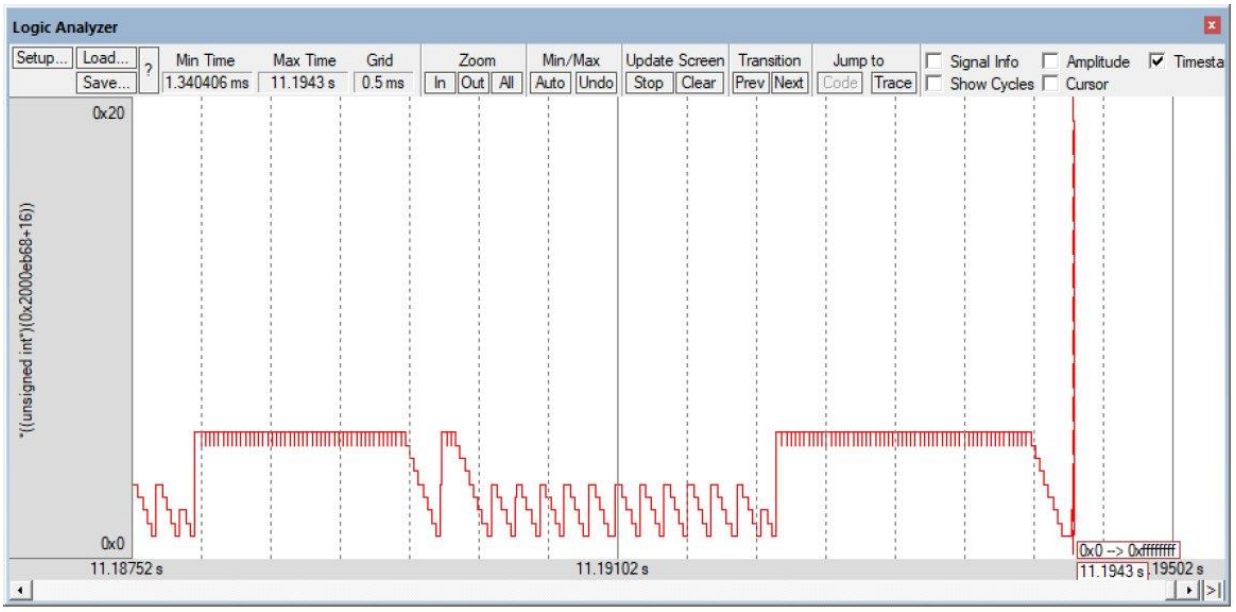
SPI8_Handle 構造体のメモリ位置はメモリマップで確認できます。(マップファイルを開くには、Project ウィンドウでプロジェクトのターゲット"Debug"をダブルクリックするだけです) :

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

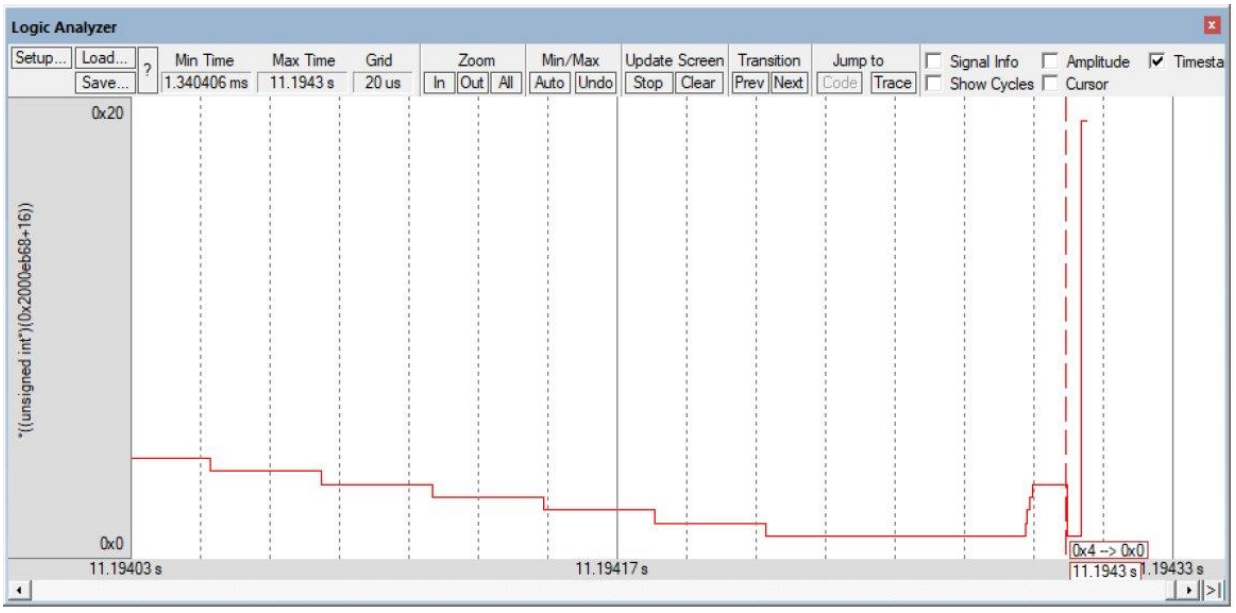
シンボル **toReceiveCount** には直接アクセスできないため、**SPI8_Handler** アドレスと **toReceiveCount** のオフセットを使用して、メモリ内の **toReceiveCount** 変数の場所 (0x2000eb68 + 16) を確定します。**toReceiveCount** 変数が時間の経過とともにどのように変化するかを観察するには、uVision の **Logic Analyzer** 機能を使用します。**Logic Analyzer** はトレースデータを使用するため、図に示すように SWO トレースをコンフィギュレーションする必要があります。



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 0xFFFFFFFF へ予期せず変わる現象を示しています。



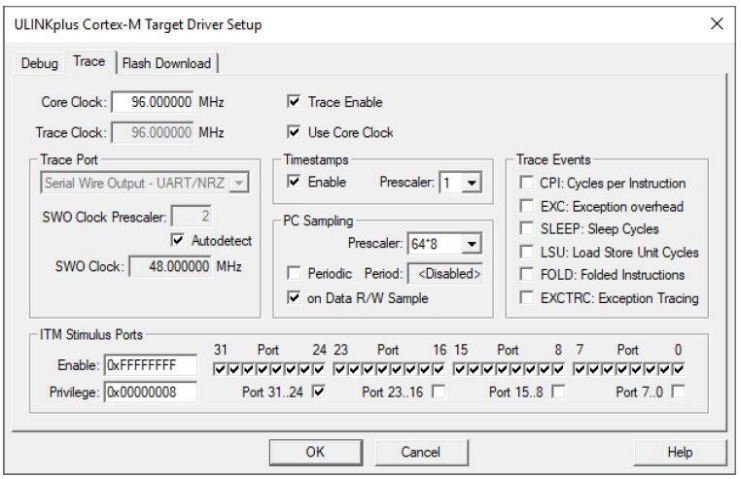
ステップ 1 で確認したところ、値 0 から値 0xFFFFFFFF への変更はソースコード上不可能であることがわかりました。その後、**`toReceiveCount`** 変数の値が 1 減らされ、値 0xFFFFFFFC になります。
次の図は、**`toReceiveCount`** 変数が値 0 から値 0xFFFFFFFF に変更される前の最後の変更を示しています。



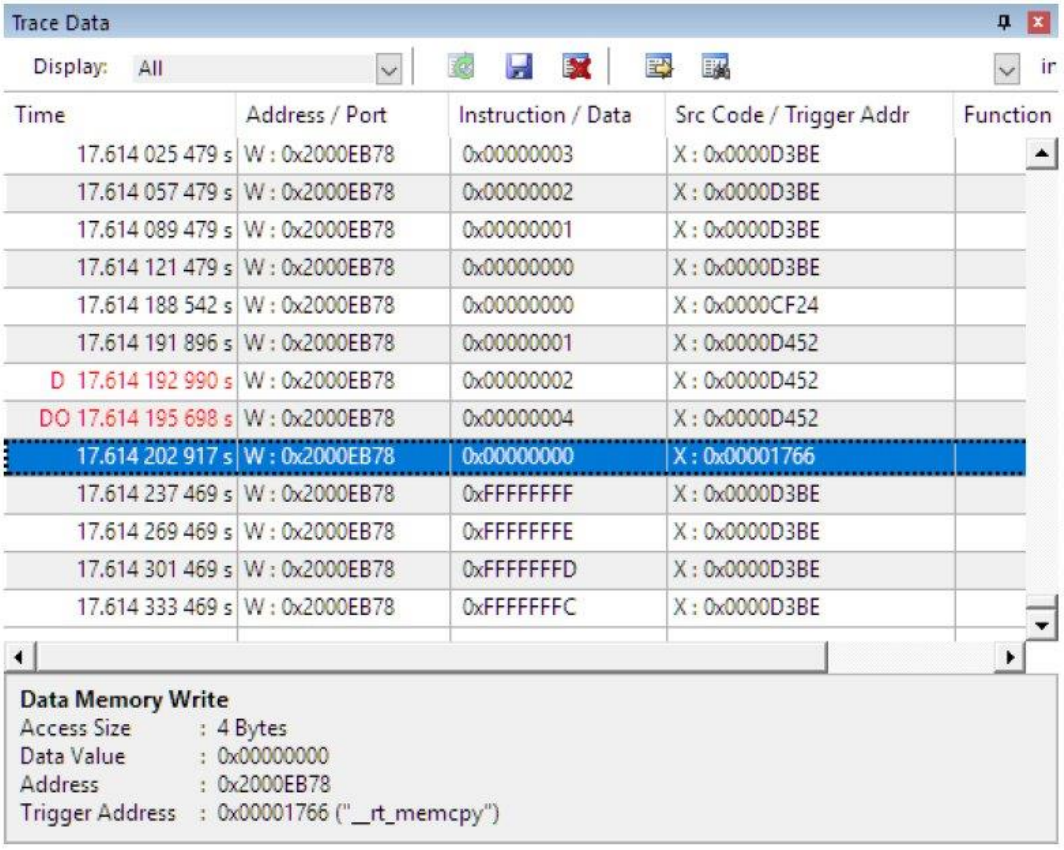
toReceiveCount 変数の値は 4 から 0 に減少します。前のサンプル同様、値が 1 ずつ減少することが予測されます。これは、**toReceiveCount** 変数がアプリケーションの別の部分によって意図せずに値 0 で上書きされた可能性があることを示しています。
次の手順に進む前に、**exit the debug session (CTRL+F5)** を選択してデバッグセッションを終了します。
多くの場合、アクセスブレークポイントを使用すると、コード内でメモリが予期せず上書きされる場所を特定するのに役立ちます。ただし、この例では、実際のところこのアプローチは適用できません。 **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 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：



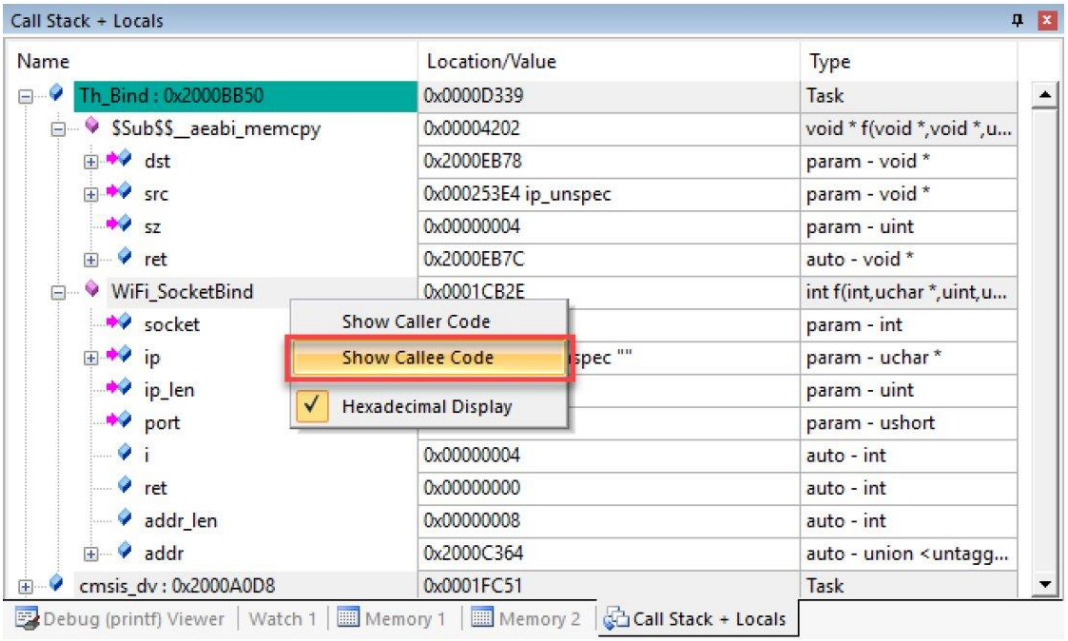
Logic Analyzer で以前に観測されたものと同じ次の動作が確認できます：
`toReceiveCount` 値が 4 から 0 に変化し、その後 0xFFFFFFFF に変化します。
Trigger Address の値がデコードされ、これが"__rt_memcpy" 関数で行われたことがわかります。
ソースコードを確認すると、**`memcpy()`** 関数が **`SPI8_Handle`** 構造体を変更するために意図的に使われた形跡はありませんでした。これは、**`memcpy()`** 関数が **`toReceiveCount`** 変数を上書きしていることを強く示しています。

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)
{
  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;
}
```

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：



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。