Catching Crashes in NET+OS using ESP

1 Document History

Date	Version	Change Description	
4/25/2014	1.0	Initial Entry	
5/1/2014	V1.1	Continued entry	
5/1/2014	V1.2	Continued entry	
5/2/2014	V1.3	Additional Editing/cleanup	

2 Table of Contents

1 Document History 2
2 Table of Contents 3
3 Introduction 4
3.1 Problem Solved 4
3.2 Audience 4
3.3 Assumptions 4
3.4 Scope 4
3.5 Theory of Operation 4
4 Catching crash-causing defects in the debugger 4
4.1 Introduction 4
4.2 Setting up the exception breakpoints (ARM7) 6
4.3 Next Steps 14
4.4 Debugging 17
4.5 Interim Conclusions 24
4.6 Viewing memory 24
4.7 Additional conclusions 28
4.8 ARM9 Differences 28
5 Example Application Explanation 41
6 Conclusion 41
7 Appendix 41
7.1 Glossary of terms 41
7.2 Citations 41

3 Introduction

This paper discusses catching NET+OS project crashes using ESP.

3.1 Problem Solved

We all try hard to ensure that our project code does not cause a crash. But inevitably something goes haywire and our code crashes. Then we have to figure out what happened. This paper discusses setting up ESP to catch a crash, hopefully putting you on the road to finding the offending code and fixing it, thus eliminating the defect.

3.2 Audience

This paper is written for developers working with the NET+OS development environment who need to dig into code following a severe defect. In the cases that this paper is concerned with, this defect results in a crash.

3.3 Assumptions

This paper assumes that the developer reading this paper, is developing using Digi's integrated development environment (IDE), namely ESP.

3.4 Scope

The scope of this paper is quite narrow. It looks at setting up ESP for catching crashes.

This paper does not describe any of the following subjects:

- TCP/IP or socket programming
- C or C++ programming
- HTTP, AWS or anything involved with the web server
- Files or the file system

3.5 Theory of Operation

This paper uses a set of screen shots for describing setting up ESP for catching crash-causing defects.

4 Catching crash-causing defects in the debugger

4.1 Introduction

As hard as you try, sooner or later, some code you write will cause your system to crash. From our experience, a large number of crashes are caused by our code either over or under filling a buffer, array or other memory area. The memory area can be malloced, or declared. Further the memory can be global or stack memory. No matter, if you write bytes beyond the bounds of the variable, sooner or later your code will overwrite something vital, such as instructions, a pointer or something else. As a result of this, one of the exception vectors will be referenced and the assembly code associated with that

vector will be called. To quote from the book ARM System Developer's Guide: "When an exception or interrupt occurs, the processor sets the PC to a specific memory address. The address is within a special address range called the vector table. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.

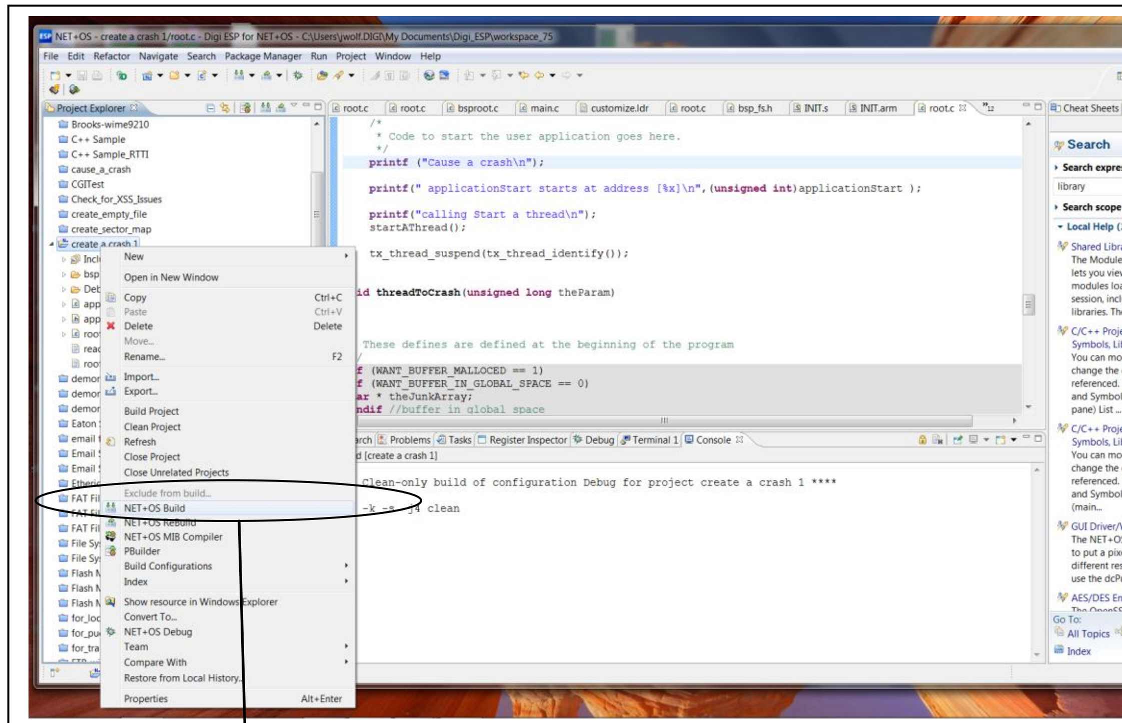
The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. …..When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table…Each vector table entry contains a form of branch instruction pointing to a specific routine".
The set of exceptions are as follows:

- Reset vector
- Undefined Instruction Vector
- Software Interrupt Vector
- Prefetch Abort Vector
- Data Abort Vector
- Interrupt Request Vector

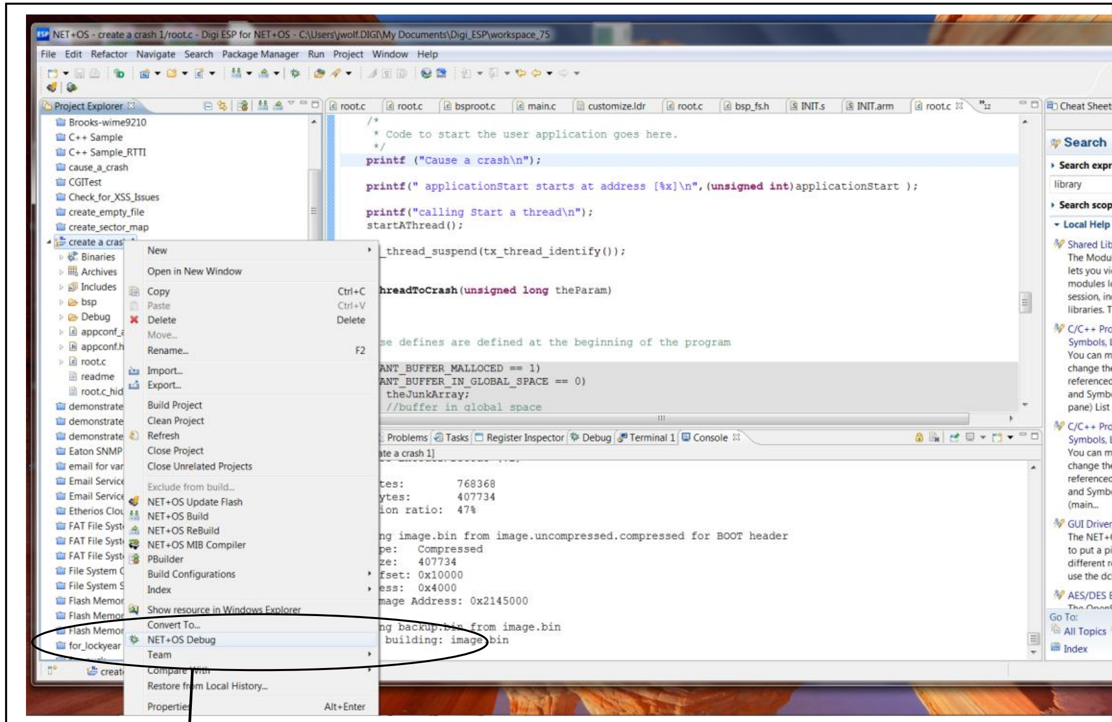
We believe the lion's share of the exceptions you will encounter are of the Data Abort Vector variety. The book ARM System Developer's Guide describes a data abort vector exception as "similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions".

4.2 Setting up the exception breakpoints (ARM7)

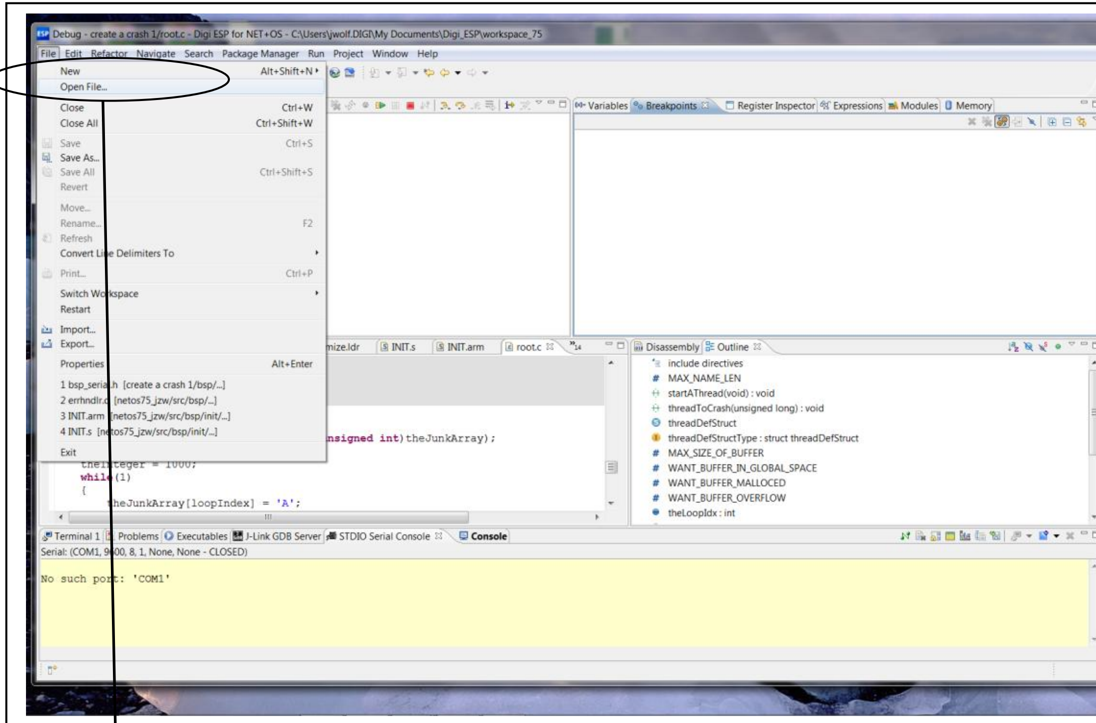
So let’s set up the breakpoints we need to catch a crash. First we need to get into the debugger view. This assumes you have a working application. You have built it and have downloaded it into your device.



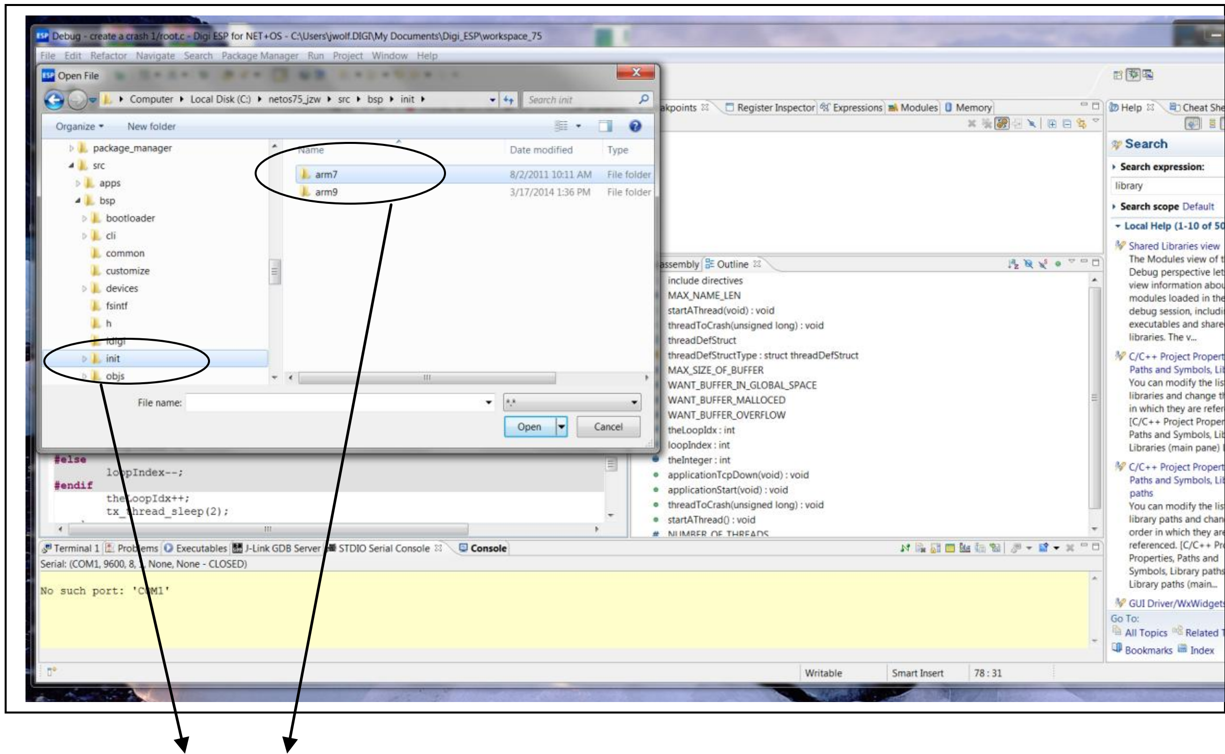
Build your application as you would normally do.



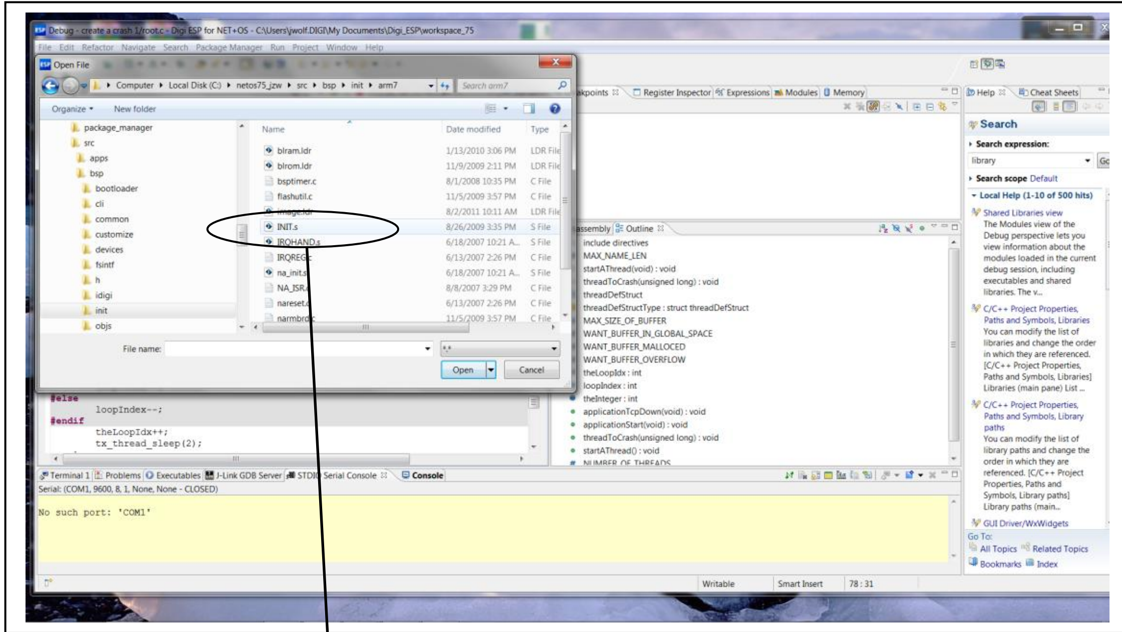
Begin debugging your application.



We need to open the file in which the vector table functions are stored. To do this click on the open file menu tab.

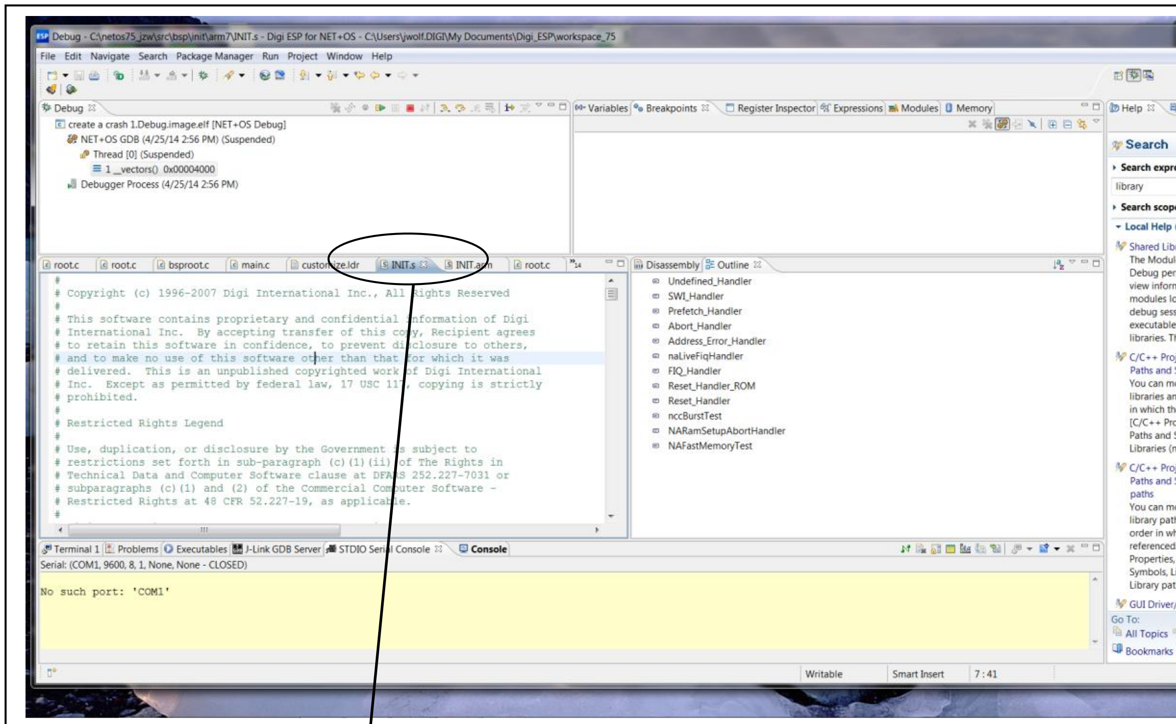


In the file widget that is displayed, select src\bsp\init\arm7. Directory src\bsp\init\arm7 is located in the directory tree where you installed NET+OS. It is not located with your ESP projects.

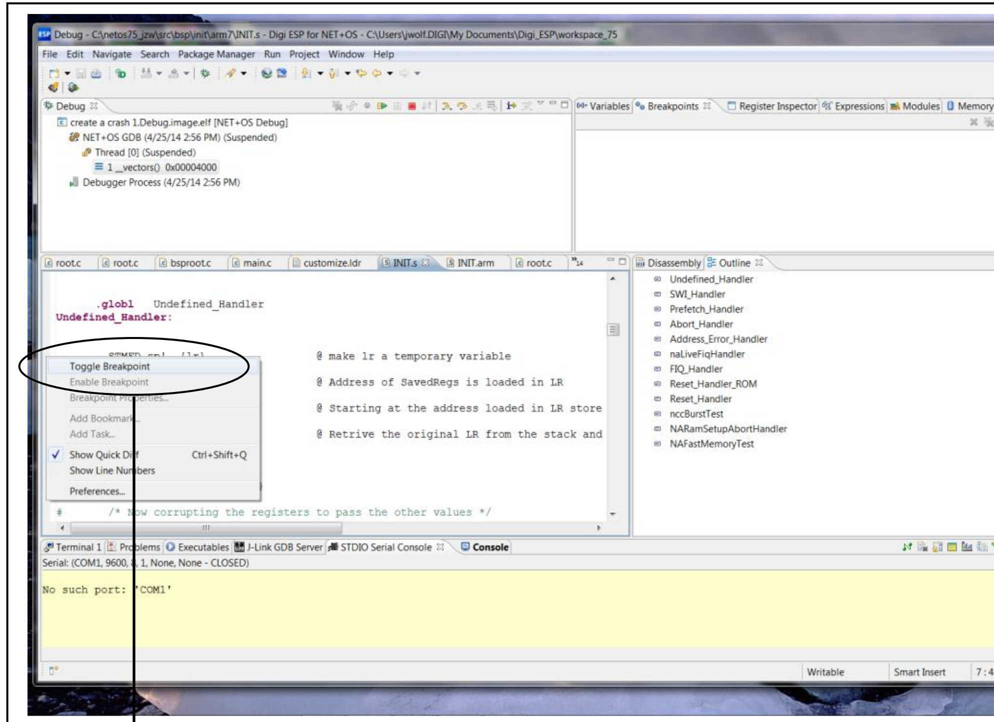


Select file INIT.s. This is the arm7 assembly file containing the vector table functions.

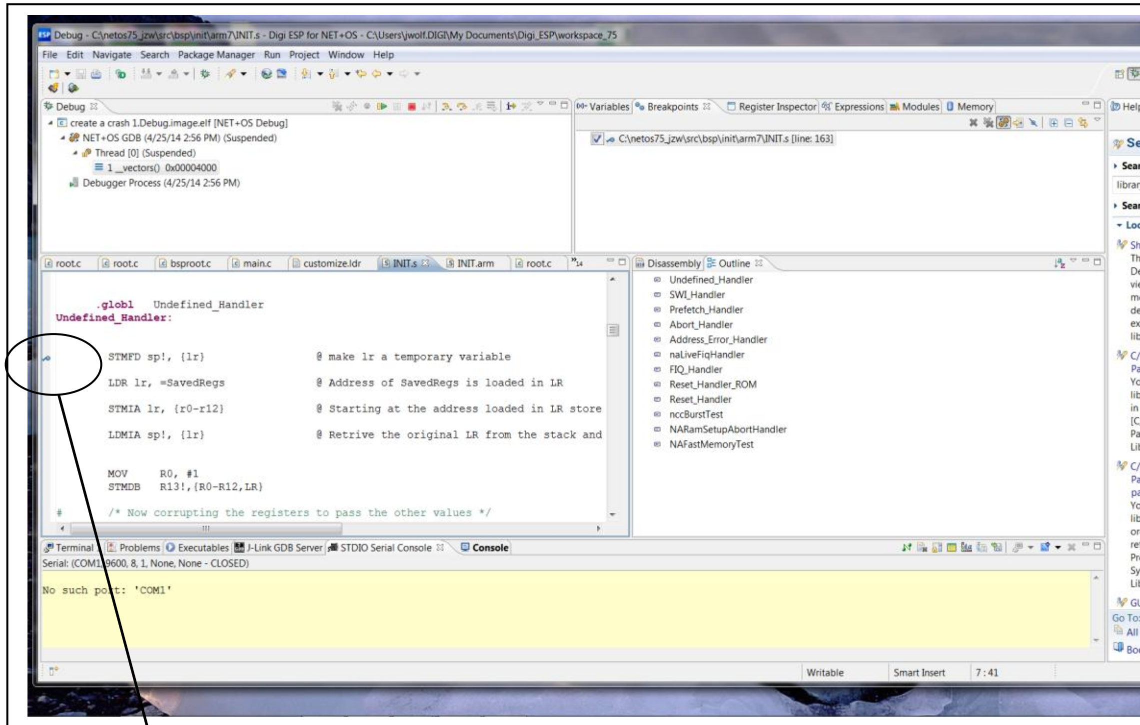
The next step(s) is to walk through the file INIT.s, find each function in which we want a breakpoint, and set that breakpoint.



We now have INIT.s in the editor and we are ready to set breakpoints.



We shall start with the Undefined_Handler. Select the first executable statement following the function label, right click and select Toggle Breakpoint.

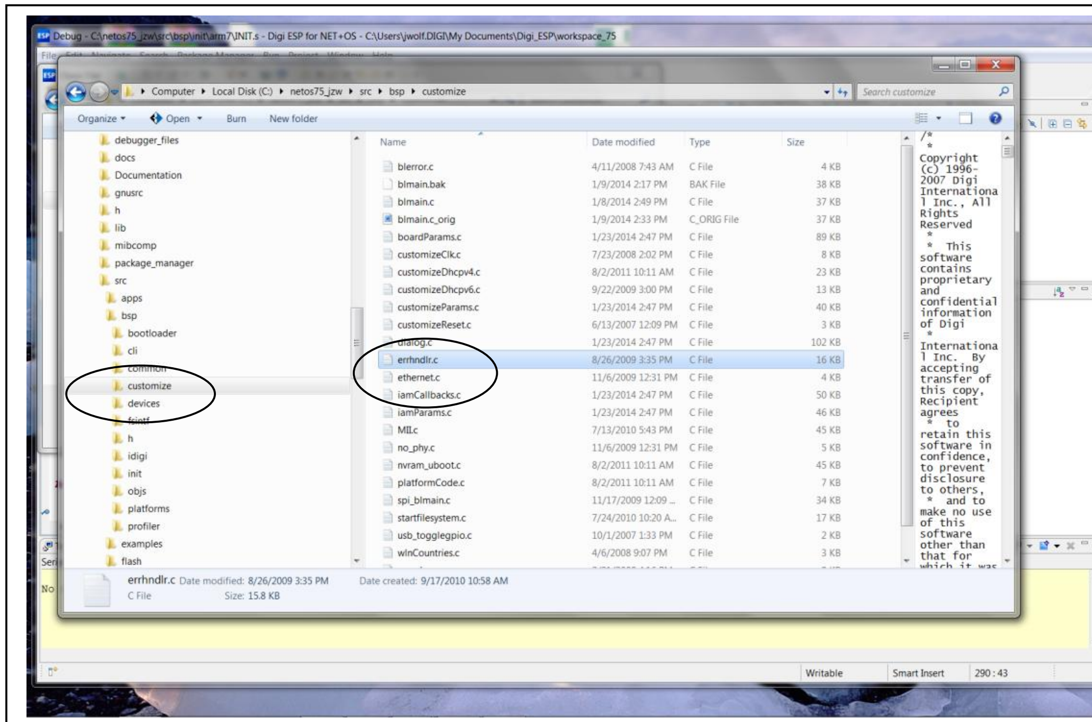


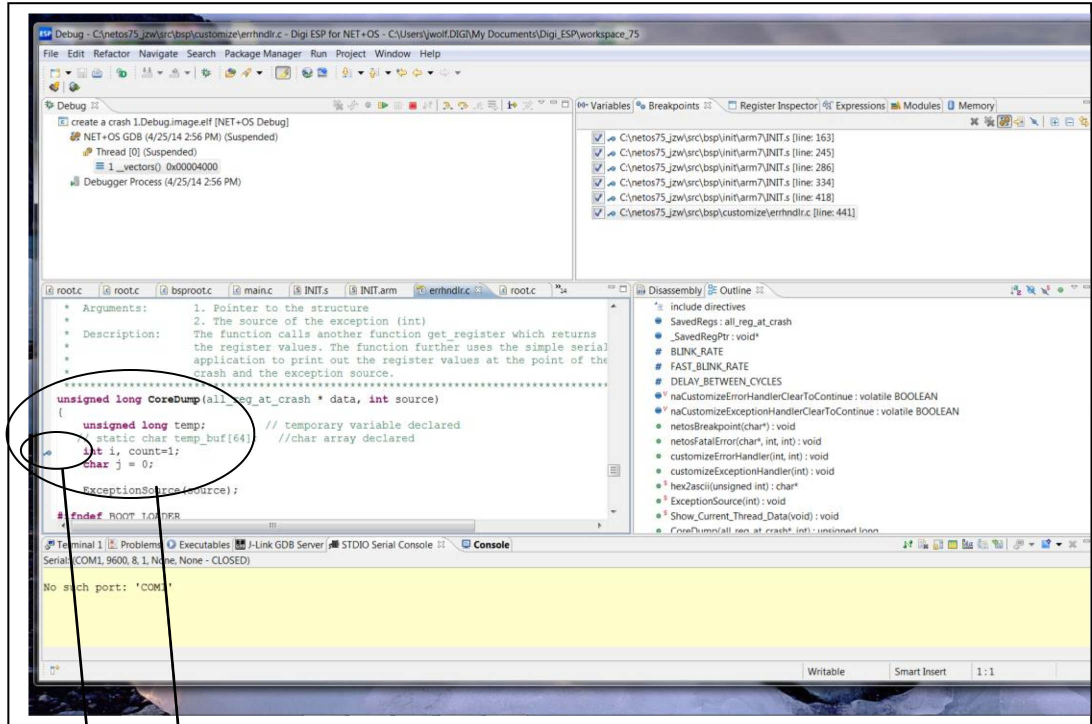
The little symbol to the left of the STMFD instruction, signifies that a breakpoint has been set.

4.3 Next Steps

To complete setting up the breakpoints, perform the steps we did with the Undefined_Handler with Prefetch_Handler. Abort_Handler, Address_Error_handler and crash.

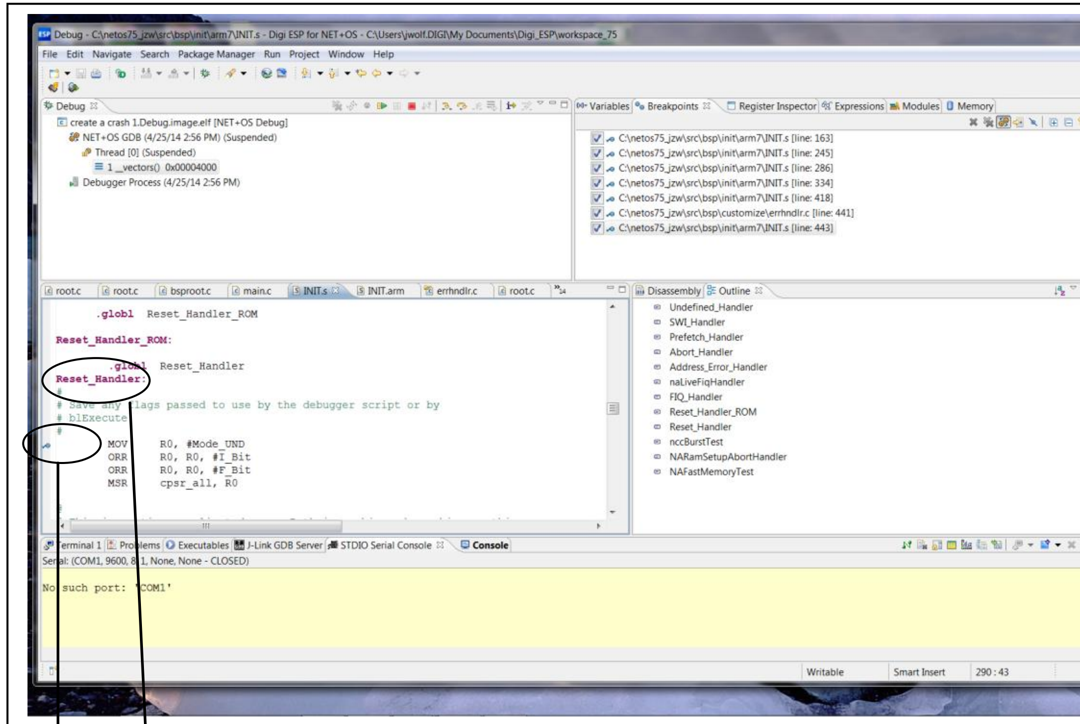
Next click on the Open File menu tab that we used earlier, and select src\bsp\customize. Select file errhndler.c. As was stated earlier, directory src\bsp\customize is located in the directory tree where you installed NET+OS.





Set a breakpoint in function CoreDump.

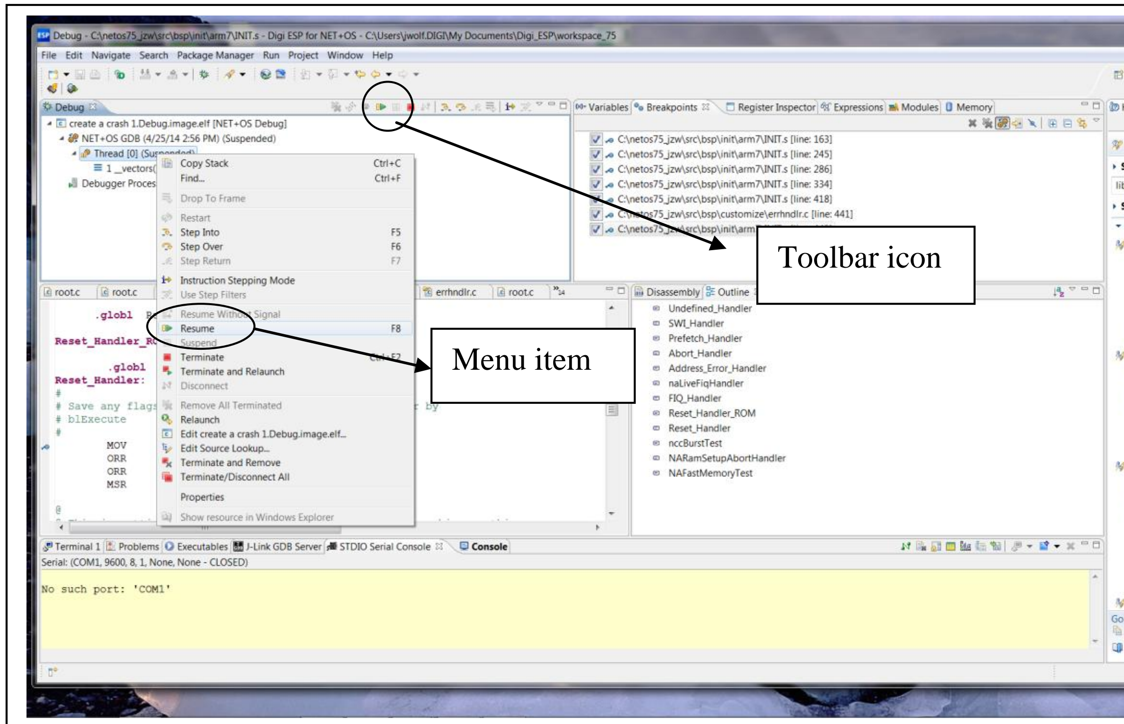
The next breakpoint is optional, depending on what you are looking for. You can set a breakpoint at Reset_Handler, as shown below:



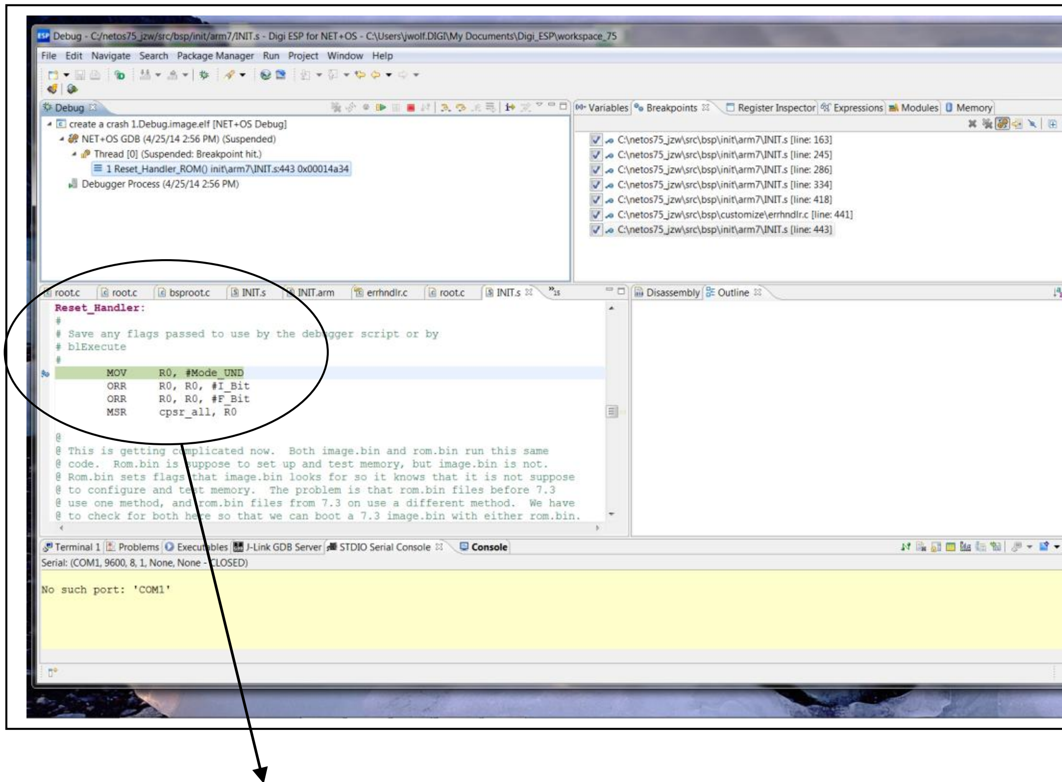
Reset_Handler is the first piece of code run after the debugger downloads the code to the device and gives control over to the device. If you are having trouble whereby you are not even getting to applicationStart, this might be a useful breakpoint.

4.4 Debugging

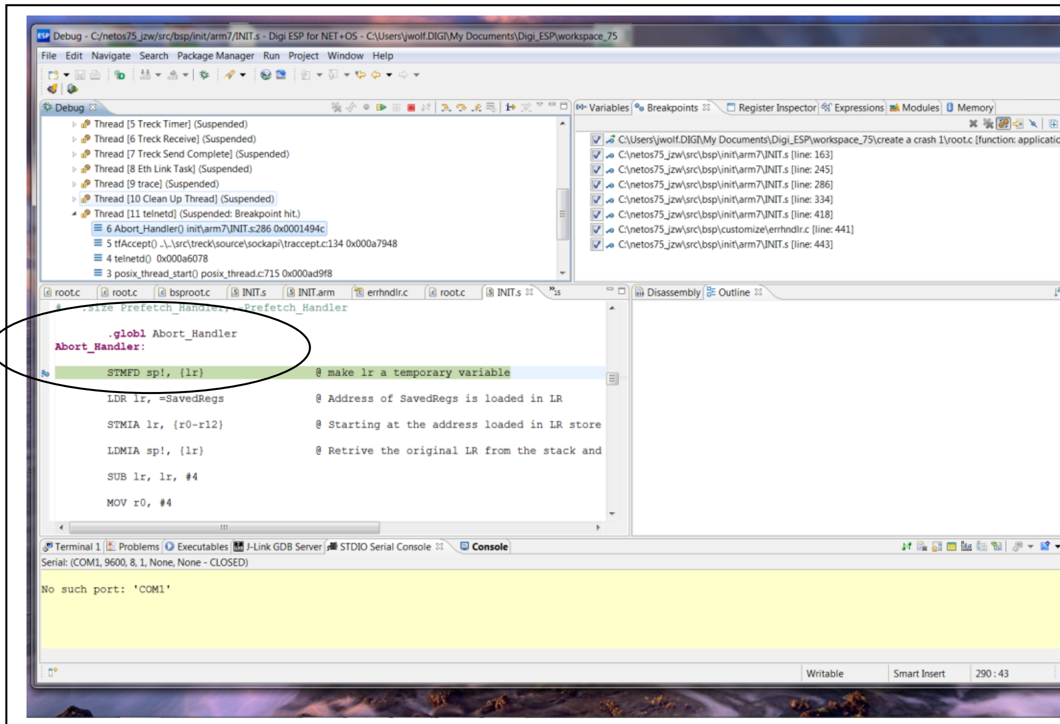
Now that we have the breakpoints set we can resume the running of the application.



You have the option of resuming from the menu list associated with your application or you can use the icon in the toolbar.

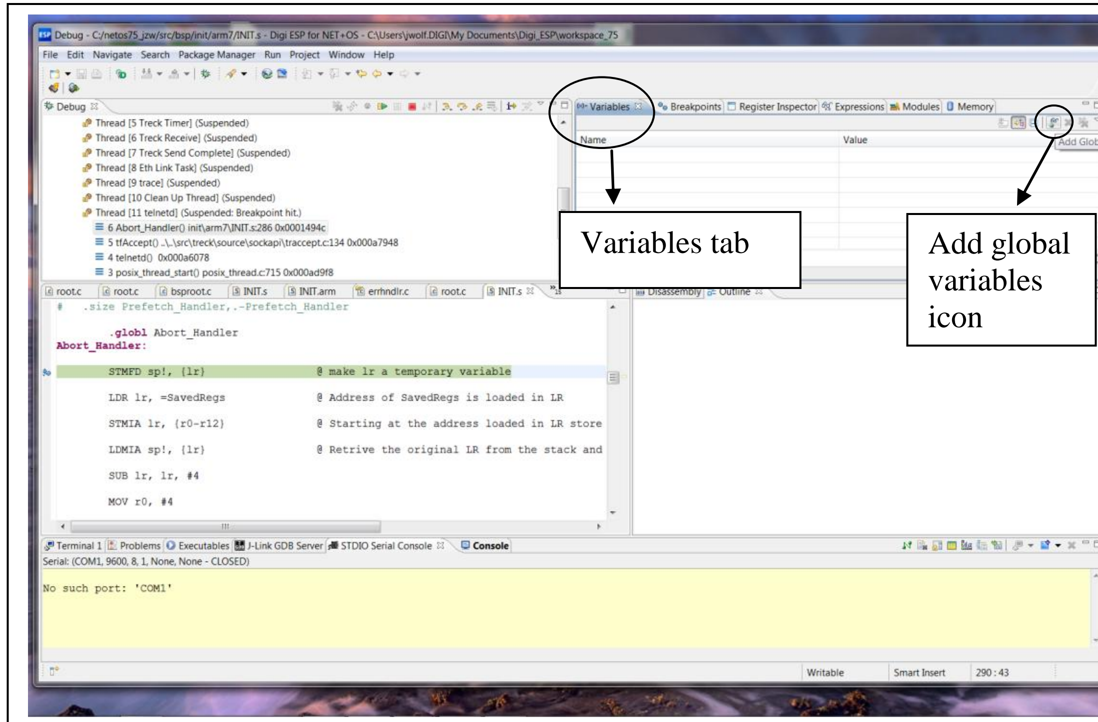


If you chose to set a breakpoint at Reset_Handler the debugger would stop at the Reset_Handler breakpoint, as shown above. Select resume (either form the menu or the toolbar) to continue. At this point, the presumption is your application will run for a while. The length of "a while" is application dependant.



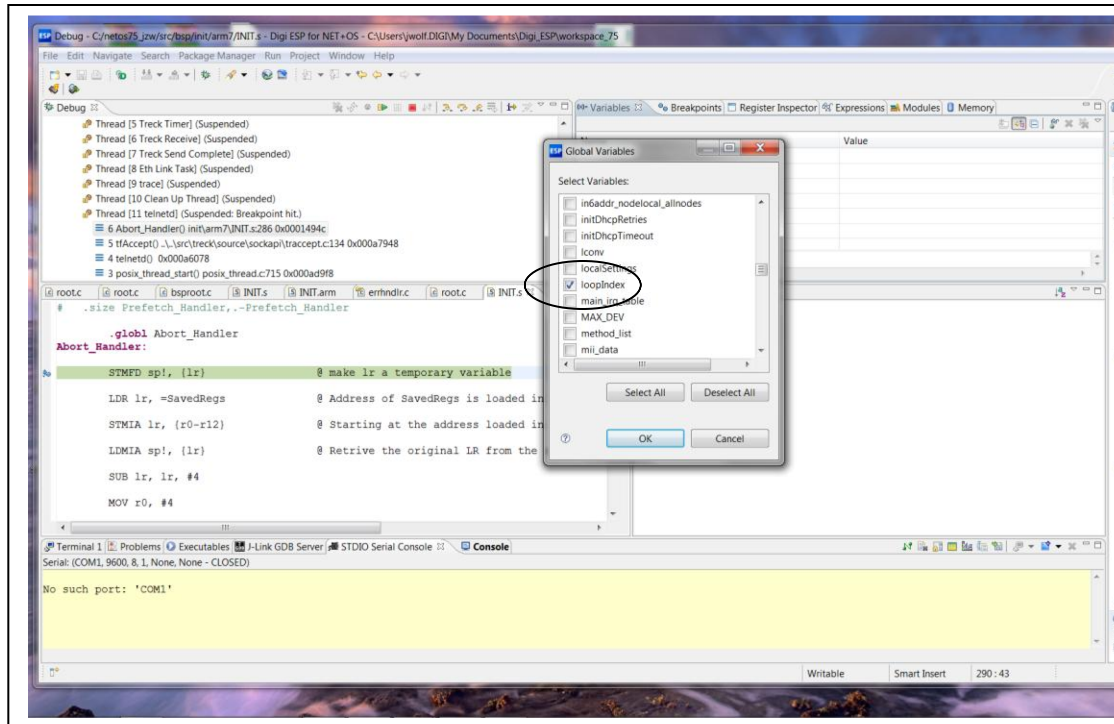
We have experienced an abort (crash). Notice that we have hit the breakpoint at the Abort_Handler function. Now what? This will depend on the nature of the crash and how badly memory was corrupted.

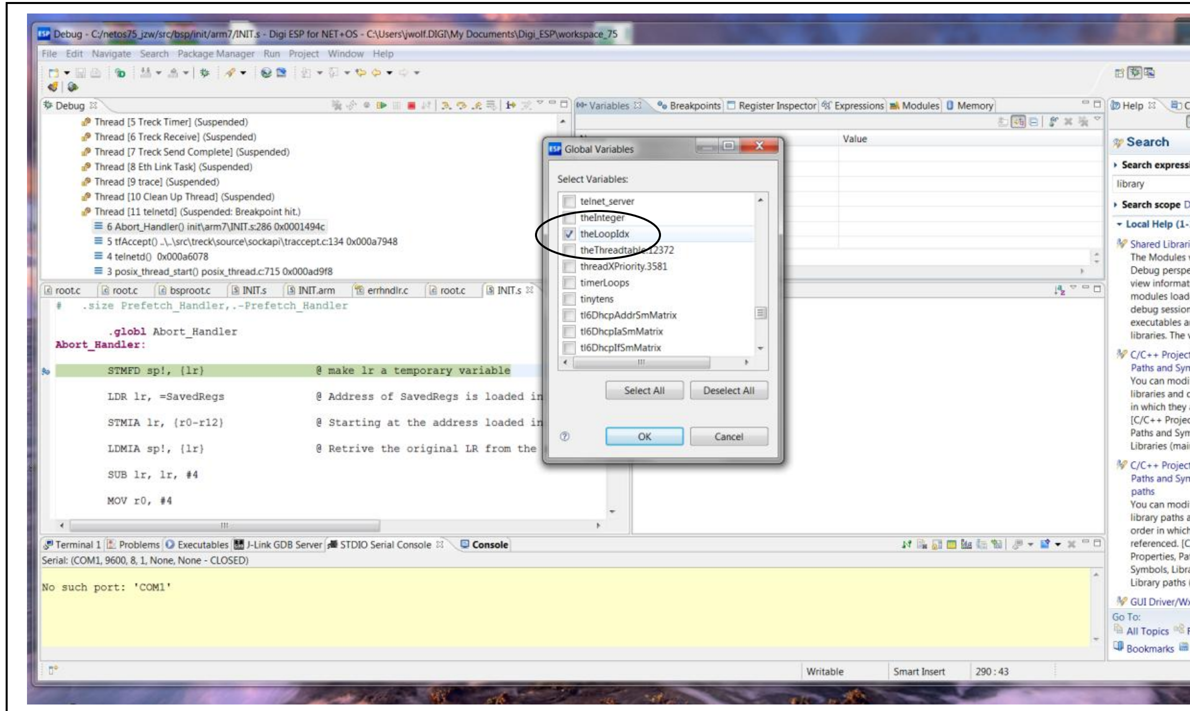
One thing you may want to do, at this juncture, is to look at your global variables. Maybe you have some global variables that might give you some insight as to what happened or how many iterations your application has performed before the crash. So let's view our global variables.

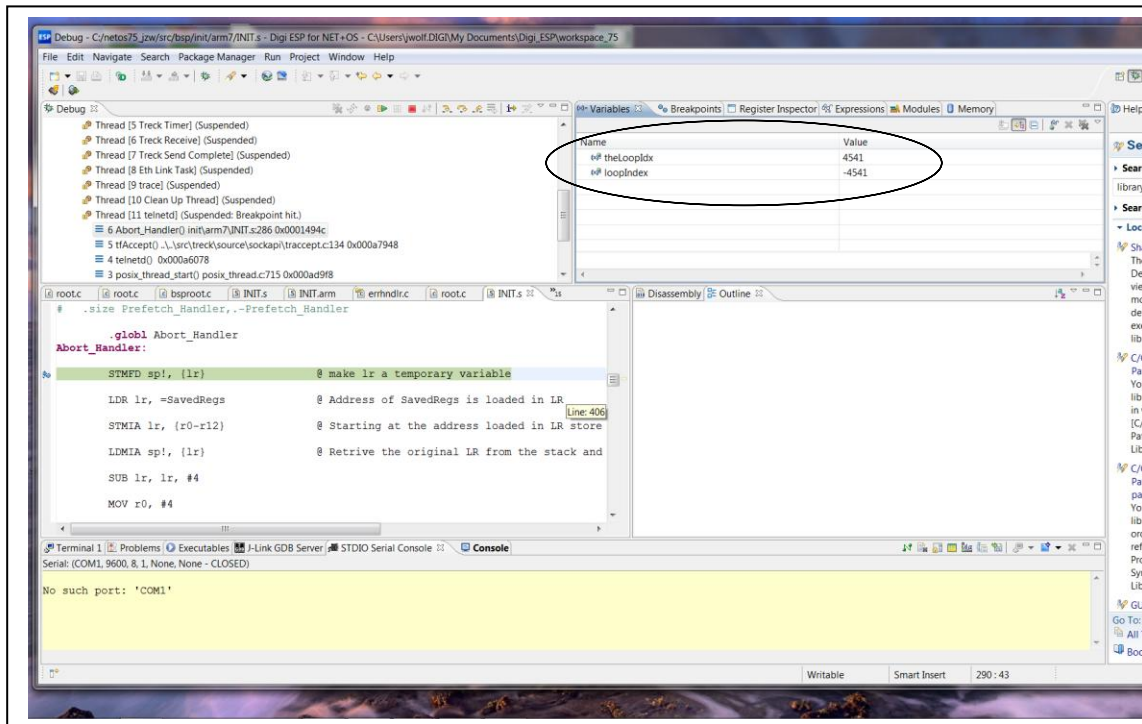


Select the variables tab in the middle right of the screen. Select the global variables icon, which looks like a pair of glasses on the upper right side of the screen.

The next three screen shots show the selecting of two global variables from the global variables widget and the displaying of the global variables in the variables section of the debugger perspective. We are selecting loopIndex and theLoopIdx.




Debug - C:/netos75_jzw/src/bsp/init/arm7/INIT.s - Digi ESP for NET+OS - C:\Users\jwolf.DIGI\My Documents\Digi_ESP\workspace_75
File Edit Navigate Search Package Manager Run Project Window Help
Debug
Thread [5 Treck Timer] (Suspended)
Thread [6 Treck Receive] (Suspended)
Thread [7 Treck Send Complete] (Suspended)
Thread [8 Eth Link Task] (Suspended)
Thread [9 trace] (Suspended)
Thread [10 Clean Up Thread] (Suspended)
Thread [11 telnetd] (Suspended: Breakpoint hit.)
6 Abort_Handler() init\arm7\INIT.s:286 0x0001494c
5 tfAccept() ..\..\src\treck\source\sockapi\traccept.c:134 0x000a7948
4 telnetd() 0x000a6078
3 posix_thread_start() posix_thread.c:715 0x000ad9f8
Variables
Breakpoints
Register Inspector
Expressions
Modules
Memory
Value
Global Variables
Select Variables:
telnet_server
theInteger
theLoopIdx
theThreadtable.12372
threadXPriority.3581
timerLoops
tinytens
tl6DhcpAddrSmMatrix
tl6DhcpIaSmMatrix
tl6DhcpIfSmMatrix
Select All
Deselect All
OK
Cancel
.globl Abort_Handler
Abort_Handler:
STMFD sp!, {lr} @ make lr a temporary variable
LDR lr, =SavedRegs @ Address of SavedRegs is loaded in
STMIA lr, {r0-r12} @ Starting at the address loaded in
LDMIA sp!, {lr} @ Retrive the original LR from the
SUB lr, lr, #4
MOV r0, #4
Terminal 1
Problems
Executables
J-Link GDB Server
STDIO Serial Console
Console
Serial: (COM1, 9600, 8, 1, None, None - CLOSED)
No such port: 'COM1'
Help
Search
Writable
Smart Insert
290 : 43



In the last two slides we selected the global variables loopIndex and theLoopIdx to be displayed. In this slide, the variables are displayed along with their current values.

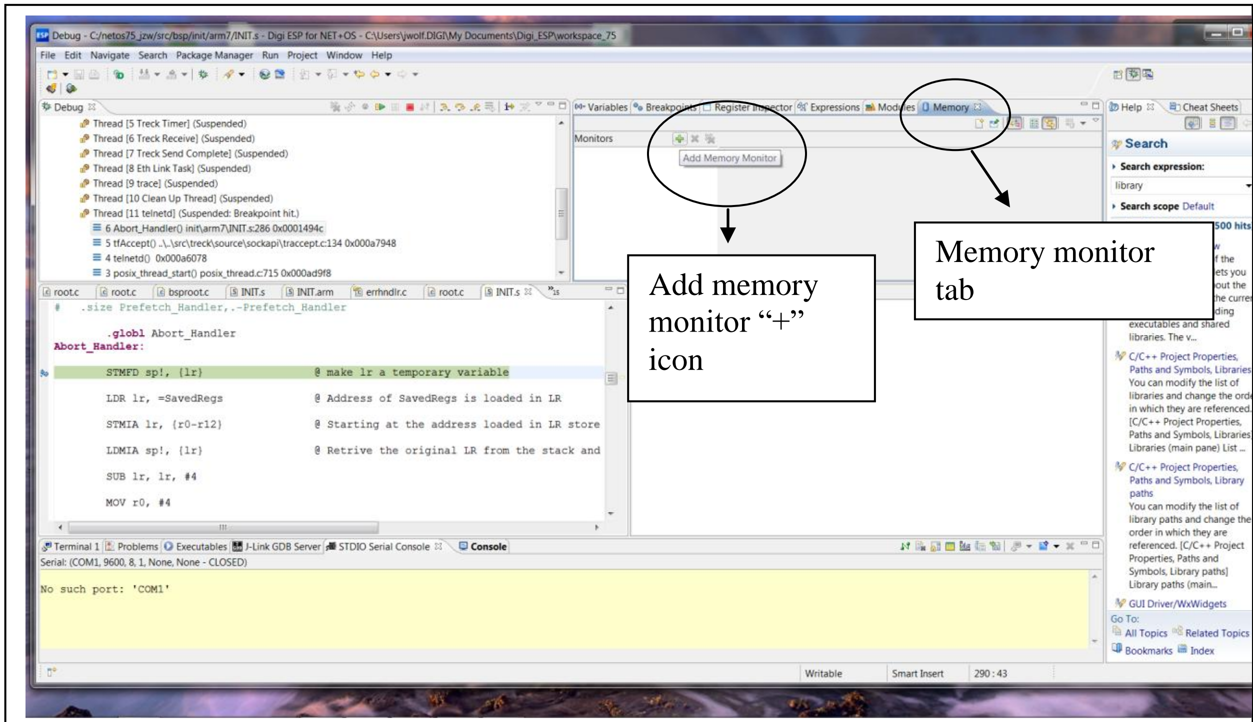
4.5 Interim Conclusions

So what can we conclude as to what event might have lead to the system crash? Well one interesting item is that the value of loopIndex (see the previous screen) is a fairly large negative number. If this was being used as an index into a buffer or array, this could be troubling. Additionally, the global variable has a value of 4500 plus. If the buffer to which it refers is 100 or 200 elements long, this might also be troubling. We should keep investigating.

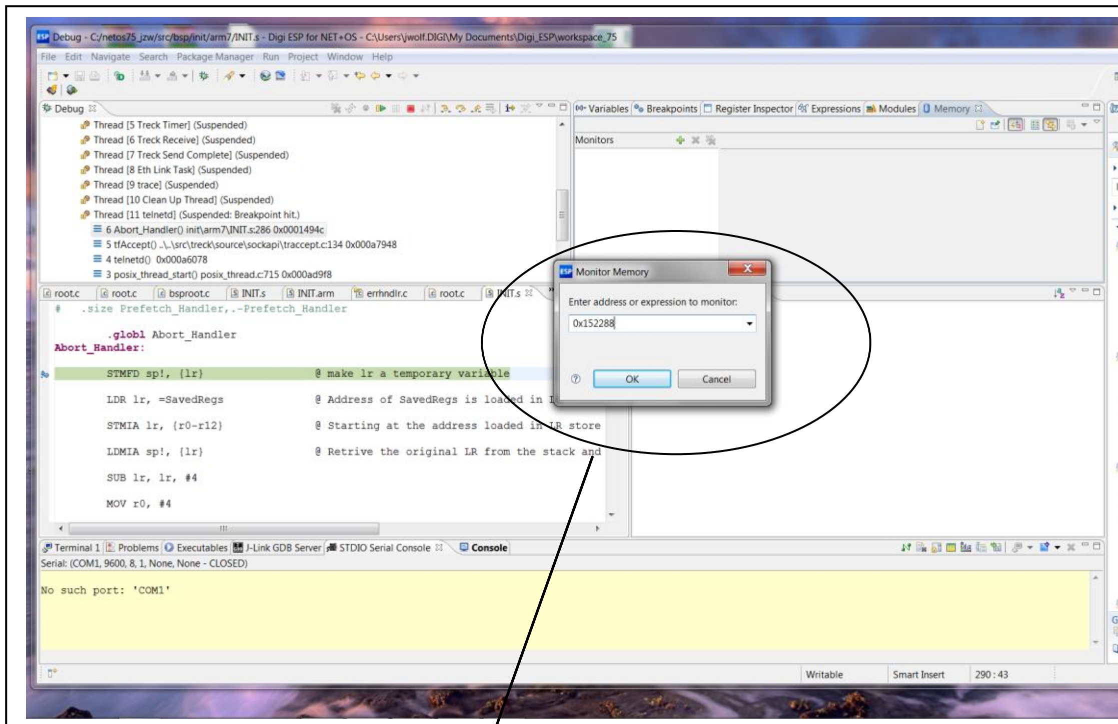
4.6 Viewing memory

The next thing we'll do is view known memory locations for anything "out of the ordinary".

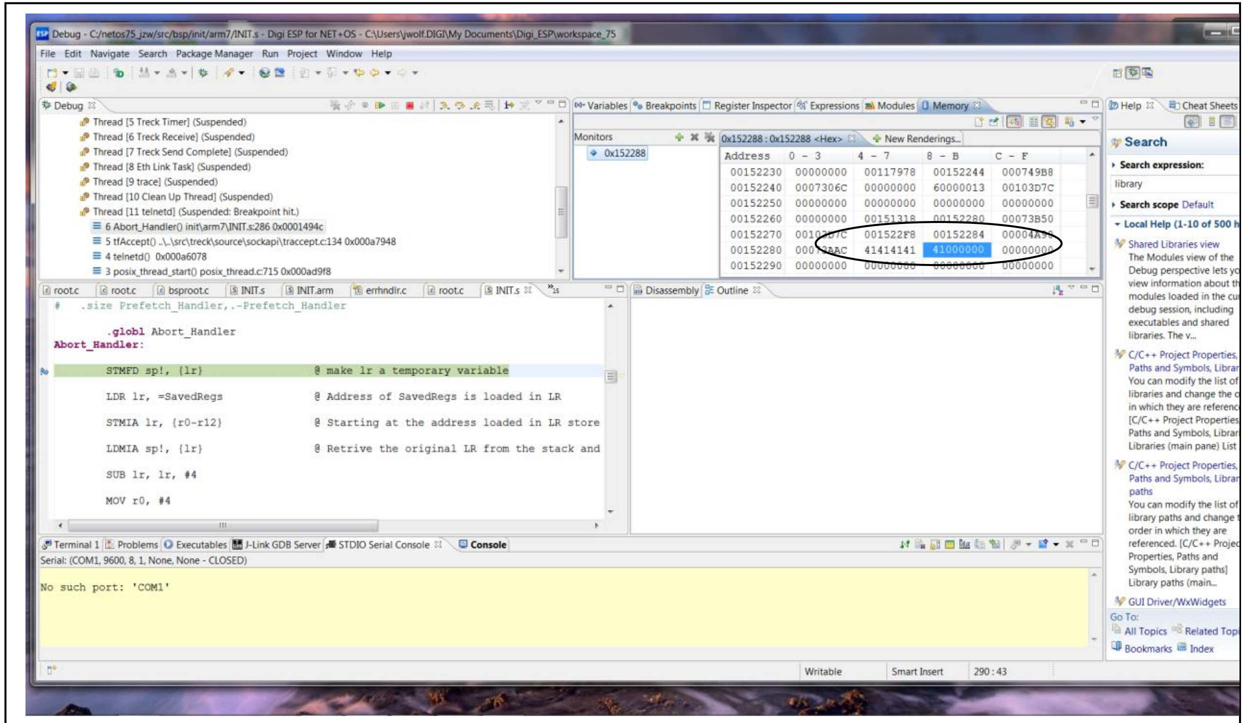
To do this we want to add a memory monitor.



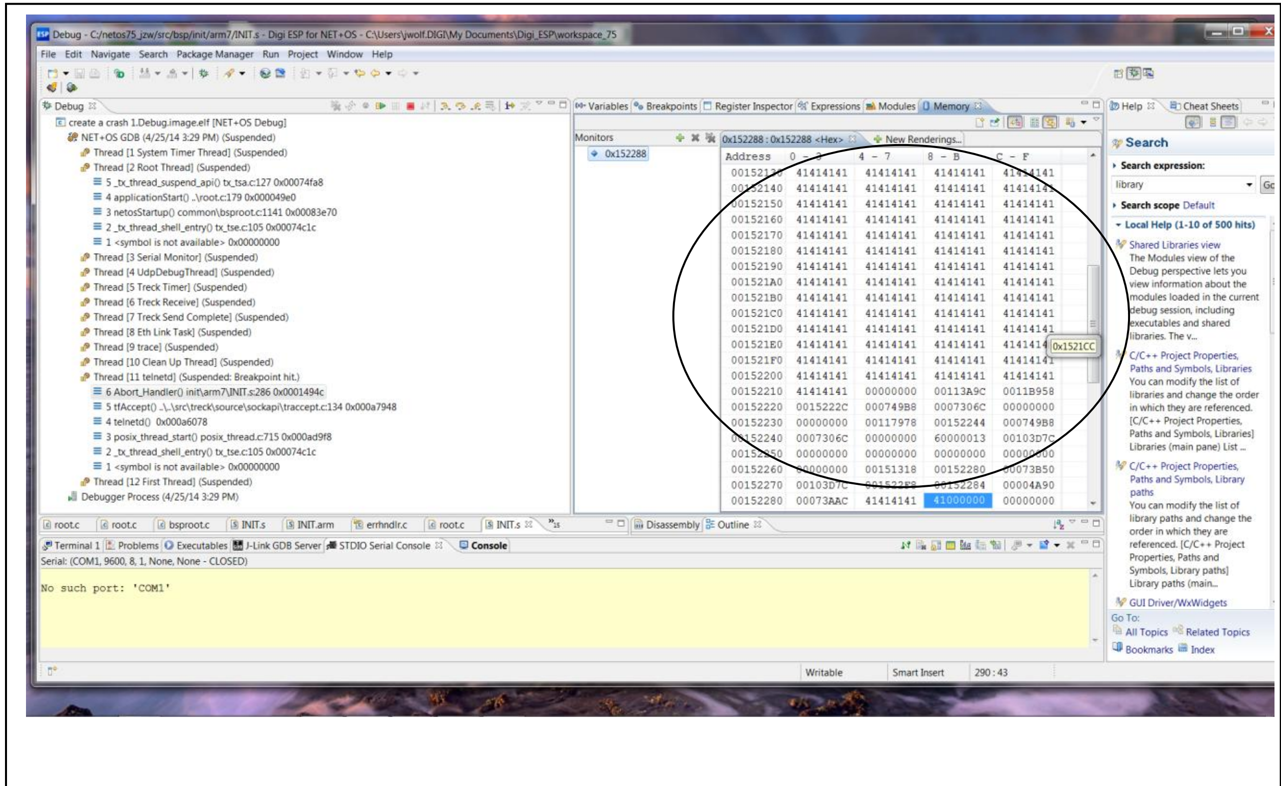
Click on the memory monitor tab in the upper right of the debug perspective. Click on the "+" sign in the memory monitor toolbar.



A child-window will pop up. Fill in the address of memory you want to start viewing. The address should be a HEX address starting with 0x. Then click OK.



Let’s look at memory in the area of the memory that we thought might be suspect. I see some bytes that contain the letter “A”. Maybe that is not what you were expecting. This might be a clue. Let’s keep looking.



I think we found something. We are probably over (or under) running a memory location. I would not expect that much data in this area of memory. It is time to go back to my source code and look for cases where I could be over or underwriting memory. Maybe the location into which we are putting some data is not large enough, to hold that data. Maybe some loop is not stopping where it should.

4.7 Additional conclusions

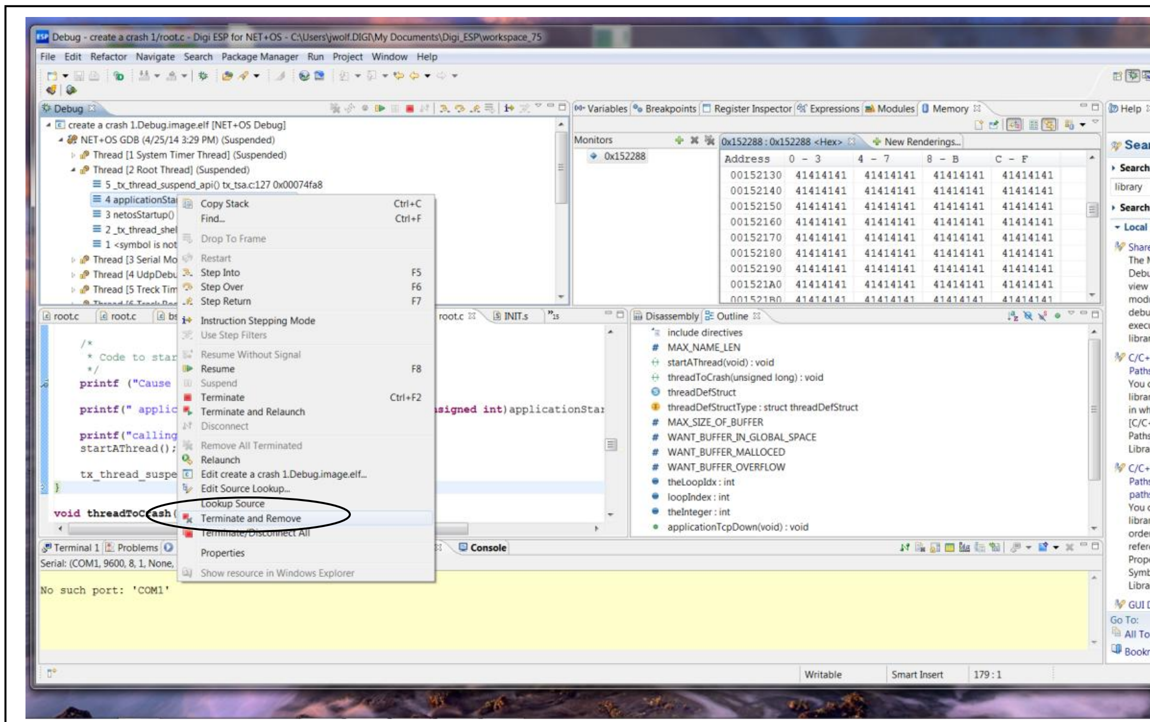
We probably have some sort of memory corruption occurring in the application. In our case, we started with a buffer of 100 bytes and allowed the code to continue overwriting or underwriting the buffer until something bad occurred. This was a mission to specifically cause memory corruption and the ensuing crash. In your case, it is not an intentional crash and it will probably take some detective work to find the culprit.

You will notice that in the debugger, in looking at the call stack, it appears that the crash occurred in the telnetd thread. In the testing to create this paper we saw instances where it appeared the crash was in the system timer thread and the treck timer thread. In another case, we lost all data in the debugger. If you believe you are done debugging and believe the problem is in either the telnetd thread, the system timer thread or the treck timer thread, you are fooling yourself. You have corrupted memory to a point where you stepped on something vital, causing that thread to crash. You will need to go back to your application and reason out what operation is going beyond its bounds.

4.8 ARM9 Differences

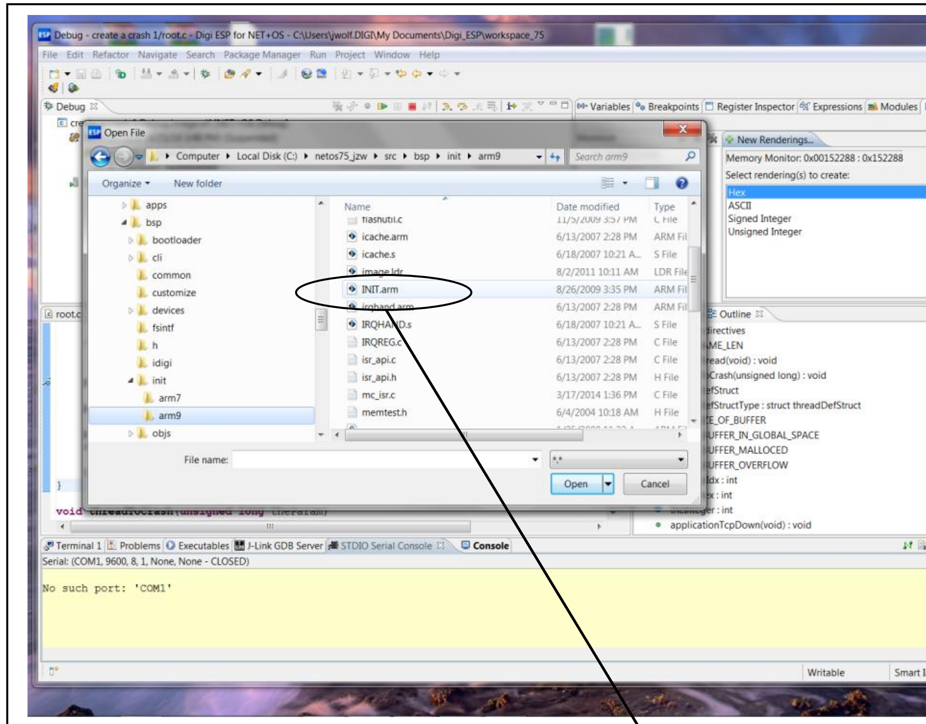
Most of what we have presented so far is applicable to both ARM7 and ARM9 devices. There is one place where this is not the case so let's go and explore that one area of deviation.

First let's stop the current application and remove it from the debugger.



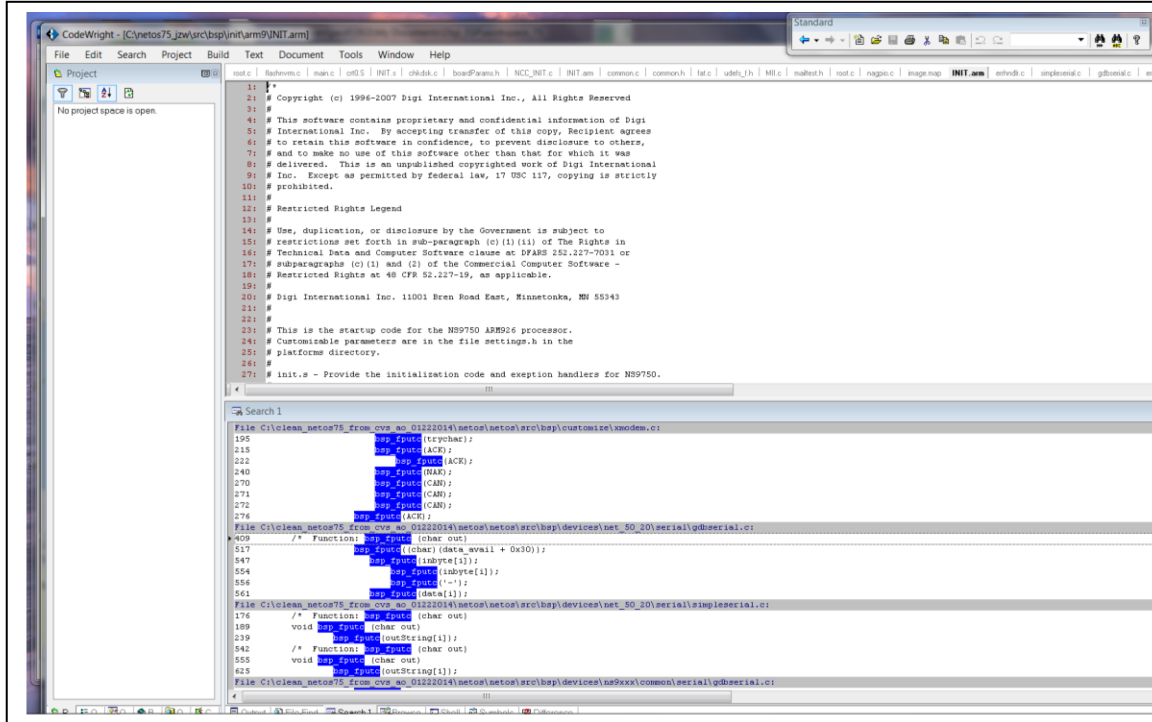
Next we will bring up an application, in the debugger, on an ARM9-based module. We will need to perform an "open file" as we did previously as the init file is different for the ARM9, as compared to ARM7-based modules.

Notice we are searching in src\bsp\init\arm9 as opposed to src\bsp\init\arm7. Also as stated previously, directory src\bsp\init\arm9 is located in the directory tree where you installed NET+OS. It is not located with your ESP projects.



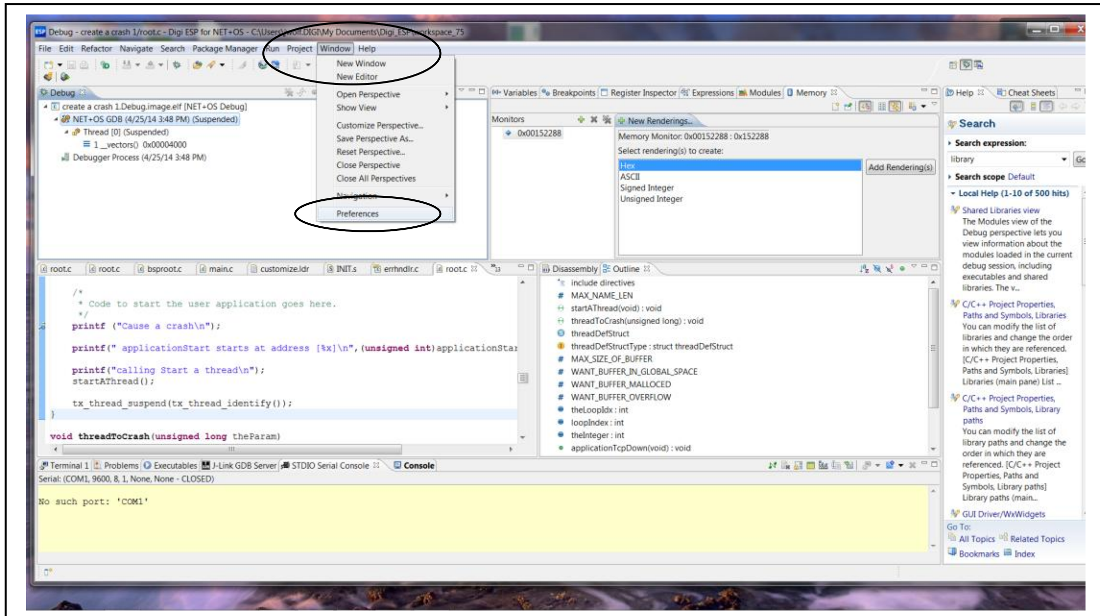
Additionally, instead of opening INIT.s we want to open INIT.arm. In the file INIT.arm the breakpoints we need to set have the same names as the breakpoints we set for the ARM7 module.

Ok, let’s hit the “open” button at the bottom of the file widget, get the file INIT.arm into the debugger, set our breakpoints and get our debugging going.

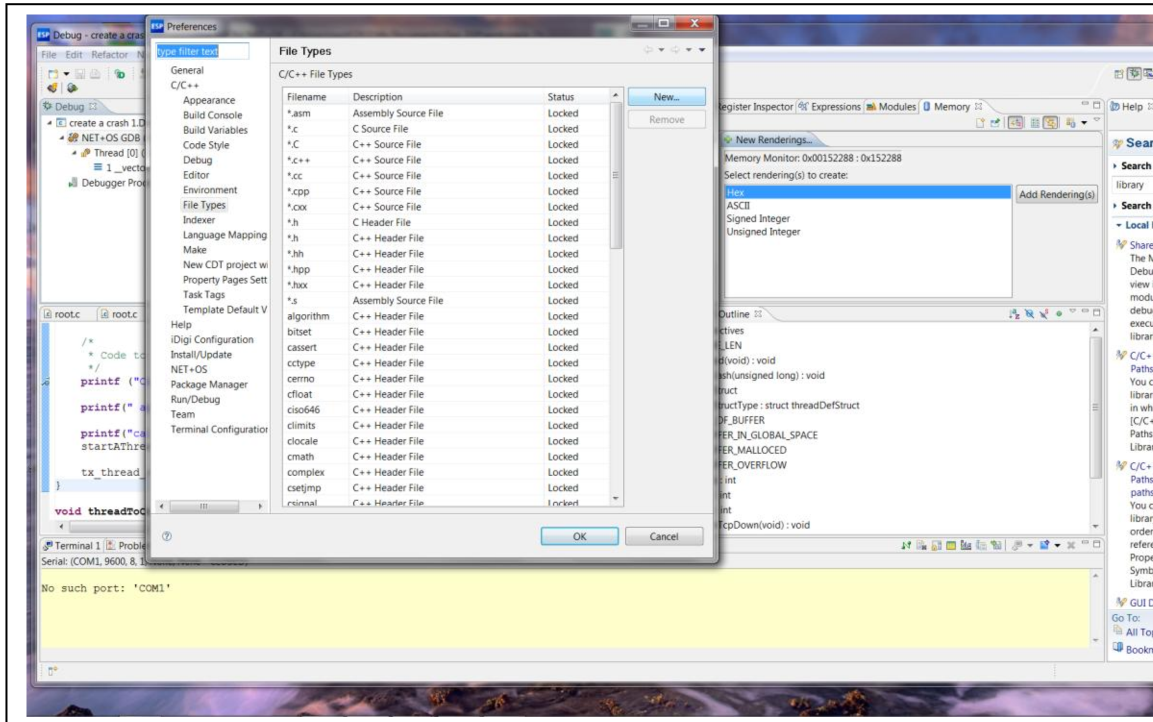


We have a problem! I hit the open button on the file widget. Instead of INIT.arm opening in the debugger, my editor opened the file instead. What is going on here?

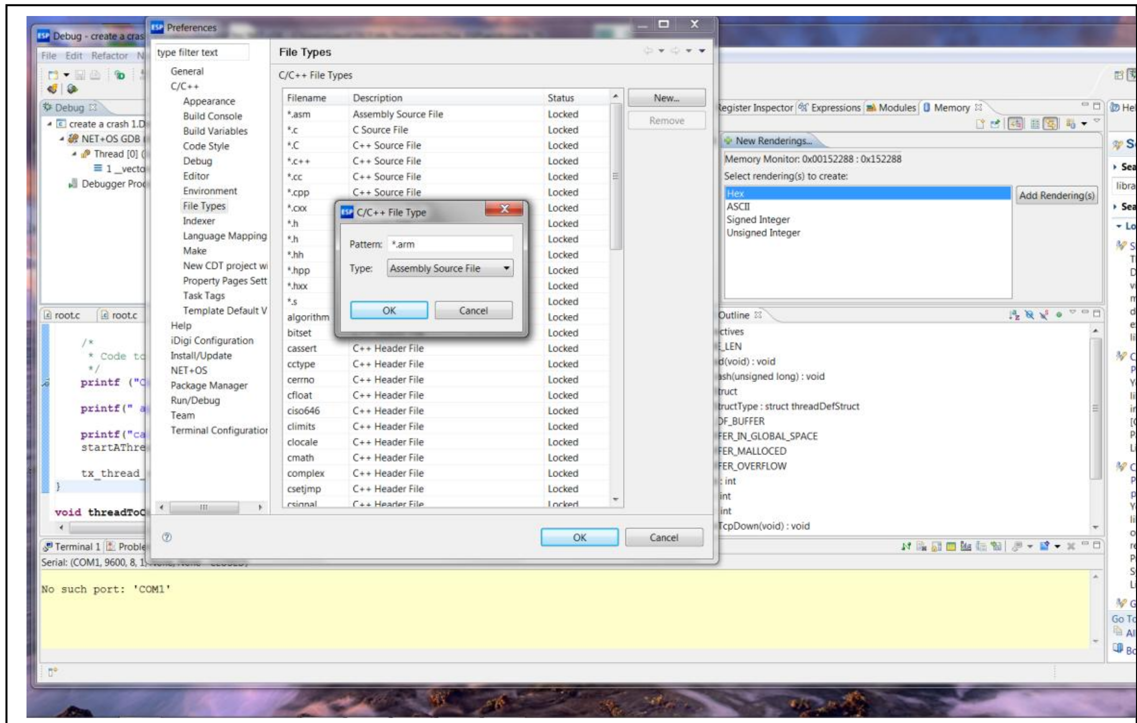
Just for the sake of completeness let us check out system preferences to see if we can find something that might catch our eye.



Click on window and under window click on preferences.

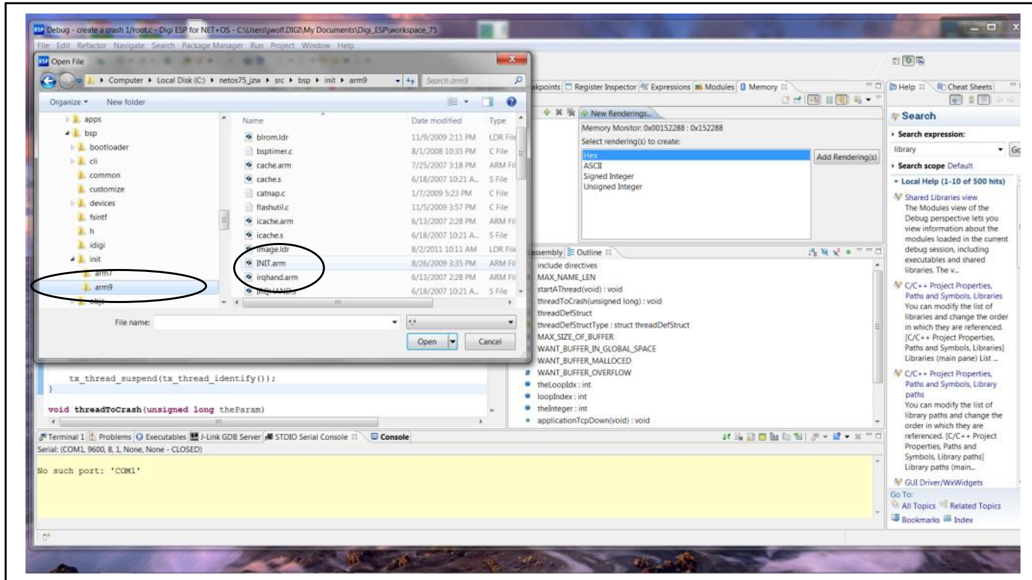


Now this is interesting. Under C/C++ and File Types, I see *.asm but I do not see *.arm. This sounds like a lead to follow. Let's try adding *.arm and see whether our results are any different.

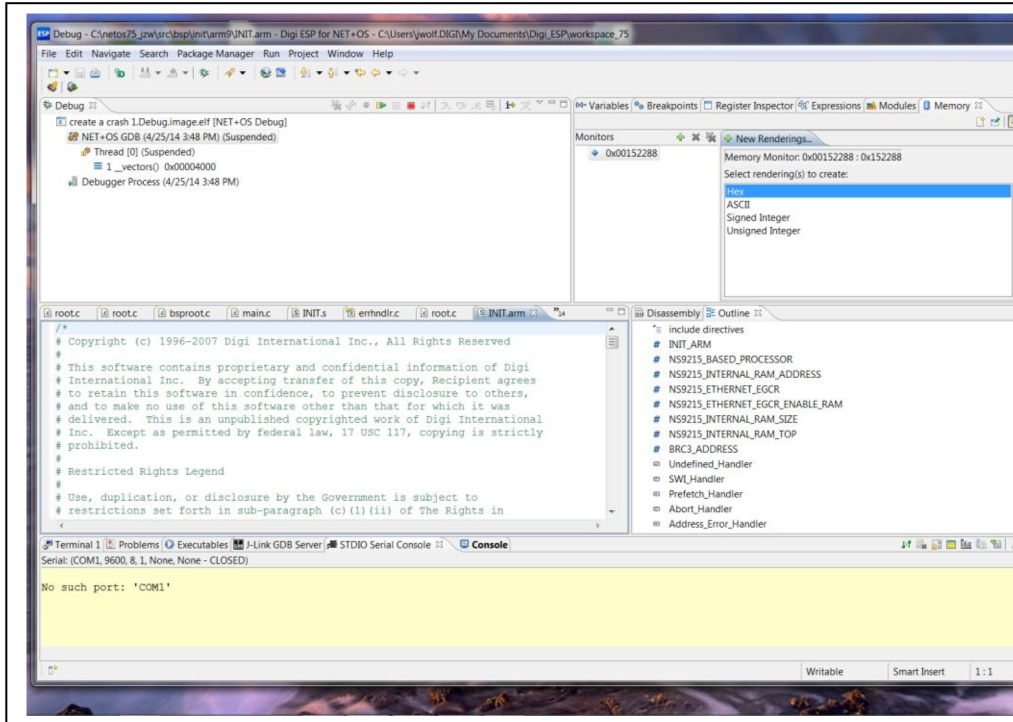


Click on new and a child-window pops up. In the pattern space type *.arm. In the type space select Assembly Source File. Then click OK.

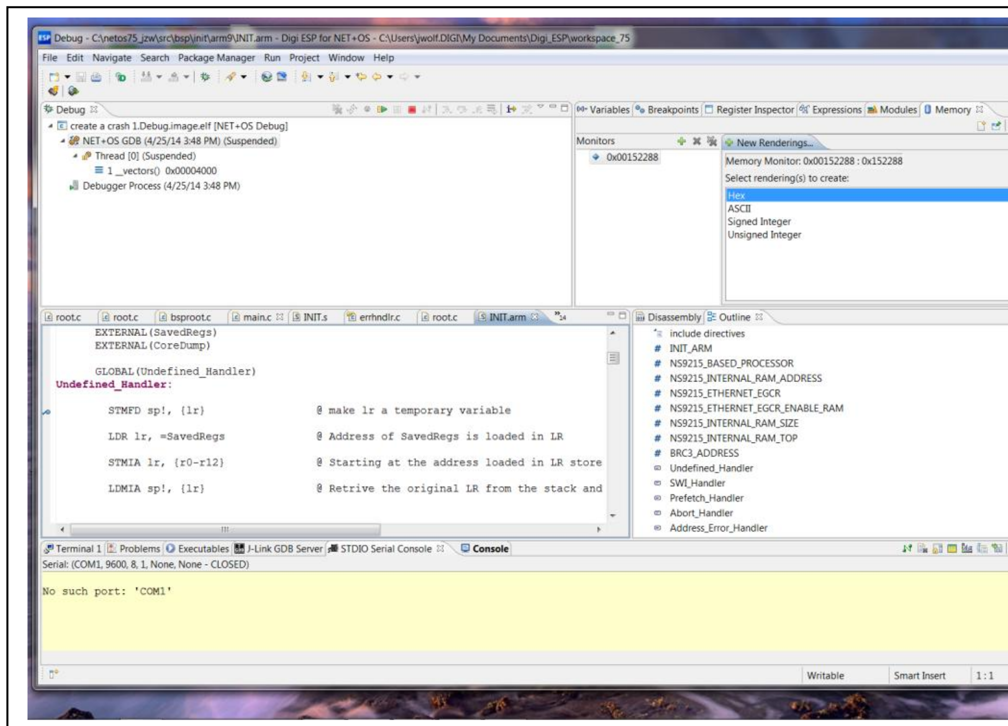
Step one is now complete as *.arm files are now included in the known file types. As an aside, notice that *.arm is marked as user defined. So now we should try opening up INIT.arm again and see whether the results are any different.



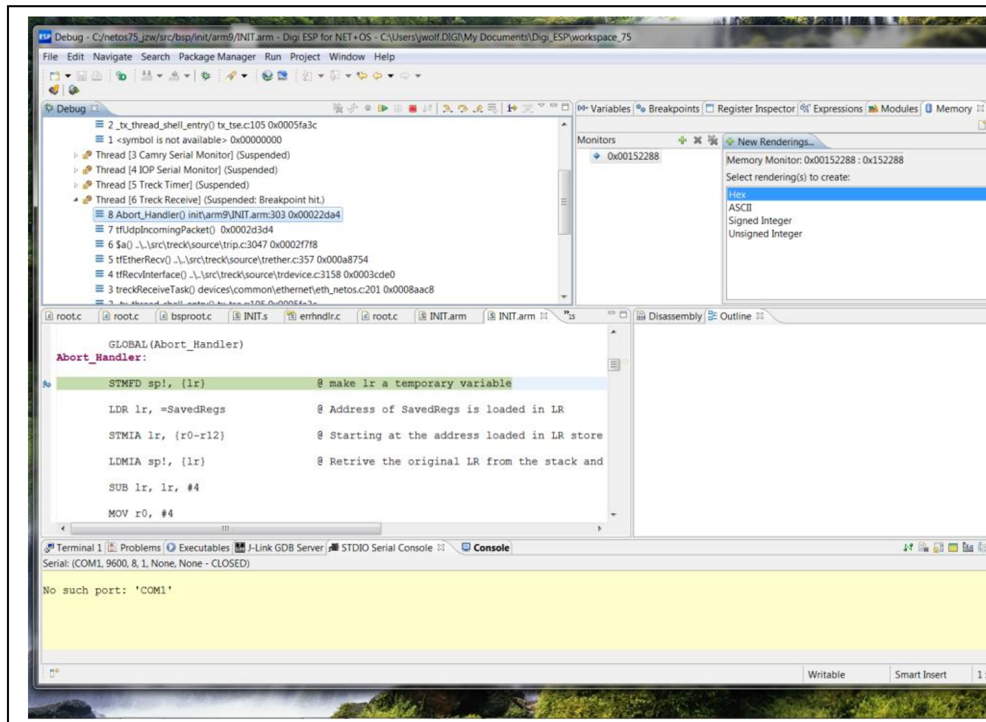
Let's try opening INIT.arm again.



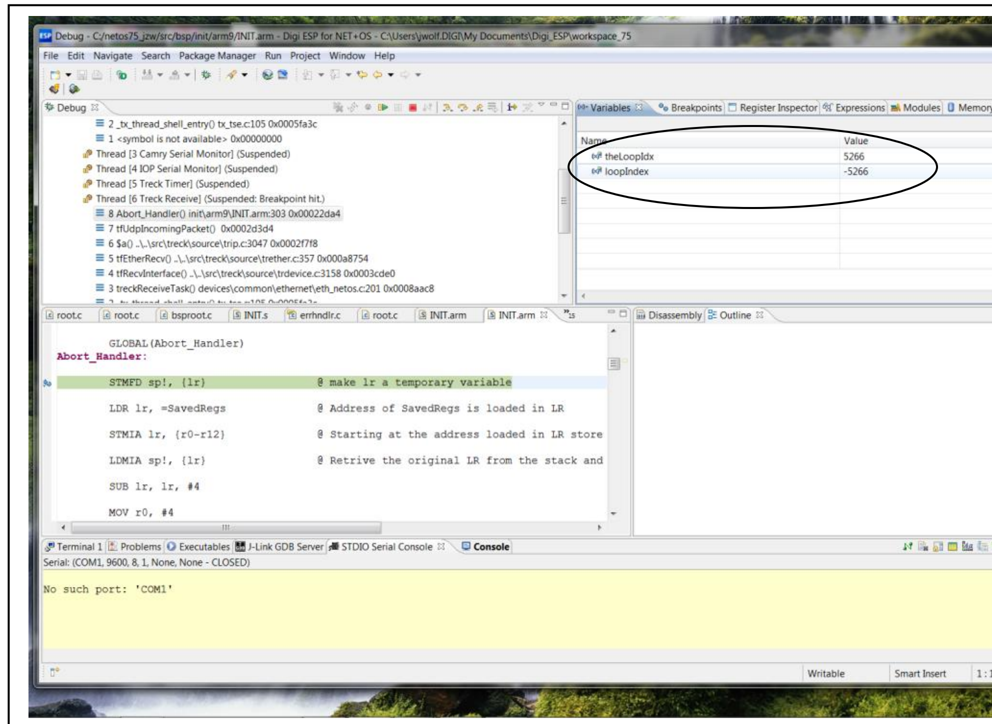
AHA! After adding *.arm to the list of known file types, file INIT.arm is now brought into the debugger. We should now be able to set the same breakpoints in INIT.arm as we did in INIT.s. So we should be able to do for an ARM9 module what we did for an ARM7 module.



Here we have set the breakpoint for Undefined_Handler for the ARM9 module. You will want to review the breakpoints we set for the ARM7 module and set the same ones for your ARM9 module. Then let the debugging begin.



Here we hit the Abort_Handler breakpoint from the Treck Receive thread. So on the ARM9 we were able to corrupt some other area of memory. Again, from here you'll need to go back to your application and try and find where you are corrupting memory and causing a crash.



You should still be able to view global variables and memory monitors when debugging your ARM9 application, just as we did for the ARM7-based application.

5 Example Application Explanation

As described earlier, the application that accompanies this paper is a fairly simple application that does something that we all strive not to do. That is overwrite or underwrite a buffer. Using a couple of macros included in the application, you can cause overflows and underflows to either global memory or local, malloced or not. You can use this to practice with the debugger and get used to evidence you can go looking for when debugging a crash.

Get example application here

6 Conclusion

No one wants to debug a memory corruption crash. Unfortunately, if you develop embedded systems long enough, you'll run into a defect that causes a crash. Once that happens, you need tools to trap the crash and then work backwards to identify the origin and a solution for the defect. We hope that through this paper, we have provided you with some additional tools for resolving your next crash-inducing defect.

7 Appendix

7.1 Glossary of terms

ESP – Embedded Software Productivity. Digi International's Integrated Development Environment (IDE). An eclipse-based tool for developing embedded software products.

File Widget – Window that is displayed, allowing a developer to select a file.

NET+OS – An embedded operating system and a development environment developed and sold by Digi International

Vector table – an area in the memory of a microprocessor that is reserved and contains functions that are called to handle exception situations. An exception situation is generally a crash.

7.2 Citations

Sloss, A., Symes, D., Write, C. ARM System Developer's Guide. Morgan Kaufmann Publishers, 2004.