



White paper

**ARM® Cortex™-M
system crash analysis**

COPYRIGHT NOTICE

© Copyright 2016 Atollic AB. All rights reserved. No part of this document may be reproduced or distributed without the prior written consent of Atollic AB.

TRADEMARK

Atollic, **Atollic TrueSTUDIO** and the Atollic logotype are trademarks or registered trademarks owned by Atollic. ECLIPSE™ is a registered trademark of the Eclipse foundation. All other product names are trademarks or registered trademarks of their respective owners.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment of Atollic AB. The information contained in this document is assumed to be accurate, but Atollic assumes no responsibility for any errors or omissions. In no event shall Atollic AB, its employees, its contractors, or the authors of this document be liable for any type of damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

DOCUMENT IDENTIFICATION

ASW-WPCA March 2013

REVISION

First version March 2014
Second version August 2016

Atollic AB

Science Park
Gjuterigatan 7
SE- 553 18 Jönköping
Sweden

+46 (0) 36 19 60 50

E-mail: sales@atollic.com

Web: www.atollic.com

Atollic Inc

241 Boston Post Road West,
First Floor,
Marlborough,
Massachusetts 01752
USA

+1 (973) 784 0047 (Voice)
+1 (877) 218 9117 (Toll Free)
+1 (973) 794 0075 (Fax)

E-mail: sales.usa@atollic.com

Web: www.atollic.com

Contents

Abstract	1
Introduction.....	2
Cortex-M fault handling	3
Exception Handlers	3
Processor status and error sources.....	4
The exception stack frame	4
The TrueSTUDIO Pro fault analyzer	6
Summary.....	10

Tables

Table 1 – Cortex-M fault status registers.....	4
--	---

ABSTRACT

Sometimes during development of embedded system applications, statement execution will force the microcontroller into a fault state, and cause the code to vector to one of the system exception handlers. Some typical reasons are division by zero, access of non-existent or illegal memory through rogue pointers, bus fault errors, or usage fault errors.

Finding the cause and location of such system crashes can be very difficult and time consuming. The ARM® Cortex™-M processor architecture has built-in capabilities that enable advanced debuggers to provide assistance in these situations. By using the methods and tools outlined in this white paper, the root cause and location of system crashes can be easily identified in seconds, rather than hours.

INTRODUCTION

Almost all software products contain errors. Modern 32-bit microcontrollers can contain hundreds of kilobytes, or even megabytes of memory, so even at low frequency error rates, it is possible that one of several lurking bugs can cause an embedded application to fail at some point. It can easily happen that some part of the software inadvertently tries to use the hardware resources of the microprocessor in an illegal way, thus forcing the processor to vector to one of the system exception handlers. It is often quite difficult to find the root cause of such problems, especially if they occur intermittently.

This white paper will discuss these types of system faults from the ARM® Cortex™-M point of view, and present tool solutions that enable embedded developers to quickly find the cause and location of such system crashes with a minimum of effort.

CORTEX-M FAULT HANDLING

When a Cortex-M processor encounters an error, it generates an internal fault and the application software execution is interrupted and execution continues in a fault exception handler. At this point, the developer is facing the following questions:

1. What characteristics of the code triggered this exception?
2. What code was being executed when the exception occurred?
3. What was the calling function of the code that caused the fault, e.g. where was the code supposed to have gone if the fault had not occurred?

Cortex-M processors contain hardware support to help answer these critical questions. However, the existence of these capabilities is often unappreciated. Even if they are, it is hard to use them with maximum effectiveness, unless the debugger implementation matches the built-in Cortex M capabilities. Information pertinent to hard faults is located deep within the ARM Cortex-M technical reference manuals. For the Cortex-M3, this document can be found at:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf

The debugger should help visualize information in a convenient, at-a-glance fashion to lead the developer to an immediate identification of the fault's cause. The alternative is a tough slog through the memory, determining the current stack pointer, and then counting backwards from it to extract relevant data, and then looking at individual bits of the CFSR register in the System Control Block, and then referring back to the relevant sections of the ARM Technical Reference manual. This is a tedious, time consuming and error prone process. ***Atollic TrueSTUDIO Pro offers an easier and faster way to get there!***

EXCEPTION HANDLERS

By default, Cortex-M processors activate the generic hard fault exception handler when an error forces the processor into a fault state. It is possible to enable 3 specific fault handlers, to handle specific categories of problems in a separate interrupt handler for each category. These are:

- Memory management faults

Memory management faults can for example occur when the software tries to execute code from non-executable memory regions, or violates the rules of the memory protection unit (MPU).

- Usage faults

Usage faults can occur when there is a program error, such as trying to execute an illegal instruction, accessing memory on misaligned boundaries, attempting to access a non-existent co-processor, division by zero or accidentally switching to the ARM instruction set mode on Thumb/Thumb2 mode-only processors.

- Bus faults

Bus faults can occur when there is an error on the bus that happens when accessing a peripheral module or memory, for example accessing an invalid memory region or byte-access to a word-only-access register. A bad pointer value for example, can cause this error.

If any of these three fault categories are enabled, a category specific interrupt handler will be invoked upon the error. If the category specific fault handlers are not enabled, the error will be escalated to a generic hard fault and be handled there. It is not absolutely necessary to enable the separate fault handlers, but some developers like the added detail of knowing the specific fault handler that is invoked by an error condition.

PROCESSOR STATUS AND ERROR SOURCES

When a system exception occurs, clues as to why this happened can be obtained by looking at the bits of the following processor status registers:

Register	Name
MMSR	Memory management fault status register
BFSR	Bus fault status register
UFSR	Usage fault status register
HFSR	Hard fault status register
DFSR	Debug fault status register
AFSR	Auxiliary fault status register

Table 1 – Cortex-M fault status registers

The meanings of individual bits are found in the ARM Technical Reference Guide. Bits in the above status registers are set according to the cause of the software error that forced the processor into a fault state. The process of using the debugger to ascertain bit status, then referring to the ARM Technical Reference Guide is a tedious and time consuming back-and-forth process that extends the time it takes to solve the problem. As we will see, the **Atollic TrueSTUDIO Pro** Fault Analyzer view makes this much easier.

THE EXCEPTION STACK FRAME

The Cortex-M processor uses a uniform way of saving the software context on the stack upon entering an interrupt or exception. The same mechanism is used for both system interrupts and fault exceptions.

When an exception (such as a system fault) occurs, a number of registers are pushed on the stack, enabling detailed analysis of the state of the CPU when the exception occurred.

The registers being saved on the stack are:

Position	Register
SP+00	R0
SP+04	R1
SP+08	R2
SP+12	R3
SP+16	R12
SP+20	LR
SP+24	PC
SP+28	xPSR

Table 2 – Registers saved on the stack when invoking an interrupt

The register values stored on the stack provides a snapshot of the processor context at the time the interrupt or exception happened:

- The working registers R0-R3 and R12 can often be useful to reconstruct the condition of the instruction that caused the fault, such as a division by zero.
- The program counter (PC) value can be used to find out which line of code drove the processor into the fault state.
- The link register (LR) often points to the parent function, and can be used to unwind the call stack at the time of the system crash.

THE TRUESTUDIO PRO FAULT ANALYZER

As the previous sections outline, it is possible, but very time consuming and error prone to manually work out the cause and location of a system crash that forces the Cortex-M processor into a fault state.

For this reason, **Atollic TrueSTUDIO Pro** introduces a new fault analysis feature in the debugger that aids developers in easily identifying and resolving hard-to-find system faults that occur when the CPU has been driven into a fault condition by the application software.

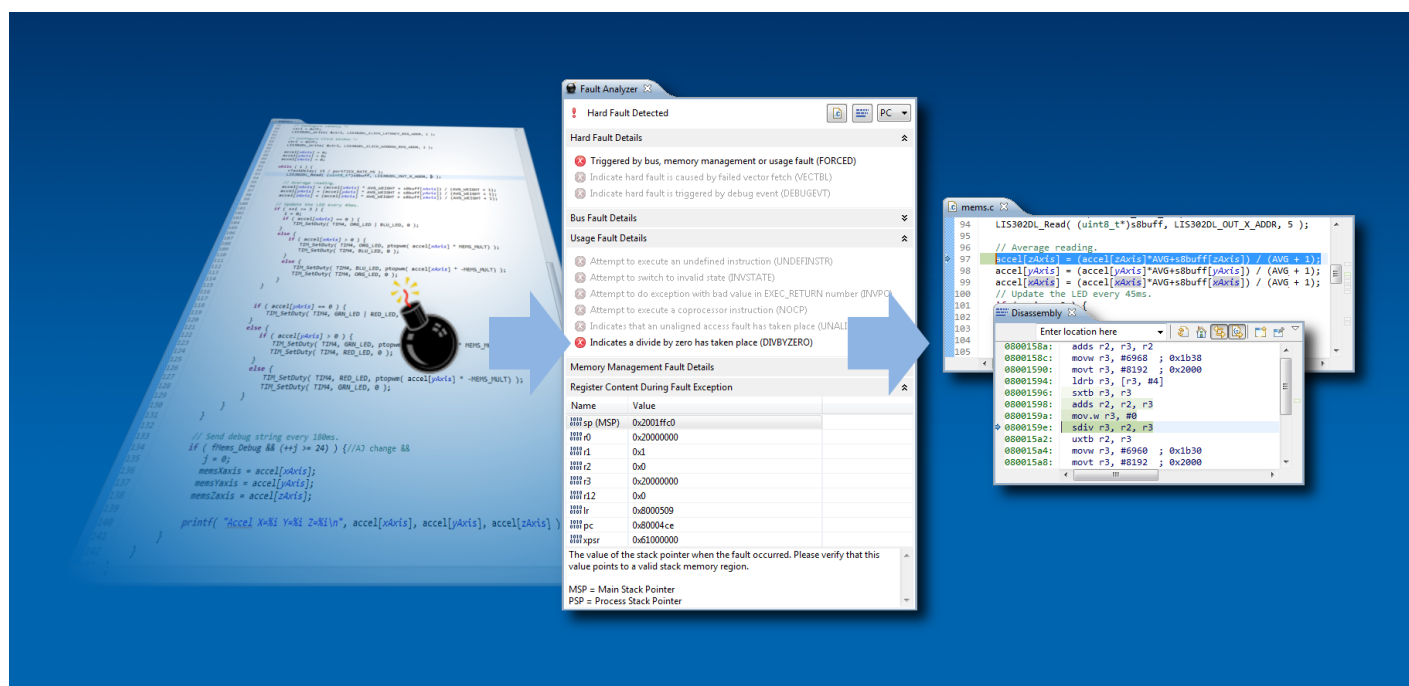


Figure 1 - The TrueSTUDIO fault analyzer find the cause and location of system crashes

The **TrueSTUDIO Pro** fault analyzer feature digs into those bits in the special function registers that are set by the Cortex-M nested vector interrupt controller when faults occur, and presents that information in a plain, easy to understand format. At a glance, the developer knows what happened, and as many of the details about the event as the Cortex-M core is able to tell.

The graphic below shows an application running under control of the **Atollic TrueSTUDIO Pro** debugger. At this point, execution is halted at the first line of `main()`, which is the typical starting point for debugging embedded applications.

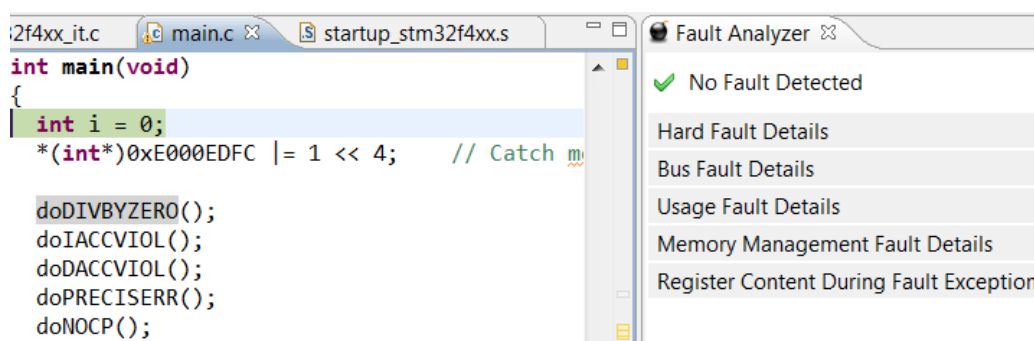


Figure 2 - The Fault Analyzer view at the start of a debug session

As we can see this is a contrived example designed to conveniently illustrate the Fault Analytical capabilities of **Atollic TrueSTUDIO Pro**. Obviously, no fault has been detected yet. Now, as the application proceeds, a fault will certainly develop. After clicking on the run toolbar button we see:

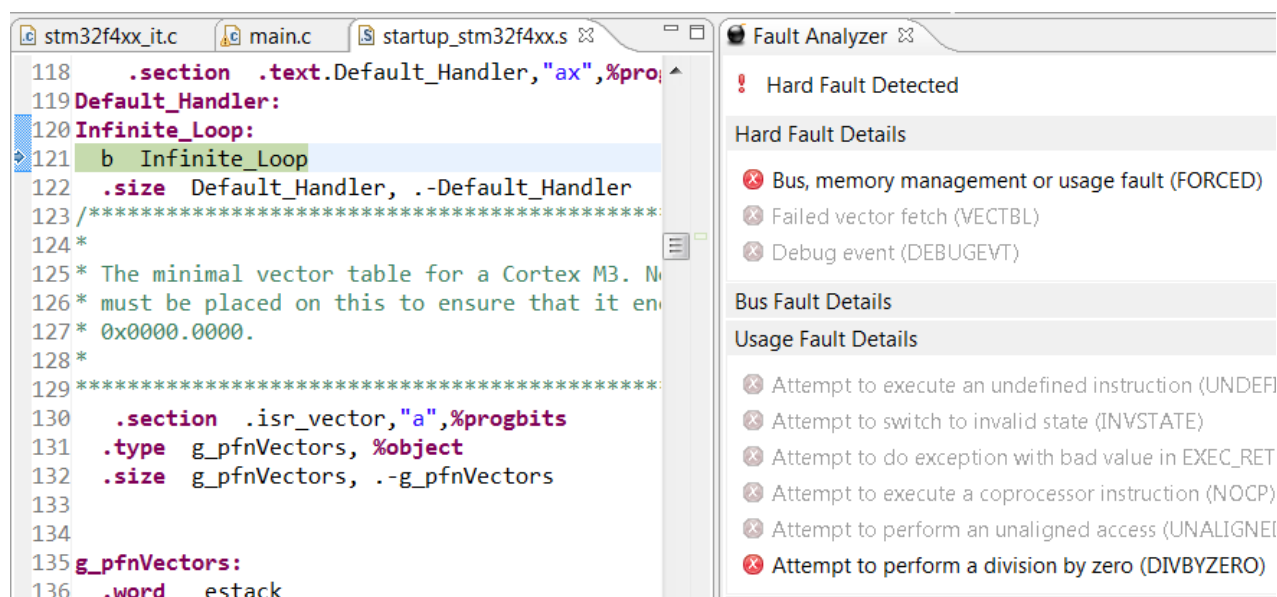


Figure 3 – The Fault Analyzer view showing the cause of processor fault

At this point, the debugger shows the processor is now in the hard fault handler implementation, typically entering into an infinite loop. Usually when a processor fault occurs, the tester notices that the application is unresponsive and halts the processor.

When this is observed, the user starts wondering about why the processor went into a fault state, and what code made it do so. This is where the Fault Analyzer view manifests its value. The screenshot of the Fault Analyzer above illustrates that an attempt to divide by zero was made. The developer now knows that somehow the denominator in a division expression illegally or inadvertently assumed a zero value. This immediately eliminates many possibilities of where there error may be, but can this be narrowed down even more?

The answer is YES!

Scroll down in the Fault Analyzer view and notice the saved PC register in the exception handler stack frame. This register holds the address of the code that brought the processor into the fault condition, in this case a division by zero being performed on the C source code line 71. Double-clicking on the PC value opens the offending source code line in the editor. Please note the PC register cannot be viewed in the CPU registers view, as it shows the code currently executing (the hard fault handler), not the address of the code that brought the processor into the fault condition.

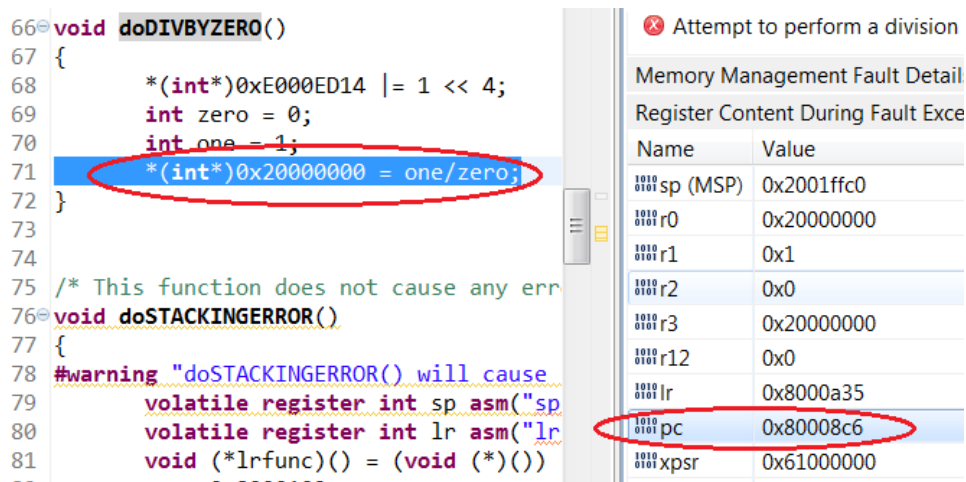


Figure 4 – The saved PC register points to the code that caused the processor fault

The function which called the code that caused the fault is also an important clue in the diagnostic strategy of the developer. This can be viewed easily by double-clicking on the link register (LR) directly above the program counter (PC) register.

But in fact, there is an even easier way to jump to the code that caused the processor into the fault condition. In the highlighted area in the screenshot below, the left toolbar button opens the offending C source code line in the editor; while the right toolbar button opens the offending assembler instruction in the disassembly view.

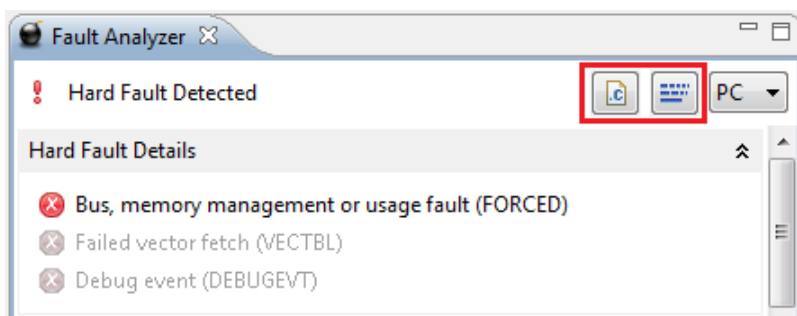


Figure 5 - Quick navigation to the code that forced the CPU into a fault state

By switching between PC or LR mode in the dropdown list-box shown in the screenshot below, the quick navigation toolbar buttons can be made to jump to either the offending code, or to the function that called the offending code, as appropriate.

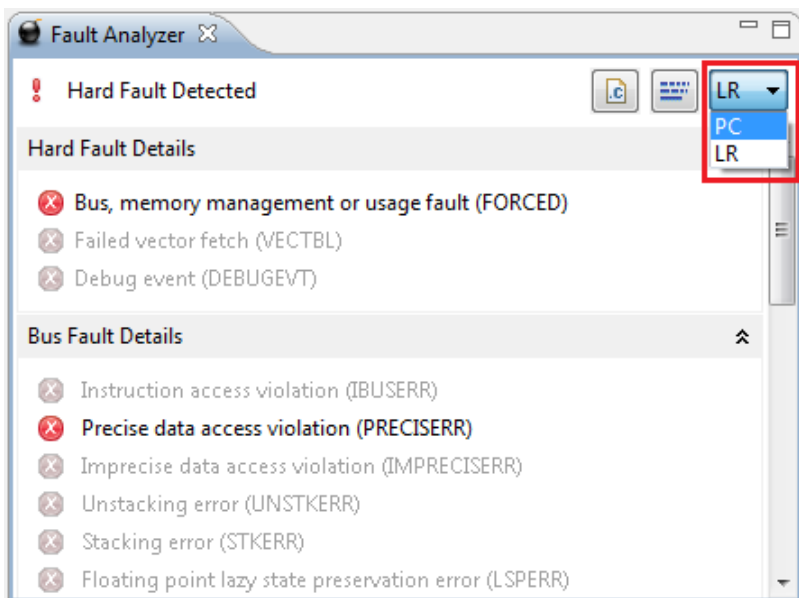


Figure 6 - Jump to offending code or to the function that called the offending code

The speed and efficiency with which this information can be obtained is orders of magnitude faster than with manual methods. More debugging power can be realized by combining this type of analysis with the capabilities of the Serial Wire Viewer real-time tracing. Using the Data Trace Timeline Graph can shed light on what values evolved in real time in various memory locations prior to the fault. The Exception Timeline Graph shows the sequence in time in which interrupts were firing prior to the fault. Information like this can be essential in fully understanding the events leading up to the fault.

SUMMARY

As complexity and size of embedded applications grow, so does the problem of debugging these kinds of software projects. The fault analyzer feature within **Atollic TrueSTUDIO Pro**'s debugger significantly aids embedded developers to quickly understand why and where the system has crashed. When this kind of problem can be solved quickly, the developer gets back working on the project sooner, spending less time “remembering” where he was before the fault occurred. This improves overall productivity, speeding development time, and cutting time to market. Tools can and should handle the drudge work for developers so that they may focus on what they do best; making their product better than their competitor's products.

Atollic provides a family of well integrated tools for professional embedded systems development and debugging. More information about Atollic and **Atollic TrueSTUDIO** is available here:

www.atollic.com

www.atollic.com/truestudio