How to use ARM's data-abort exception

By Roger Lynx
Engineering Consultant
and Embedded Systems Designer

The late Joseph Campbell, well-known scholar of comparative religion and mythology, once expressed his sentiments about computers from his perspective: "Computers are like Old Testament gods; lots of rules and no mercy." Add to his observations the familiar wisdom "where there are rules, there are also exceptions," and your Old Testament machine becomes more forgiving.

The data-abort exception (with the help of an exception handler) may be God's gift to ARM programmers. A data-abort exception is a response by a memory system to an invalid data access. The data-abort exception handler is a program that can inform the programmer where in his or her code this exception has occurred (after the application has crashed). The exception handler ought to handle the consequences of the aborted instruction gracefully, rather than forcing the processor to hang in an infinite loop. If you understand the fundamental rules of the ARM architecture and data-abort exception handling, you'll spend less time begging for mercy.

Not all ARM processors, however, come with these data-abort exception handlers. The exception handlers are usually not provided by the compiler, RTOS, or silicon vendors since it would necessitate quite a high level of integration. Ideally, a perfectly designed system doesn't need an exception handler. However, in a process of striving for perfection, engineers can come across moments when the processor showers them with a slew of undesired abort exceptions. These exceptions may originate in software, such as improper C-structures, or appear in code ported from a different processor architecture. Alternatively, exceptions may become signs of improper memory system design or manifestations of environmental effects on a marginal hardware design or on a specific component.

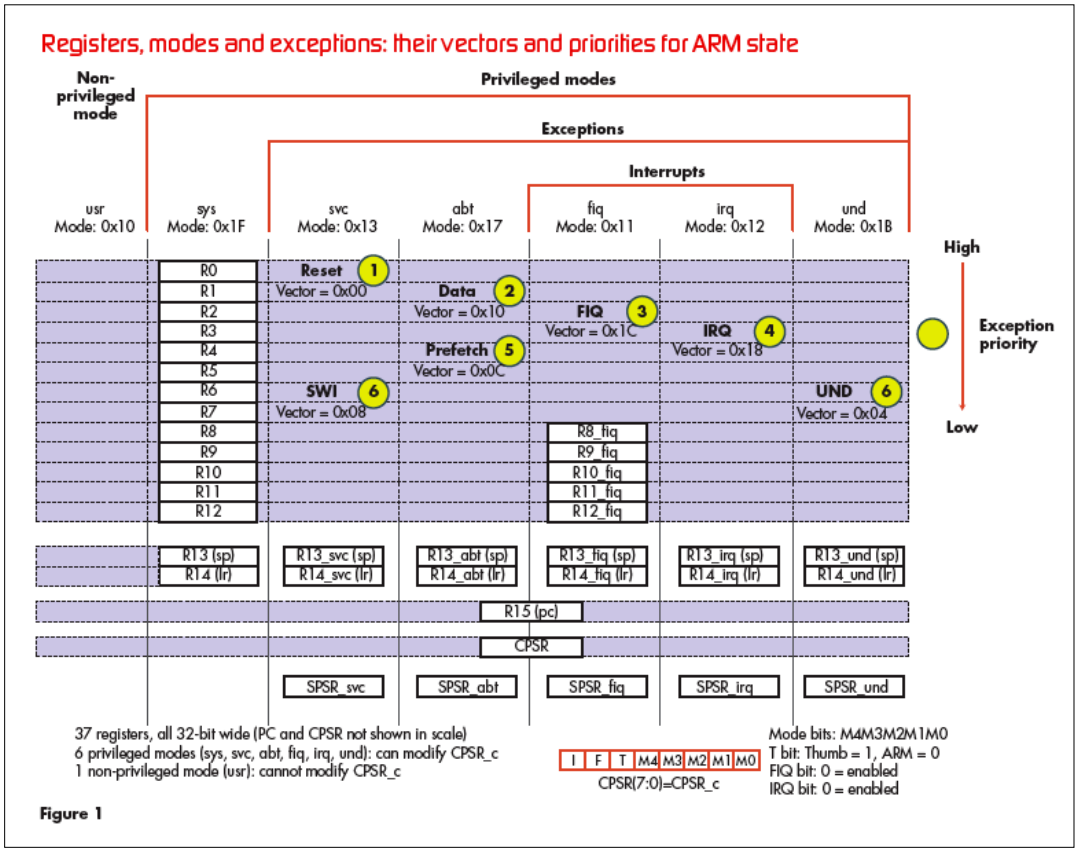

Figure 1

So I started to write my own exception handler. While writing and testing it, I stumbled on some undocumented features of LPC 2148, such as address aliasing of USB RAM and the location for corruptible Special Function Registers (SFR).

During my research many side issues arose from my empirical verifications of the researched material. When a result didn't support the theory, I wanted to know why. I hope that some of my findings help you save debugging time, since the data-abort exceptions and memory Reserved Areas are related yet not always accurately documented. You can also use this information as a springboard for developing other exception handlers for prefetch abort and exception handler for the LPC 2148 (and other ARM processors). The material presented here may help you develop your own exception handler for a similar ARM7TDMI processor and reduce your debugging time.

Off the beaten path

I was motivated to explore the topic of ARM exceptions after a prolonged debugging session when the processor would end up frequently in the default (dummy) data-abort exception handler, implemented by an infinite loop.[1] I wanted to know more about the state of affairs prior to this dead-end, glory-free execution of my application, and I suspected there was a way to construct a more efficient exception handler. Despite my large library of ARM literature and 6Mbps connection to the "information super-highway," I was dismayed to find very little coverage or documentation in the ARM literature related to this broad but important topic of exception handling.

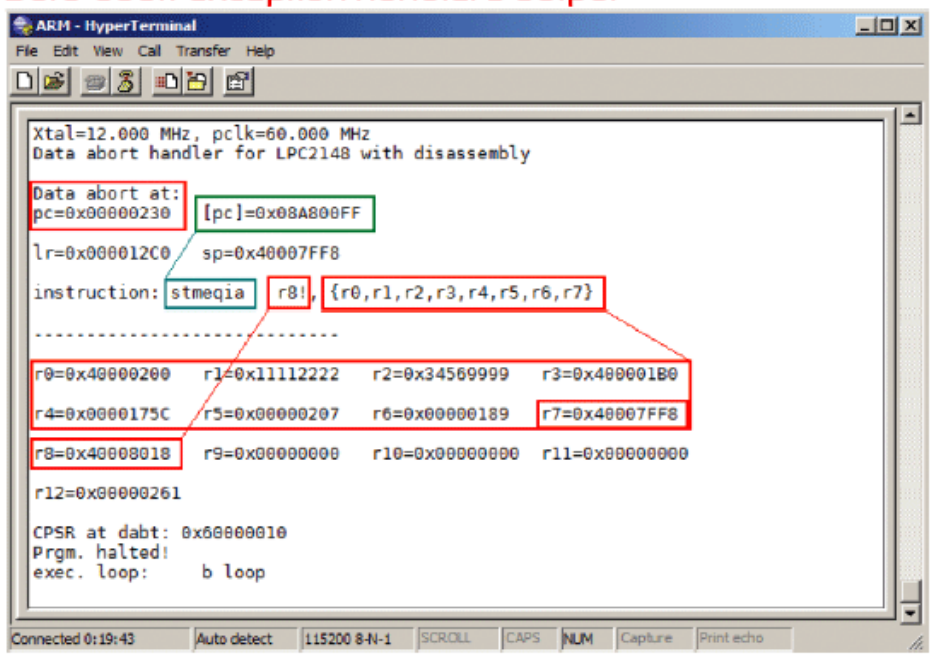

Figure 2

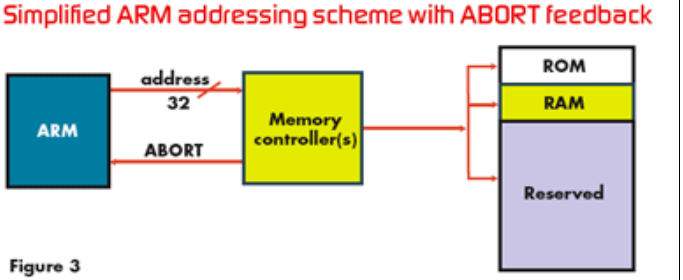

Figure 3

undefined instruction exceptions. The two remaining exceptions—the interrupt request (IRQ) and fast interrupt request (FIQ)—already have very good coverage in the literature.

Ultimately, as you'll see, I accomplished my goal to implement a data-abort exception handler that provides insight into the fault using a simple RS-232 connection to my PC instead of any high-level debugging tool (such as a debugger connected to JTAG port).

Exception handlers

Of the six exceptions that an ARM-based processor can raise, two abort exceptions signal that the current memory access cannot be successfully completed. The first one, data-abort exception, has the second-highest priority, just after reset, as shown in **Figure 1**. This exception conveys that the data access transaction was unsuccessful. The second is the prefetch-abort exception, which has the second-lowest priority, just one notch above the software interrupts. This abort is invoked when the processor is unable to fetch an instruction from memory. In this article, I concentrate on explaining why data-abort exceptions occur; the prefetch-abort exceptions are beyond our scope.

Many 32-bit embedded systems use a real-time operating system (RTOS), but not every RTOS comes with a data-abort exception handler, thus putting the responsibility for those aborts in the hands of the programmer. After all, the experts tell us "efficient handlers can dramatically improve system performance."[2]

Figure 2 shows a sneak-preview of the data-abort exception handler's output.

The exception handler is a simple UART driver that performs a register dump with the disassembled instruction that caused the data-abort exception. For example:

```
Processor aborted due to execution of instruction stmeqia located at address 0x0000 0230. Reason: a memory write was initiated at the top of SRAM (register r8=r7, before write) extended into Reserved Area range (register r8, after write).
```

ARMed with a brief history

During the past decades, the ARM architecture has undergone numerous revisions to the instruction set and hardware design. One of the many significant hardware changes was a move from the von Neumann architecture of ARM7 to the Harvard architecture starting with ARM9. Similarly, the Thumb-2 instruction set is the latest advancement and improvement of the first Thumb instruction set.

ARM processors are used in a system-on-chip (SoC) custom/proprietary designs (such as iPod, cell phones, hard disks) and in general purpose off-the-shelf MCUs. In either case, the ARM processor is in the center of a larger system. One of the synonyms for "system" is "complex." In a context of the data and prefetch aborts, dealing with both exceptions can be complex. Yet, they are both conditionally invoked based on a state of only one simple input to the ARM core--the ABORT signal asserted by the memory subsystem, shown in **Figure 3**. It forces the ARM core to respond to an event evaluated by the memory sub-system as a fault. In other words, what constitutes a failed memory transaction is decided not by the ARM core, but by the memory subsystem: its design and level of intelligence. The more the memory controller scrutinizes the ARM core's requested address, the easier it will be for the programmer to pinpoint a fault in the software (or hardware) that causes the abort exception. The ARM core's response to unsuccessfully completed memory transaction is uniform across all versions. However, the memory subsystem design's common point is only the ABORT output.

I've used the LPC 2148 as a primary test subject for this article. The LPC 2148 silicon design lacking any advanced memory controller features makes the handling of data-abort exceptions somewhat limited and challenging, as we'll see soon. Briefly, the key features of the LPC 2148 are:

- A simple three-stage instruction pipeline
- No cache, MMU or MPU
- 512KB internal flash
- 32KB + 8KB of internal SRAM, USB interface
- A convenient UART-based boot-loader

Let's begin by taking an inventory of instructions that are capable of raising a data-abort exception. These instructions

listed in **Table 1**.

For quick reference, I've included page numbers from David Seal's ARM Architectural Reference Manual (also known as the ARM ARM).[3] For completeness, I've listed two coprocessor-related instructions LDC/STC of Addressing Mode 5, even though the MCU used in development of this data-abort exception handler doesn't have one.

ARM instructions that can generate a data-abort exception

ARM state

LoaD	ARM ARM page reference	STore	ARM ARM page reference	Transfer type
ldm	A4-30/34	stm	A4-84/86	Multiple-register transfer
ldr	A4-37	str	A4-88	
ldrb	A4-40	strb	A4-90	
ldrbt	A4-42	strbt	A4-92	
ldrh	A4-44	strh	A4-94	Single register transfer
ldrsb	A4-46			
ldrsh	A4-48			
ldrt	A4-50	strt	A4-96	
		swp	A4-102	Swap word R<->Mem
		swpb	A4-104	Swap byte R<->Mem
ldc	A4-28	stc	A4-82	co-processor related instruction(*)

Thumb state

LoaD	ARM ARM page reference	STore	ARM ARM page reference	Transfer type
ldmia	A7-40	stmia	A7-84	Multiple-register transfer
pop	A7-75	push	A7-78	
ldr	A7-42/48	str	A7-86/90	
ldrb	A7-50/51	strb	A7-92/93	Single register transfer
ldrh	A7-52/54	strh	A7-94/96	
ldrsb	A7-56			
ldrsh	A7-57			

Table 1

How exceptions are raised

As I said earlier, a data-abort exception is a response from a memory system to an invalid memory access. For our discussion, it's useful to itemize the generic memory access as a "read" (load) and a "write" (store). Under certain circumstances, these instructions behave differently, as I'll explain later. Let's begin with a few simple definitions.

To write to memory means any possible form of store-type (ST) instruction. It may be a single-register store for a byte, a half-word (16 bits), or a word (32 bits); it may also be a multiple-register store or a special instruction "swap" that indivisibly (atomically) reads and writes. This definition applies to all addressing modes and to both states, ARM and Thumb. For a Thumb state, one more instruction, PUSH, expands this list.

Similarly, to read from memory implies that any possible form of load-type (LD) instruction is being executed, as applicable to ARM and Thumb states. For a Thumb state, one more instruction, POP, expands this list.

When can we expect a data-abort exception to be raised on the LPC 2148? These are the three general classes:

- When executing an unaligned memory access using instructions summarized in **Table 1**
- While performing a write to ROM (flash) space
- When accessing any of the Reserved Areas defined in the LPC 2148 User Manual[4]

Unaligned memory access

Actually, this was a trick question. In general, the ARM architecture does not support unaligned accesses. However, due to its design, the LPC 2148 won't inform you that this event took place; rather it will produce an output which I'll analyze in the following section.

An unaligned RAM data access is a read or write that involves an address that's not a multiple of the data size. For a word-aligned memory access, the last two bits of an address have to be zero. Thus, the address's last nibble will be multiples of four: 0x0, 0x4, 0x8, and 0xC. For a half-word-aligned memory access, the last bit of an address must be zero. The address's last nibble will be multiples of two: 0x0, 0x2, 0x4, 0x6, and up to 0xE. Accessing a byte is trivial.

What are the outcomes of an unaligned RAM memory access and when do they occur?

Whether the processor runs in the ARM state as it fetches 32-bit instructions or in the Thumb state fetching 16-bit instructions, the memory controller may allow the processor to access the memory using:

- A word-aligned address (ldr/str) or non-word-aligned with byte rotation[5] (ldr)
- A half-word aligned address (ldrh/strh) or non-half-word aligned address yielding unpredictable[6] results
- A byte address (ldrb/strb)

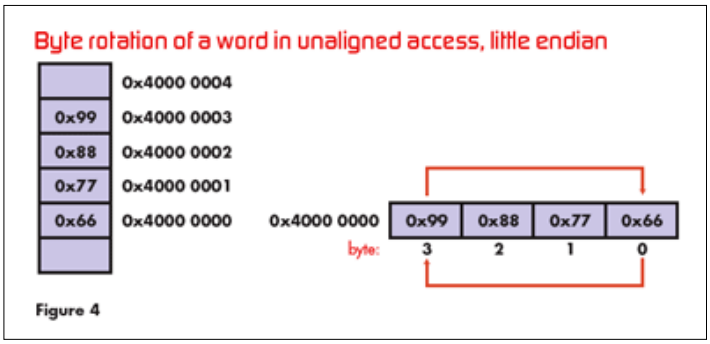

Figure 4

Alternatively, the memory controller may abort all unaligned accesses. The effects of word and half-word unaligned access will become clearer from an actual example, which I'll take you through step by step.

A word on storing words: LPC 2148 is a little-endian[7] MCU, which means that the least significant byte (LSB) of a word is stored at the lowest memory address, whereas the most significant byte (MSB) is stored at the highest memory address of a word.

Case 1, word access

Read a word from 0x4000 000n to register r1, where n is the value of the last two bits of the accessed memory address. **Table 2** shows you the results using instruction ldr r1, [r0].

You can see in **Table 2** that it's only in the first case, when n=0, that we perform word-aligned access. In the remaining three cases, n=1, 2, 3, the silicon-vendor-specific implementation of the memory controller will decide whether the register is loaded with byte-rotated value (LPC2000), or the memory controller detects the nonaligned address and aborts the read (AT-91SAM7S). The byte rotation of the instruction ldr as defined in ARM ARM is depicted in **Figure 4** and **Table 2**.[8]

For memory writes, such as str r1, [r0] the str instruction will produce a word-aligned address by masking the last two bits with 0xFFFFFFFE. This behavior, as I've described, applies to all nine addressing modes for loading and storing a word in Addressing Mode 2.

For instruction ldm (multiple load), the last two bits of the ad-

dress are ignored, so no rotation of bytes occurs, unlike for the instruction ldr. For instruction stm (multiple store), the last two bits are ignored for unaligned memory access. These are the only two instruction types of the Addressing Mode 4.

No data-abort exceptions are generated by LPC 2148 in any of these cases.

Case 2, half-word access

Read a half-word from 0x4000 000n to register r1, where n is the value of the last two bits of the accessed memory address. **Table 3** shows you the results using instruction ldrh r1, [r0]. This byte rotation of the instruction ldrh, defined by ARM ARM as "unpredictable," is depicted in **Table 3**.[9] For memory writes, using strh r1, [r0] the strh instruction will produce identical results as described for ldrh. This behavior, as described in **Table 3**, applies to all six addressing modes for the loading and storing of a half-word in Addressing Mode 3.

To briefly summarize, word-unaligned access to defined memory space on the LPC 2000 results in byte-rotated read, as shown and referenced in **Table 2**. Half-word unaligned access produces an unpredictable result, shown in **Table 3**. There is no intervention of the memory subsystem to abort an unaligned data access with a raised exception.

For ARM implementations with more advanced memory systems than the one used on the LPC 2000, unaligned access may yield a data-abort exception.

A write to ROM (flash) space

This scenario is perhaps too obvious to discuss, yet just such a situation arose when I was debugging my data-abort code. I faced a stray pointer that insisted on writing to ROM. The system defended itself by generating data-abort exceptions. I'm not talking about the programming of the flash, but the runtime misdirected write operation. This stray pointer is the only one of the three discussed in this article that will produce data-abort exception most reliably--100% coverage of the flash memory space.

Word access using instruction ldr r1, [r0]

[0x40000000]	r0	n r0 [1:0]	r1	operation
0x99887766	0x4000 0000	00	**0x99887766**	ret. val=mem[adr,4] = word aligned
	0x4000 0001	01	0x66998877	ret. val=mem[adr,4] ror 8
	0x4000 0002	10	0x77669988	ret. val=mem[adr,4] ror 16
	0x4000 0003	11	0x88776699	ret. val=mem[adr,4] ror 24

Table 2

Half-word access using instruction ldrh r1, [r0]

[0x40000000]	r0	n r0 [1:0]	r1	operation
0x99887766	0x4000 0000	00	0x00007766	ret. val=mem[0] = half-word aligned
	0x4000 0001	01	0x66000077	unpredictable
	0x4000 0002	10	0x00009988	ret. val=mem[2] = half-word aligned
	0x4000 0003	11	0x88000099	unpredictable

Table 3

Summary of read and write operations to Reserved Areas

Reserved area	Address range	Range size [words] (bytes)	Number of trapped data abort exceptions; read [words]	Number of trapped data abort exceptions; write [words]	Delta = range size – number of trapped exceptions [words] (bytes)
1 (mem. top)	0xDFFF FFFF 0x8000 0000	0x1800 0000 (0x6000 0000)	0x1800 0000	0x1800 0000	0
2*	0x7FFF CFFF 0x7FD0 2000	0x000B EC00 (0x002F B000)	0x0007 F400	0x0007 F400	**0x3 F800** (0xFE000)
3	0x7FCF FFFF 0x4000 8000	0x0FF3 E000 (0x3FCF8000)	0x0FF3 E000	0x0FF3 E000	0
4 (mem. bottom)	0x3FFF FFFF 0x0008 0000	0x0FFE 0000 (0x3FF8 0000)	0x0FFD E000	See note**	**0x2000** (0x8000)

* Tested with USB RAM disabled: PCONP[31]=0

** Test program crashed in an unrecoverable way

Table 4

Memory access to reserved areas

ARM is a 32-bit architecture designed with thirty-seven 32-bit registers; its 32-bit program counter (pc) is capable addressing a memory space of 2^32-1 = 4,294,967,295 bytes (4GB). In the LPC 2000 family this linear memory space is divided into four distinct memory regions:

- ROM (flash) for the nonvolatile code storage
- SRAM for volatile data
- VLSI Peripheral Bus (VPB) and Advanced High-Performance Bus (AHB) peripherals
- Four Reserved Areas

Examples of VPB peripherals are UART0/1, I2C, SPI, and timers. An example of an AHB peripheral is the Vector Interrupt Controller (VIC) and an 8KB USB SRAM.

The sum of all four Reserved Areas is 3,757,518,848 bytes (0xDFF7 3000), representing 87% of the 4GB total memory space. The probability of a stray pointer hitting this area is reasonably high. But if we place our data-abort exception handler in its way, we'll debug our program quicker, right? Actually, it's not so simple.

When I finished my data-abort exception handler, I needed to test it. "There is a huge test area," I thought, "It will go fast." Two days later I realized my code was solid, but I ran into some undocumented features of the chip. Have you noticed how a five-minute software upgrade easily turns into many hours (or days) of tweaking?

All of a sudden it's déjà vécu all over again.[10] My original test approach to the data-abort exception handler behavior was to intercept the exception, count it, and return to the execution of the next instruction (more on this later).

I set my pointers to read and write the Reserved Areas, expecting to get an exception for every word that accessed the "forbidden" memory region. But it did not happen this way; see the test data in **Table 4** and the memory map shown in **Figure 5**.

For Reserved Areas #1 and #3, there were no surprises; I received an exception for every read and write to those Reserved Areas. Reserved Area #2 and #4, however, produced what I first called "false positive" exceptions--they didn't occur when they should have.

When I quantified the results, I stepped back from the problem and modified the memory map, as shown in **Figure 5**. I needed to investigate this further.

Hic sunt leones

Now I was entering uncharted territory.[11] A few days later, after I finished my memory map and code testing, I came across

a document published by Philips.[12] A figure in the document marked the memory area between 0x3FFF 8000 to 0x3FFF FFFF as "Special Registers," which I've labeled on my memory map (**Figure 5**) as HSL #4a. This area accounted for the unbalanced count in data exception generated in reads. But how about writes to this area? There were problems encountered when a constant 0x11223344 was written to any address below 0x4000 0000.

The type of problems encountered was that the debugger's memory window (with view of processor's stack RAM) got cleared, and the data-abort exception handler experienced data-abort exception.

It wasn't my goal to find out why my data-abort exception handler couldn't handle this small segment of Reserved Area, but I can attest that debugging this type of problem can be a real challenge.

More lions

The LPC 2148 has a dedicated USB controller with a DMA transfer and a dedicated 8KB RAM. This RAM can also be used as a generic storage area. However, this memory is inaccessible unless the USB controller is also enabled by setting PCONP [31] = 1. The reason for this behavior is that this RAM is a part of the USB controller on the AMBA bus, unlike the standard system SRAM, which resides on the ARM7 local bus.

After setting this bit, the SRAM is enabled and filled with random values. During my attempt to initialize this region, I noticed that when the value is being written to the address 0x7FD0 0000, it is also being "echoed" at address 0x7FD0 2000, 0x7FD0 4000, 0x7FD0 6000, and so forth. Apparently, some address aliasing is taking place. If we pull out the calculator and perform the following math, the result makes sense:

```
0x7FE0 0000 (top of
HSL #2a) -
0x7FD0 0000 (bottom
of USB SRAM) = 0x0001
0000.
```

But the size of the USB RAM is 8KB (0x2000), so we divide 0x10000 by this value. Voila! There are 0x80 (128) segments of 8KB each in the bottom of the Reserved Area #2; only one of them is filled with actual RAM. This explains the peculiar behavior of my data-abort scanning procedure for reporting "false positives" during the test when the SRAM was disabled.

Review

Let's review our findings so far:

- Unaligned memory access doesn't generate data-abort exception
- A write to ROM does generate data-abort exception
- A read or write to the entire memory segment defined as Reserved Area #1 and #3 will generate data-abort exceptions, whereas a read or write to specific subsets of the Reserved Area #2 and #4 will not. Those subsets are defined in **Figure 5**.

Effects of data-abort exception

To examine the effects of data-abort exceptions, let's look at some basic background material. There are two models for data abort:

- Base restored abort model
- Base updated abort model

Both models are implementation-dependent, so you should refer to your chip's manuals for details. What's the difference between the two? The ARM7 in general uses the base updated abort model, which ARM ARM defines as "If a Data Abort occurs in an instruction, which specifies base register writeback, the base register writeback still occurs."[13] On the other hand, ARM9 uses the base restored abort model, defined by ARM ARM as "If a Data Abort occurs in an instruction, which specifies base register writeback, the value in the base register is unchanged." Notice how quickly we're getting deeper into apparently unrelated issues. What is a base register writeback? Consider the following instruction; for a moment we step outside of a data-abort exception context:

```
ldr r0, [r1, #4]!
```

This instruction (auto-indexing) loads the register r0 with the content of the memory located at the address stored in r1, which is automatically incremented by four, after its execution. For example, if this instruction were placed in a loop, the effect would be an automatic stepping through a lookup table with a starting address pointed to by a value in r1. This feature is built into the ARM hardware; the exclamation mark in this example simply means base register writeback update. Let's return to data-abort exception effects and rules. There are two cases to be considered: first, how the data abort affects the memory content (write); and second, how the data abort affects the content of registers (read). Furthermore, we can analyze these two cases for a single register transfer and a multiple register transfer. David Seal's treatment of this topic is very precise.[14] I won't attempt to re-interpret the rules; rather you can study these rules on your own. However, we'll examine two cases--multiple writes and a single read--to gain some background for my implementation of the data-abort exception handler for the LPC 2148. Let's consider the following instruction for storing multiple registers involving the memory addresses region of boundary #4:

```
stmda r8!, {r0-r7}
//store multiple
decrement
after and update;
with '!'
```

If we let r8=0x7FD0 0010 (pointing to the bottom of USB RAM range), this instruction would write out to memory only five registers[15] (r7 to r3), followed

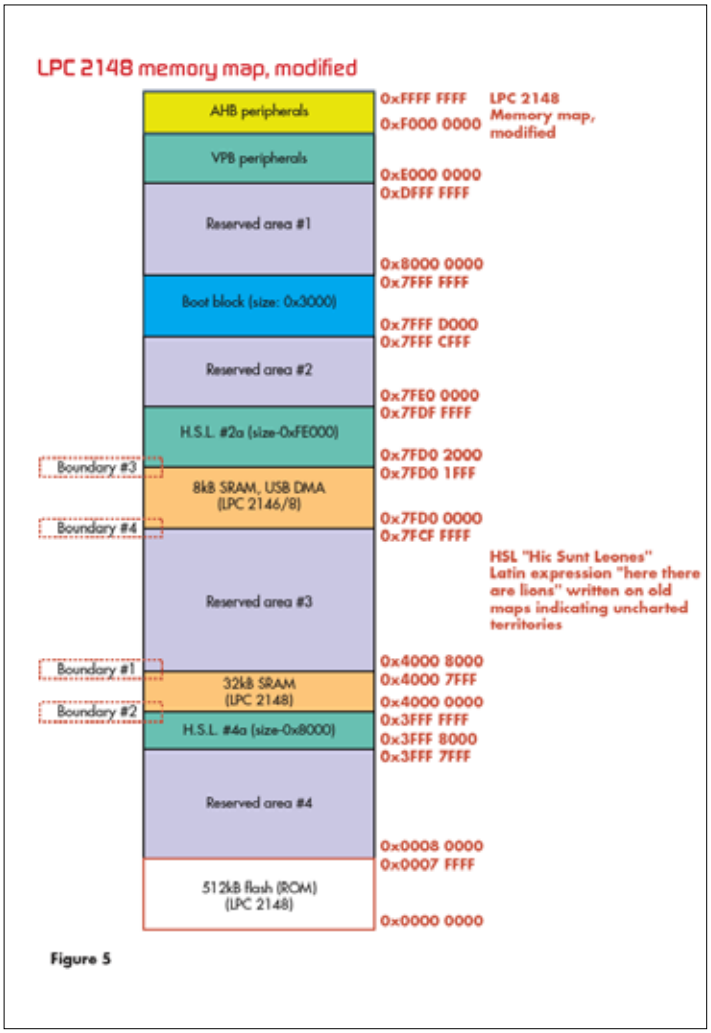

Figure 5

by data-abort exception. However, the register r8 would be updated though all eight registers were transferred to memory, namely the contents of r8 would be updated to:

```
0x7FD0 0010 - (8 reg
isters*4 bytes=
0x20) = 0x7FCF FFF0
```

as **Figure 6** depicts.

This instruction should cause a data abort, and it does. Register r8 was predictably updated to 0x7FCF FFF0 as if the last write to the memory address at 0x7FCF FFF4 was successful (the "decrement after" mode caused the register r8 to be updated to the next word address after 0x7FCF FFF4, which is 0x7FCF FFF0). In the example just cited, I've intentionally selected the most important of the six rules for the data-abort effects since this rule specifically addresses the unique property of ARM7TDMI-S. Later, I'll analyze the specifics of this property and its consequences in the remaining three boundary cases in the context of the LPC 2148 implementation. I decided not to process the writeback update condition because it would have an impact on the code size of the data-abort exception handler. Now, let's look at the outcome when the writeback is not specified (absence of an exclamation mark), as shown here:

```
stmda r8, {r0-r7}
//store multiple
decrement
after without an up
date
```

Using identical initial conditions as before, the value stored in r8 would be unchanged after the data-abort exception; it would still be 0x7FD0 0010. For a case involving memory read with a destination a single general-purpose register, the value of that register is unchanged; for example, consider the following code segment:

```
Initially
r0=0x00000101
ldr r2, =0x7FE00000
;address in Reserved
Area #2
ldr r0, [r2]
;data abort excep
tion raised
After data abort
exception raised,
r0=0x00000101
```

The initial value in r0 was 0x00000101. Execution of these two lines would not affect the register's content, but the data-abort exception would be raised since we attempted to load r0 from an address in Reserved Area #2.

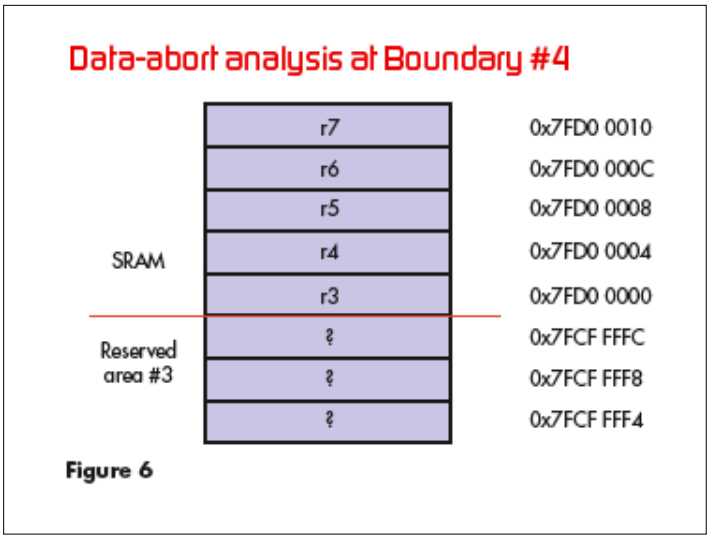

Figure 6

Non-unified theory

There is a schism even in the ARM literature about what ought to be done by the data-abort exception handler. One view states the exception handler should "fix" the error and return to re-execute the instruction that caused the exception.[16] The other says that the exception handler should report the error and stop further program execution.[17] This divide is understandable because so many variations of hardware are available; there is no silver bullet. The final decision rests with the system programmer and depends the context of the problem at hand.

I agree with the second view that emphasizes reporting the error and stopping further program execution. However, I do process the content of all 16 registers, the opcode of the instruction causing the exception, and the disassembled opcode.

The bulk of my data-abort exception handler's code uses the exception's native 32-bit ARM state. After all, it is only 104 bytes in code size. so there's no incentive to try to compress it using Thumb. The second reason for using the ARM state for most of the data-abort exception processing is that one has access to MSR and MRS instructions capable of modifying control bits of CPSR necessary for switching in and out of the data-abort mode.

The state (and mode) switching is performed for the purpose of obtaining the content of the stack pointer and link register (r13=sp and r14=lr, respectively) at the time of exception; the contents of registers r0 to r12 are transparent to the data-abort mode.

A myth is being perpetuated by some people working with ARM processors that there are only eight registers when the processor is in a Thumb state. The source of this myth is perhaps rooted in the original ARM block diagram for Thumb state showing only eight registers in any mode.

My first question when I saw this diagram was: where did the remaining registers go? They are there; they didn't vanish. Actually, their names are Lo; r0-r7 and Hi; r8-r15.[18] However, only three instructions can operate on those Hi registers: MOV, ADD, and CMP. Perhaps for the compiler designer, it might be easier not to use those Hi registers, but for someone programming in assembly, any available register can be a great gift.

The data-abort exception handler, therefore, processes and displays the content of all 16 registers, regardless of whether the data-abort exception was raised in an ARM or the Thumb state.

Insert code here

I wanted to write an exception handler that could process data aborts not only from either state but also from any mode (sys, usr, swi, irq, and so on). Furthermore, I wanted to write a modular exception handler, independent of an output device driver. At the end of the exception handler routine, the processor would switch to Thumb state and an additional umbrella-like function would be called that might contain additional processing; in my implementation, it would be a call to the disassembler and to the UART output driver. For further processing of the crash data, I chose the Thumb state based on the merit of the code density; the processing speed is irrelevant because the application had already crashed. The data-abort exception handler fills the global array of 72 bytes with the contents of all 16 registers at the time of exception plus the mode of the processor (in last five bits of cpsr) as well as the offending instruction's opcode. The data-abort exception handler produced the output I described earlier, based on two facts:

- First, when the data-abort exception is raised, the processor stores the address of the aborted instruction plus eight bytes (pipeline offset) in lr, and then fetches the address of the data-abort exception handler and executes its code. Thus the address of the offending instruction can be calculated by simple subtraction:

```
adr_dabt_instr
= lr - 8
```

- Secondly, the mechanism of reading 32-bit constants from the literal pool guarantees that reading the code memory produces, obviously, no exceptions. Therefore, once the address of the aborted instruction is known, we can

obtain the opcode of this instruction using the "load register-offset" instruction. This opcode is then disassembled and the result is presented to the output driver.

Based on the opcode, the disassembler provides the decoded format of any load-store instruction. All nineteen addressing modes of ARM state load-store instructions defined by Addressing Mode 2, 3, and 4 are decoded. For the Thumb state, all four formats of load-store instructions are also decoded. The size of this Thumb-compiled disassembler is 4.3KB.

The disassembler is almost complete, but because the data processing operands of Addressing Mode 1 (ex: MoVe, ADD and SUBtract) do not generate data-abort exceptions they were omitted.

The disassembler outputs a string with a maximum length of 62 characters that contains the disassembled instruction; the longest possible string may be in a form shown in **Listing 1**.

In the following section, we dissect the logic of the data-abort exception handler (filename data_abort.s79). After raising the exception, the processor fetches the address of the data-abort exception handler from the vector table, namely, the vector at address 0x10; and from this address, the processor then executes the data-abort exception handler's code.

For the sake of simplicity, I chose to implement the data-abort exception handler without concern for fast interrupt request (FIQ) processing. Even though the data-abort exception is the second highest one, as mentioned earlier, it can be interrupted by FIQ. It would be the responsibility of the FIQ handler to preserve the context of the data-abort exception handler.

The data-abort exception handler's prologue starts with instructions on Lines 2, 4, and 5 of **Listing 2**. First we push registers r0 through r12 on stack together with link register. Even though the ARM-Thumb Procedure Call Standard (ATPCS) requires preserving only registers r4 through r11, saving a full context simplifies this system code (in contrast to user code where the exception originated).

In the following step we load a stack pointer with the address offset to the sixth element of the array. Finally, we save the contents of all pertinent registers to the array dabort_dump. Thus, we saved the first subset of the full dataset.

Now we proceed to build the second dataset starting with the address calculation of the offending instruction. The result is stored in r0 by instruction shown on Line 6 of **Listing 2**. This step is inherently beautiful because regardless of whether the processor is in ARM or Thumb state, it will always save pc+8 in link register (lr).[19] What we need in this data-abort exception handler is the address of the instruction that raised the exception; this address is pc = lr - 8.

Comment: don't look back

If we wanted to return from the data-abort mode to continue executing the subsequent user code instruction, we would subtract four (instead of eight) from the link register to get the ARM state's return address. Thus, ARM state return address is ARM_return = pc = lr - 4. For the Thumb state, the return address calculation would also need to subtract two to compensate for the fact that the program counter is incrementing in multiples of two. Therefore, the Thumb state return address is equal to: Thumb_return = pc = lr - 6, or Thumb_return = ARM_return - 2.

The motivation for my brief math lesson above is cautionary. The current LPC 2148 silicon design has an anomaly, which is described by the erratum for each LPC MCU individually, globally referenced as "Core.1"[20]

Here's a brief description of the anomaly. When the processor is in the Thumb state, the data-abort exception sequencing by the hardware will save to link register value pc+6 instead of pc+8 (other conditions also have to be met). This means that implementing the correct return address model (as described) will bring the processor back to execution of the same "faulty" instruction instead of the subsequent one. Concerning the Core.1 erratum: my tests of its description indicate that it's not quite correct.

Using my workaround, I obtained results that led me to the summary in **Table 4**, to the redrawn memory map shown in **Figure 5**, and to my conclusions as stated at the end of this article. An alternative, but Core.1 erratum affected data-abort exception handler is also available (the filename is data_abort_alt.s79). A suggestion for anyone thinking about implementing the

Listing 1 Longest possible string of disassembled instruction output by a disassembler

```
ldmeqia___sp!,_{r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,lr}^;     where '_' denotes a space
```

Listing 2 Code for a data-abort exception handler

```
1: T_bit    EQU (0x01<<5)           ; T Bit def (0x20) in psr
2:     stmfd sp!, {r0-r12, lr}      ; save superset of ATPCS scheme (r4-r11)
3:                                  ; #define _dabt as _abt (no prefetch abort here)
4:     ldr sp, =(dabort_dump+5*4)   ; set sp_abt to data array w/offset (restore later)
5:     stmia sp, {r0-r12}           ; save 1st dataset in r0-r12 registers to array
6:     sub r0, lr, #8               ; calculate pc value of dabort instr: r0 = lr-8
7:     mrs r5, cpsr                 ; save current mode to r5 for mode switching
8:     mrs r6, spsr                 ; spsr_abt = CPSR of dabt originating mode:
9:                                  ; save to r6 for mode switching
10:    mov r2, r6                   ; building 2nd dataset: r2 = CPSR (of exception)
11:    tst r6, #0xF                 ; test for the mode which raised exception:
12:    orreq r6, r6, #0xF           ; if EQ => change mode usr->sys; else do nothing
13:    bic r7, r6, #T_bit           ; go to forced ARM state via r7
14:    msr cpsr_c, r7               ; switch out from mode 0x17 to
15:    mov r3, lr                   ; dabt generating mode and state
16:    mov r4, sp                   ; get lr (= r3) and sp (= r4)
17:    msr cpsr_c, r5               ; switch back to mode 0x17
18:    tst r6, #T_bit               ; Thumb state raised exception?
19:    ldrneh r1, [r0]              ; yes T-state,load 16-bit instr: r1 = [pc](dabt)
20:    ldreq r1, [r0]               ; no A-state,load 32-bit instr: r1 = [pc](dabt)
21:    ldr sp, =dabort_dump         ; reset sp to array's starting adr.
22:                                 ; and save the 2nd dataset from r0 to r4
23:    stmia sp, {r0-r4}            ; clean up & restoration follows:
24:                                 ; restored full context, sp first
25:    ldr sp,=SFE(ABT_STACK)       ; sp uses ABT_STACK definition from linker file
26:    ldmdb sp, {r0-r12, lr}       ; r0-r12 and lr using restored stack pointer
27:                                 ; ARM processing ends here
28:    ldr r0, =SwTh                ; switch to Thumb & execute Thumb compiled dabt.c
29:    bx r0                        ; w/output drivers (UART, SPI, I2C, USB...)
30:    CODE16                       ; dabt.c contains call to dissembler
31: SwTh:                           ; Note:
32:     bl dabt                     ; dabt.c must compiled as Thumb! No veneer here
33:                                 ; after return from dabt go to infinite loop
34: inf_loop:                       ;
35:     mov r0, r0                  ; "He is dead, Jim" --Doctor Leonard "Bones' McCoy
36:     b inf_loop                  ;
```

return from data-abort exception handler back to application for the LPC 2148's current silicon: have mercy on yourself and forsake that idea.

Two instructions, on Lines 8 and 10, show how we preserve the saved processor status register (spsr) of the current data-abort mode 0x17. The spsr is the current processor status register (cpsr) of the mode from which the data-abort exception originated. One copy of spsr is saved to r2, the second one becomes our working copy, stored in r6.

We've saved all the registers that we could from the data-abort mode. We now have to set up the registers for mode or state-switching cases (or both cases as needed). The only reason for switching to the mode or state that generated the exception is to obtain the values of its stack pointer and link register.

Mode test

All ARM privileged modes can modify the control bits of cpsr. If the data-abort exception was raised in the nonprivileged mode (usr) we could not return to the data-abort exception handler mode. Therefore, we test for the usr mode; if we detect it, we switch the processor to sys mode, since the link register and stack pointer for usr and sys modes are shared as shown in **Figure 1**. This is being accomplished by the instructions shown on Lines 11 and 12.

Forced ARM state

The trick for the state switching is to use "forced" ARM state. This step is necessary because the Thumb state doesn't support access to control bits of cpsr. If the exception was generated in Thumb state of some mode, we clear the mode's Thumb bit and enter that mode in ARM state. Then we can get the desired sp and lr values. The register r7 serves as an intermediary for this process. Having the Thumb bit cleared in r7, and subsequently loaded its value to cpsr, we don't let the processor know that we are "roaming around" its modes and collect crash data. The instruction

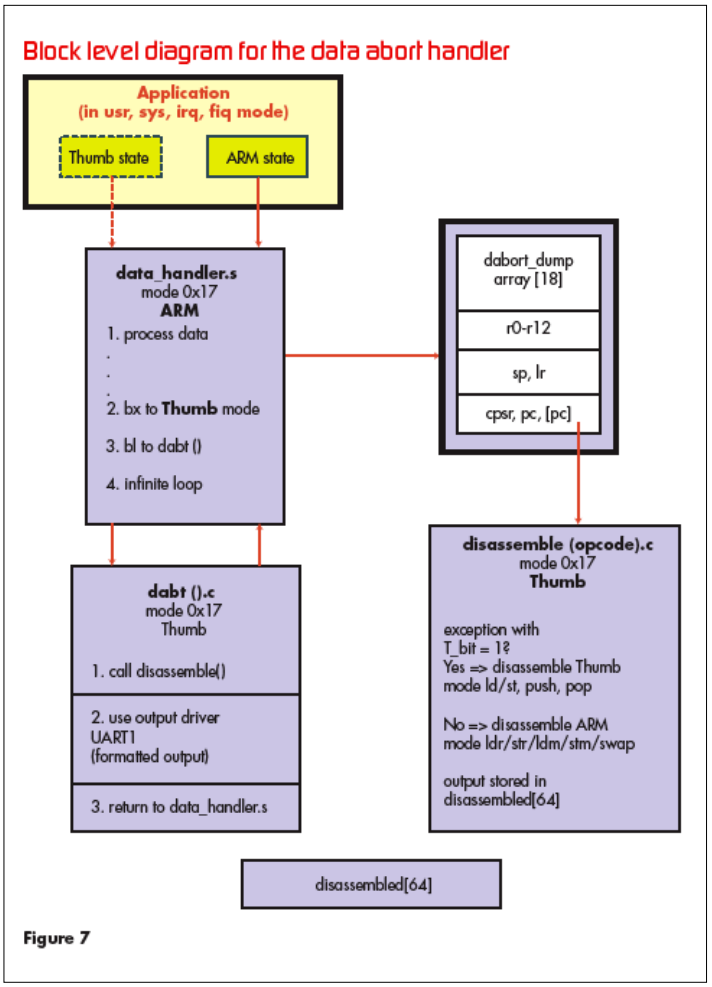

Figure 7

shown on Line 13 accomplishes Thumb bit clearing function, while the instruction on Line 14 performs the desired switching action without causing unwanted change of states. Using this method we do not forfeit access to cpsr for switching back to mode 0x17, the abort mode.

Now we're ready to collect the data we need: the content of the link register and the stack pointer. We'll accomplish this transparently, without reference to any mode and state, as shown by the instructions on Lines 15 and 16. In this manner, we saved lr and sp to registers r3 and r4, respectively.

Data collection is done; now we switch back to the data-abort mode to finish the data processing. We restore cpsr (Line 17) using the temporary scratch register r5, to which the original cpsr was saved on Line 7.

Lastly, we have to find out which state generated the exception. If it was the ARM state, the opcode is 32 bits and we load the word into r1; if it was the Thumb state, the opcode is 16 bits, and we load a half-word. We load r1 from the memory address that is pointed to by the value in r0. The register r0 holds the address of the offending instruction as we calculated it earlier in Line 6. The opcode that we'll obtain now will be disassembled in a C-coded function later. We store the opcode with the conditional load shown by the instructions on Lines 19 and 20, which follow immediately the Thumb bit test on Line 18.

Now we have the second set of data (stored in r0-r4) ready to save in our dabort_dump array. After we adjust the pointer to the beginning of this array (Line 21) we again use the store-multiple instruction, shown in Line 23. The state of the CPU registers at the time of abort is now fully saved in dabort_dump[18] array; this is a globally accessible array.

It's now time to do "housecleaning": restore the stack pointer, restore registers to comply with ATPCS function epilogue and finally we switch the processor to Thumb state. Since I use IAR's EWARM compiler, I took advantage of reloading the stack pointer with linker file script directive SFE(ABT_STACK), shown on Line 25. Once the stack pointer is restored to its original value, we are able to restore full register context. Using this feature makes the epilogue simple; see Line 26. Then we switch the processor to Thumb state and call the Thumb-compiled function dabt.c, Lines 28, 29 and 32. When the function dabt.c returns, we put the processor in to an infinite loop, Lines 35 and 36.

The block diagram of the data-abort exception handler's dependencies and process flow is shown in **Figure 7**.

As I close this section, I'd like to express my deep appreciation for the encouragement and technical advice of Peter Maloy of CodeSprite. When I started this project, I was aware of my exception handler's limitations. I now believe the energy I had invested in writing the disassembler paid off. Peter was very generous and helpful with his comments on the structure of the disassembler and in clarifying some concepts when the reference books seemed wrong to me.

Fortunately, the abort model is clearly documented in ARM7T-MDI-S Technical Reference Manual, and I was able to confirm it with my tests later. Thanks to my interactions with Peter Maloy, this paper is a comprehensive collection of information related to data-abort exception on LPC 2148 specifically and on ARM7 in general.

Potential risks

As I said earlier, I started to write my own exception handler and during while testing it, I stumbled upon undocumented features of the chip I was using. Test by test, day by day, I was getting closer to explaining some of my handler's outcomes, such as the difference (delta) between the number of generated exceptions and trapped ones, as shown in Table 4. This delta

led me indirectly to issues of the address aliasing of USB RAM and special function registers (SFR) corruption. In the absence of any authoritative documentations or explanations, I could only describe the symptom but couldn't cure the root cause of the decreased efficiency of my data-abort exception handler.

To conclude, there are risks of stack overflow in this data-abort exception handler. Let's take a closer look at the boundaries and propos e possible consequences of the stack behavior at these junctions.

Case 1

The stack segment is located at the bottom of SRAM, 0x4000 0000 extending upwards, and data zero-initialized and un-initialized segments are placed above the stack segment (at higher address). See Boundary #2 in **Figure 5**.

Stack "spill" to the region of the "special registers" area (0x3FFF 8000 to 0x3FFF FFFF) does not generate a data-abort exception. Furthermore, due to the nature of this region's purpose and content, the processor's behavior will become erratic when hardware-configuration values are overwritten.

Potential risk for crash is high (but unique to the chip design).

Case 2

The data zero-initialized and un-initialized segments are located at the bottom of SRAM, 0x4000 0000 extending upwards; the stack segment located above these segments at higher addresses.

Stack "spill" will overwrite data. This is unrelated to the data-abort exception context.

Potential risk for crash: conventionally high (not unique to the chip design).

Case 3

The stack is located at the top of SRAM, 0x4000 7FFF extending downwards with data zero-initialized and un-initialized below it at lower addresses.

Undesired memory access to some of the "special registers" will reduce the available SRAM; in other words, the 32KB LPC 2148 part might become a 16KB or a 8KB LPC 2142/1, or the SRAM becomes nonexistent.

Swapping the zero-initialized and un-initialized segments with the stack placement has limited merit.

Potential risk for crash: high due to the fact the stack disappears completely (unique to the chip design).

Case 4

The stack is located at the top of USB RAM, 0x7FD0 1FFF extending downwards, with data zero-initialized and un-initialized below it at lower addresses. See Boundary #3 in **Figure 5**.

The stack is located at the bottom of USB RAM, 0x7FD0 0000 extending upwards, with data zero-initialized and un-initialized above it (at higher address). See Boundary #4 in **Figure 5**.

Due to the apparent address aliasing occurring in the region from 0x7FD0 0000 to 0x7FDF FFFF, the undesired writes to any area above the USB RAM will not raise the data-abort exception, but the stack data will be modified.

Potential risk for crash: high (unique to chip design).

Comments:Of course, you could use either the compiler's option to generate stack overflow checking code, or use an RTOS's function to perform the same. However, an unwanted and undetected pointer incursion to Reserved Area HSL #4a may result in "disappearance" of the stack. I am not aware of any available defense against this.

Heroes with a 1k faces

The idea for this work was born out of necessity to simplify my debugging and my curiosity about exceptions on ARM7. I was not content with trapping data-abort exceptions by an exception handler with an infinite loop. I searched for a better way.

What I discovered was that the data-abort exception handler's performance and ability to capture offending pointer incursions to Reserved Area #2 and #4 are reduced by virtue of the LPC 2148's unique hardware design and its boot-loader implementation. My findings confirmed that the concept of the data-abort exception handler works on other ARM processors, but that the LPC 2000 family has a deficiency that makes it harder to use in general; with respect to exception handling, the LPC 2000 family is more difficult to debug. Therefore, I've presented some of the undocumented features here in hopes that it makes your job easier.

I hope my findings will be of help to you and hope you, too, will be restless enough to question and add your own improvements to this work. I'm eager to hear what you discover.

Endnotes:

1. Sloss, A., D. Symes, and C. Wright. ARM System Developer's Guide. San Francisco, CA: Elvesier, 2004. page 374 of Chapter 10 "Firmware" shows use of this type of dummy handler implemented with an infinite loop.
2. ARM System Developer's Guide: Preface, page xii: "At the heart of an embedded system lie the exception handlers. Efficient handlers can dramatically improve system performance." Also see Chapter 9 "Exceptions and Interrupt Handling," page 330: Chapter 9 covers the theory and practice of handling exceptions and interrupts on the ARM processor through a set of detailed examples.
3. Seal, David. ARM DDI 0100E, ARM Architecture Reference Manual (a.k.a. ARM ARM), Harlow, UK: Pearson Education Limited, 2001.
4. LPC2141/2/4/6/8 User Manual UM10139, Rev. 01, Aug 15, 2005, Philips: System Memory map, **Figure 2**, page 8 and **Figure 5**, page 14.
5. Byte rotation for non word-aligned LoaD is implementation-specific; LPC 2148 does perform byte rotation as described here.
6. "Unpredictable means that the result of an instruction cannot be relied upon. Unpredictable instructions or results must not represent security holes. Unpredictable instructions must not not halt or hang the processor, or any parts of the system." (ARM ARM, Glossary-xiv).
7. Some other ARM versions are bi-endian.
8. ARM ARM, pp. A4-38.
9. ARM ARM, pp. A4-44.
10. Déjà vécu: sensing that everything is just as it was before, as opposed to Déjà vu, same appearance as before.
11. Latin: "here there are lions" often found on old maps indicating uncharted territories.
12. Khan, Ata R. "Representing a Microcontroller in C," ARM IQ, Volume 3, Number 5, 2005. You can find this article on ARM IQ Online, www.arm.com/iqonline
13. Seal, David. ARM ARM, page A2-18.
14. Seal, David. ARM ARM. Effects of data-aborted instruction, page A2-17.
15. A higher register number is assigned to a higher memory address.
16. ARM DUI 0041C ARM Software Development Toolkit, Version 2.50 Reference Guide, Cambridge, England: ARM Ltd., 1998, page 12-39: "If the exception is an interrupt, or a prefetch or data abort, the user function should make the ARMulator retry the instruction, rather than continuing from the following instruction."
17. ARM DUI 0040D ARM Software Development Toolkit Reference Guide, Cambridge, England: ARM Ltd., 1998, page 9-37: "If there is no MMU, the data-abort handler should simply report the error and quit."
18. Furber, Steve. ARM System-on-chip architecture, second edition. Addison-Wesley Professional, 2000, page 190.
19. Furber, Steve, page 191.
20. Philips LPC 2148 Errata, version 1.2, Feb 27, 2006.