m0FaultDispatch

Table of Contents

ARMv6-M architecture history

History & the ARMv7-M

In 2004, ARM released the Cortex-M3 core. It was designed as a 32-bit microcontroller core. It was quite cheap, quite powerful, and an instant hit. Cortex-M3 ran the newly-created ARMv7-M architecture. It was basically the unmodified ARMv7 architecture's Thumb-2 instruction set at the user level, bolted unto a completely new system-level exception model. The idea was quite clever. With some ABI trickery, all the code, even interrupt handlers could be written in C, with no need for complex assembly interrupt dispatchers and other such things. Exceptions had priorities: some - configurable, others - fixed. When something went wrong, an exception was taken.

MemManage fault is for when a forbidden memory access was attempted, **BusFault** for when the underlying memory subsystem failed to execute a transaction (for example when you access a memory that is simply not there), **UsageFault** is for when an invalid instruction is executed, an attempt is made to switch to ARM state, or an unaligned access is attempted when those are disabled. These can each be disabled, and if so, the exception auto-escalates into a **HardFault** - the catch-all "shit went south" exception. Good code would likely have a handler for each of these exception types to properly log what went wrong. Some of these even tell you where things wen wrong. Many types of **BusFault** and **UsageFault** exceptions provide to you, in a special register, the address that was attempted to be accesed.

Cortex-M0 is born

In 2009, ARM released the Cortex-M0 core. The idea was simple. Cortex-M3 was very powerful, but also kind of big - lots of transistors. There was a lot of pent up desire in the market for a smaller core to take on the 8-bit and 16-bit microcontrollers that ruled the low end. ARM responded with the Cortex-M0. It removes almost all of the 32-bit-long instructions from the Cortex-M3 core, basically taking it back to ARMv5T Thumb-1 instruction set. This generated a lot of size savings since the decode logic for Thumb-2 was complex (if you do not believe me, take a look at all the instruction formats that ARMv7-M has). They called it the ARMv6-M acrhitecture. Another thing ARM cut was the number of available interrupts. Also makes sense - smaller core needs fewer interrupts. And, the last thing that was cut was the (I guess) complex logic of recording what and where went wrong. No more **MemManage** fault, **BusFault**, or **UsageFault**. There is just **HardFault**. If anything anywhere goes wrong, a **HardFault** is generated and that is it. No information is provided as to what specifically went wrong. ARM's docs basically say that they consider **HardFault** to be unrecoverable. You can indeed return from it, but since you do not know what went wrong, why would you?

Exception handling

How things normally work

Every exception has a number. Generally higher exception numbers are pre-empted by lower exception numbers in Cortex-M0. Reset is exception number 1, pre-empting anything (obviously). **NMI** (the non-maskable interrupt) is number 2, and pre-empts every thing (including most faults, yes!). **HardFault** is number 3, pre-empting anything that is not an **NMI** or a reset. Numbers 4-10 are reserved, and **SVCall** is number 11. It goes on, but this is as far as we care. If an exception is caused while executing at a lower-or-same exception number, the CPU will take a **HardFault**, since an exception cannot take the CPU to as higher exception number

- this is not allowed. But, the same rule applies to **HardFault** itself. So what happens if you cause a **HardFault** while executing your **HardFault** handler, or **NMI** code? ARM thought of this too. The CPU enters a state known as **Lockup**. Basically this prevents it from getting itself into any more trouble. No more insutrctions will be executed until a reset occurs.

What happens when an exception is actually taken? Eight words are pushed to the current stack: **r0**, **r1**, **r2**, **r3**, **r12**, **lr**, **pc**, and **APSR**. This may seem like an arbitrary set of registers, but actually this is precisely the set of registers that are caller-saved in ARM (plus the status register, which the ABI specifies anyone may clobber anytime). Having the core push them allows the handler to be written in C with no special compiler help, cause all caller-saved state is auto-saved. On exception entry, **lr** is set to a special magical value that, when jumped to at the end of the handler, will tell the CPU to return from exception (pop state off the stavk). This value is **0xFFFFFFFx**, where x determines whether to go to Thread (normal code) or Handler (exception handler) mode, and which of the two stack pointers to use to load an exception frame from.

So what if we try to figure out why we faulted?

Ok, so Cortex-M0 will not tell us why we faulted, but what if we try to figure it out ourselves? Logically, we can take a look at the **lr** value we have, use an **MRS** instruction to read the proper stack pointer register, and there find our exception frame (those 8 words the CPU auto-pushed). What now? There is nothing there to tell us why we faulted really. Ok, we can check the pushed **APSR** for the **T** bit, and if it is not set, we know the CPU tried to enter ARM mode. Ok, easy enough. What else can we tell? "Well, we can look at the **PC** value to tell if it is in a valid range," you might say. But, what is a valid range? That may differ based on available external memories, time of day, and the phase of the moon. No, this is a bad approach. And even if it were not, that gives us surprisingly little information.

Ok, so what if we try to read the actual instruction that faulted. The exception frame contains **PC**, so we know where that it. What happens if we try to read it? Well, it might be unaligned. We'll fault! Bad. But, we can check for that. But what if **PC** points to some location we cannot read? We'll fault! But, we're already in the **HardFault** handler. So if we fault, lockup! We're screwed. Nope, not a good idea. Maybe ARM was right, no way to recover from a **HardFault** on a Cortex-M0...

How about a dirty hack?

Is a good fake as good as the real thing?

So, what happens if we craft a custom exception-return frame on stack, and try to return to it? This works well actually. The CPU does some checks, and, if all goes well, uses it. This logically makes sense, since there is nothing magical about an exception frame.

Can we go to a hidden level?

Allright, the exception numbers 4 through 10 are reserved, but what happens if we actually try to go to one of them. Obviously we cannot take an exception to get there, since no existing exception type takes us there, but we can RETURN to there from any exception whose number is lower, say a **`HardFault`**? What happens? As far as I can tell (I tested every revision of Cortex-M0 and Cortex-M0+), the CPU compares levels numerically, so even if those levels are in theory reserved, we can indeed go to them, execute in them, etc. But why?

Putting it all together

Let's use exception number 4. This is as close as we can get to **`HardFault`** level without being in it. Why? This way we're still more important than anything else in the system (nothing will interrupt us), but at the same time, if we fault, no lockup - we just re-enter the **`HardFault`** handler. So, we can now live dangerously! We can try things! We just need to be careful to not end up in an endless loop of repeated faults.

And, we, of course, we need to keep the original exception frame around for when we want to return to the original faulting context. "Return!?" you might exclaim. Yes, if we can classify faults, we can fix some, and resume execution. For example, this can be used to emulate instructions, or skip faulting memory accesses. There are a lot of moving parts, but in realty this all works out pretty well.

m0FaultDispatch

The complex moving parts

First, we need some global state. One bit actually (**`mInExcHandlerAlready`**). But we use a byte for convenience. It starts at zero, and anytime we take a fault, while we're processing, we have it set to one. This tells us if we're in a nested fault, if we get another fault. First, in our **`HardFault`** handler, we check for the **`T`** bit, and report that as an attempt to enter ARM mode. Easy. Next we read the exception **`PC`** and see if it is unaligned. If so, we report that. If not, we set **`mInExcHandlerAlready`** to one, and craft an exception frame to go to exception number 4. We use the **`lr`** value of **`0xfffffff1`** to indicate that we want to use the main stack and go to

handler mode. We set the pushed **APSR** to **0x01000004** to indicate thumb mode, exception 4. And we return. At this point we can analize the exception further, with no worry about faults.

We structure any code that can fault in a special way. If any instruction faults, *it and the next one* will be skipped. This allows use to easily write code to handle failures. For example the function that safely tries to read an 8-bit value, and tells us if it failed or not looks simply like so:

```
//on fault, 2 instrs are skipped (faulting instr and next one)
static bool __attribute__((noinline)) m0FaultDispatchRead8(uint32_t addr, void *to)
{
        bool ret = false;
        uint32_t t;

        asm volatile(
                "       mov  %1, #0       \n\t"
                "       ldrb %0, [%2]     \n\t"
                "       mov  %1, #1       \n\t"
                "       strb %0, [%3]     \n\t"
                :"=&l"(t), "=&l"(ret)
                :"l"(addr),"l"(to)
        );

        return ret;
}
```

It is clear that if the fault does not happen, the value is copied to whatever "to" points to, and 1 (true) is returned. If that load fails, the **mov %1, #1** will be skipped, and thus **%1**(ret) will stay as 0 (false). Nice and easy. "Why," you might ask, "would two instructions get skipped in case of a fault?" Well, our hard fault handler does that, if it is entered while **mInExcHandlerAlready** is set, since our code can only fault due to invalid memory accesses (we carefully craft it for that to be the only such case).

Interpreting it all

Given that we can now safely read memory, or know it is unreadable, the next obvious thing to do is to read the faulting instruction. We do. The first halfword we read can immediately tell us if this is a 16-bit-long or a 32-bit-long instruction. Cortex-M0 has no valid 32-bit-long instructions that can cause a fault, so that means that if we see a 32-bit-long instruction at the fault location, the fault was due to it being an invalid instruction. We report this. If we see a 16-bit-long instruction, there is more work to be done.

How do we know why an instruction faulted? Well, we can intepret it and see what it was trying to do. This can tell us why it failed to do that. That is precisely what m0FaultDispatch does. It properly interprets all 16-bit instructions that can cause a fault. Invalid ones are immediately reported. For memory-access instructions, we check for alignment as we work out the addresses used. If there are alignment issues, we can and do report them immediately. If not, we then attempt the requested accesses. If a write was requested we first try a read, so we can write back the same value and not corrupt data. We access data using the same-size access so that even access to weird peripherals that only accept accesses of a certain size work properly. A clever reader might notice that writing back data that was read back may not be safe for some HW regs. True, but if the write faulted, it will fault again and thus no data will be overwritten afterall! For load-multiple and store-multiple instructions, multiple accesses are attempted and m0FaultDispatch can properly report which one failed.

The short road to recovery

Before calling the user-provided handler, m0FaultDispatch will stash all the registers not alreay in the exception frame on stack, so that the handler has access to them. This means that your handler gets access to the entire exception frame and all the registers at the time of fault. This allows you to emulate instructions, fix fault reasons, etc. All of this state is modifiable, and when your handle returns, it will be all put back into the proper place and form.

The practical upshot

With m0FaultDispatch, your Cortex-M0 projects can now have better-than-Cortex-M3 fault reporting, with ability to recover and better understand faults. Faults are classified into:

- **`M0faultMemAccessFailR`** - A data read failed. The faulting address is provided to you.
- **`M0faultMemAccessFailW`** - A data write failed. The faulting address is provided to you.
- **`M0faultUnalignedAccess`** - A data access was attempted with bad alignment. The faulting address is provided to you.
- **`M0faultUnalignedPc`** - **`PC`** was seen unaligned
- **`M0faultInstrFetchFail`** - **`PC`** points to memory that could not be fetched and thus execution failed (you jumped to nowhere)
- **`M0faultUndefInstr16`** - An undefined 16-bit-long instruction was encountered
- **`M0faultUndefInstr32`** - An undefined 32-bit-long instruction was encountered
- **`M0faultBkptInstr`** - A breakpoint (**`BKPT`**) instruction was encountered
- **`M0faultArmMode`** - You attempted to enter ARM mode
- **`M0faultUnclassifiable`** - Something else happened (unlikely)

Gimme!

The code, and demos are available for download here: [LINK (/images/m0FaultDispatch.zip)]. The license is simple: This code is free for use in hobby and other non-commercial products. For commercial use, contact me (mailto:me@dmitry.gr).

post a comment