Using the Cortex-M3/M4 Flash Patch and Breakpoint Component for Firmware Updates

Marcus Harnisch, Doulos, 2010

Abstract

ARM processor cores implementing the v7-M architecture, currently Cortex-M3 and Cortex-M4, include a component called the “Flash Patch and Breakpoint” Unit (FPB). Besides being used by debug tools to provide a hardware breakpoint mechanism, FPB provides a mechanism for patching immutable program code or literal constants in firmware by redirecting memory accesses. This paper will show you an MCU-vendor independent approach to do this.

Introduction

Case 1: Part of the firmware of a Cortex-M3 based device is located in a ROM to protect it from any form of modification. The code is suspected to contain a bug which needs to be fixed in the next product release. Development devices with programmable memory instead of the ROM might not be available. Perhaps verification could be simplified if it were possible to instrument the code.

Many man-hours will have to be spent on verification that the proposed fix will work as expected in the actual product. Having a mechanism that would allow remapping instruction access from the affected function to a function containing the code to be verified would enable developers to test the change *in situ*.

Case 2: Another product with firmware in a memory region that cannot be modified for a variety of reasons provides an upload mechanism for program code of some sort. In a specific product, part of the firmware is located in ROM, another part will be factory programmed into an OTP NVM (One-Time Programmable Non-Volatile Memory). Or perhaps a USB mass storage device might be polled by a bootloader. If a suitable integrity-checked firmware update can be found, it will be copied into a reserved memory region. The existing firmware will have to access new functions without knowing their exact locations in advance.

Users of common microcontrollers might shake their heads in disagreement, since those devices offer a variety of memory remapping options (RAM/Flash booting), and plenty of flash which can be reprogrammed often enough. However, v7-M cores can also be found in ASICs which have been tailored for a very specific application, with only the very functionality needed in the mass-produced end product.

In many of the situations outlined above and other, similar ones, the traditional workaround is either to provide entry points at fixed addresses where possible. Alternatively all relevant functions (which need to be identified first) could be accessed through a programmable call table containing function addresses. This is rather inefficient since the size of the call table has to scale with the number of functions that *might* be replaced, not the actual number of functions that *will* be replaced in the end.

Lastly, all traditional solutions replace entire functions, while FPB can replace individual instructions.

FPB Theory of Operation

The ARM architecture v7-M (1) defines “Flash Patch and Breakpoint” (FPB) as a component that monitors instruction fetch or data read (literal load) to CODE memory in the address range between 0x0 and 0x1FFFFFFF. If an instruction address in CODE memory matches one of the programmable FPB instruction comparators, depending on FPB configuration either of two things may happen:

1. FPB returns a BKPT instruction. This is perhaps the most common use of FPB and is normally controlled by the debug environment to implement common breakpoints.
2. FPB returns an instruction from a remap table. The remap table must reside in SRAM memory space (0x20000000-0x3FFFFFFF) and contains instructions or literals to be returned

if the corresponding comparator matches. Since all instruction fetches are word accesses, care must be taken when remapping narrow instructions. Remapping adds extra latency to the instruction fetch.

In case of data access (literals), only remapping is allowed. It is worth noting that only read accesses will be monitored and remapped that way. Write accesses to addresses matching a comparator will remain unaffected.

FPB provides an implementation defined number of address comparators for matching either instruction or data addresses, respectively. In order to keep software generic, the exact number of comparators of each type can be determined by reading the FP_CTRL Register fields NUM_LIT and NUM_CODE1[1].

Remapping is an optional feature of FPB. The bit FP_REMAP.RMPSPT indicates whether besides generating breakpoints a particular FPB implementation supports remapping at all.

How it really works

In order to remap an instruction, the instruction address has to be programmed into one of the comparator registers and enabled (FP_COMP*x*), the remap table has to be created in SRAM region and the replacement instruction has to be placed in the slot corresponding to the comparator register. The FPB has to be pointed to the table (FP_REMAP) and the FPB has to be enabled globally (FP_CTRL).

Whenever the programmed address is fetched from, the word from our remap table will be returned instead of the one at the requested address.

If all we can do is replacing individual instructions, we can replace a function call (BL <*label*>) with another, leaving all other calls to the same function unaffected. In many situations, however, we will want to replace an entire function no matter where was called from. In this case we don't just replace all calls that take us to the function (which would be a rather inefficient use of our limited number of comparators), but instead we will remap the address of the first instruction of a function. That way, when the processor fetches the function's first instruction, FPB will remap that memory access to SRAM and return another branch[2] to a replacement function instead. Figure 1 shows the process and the individual steps. Incidentally, the address of the first instruction in a function is the address of the function itself, which makes it rather simple to determine this address even from higher level languages such as C.

[1] The architecture defines another related bit-field NUM_CODE2 (number of FPB banks) which always reads zero in both Cortex-M3 (3) and Cortex-M4 (2)

[2] Don't use BL here, since that would spoil our link register.

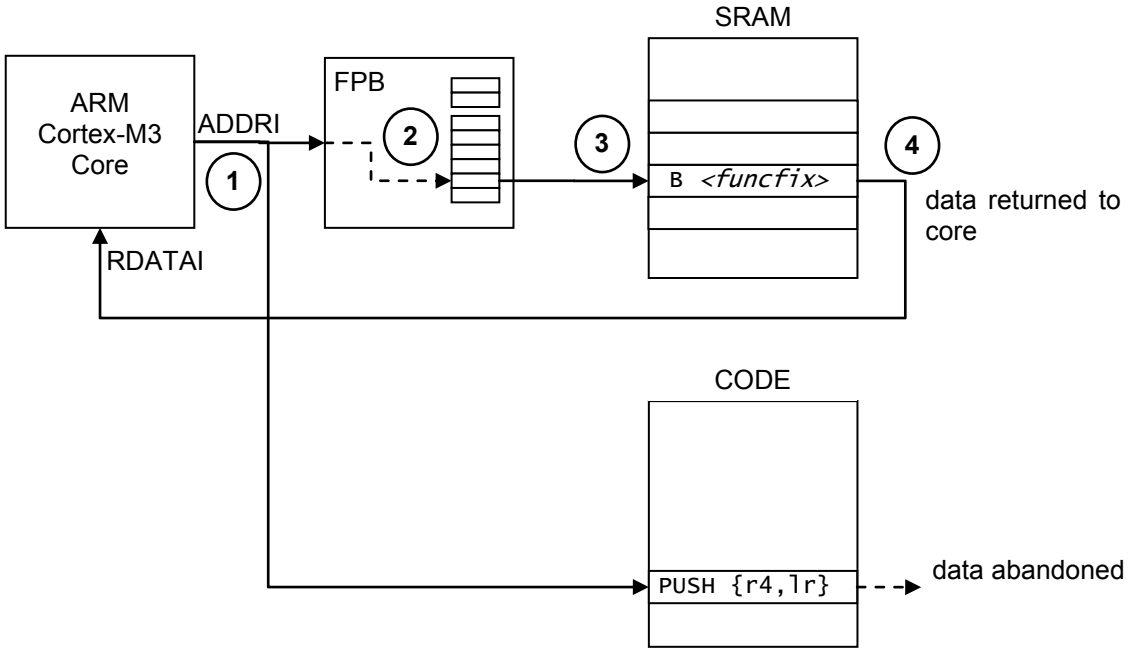

Figure 1FPB Operation

Example FPB usage

In the first use case above, reprogramming the FPB registers would perhaps be part of a debugger script that is used to initialize the device for a specific kind of test. Since all debug-relevant registers are memory mapped, they can be accessed easily using memory access commands provided by every debug tool. In this scenario, the firmware itself will not have to be aware of FPB.

The second application scenario will require programmatic FPB configuration as part of the system initialization in a boot loader. Let us assume for a moment that our system memory map consists of a ROM region in which our firmware is stored and an extra Flash/OTP region for updates. Of course RAM is available, too. ROM and OTP are part of the CODE region (first .5GB in the memory map), whilst RAM is part of the internal SRAM region (1). A diagram of this memory map is shown in Figure 2.

An example application has been implemented which realizes the essential steps of such an update mechanism. The ROM function *func1()* contains a bug and is to be replaced by *func1_fixed()* using FPB.

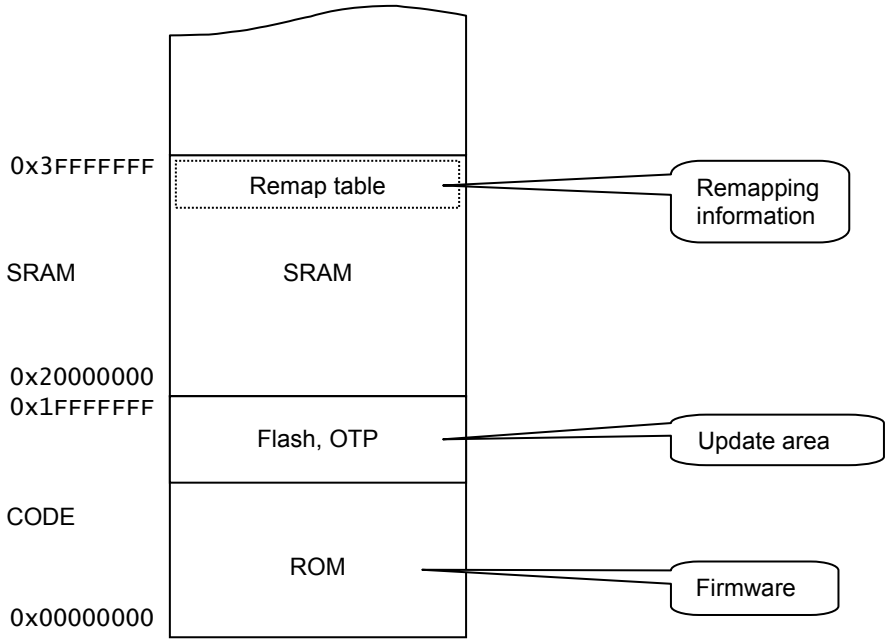

Figure 2 Example memory map

Normally our example system will execute code from ROM only. During system boot a routine will have to detect whether the Update area has been programmed and set up FPB as well as the remap table containing the instructions to be returned.

The partial image loaded into the Update area contains a table of address pairs. The first address in this tuple is the original function that we want to replace with another one, perhaps located in the Update area. The second address is that of the replacement function. We need this address for two reasons: 1. It is the value to program into one of the FP_COMP*x* registers and 2. the synthesized branch instruction is PC relative so the value in the B-opcode must be calculated relative to the address of the original instruction.

This table will have to be located at a fixed address so it can be found by the boot-loader. This could be achieved using an appropriate scatter loading file (aka linker script) or some other tool specific mechanism.

The boot-loader will read the table skipping all slots that are “empty”. Both Flash and OTP read ‘1’ when not programmed, so an address value of 0xFFFFFFFF has been chosen as ”empty” value. It turns out that in ARMv7-M this is perfect, since addresses above 0xA0000000 carry the hard-wired attribute XN (eXecute Never) and therefore couldn’t possibly be legal function addresses anyway.

For each valid entry, the address field in the corresponding FP_COMPx register will be set to the original function address. The register will also be configured to remap the instruction (as opposed to generating a BKPT), and enabled.

The routine will calculate the branch offset from the current PC at the original function address to the new function and generate a branch instruction (B) in the corresponding entry in a remap table in SRAM. It is important to remember that the PC position in Thumb-2 is always calculated from the executed instruction (Ex+4)[3].

[3] This is different in Thumb-2 compared to traditional ARM cores, where the reference point was not the address of the executed instruction but the address of the instruction fetched from memory.

FPB itself will be informed (in FP_REMAP) about the location of the remap table, which we created as a normal C data structure and which could be anywhere in RAM[4]. Finally, FPB will be enabled using the global enable bit in FP_CTRL.

When the program executes, a function call to *func1()* will be successfully redirected to *func1_fixed()*.

Caveats

FPB is part of the debug infrastructure and in particular it will be used by debug tools to generate breakpoints. If you are debugging code which uses the FPB, like our example application, it could happen that the debug tool overwrites your FPB configuration or conversely that your FPB configuration overwrites breakpoints. Before implementing your boot-loader, make sure which order your debugger assigns the FP_COMP*x* in.

At this point there doesn't seem to be a standard for communicating this dual use between debugger and application. I recommend that tool vendors determine for each FP_COMPx register, whether it is used for remapping and only use the remaining comparators. In case of a conflict the debug tool could issue a warning, giving users the option of overwriting the register, or not setting the breakpoint.

Conclusion

All v7-M cores support FPB although implementations might differ in terms of size and availability of the remapping functionality. FPB is useful for setting breakpoints even in read-only memory regions without having to differentiate between software and hardware breakpoints.

Beyond its debugging aspect, FPB *with* remapping can be used in ROM-based end-products to replace call tables. Setting up FPB is rather straightforward as demonstrated by the example code provided along with this document.

Bibliography

1. **ARM Ltd.** *ARMv7-M Architecture Reference Manual.* Cambridge, UK : s.n., 2010. ARM DDI 0403D.

2. —. *Cortex-M4 Technical Reference Manual.* Cambridge, UK : s.n., 2010. ARM DDI 0439C.

3. —. *Cortex-M3 Technical Reference Manual.* Cambridge, UK : s.n., 2008. ARM DDI 0337G.

[4] *Almost* anywhere, that is. The table must be aligned to an address which is a multiple of 32. We achieve this using the *aligned* attribute in a syntax common to GCC and RVCT.