Chapter 6: Introduction to ARM Memory and Assembly Language

In this chapter we jump into the core (pun intended) of MPG and examine the very low level details of the ARM processor. We start by looking at the memory structure in the ARM7 and understand how registers, peripheral special function memory, RAM and ROM work together in the microcontroller. Then we will examine the full instruction set on the ARM7TDMI core and focus in on the instructions that are of greatest interest and most commonly used.

6.1 ARM7 Memory Space

Before writing code, you must understand the system that you will be working with. Writing code in high-level languages on PCs often provides the luxury of removing the necessity of knowing how memory in the system is being used not to mention what is actually happening on the bit-level of the microcontroller. The operating system and compiler tends to take care of RAM allocations, you know your program lives somewhere on the hard disk, and any time you need to make use of hardware you can call an API or function in a .dll. You probably do not have a clue about how all the hidden code works and what the operating system is doing in the background – and that is perfectly fine.

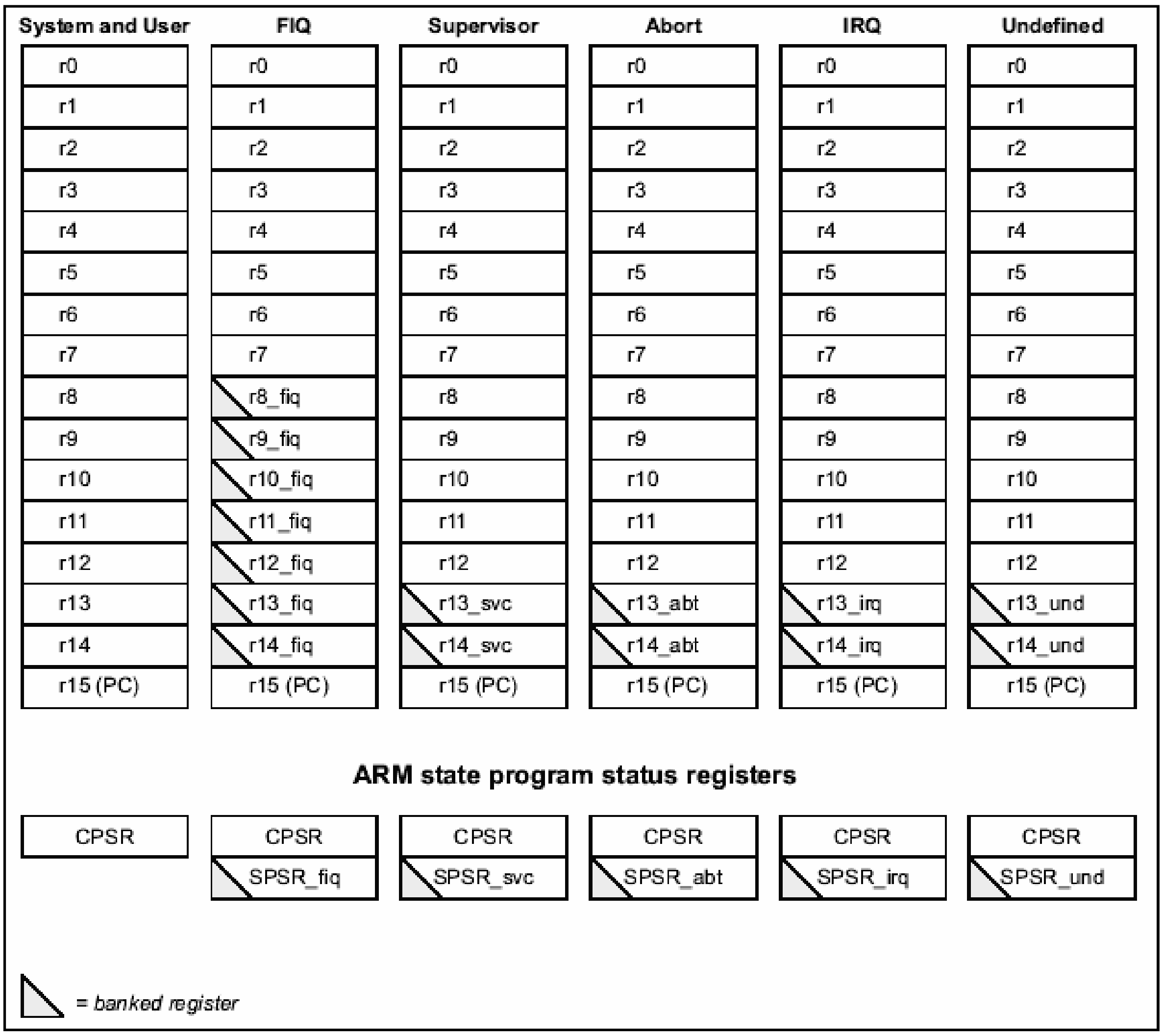
In MPG and in small embedded systems in general, you DO need to be aware of all the details like this. It often becomes up to the embedded designer to manage memory and know exactly what resources are available and how they are used in the system. It is not uncommon to build a system from scratch, which means you will have to write all of the drivers that your higher-level functions will call. Though it might sound a bit daunting, microcontrollers are setup and documented to make the process go quite smoothly. If you build your system of drivers intelligently, you can re-use them for different applications running the same processor.

The first step is to know the definitions between the different memory resources that you have available.   Registers, RAM, Flash, cache – all these terms refer to different places where data will live and can be accessed in various ways. You also must learn things like how fast accesses to different memory locations are, and whether or not your data will be volatile or non-volatile.

6.1.1 Core Registers

Registers are volatile memory spaces that have specific purposes, though the implementation of a register can vary between processors or even vary within the same microcontroller. Some registers are connected to the processor bus and are addressed with pointers to access the data they hold. In essence, they are RAM locations with special functions. Other registers may live entirely inside the processor core and are accessed in a single clock tick during the execution phase of the instruction cycle.

In the ARM7, there are 17 core registers available in the standard mode of operation, and another 20 registers for the other operation modes that become available at different times. Figure 6.1.1.1 shows the complete list of registers for all the ARM modes.



System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	◣ r8_fiq	r8	r8	r8	r8
r9	◣ r9_fiq	r9	r9	r9	r9
r10	◣ r10_fiq	r10	r10	r10	r10
r11	◣ r11_fiq	r11	r11	r11	r11
r12	◣ r12_fiq	r12	r12	r12	r12
r13	◣ r13_fiq	◣ r13_svc	◣ r13_abt	◣ r13_irq	◣ r13_und
r14	◣ r14_fiq	◣ r14_svc	◣ r14_abt	◣ r14_irq	◣ r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◣ SPSR_fiq	◣ SPSR_svc	◣ SPSR_abt	◣ SPSR_irq	◣ SPSR_und

◣ = *banked register*

Figure 6.1.1.1: ARM state registers

Source: LPC214x User Guide UM10139 Rev. 3 - 4 October 2010

The system modes are discussed a bit more later on, but for now what you need to understand is that some registers are common to all modes while others are unique. As annotated above, there are some registers that are "banked." Any banked register is unique to its particular state and would actually be a different physical memory location even though the instruction address to write to it would be the same regardless of mode. For example, if you wanted to write to r13 in whatever mode, you would use r13 = some_value, and not actually specify the unique name r13, r13_fiq, r13_svc, etc. So if you wrote a value into r13 while in User mode then switched to FIQ mode, the value in r13 User mode would not be available. While in FIQ mode, you could write to register r13 again and not impact the value you wrote during user mode.

On the other hand, non-banked registers are shared, so regardless of what mode you are in if you write a value to, say, r0 while in User mode and then switch to FIQ mode, the value you wrote is still available in r0 and would be overwritten if you chose to write to r0 while in FIQ mode. It is as if "mode" controls a bus multiplexer that for some registers will select a different location. Figure 6.1.1.2 might help clear that up a bit.

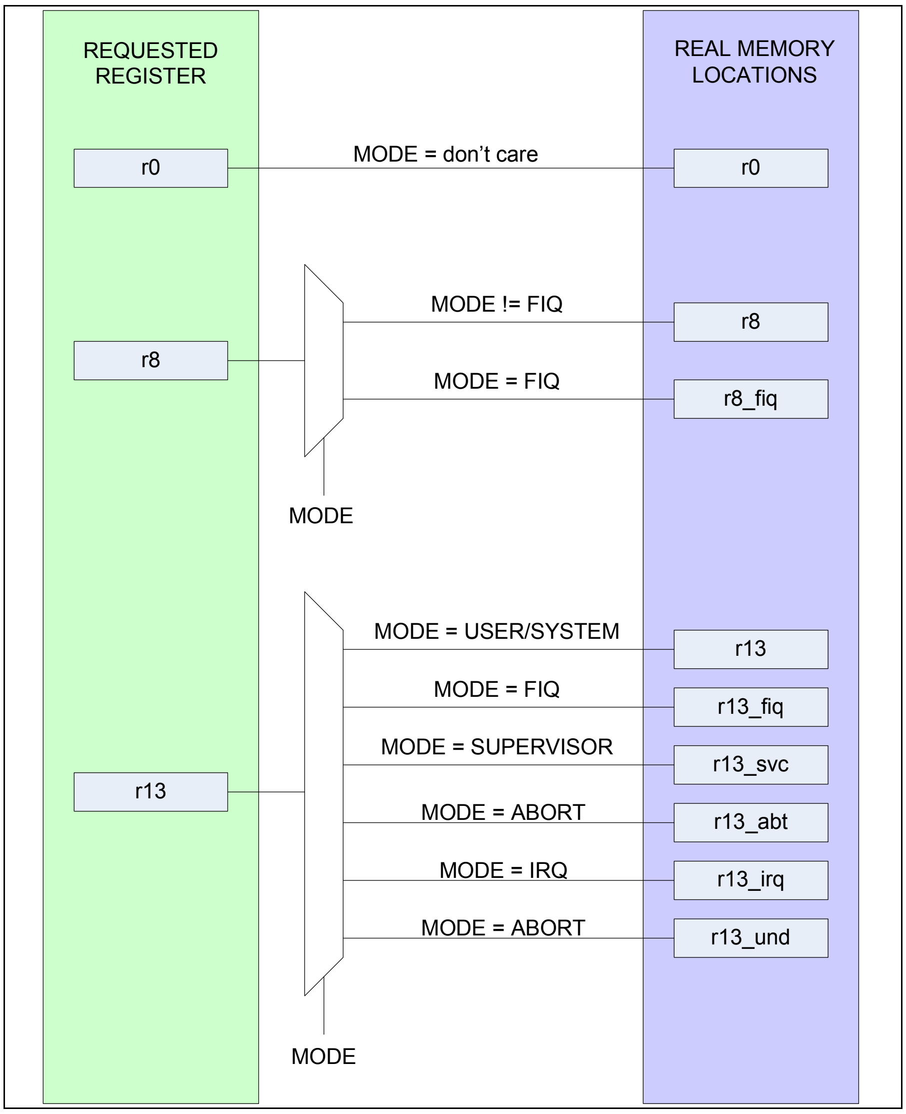


Figure 6.1.1.2: representation of memory accesses to banked registers

For a banked register, your code would always specify the same register to access, but the current mode of the processor would direct that access like a multiplexer to select the value particular to the mode the core is currently in.

You will use the processor in System or User mode most of the time. Here, the first 16 registers are general purpose and used to hold data or addresses that will be used with arithmetic or logical operations. Of these, the first 13 are generic and denoted r0 thru r12. The registers r13