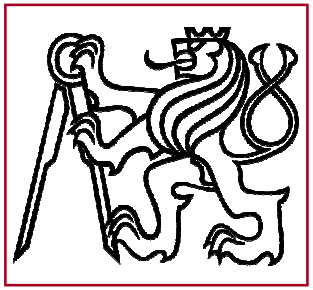
Y35PES

Programming for Embedded Systems

Ondřej Špinka, Pavel Němeček
spinkao@fel.cvut.cz nemecp1@fel.cvut.cz
http://dce.felk.cvut.cz/pes



Exceptions Handling (1)

Exceptions are triggered by internal and external sources to cause the processor to handle an event. When a exception occurs, the execution is forced to go on from a fixed memory address, corresponding to the exception type. These addresses are called **exception vectors**, stored in so-called **vector table**. Seven types of exceptions exist in ARM (aligned in priority order):

- Reset (supervisor mode)
- Data memory access abort (abort mode)
- FIQ (FIQ mode)
- IRQ (IRQ mode)
- Instruction prefetch abort (abort mode)
- Undefined instruction (undefined mode)
- Software interrupt (SWI) (supervisor mode)

When an exception occurs, the LR (R14) and CPSR registers are automatically stored into the LR_<mode> and SPSR_<mode>. LR is automatically decremented by 4 before being saved into LR_<mode>.

Exceptions Handling (2)

If multiple exceptions occur simultaneously, they are serviced in order of their priority. Exception handlers could be nested – should a higher priority exception occur when a lower priority exception is being serviced, it would interrupt the lower priority service routine.

When executing user program, the processor runs in the user mode. When a exception occur, the mode is changed automatically into some kind of privileged mode. This also gives access into a dedicated set of banked registers:

- R13 (Stack Pointer) – SP_<mode>
- R14 (Link Register) – LR_<mode>
- SPSR (Saved Program Status Register) – SPSR_<mode>
- In case of FIQ also dedicated set of R8-R12 (R8_FIQ - R12_FIQ)

Each exception handler must ensure that other registers are restored to their original state upon exit. This can be done by storing the contents of any registers the handler needs to use onto its stack and restoring them before returning.

Exceptions Handling (3)

The processor's response to an exception:

When a exception occurs, the processor does following:

- Copies CPSR into SPSR_<mode>
- Sets appropriate CPSR bits
  - To change the processor mode
  - To disable interrupts
- Stores the return address (PC - 4) into LR_<mode>
- Sets the PC to the appropriate vector address

All actions noted above are performed automatically by the processor hardware.

Exceptions Handling (4)

Returning from the exception handler:

The exception handler must perform following two actions prior to returning:

- Restore CPSR from the SPSR_<mode>
- Restore PC using the return address from LR_<mode>

These can be achieved within a single instruction, because adding the S flag (update condition codes) to a data - processing instruction when in a privileged mode with the PC as the destination register also transfers the SPSR to CPSR as required.

Exceptions Handling (5)

The return address and return instruction:

The actual value in the PC which causes a return from a handler varies with the exception type. When an exception is taken, the PC may or may not have been updated (depending on the exception type), and the return address may not necessarily be the next instruction pointed to by the PC, because of the way the ARM loads its instructions. When loading the instructions, the ARM uses a pipeline with a fetch, decode and execute stage. At any time, there will be one instruction in each stage of the pipeline. The PC actually points to the instruction being fetched. Since each instruction is a word long, the instruction being decoded is at address PC – 4 and the instruction being executed is at PC – 8.

Interrupts Handling (1)

Interrupts form a special subset of exceptions. There are two types of interrupts:

- **FIQ** – Fast interrupt service. The processor is set to the FIQ mode. Exception vector is 0x0000001C.
- **IRQ** – Normal interrupt service. The processor is set to the IRQ mode. Exception vector is 0x00000018.

FIQs have higher priority then IRQs. If they occur simultaneously, the FIQ is serviced first. If a FIQ occurs during the IRQ servicing, it interrupts the IRQ service routine.

Interrupts Handling (2)

Returning from an interrupt handler:

The IRQs or FIQs occurrence is checked after execution of each instruction. This means that a FIQ or IRQ is generated always after the PC had been updated, meaning that the next instruction is located at the address PC – 8. Because the PC is automatically decremented by 4 prior storing into the LR_<mode>, it must be decremented again by 4 prior to FIQ or IRQ handler return. Therefore, the return instruction from a FIQ or IRQ handler would be

subs pc, lr, #4

Interrupts Handling (3)

IRQ handling sequence:

Assume that an IRQ occurred when executing instruction at address 0x00001004.

Code

0x00000018	mov pc, #0x00000D00
	…
0x00000D00	mov r4, r1
	…
0x00000F00	subs pc, lr, #4
	…
0x00001000	ldr r10, [r0]
0x00001004	mov r8, r10 IRQ event
0x00001008	add r5, r5, r8

Vector Table

0x00000000	Reset_handler
0x00000004	Undef_handler
0x00000008	SWI_handler
0x0000000C	Pabort_handler
0x00000010	Dabort_handler
0x00000018	IRQ_handler
0x0000001C	FIQ_handler

PC (after execution)

0x0000100C	0x00000D00
0x00001010	…
0x00000018	0x00000F0C
…	0x00001008

Writing an Interrupt Handler (1)

There is a special keyword in the C language *__irq*, specifying that the following function is an interrupt handler (IRQ or FIQ).

The *__irq* prefix means that all registers would be automatically stored/restored on the stack, CPSR would be restored from SPSR_<mode> at the end of the function and that the return address to be used is LR_<mode> - 4.

```
void __irq irq_handler ( ) {
        ...
}
```

Writing an Interrupt Handler (2)

Alternatively, the IRQ (FIQ) handling function might be declared as:

```
void irq_handler ( ) __attribute__ ( ( interrupt ) );

void irq_handler ( ) {
        ...
}
```

Writing an Interrupt Handler (3)

Now the IRQ handler must be registered in the **vector table**, in order to be called when respective exception occurs. The simplest way to do that is to place a branch instruction to the handler into the address of the exception vector in the vector table. This only allows for branching within 32MB memory space, however, this is sufficient in most cases. The required instruction can be constructed as follows:

- Take the address of the exception handler
- Subtract the address of the corresponding exception vector
- Subtract 0x08 to allow for the pipeline
- Shift the result two bits right to produce correct word-aligned offset
- Check the top 8 bits to be clear, to allow for the instruction opcode
- Logically or this with 0xEA000000 (BL instruction opcode)

Now the resulting value can be placed to the vector address.

Writing an Interrupt Handler (4)

```
unsigned install_handler ( unsigned routine, unsigned *vector ) {
        unsigned vec, oldvec;

        vec = (( routine – (unsigned) vector – 0x08 ) >> 2 );
        if ( vec & 0xFF000000 ) {
                printf ( "installation of the handler failed!\n" );
                exit ( 1 );
        }

        vec |= 0xEA000000;
        oldvec = *vector;
        *vector = vec;
        return oldvec;
}
```

Writing an Interrupt Handler (5)

```
void __irq irq_handler ( ) {
        ...
}

void main ( ) {
        unsigned old_vec;
        unsigned *irq_vec = (unsigned *) 0x18;

        old_vec = install_handler ( (unsigned) irq_handler,
        (unsigned) irq_vec );
        ...
}
```

Writing an Interrupt Handler (6)

Registering a FIQ handler:

To allow fastest possible FIQ reception and servicing, the FIQ vector is located at the very end of the vector table. Hence there is no need to perform any branch to the handler, instead the FIQ handler can (and should) begin directly at the FIQ vector address. This is achieved by copying the handler code directly to the address 0x1C. The ARM code is inherently relocatable, however, it must use PC relative addressing only (and therefore the data it uses must be relocated too).

Writing an Interrupt Handler (7)

Registering a FIQ handler in C:

```
void __irq fiq_handler ( ) {
        ...
}

void main ( ) {
        memcpy ( 0x1C, fiq_handler, ? length_of_handler? );
}
```

Writing an Interrupt Handler (8)

What should an IRQ/FIQ handler do:

- Determine what caused the interrupt
- Branch to respective event handler (optional)
- Handle the event
- Return to normal program flow

Interrupt handler time considerations:

An interrupt handler should be kept as short and simple as possible, to prevent excessive latencies. Note that an interrupt in processing blocks any lesser and equal priority pending interrupts, therefore delaying their processing and introducing latency into the event handling.

Lecture Summary – Essential Things to Remember

- **Exceptions** provide a way to handle **events**, either internal or external.
- When an exception occurs, the execution is forced to go on from a fixed memory address, corresponding to the exception type. These addresses are called **exception vectors**, stored in so-called **vector table**.
- **Interrupts** form a special subset of exceptions. There are two types of interrupts: **normal (IRQ)** and **fast (FIQ)**.
- FIQs have **higher priority** then IRQs. If they occur simultaneously, the FIQ is **serviced first**. If a FIQ occurs during the IRQ servicing, it interrupts the IRQ service routine.