Topic 3:
I/O and Peripheral Devices Management

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

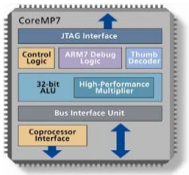
Content of Session

- Types of I/O and peripheral management
  - I/O interfaces and required management operations
  - Synchronization between I/O devices and microprocessors' CPUs
  - Prioritizing multiple interrupt sources: multi-level interrupts handling
- I/O management for ARM architectures
  - I/O peripheral subsystem in ARM microprocessors
  - Management of timers in the NDS using the *libnds* library
  - Use of timers and screen together in the NDS: designing a chronometer game

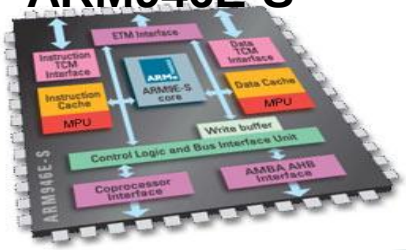
I/O and peripheral management on the NDS

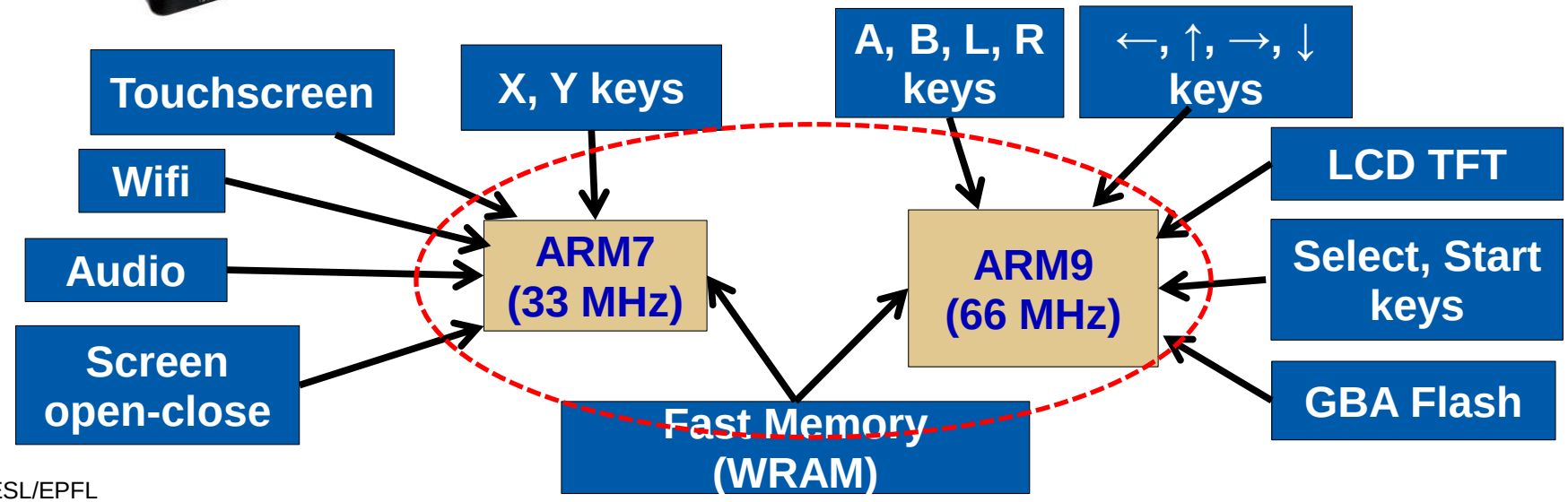
- Two 32-bit ARM cores manage the I/O and peripherals
  - ARM 946E-S: most of the keys, LCD TFT, GBA flash
  - ARM 7TDMI-S: sound, wifi, touchscreen, screen open-close and X-Ykeys

ARM7TDMI-S



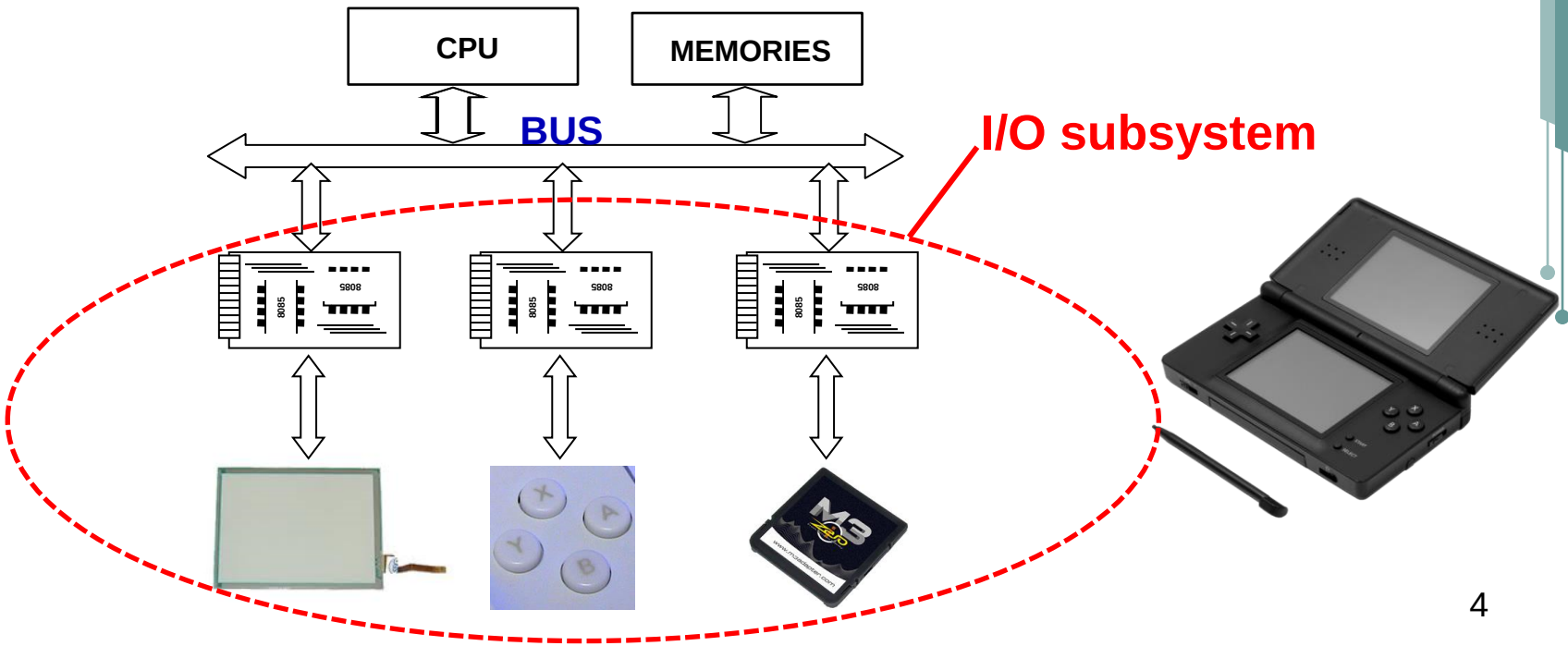
ARM946E-S





Interface to I/O subsystem and design considerations

- I/O subsystem enables the interaction of the CPU with the “external” world in Von-Neumann and Harvard computing architectures
  - Access through a bus (shared medium) to connect variable number of peripherals
- Three main design considerations
  - Request and transfer of operation and command/data with the CPU
  - Synchronization of component status (being utilized, ready or not)
  - Prioritize handling from different peripherals that request concurrent transfers

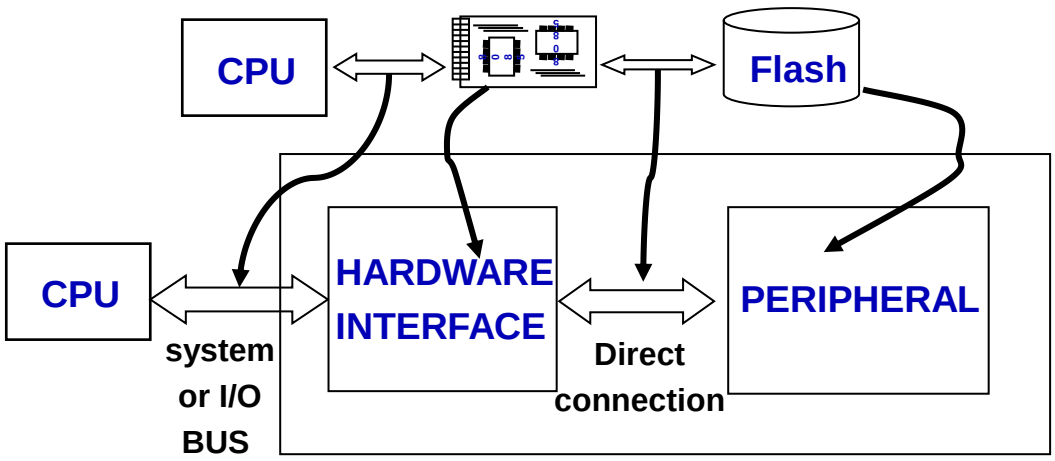


Basic functionality needed in I/O peripheral to process request from CPU

- Addressing: selection of I/O device for the microprocessor to do the transfer
- Transfer: data communication between microprocessor and peripheral
  - Type of data transfer
    - Read: microprocessor ← peripheral
    - Write: microprocessor → peripheral
  - Conversions between data formats
    - Electrical levels (TTL: 1 > 2.0V; 0 < 0.8V; RS-232-C: 1 < -3.0V; 0 > +3.0V)
    - Coding conversion (e.g., ASCII-Binary, float and double, etc.)
    - Serial-parallel vs. parallel-serial conversion
    - Analog-digital vs. digital-analog conversion
- Synchronization: transfer control mechanism for the microprocessor to know
  - If the peripheral is ready to receive and send data
  - If the peripheral has finished a transfer and it is ready to receive a new one

Implementation of I/O peripheral functionality: interfaces, drivers and libraries

- Peripherals are always connected with the microprocessor through a hardware interface (= hardware controller = adapter = I/O card)
  - Translation of CPU orders to the peripheral
- Software driver: defines the set of basic operations and configuration operations of the hardware interface (e.g., *read()*, *write()*, etc.)
  - Reads/Writes in a set of registers of the hardware interface
- Function library: set of high-level operations where each operation needs to call a group of basic operations of the driver (e.g., *printf()*)



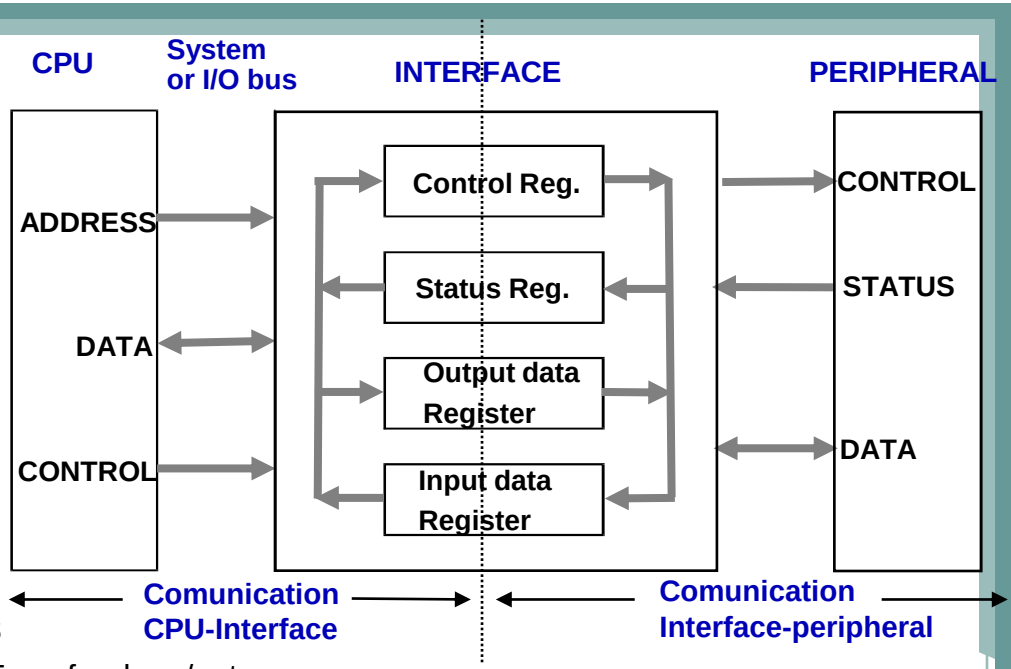
Example

CPU Orders (SW driver) → HW interface
- **Read N bytes starting from**
  - **Surface S**
  - **Cylinder C**
  - **Sector T**

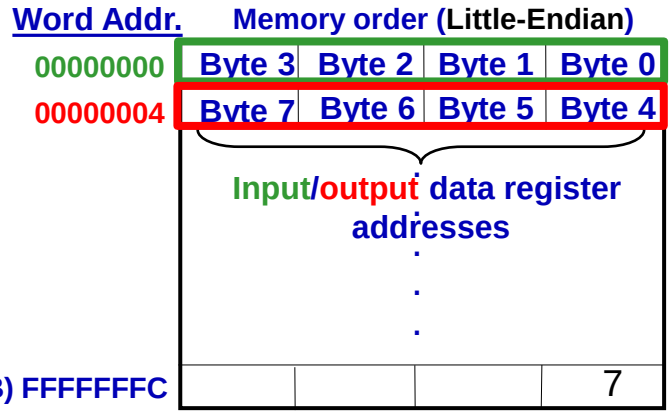
Orders HW Interface → peripheral
- **Position heads in cylinder C**
- **Position heads in sector T**
- **Select head of surface S**
- **Read N bytes**
- **Remove heads**

Structure of the I/O Interface

- Communication from CPU to peripheral requires four interface registers:
  1. Output data register
     - Used by CPU to write data to be sent
  2. Input data register
     - Used by peripheral to write data to be returned to the CPU
  3. Status register
     - Read by CPU to know peripheral status
       - Device ready/not; Data reg. full/empty; Transfer done/not
  4. Control register
     - Writen by the CPU to transmit orders to the peripheral
       - DRAM: read/write N bytes in row R, column C
       - Printers: print character, jump line, jump page, etc.



- Two options to access them:
  - Access using memory addresses: memory-mapped peripherals (ARM)
  - Special assembly instructions (x86)



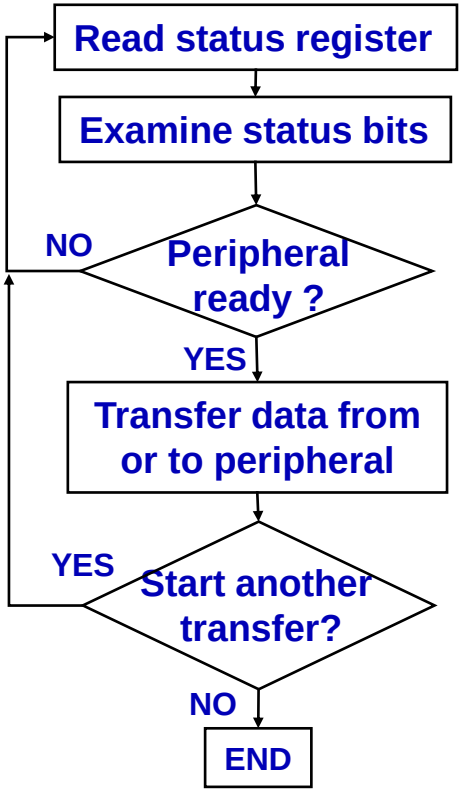
Synchronization I/O device and CPU

- Mechanism required each time the CPU wants to send/receive data to/from a peripheral in order to ensure that the device is ready to perform the transfer

- Two basic mechanisms for I/O subystem syncronization
  - Programmed I/O with active waiting response
  - I/O interrupts
    - Exceptions are high-priority interrupts in ARM processors (typically due to hardware errors or unexpected external system events)

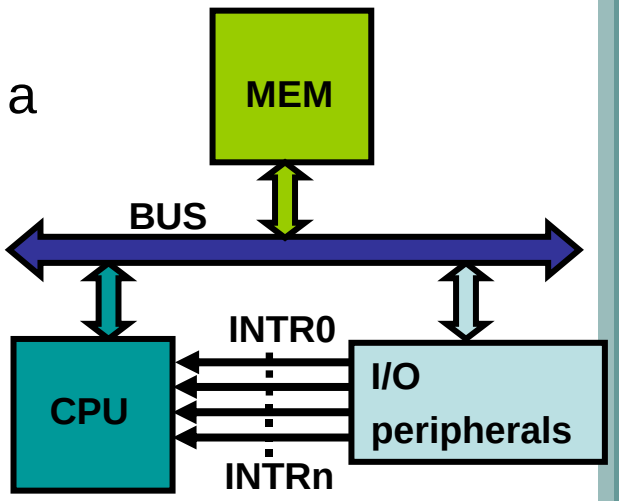
Synchronization I/O device and CPU: Programmed I/O with active waiting response

- Use of a loop in the CPU each time it wants to perform a transfer
  - Check continously in software the status of the peripheral until it becomes ready to perform the I/O transfer
- Problems
  - The CPU does not do useful work during the waiting loop
    - Slow peripherals make the loop repeat thousands of times
  - Program execution is stopped during I/O operations
    - E.g., in a videogame it is not possible to stop the game dynamism while user presses a key or moves the touchpad
  - Impossibility to attend multiple peripherals
    - While the CPU is waiting for a peripheral to get ready for a transfer, it is not possible to serve another peripheral

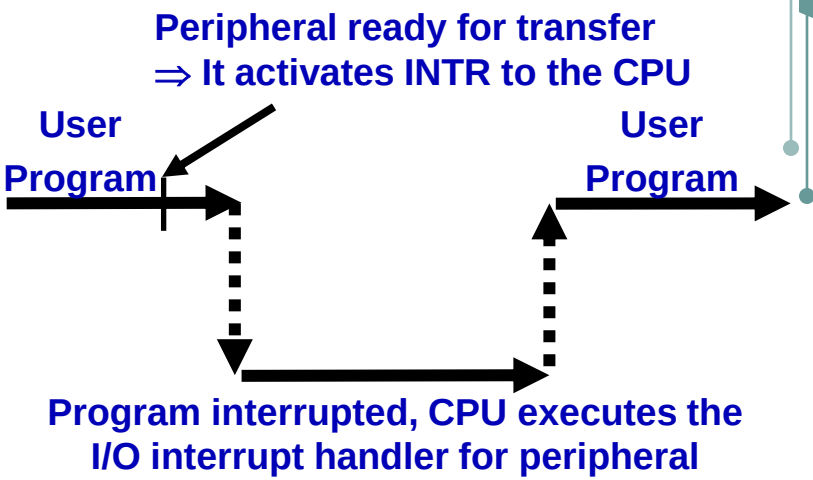


Synchronization I/O device and CPU: I/O interrupts

- I/O Interrupts/exceptions: peripheral indicates CPU when it is ready to perform a transfer, by activating a special dedicated line: *Interrupt Request Line* (*INTR*)
  - No waiting loop
  - **Multi-source interrupt:** Several INTR lines possibly arrive to the CPU, which decides which peripheral can interrupt the current program execution

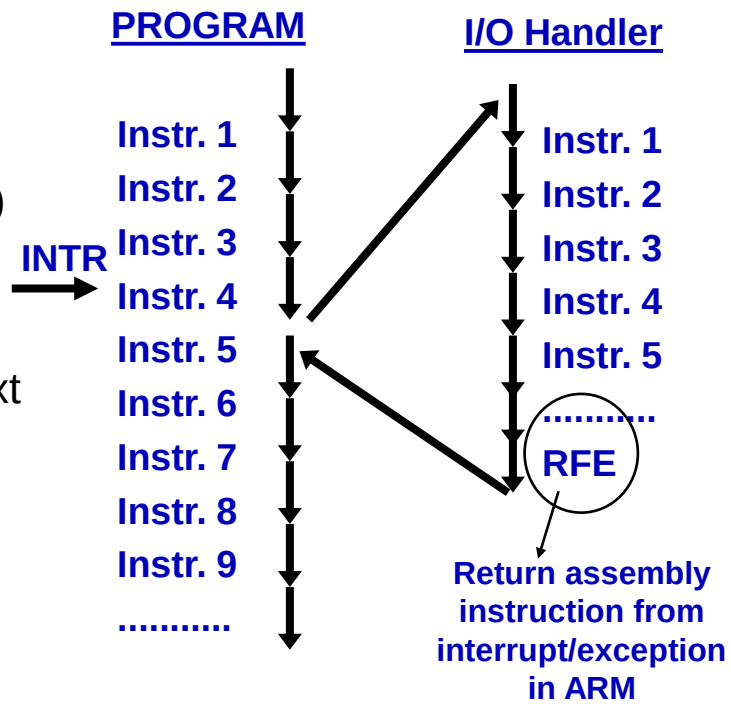


- Interrupts are managed using a special type of CPU function (set of assembly instructions): *Interrupt Handler*
  - Each time a CPU receives an interrupt request, it jumps to execute the interrupt handler after the current program assembly instruction
  - The interrupt handler performs the I/O operation and reconfigures the peripheral control register to get ready for next transfer
    - Handlers should last as little as possible



Synchronization I/O device and CPU: I/O interrupts vs. user functions

- Similarities
  - Both break the usual program sequence
    - We need to save the registers that we use inside the handler of the user function (***r0…rn***)
  - Both require saving program counter (***PC)*** in the stack:
    - At the end of both we need to return to the next instruction from where the program jumped



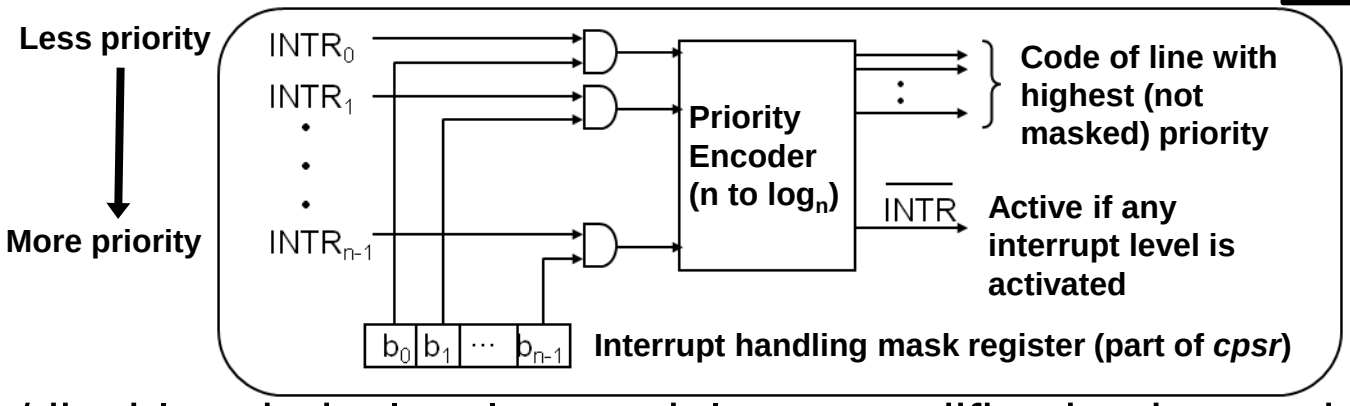
- Differences
  - An interrupt handler can run without the control of the system designer
    - A user function only runs when the program requests it
  - Interrupt handlers need to save the current program status register (***cpsr***) in the stack and restore it before returning to the program
    - Their execution status is not related to the user main program, while in functions it is related

Prioritizing multiple interrupt sources: Multi-level interrupts handling

- Multiple levels of interrupts: exception vectors table
  - Several INTR lines: one handler for each of them
  - Each level has a different priority according to how urgent handling the peripheral is (system designer decides)
- Use of *priority decoder* to handle priorities among lines
  - Handling always the highest priority device
    - Store in stack ***PC*** and ***cpsr*** if a higher priority request arrives
  - If two devices use the same priority: first-in first-serve policy

Exception vectors table

Address	
0x1C	Address handler INTR0
0x18	Address handler INTR1
0x14	Address handler INTR2
0x10	Address handler INTR3
0x0C	Address handler INTR4
⋮	
0x00	Address handler INTR$_{n-1}$



- Enable/disable priority levels: special *cpsr* modification instructions in ARM
  - Enabling or disabling certain bits in the ***interrupt handling mask register***
  - 1 mask bit (b_k) per level { If $b_k = 1 \rightarrow INTR_k$ level is enabled; If $b_k = 0 \rightarrow INTR_k$ level is disabled (or masked)

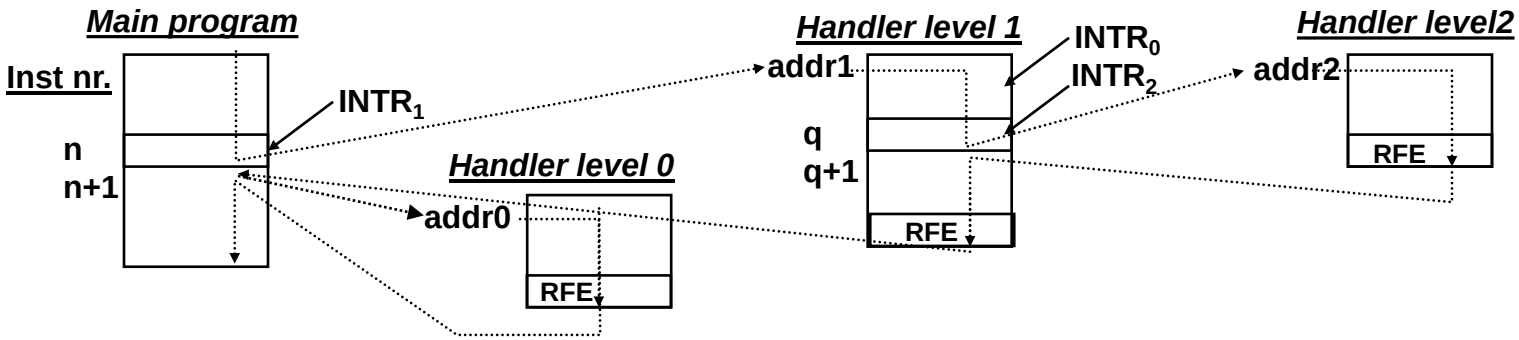
Example of multi-level interrupt handling

- System with 3 levels of interrupts { $INTR_0$, $INTR_1$, $INTR_2$ } ($INTR_0 < INTR_1 < INTR_2$)

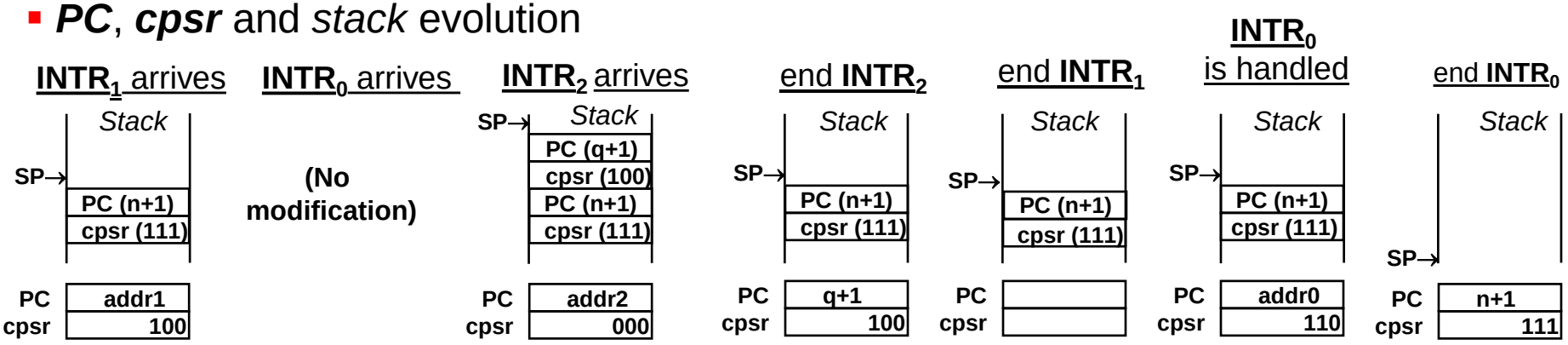
Control Program Status Register

cpsr: ... | b_2 | b_1 | b_0 → Mask bits { If $b_k = 1 \rightarrow INTR_k$ level enabled; If $b_k = 0 \rightarrow INTR_k$ level masked }

- Suppose that 3 interrupt requests arrive: $INTR_1$ - $INTR_0$ - $INTR_2$



- ***PC***, ***cpsr*** and *stack* evolution



I/O peripheral subsystem in ARM microprocessors

- I/O peripheral subsystem is memory-mapped
  - The peripheral ports are part of the 32-bit (4GB) memory address space

- Number of INTR lines/devices: ARM exception vectors table
  - Multiple devices can share the same INTR
  - Exception priorities (smaller number means higher priority)
    1. Reset
    2. Data Abort
    3. FIQ
    4. IRQ
    5. Reserved – Used only in ARM v6
    6. Prefetch Abort
    7. Undefined Inst / SWI (software interrupt, defined by the system designer)

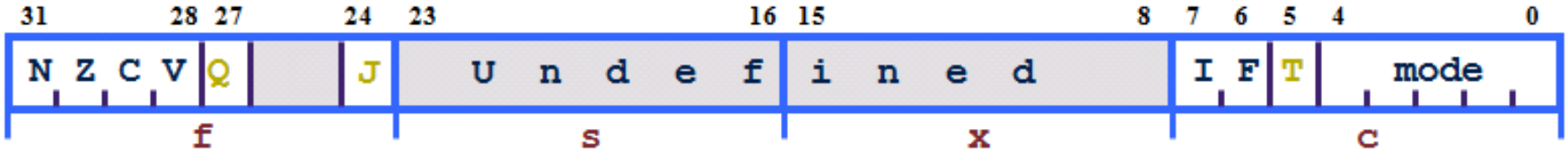
Exception vectors table

Address	
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruct
0x00	Reset

- In NDS we can only control one interrupt priority (*SWI*) for all peripherals
  - We define how we assign priorities between devices interrupts on the handler code
  - The rest of interrupt levels are all managed automatically by the NDS

All interrupt sources handled in the NDS using the *libnds* library methods

Program status registers (***cpsr***): Information about ARM microprocessor status



- Condition code flags (or bits)
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed

Managed automatically by NDS (*libnds* library)

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- Mode bits
  - Specify the processor mode

- **Sticky overflow flag - Q flag**
  - Architecture 5TE/J only (ARM9)
  - Indicates if saturation has occurred
- **T Bit**
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - Architecture 5TEJ only (ARM9)
  - J = 1: Processor in Jazelle state

Sources of interrupts in the NDS

- 25 sources:
  - Graphic (x2)
  - Timer (x4)
  - DMA (x4)
  - Keypad
  - GBA Flashcard
  - FIFO
  - DS Card
  - GFX FIFO
  - Power Management
  - SPI
  - WIFI
  - …

Examples of interruption events (IRQ_MASK) defined in the *libnds* library

Libnds	Description
IRQ_VBLANK	Vertical blank
IRQ_TIMER0	Timer 0
IRQ_KEYS	Keypad
IRQ_WIFI	WIFI
IRQ_DMA0	DMA 0

All of them can be consulted in the VM in *`/opt/devkitPro/libnds/include/nds/interrupts.h`*

How to handle NDS interruptions: *libnds* application programmer interface (API)

- Initialize interrupts subsystem
  - void irqInit();
    - Usual way to initialize peripherals **(except when using sound)**
- Specify the handler to use for the given interrupt
  - void irqSet(IRQ_MASK irq, VoidFunctionPointer handler);
- Remove the handler associated with the interrupt
  - void irqClear(IRQ_MASK irq);
- Allow the given interrupt to occur
  - void irqEnable(uint32 irq);
- Prevent the given interrupt from occurring
  - void irqDisable(uint32 irq);

Example of interruption in libnds API: Handling a key interrupt

```
//Handler function
void keys_ISR () {
printf ("a key was pressed");
}

//main function
int main(){
irqInit();
irqSet(IRQ_KEYS, &keys_ISR);
irqEnable(IRQ_KEYS);
…
irqClear(IRQ_KEYS);
irqDisable(IRQ_KEYS):
}
```

Keypad Handler function

Initializing the interruption

Mapping the keypad **interruption to its handler**

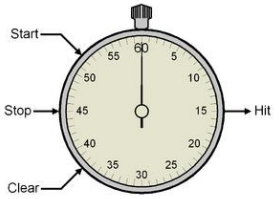
Enabling the keypad interruption

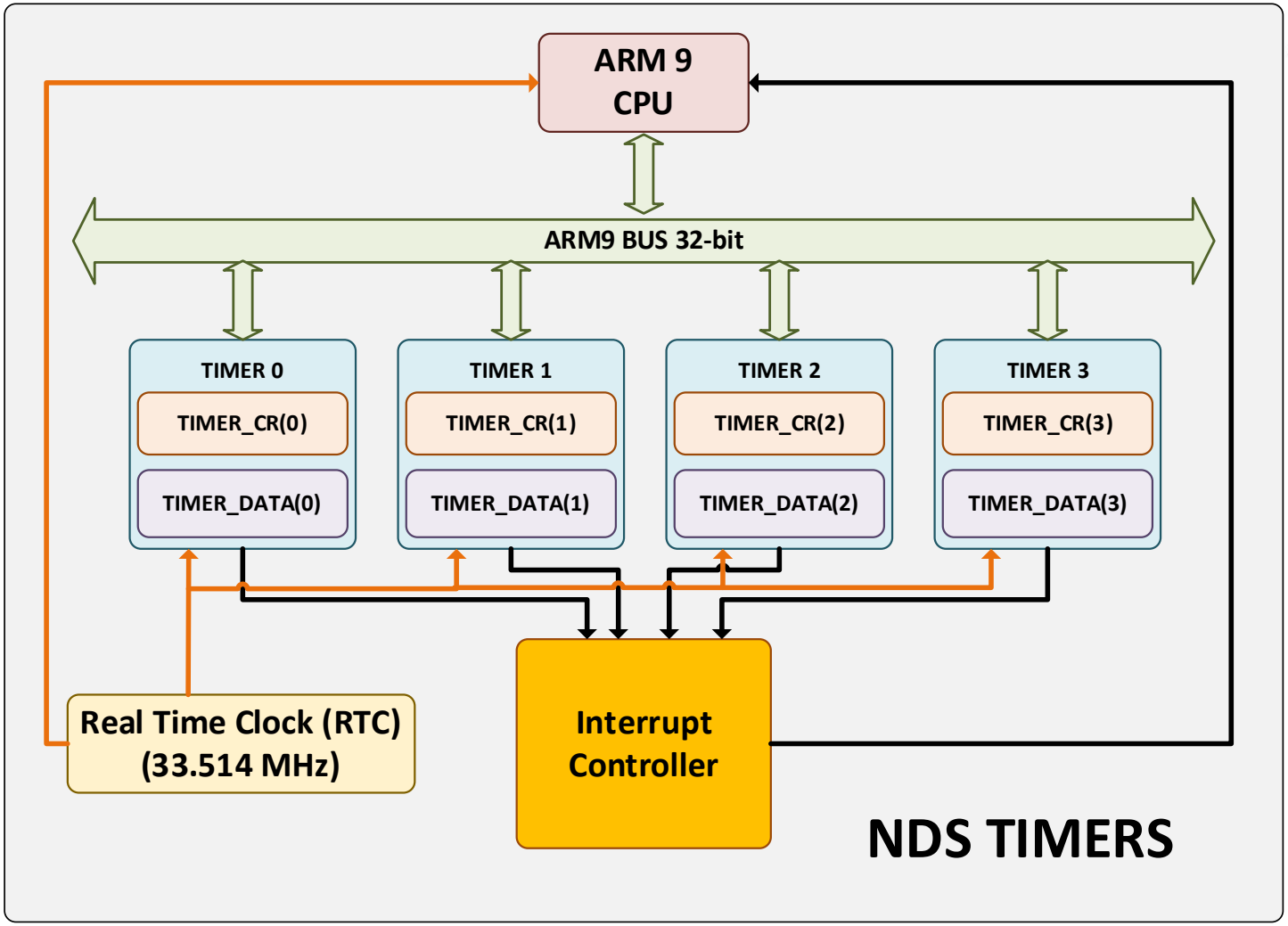
Removing *key_ISR* handler. (Keypad still triggers interrupt)

Keypad stops triggering interrupt

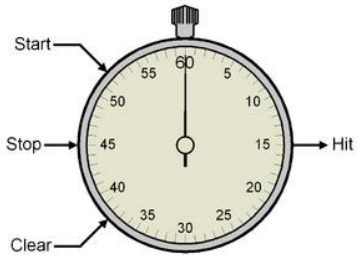
Timers in NDS

- 4 timers available in the NDS





Handling interrupts using the libnds API: the Timers



- 4 timers available in the NDS
- Libnds provides several macros to deal with timers:
  - TIMER_CR(n)
    - Returns a de-referenced pointer to the timer control register *n*
  - TIMER_DATA(n)
    - Returns a de-referenced pointer to the data register for timer *n*
  - TIMER_ENABLE
    - Enable the timer
  - TIMER_DIV_VALUE, with VALUE= 1, 64, 256, or 1024
    - Timer will count at (33.514 / VALUE) MHz
  - TIMER_FREQ_VALUE (freq), with VALUE= 64, 256, or 1024
    - Set up the register value to start and overflow each 1/freq second
  - TIMER_IRQ_REQ
    - Timer will request an interrupt on overflow (see TIMER_FREQ_VALUE (freq))

Handling interrupts using the libnds API: the Timers

- Basic macros that *libnds* provides to deal with timers:

- Macros to access specific timer registers

  - ***TIMER_CR(n)***

    Returns timer control register for timer *'n'*

  - ***TIMER_DATA(n)***

    Returns data register for timer '*n*'

- Macros to set up the registers

  - **TIMER_DIV_1**
  - **TIMER_DIV_64**
  - **TIMER_DIV_256**
  - **TIMER_DIV_1024**

  Timer will count at (33.514/number) MHz

  - **TIMER_ENABLE**

  Enable timer

  - **TIMER_IRQ_REQ**

  Timer will request an interrupt on overflow

  - **TIMER_FREQ(n)**
  - **TIMER_FREQ_64(n)**
  - **TIMER_FREQ_256(n)**
  - **TIMER_FREQ_1024(n)**

  Compute starting point of the register for a given frequency 'n'

Interruption: libnds API (Timer Example)

- How to use these macros ?

  Example:

```
TIMER_CR(0) = TIMER_ENABLE | TIMER_DIV_64 | TIMER_IRQ_REQ ;
```

Access to Timer 0 control register

Enable Timer 0

Timer 0 will count at (33.514 / 64) MHz

Timer 0 will get triggered on overflow

```
TIMER_DATA(0) = TIMER_FREQ_64 ( 125 ) ;
```

Access to Timer 0 data register

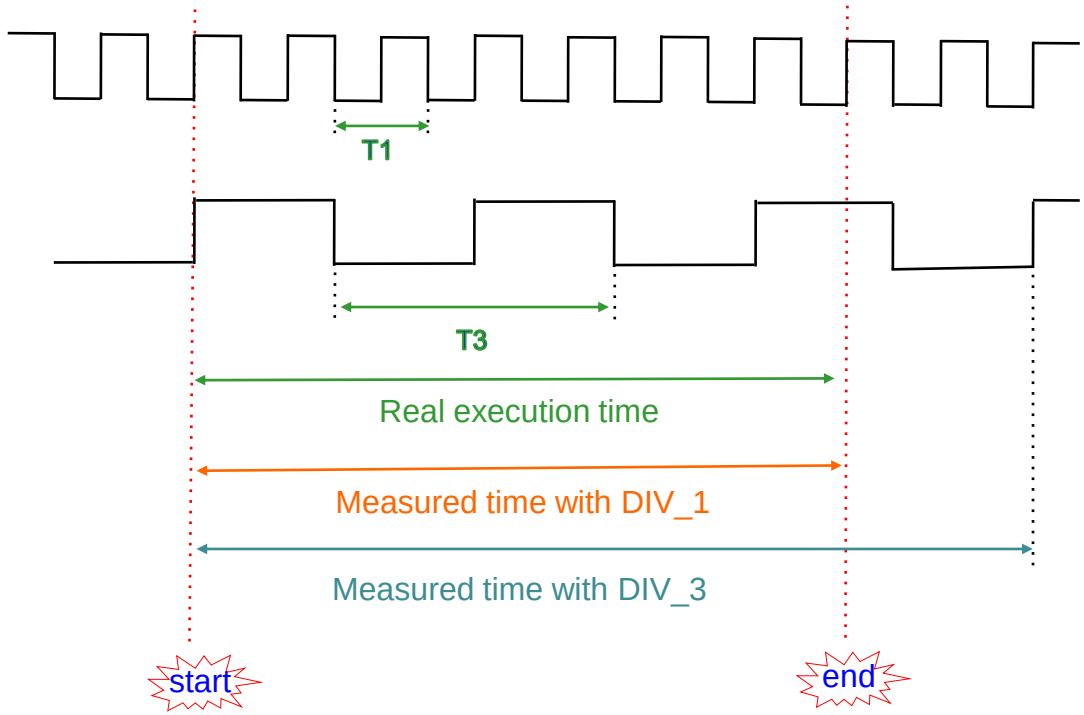
Set up the register value (in Hz) to a starting point which will make it overflow each 1/125 = 0.008 second

Interruption: libnds API (Timer Example)

- Example configuring two timers: DIV_1 and DIV_3

DIV_1 : F1 = 33.514 MHz
T1 = 1/F1 second

DIV_3 : F3 = 11.171 MHz
T3 = 1/F3 second



Higher frequency
↓
Higher resolution
↓
More accurate timer

	DIV_1	DIV_3
end - start = nr. of periods	7	3
Measured execution time	7 * T1 = 0.208 µs	3 * T3 = 0.268 µs

Interruption: libnds API (Timer Example)

- TIMER_DATA(0) is incremented each TIMER0 tick
  - Maximum measured ticks = $2^{16} - 1$
  - An interrupt is fired on overflow

	DIV_1	DIV_1024
Maximum measured time	$(2^{16} - 1) / 33514000 = 1.9ms$	$(2^{16} - 1) * 1024 / 33514000 = 2s$

- How to make it fire an interrupt each **X** seconds for **DIV_1024** ?

X = (number of ticks) * period

(number of ticks) = X / period = X * frequency

TIMER_DATA(0) = $(2^{16} - 1)$ – (number of ticks)

Equivalent

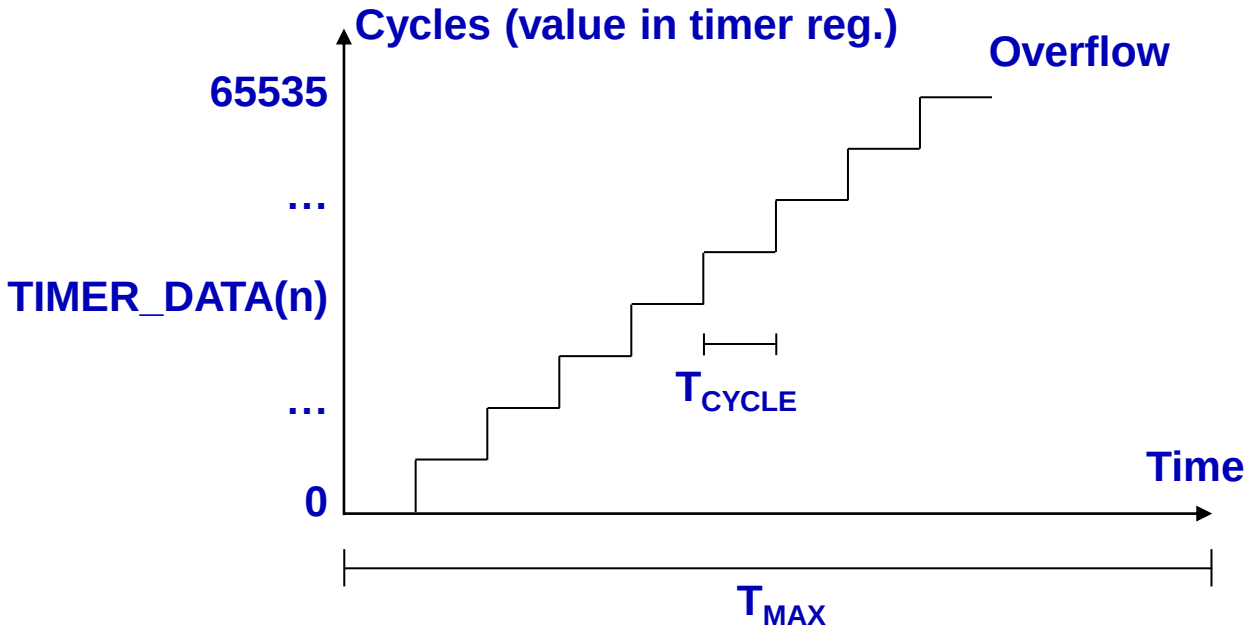
TIMER_DATA(0) = TIMER_FREQ_1024(Y) ;

Y = 1 / X (in Hz)

- Interrupt each 1s
  Y = 1 / 1 = 1
- Inter. each 100ms
  Y = 1 / 0.1 = 10

Possible Timer Resolutions in libnds API

- Which values are possible to use with TIMER_FREQ_X(n)?



Divider	$F_{CYCLE}=(33.514/DIV)MHz$	$T_{CYCLE}=1/F_{CICLE}$	$F_{MIN}=F_{CYCLE}/2^{16}$	$T_{MAX}=2^{16}\ T_{CYCLE}$
TIMER_DIV_1	33.514 MHz	29.838 ns	511.383 Hz	1.955 ms
TIMER_DIV_64	523.656 kHz	1.910 us	7.990 Hz	125.151 ms
TIMER_DIV_256	130.914 kHz	7.639 us	1.998 Hz	500.603 ms
TIMER_DIV_1024	32.729 kHz	30.554 us	0.499 Hz	2.002 s

Practical Work 5: Interrupts in the NDS to control multiple Timers

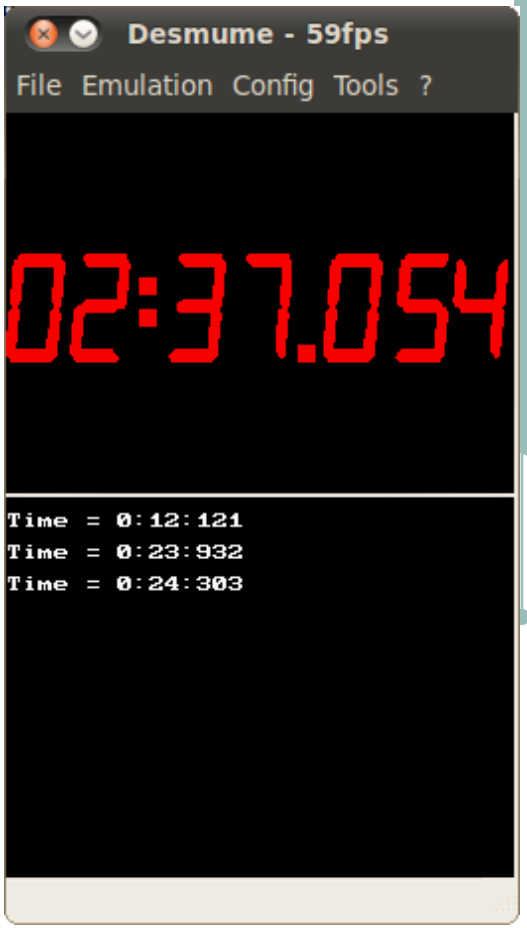
- Measuring Time
  - Precise way to measure time using the timers

```
int result, number;

result = floor(sqrt((double) number));

result = iSqrt(number);
```

  - Which function takes more time?
  - How much?



- Need to handle interrupts for accuracy!
  - Timers, Graphic (VBlank), Keys….
- Can we use them to create a small game?

Practical Work 5: Interrupts in the NDS to control multiple Timers

- Exercises (and homework)
  - Exercise 1 – Measure time using timers
  - Exercise 2 – Implementing a chronometer using 2 timers' interrupt
  - Exercise 3 – Implementing a chronometer using 1 timer
  - Exercise 4 – Refreshing the screen properly using graphic interrupts
  - Exercise 5 – Changing the color of the clock periodically
  - Exercise 6 – Printing lap time with the keys

  - Additional Exercise:
    - **Time challenge game**

Questions?

Let’s use the ARM interrupts to create a chronometer in NDS!