DESN2000: Engineering Design & Professional Practice (EE&T)

Week 7

Procedural call standard

Input and output interfaces

David Tsai
School of Electrical Engineering & Telecommunications
Graduate School of Biomedical Engineering
d.tsai@unsw.edu.au

Biomedical Microsystems Lab

This week

- Function call examples
- Input / output background
- Polling
- Interrupts
- LPC2478 microcontroller

Revision: AAPCS

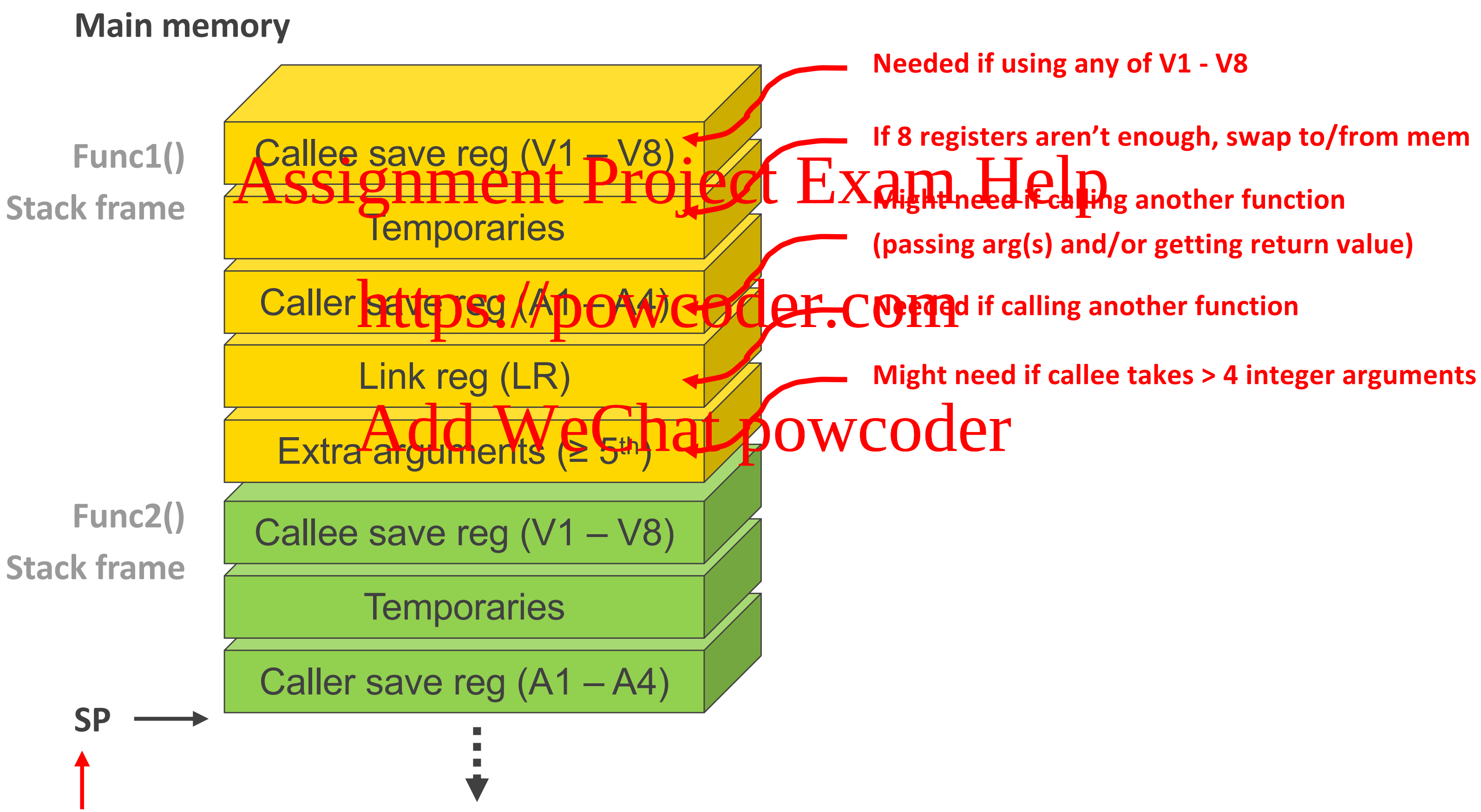
- Caller's rights:
  1. Use V1 – V8 freely.
  2. Assume that the return values are placed in A1 – A4 by the callee.
- Callee's rights:
  1. Use A1 – A4 freely.
  2. Assume that the arguments are available in A1 – A4 (additional arguments are on stack).

Revision: AAPCS

- Caller's responsibility:
  1. Save LR before BL.
  2. Save A1 – A4 if these registers are used for any operations after BL to callee. Because callee might modify them.
  3. Place first 4 arguments in A1 – A4. Use stack if more than 4 arguments, e.g.:
     - 5^{th} at [SP, #0]
     - 6^{th} at [SP, #4]
- Callee's responsibilities
  1. Save V1 – V8 before using them and restore the original values before returning.
  2. If not void, place return values in A1 – A4.
  3. Return to caller by performing `MOV PC, LR`.

Summary of stack frame

- Func1() called Func2(), processor is executing Func2(). Stack frame might look like the following, in the most general case:



Stack pointer advances with function call, rewinds when returning.

Assembly example

- Build a recursive function to compute the nth Fibonacci Number.
- The Fibonacci numbers *F*(*n*) are defined as

$$F(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- Implementation in C

A1 A1


```
int fib(int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

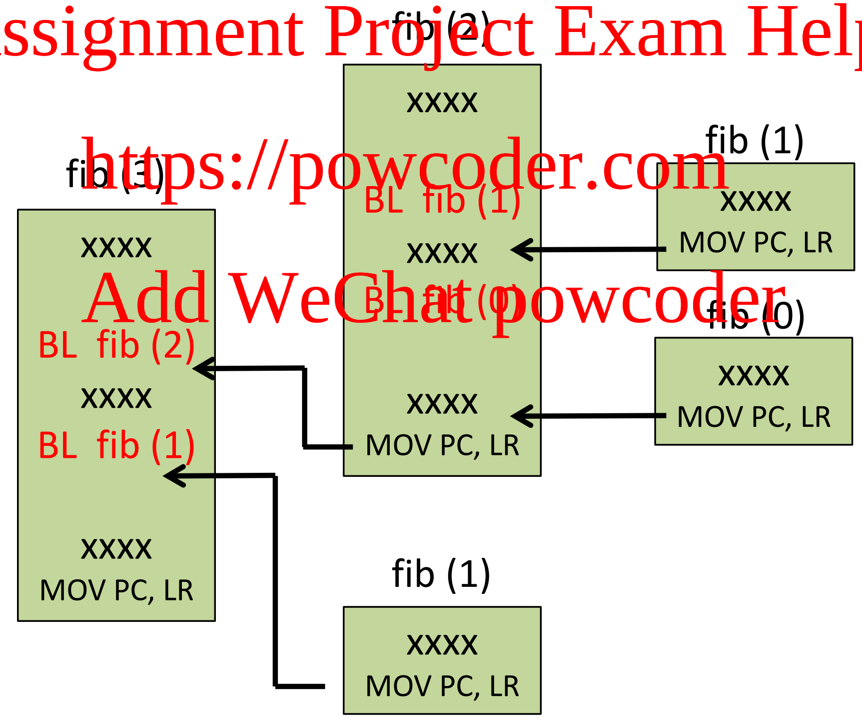
1st A1 ← (n-1)   2nd A1 ← (n-2)

- A recursive function – calls itself repeatedly until leaf conditions (n = 0, n = 1) are reached.

Assembly example

- How does the recursion work?

Example:
F(3) = F(2) + F(1)
= F(1) + F(0) + 1
= 1 + 1 + 1 = 3



Assembly example

- Being AAPCS-compliant is critical for recursive functions, to avoid resource conflicts with nested function calls.
- **Step 1: Identify registers to be saved and frame size.**
  - Save LR, because *F(n)* calls *F(n-1)* and *F(n-2)*.
  - Save V1 for intermediate result computation.
  - Save A1 to pass argument to *F(n-1)* and *F(n-2)*, and to return result.
  - Therefore a stack frame of 3 words.

  - So assembly code must have:

```
STR  LR, [SP, #-4]!
STR  V1, [SP, #-4]!
...
STR  A1, [SP, #-4]!
```

NOTE: this line appears in function body

  - Identically:

```
STMFD SP!, { A1, V1, LR }
```

Assembly example

$$F(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ \underline{F(n-1) + F(n-2)} & \text{otherwise} \end{cases}$$

- **Step 2.1: Implement the function body**
  - Starting with the leaf cases
  - Assembly:

```
fib_body    CMP    A1, #0          ; if (n==0)
            CMPNE  A1, #1          ; if (n==1)
            MOVEQ  A1, #1
            BEQ    fib_fin         ; return 1
```


- **Step 2.2: Implement the function body**
  - The recursive part
  - Assembly:


```
            STR    A1, [SP, #-4]!  ; need A1 after BL
            SUB    A1, A1, #1      ; A1 = n-1
            BL     fib
            MOV    V1, A1          ; save ret value
            LDR    A1, [SP], #4    ; restore A1
            SUB    A1, A1, #2      ; A1 = n-2
            BL     fib             ; fib(n-2)
            ADD    A1, A1, V1      ; A1 = fib(n-1) + fib(n-2)
```

ret. F(n-2) F(n-1)

Assembly example

- **Step 3: Returning**
  - Removing stack frame.
  - Placing return value in A1.
  - Adjust PC, leaving fib().

  - So assembly code must have:

```
LDR   A1, [SP], #4    ; restore A1
...
LDR   V1, [SP], #4
LDR   LR, [SP], #4
MOV   PC, LR
```

NOTE: this line appears in function body

  - Identically:

```
LDMFD SP!, { A1, V1, LR }
```

Assembly example

- The complete assembly program

```
fib         STR   LR, [SP, #-4]!
            STR   V1, [SP, #-4]!

fib_body    CMP    A1, #0           ; if (n==0)
            CMPNE  A1, #1           ; if (n==1)
            MOVEQ  A1, #1
            BEQ    fib_fin          ; return 1

            STR    A1, [SP, #-4]!   ; need A1 after BL
            SUB    A1, A1, #1       ; A1 = n-1
            BL     fib
            MOV    V1, A1           ; save ret value
            LDR    A1, [SP], #4     ; restore A1
            SUB    A1, A1, #2       ; A1 = n-2
            BL     fib              ; fib(n-2)
            ADD    A1, A1, V1       ; A1 = fib(n-1) + fib(n-2)

fib_fin     LDR    V1, [SP], #4
            LDR    LR, [SP], #4
            MOV    PC, LR
```

Function name

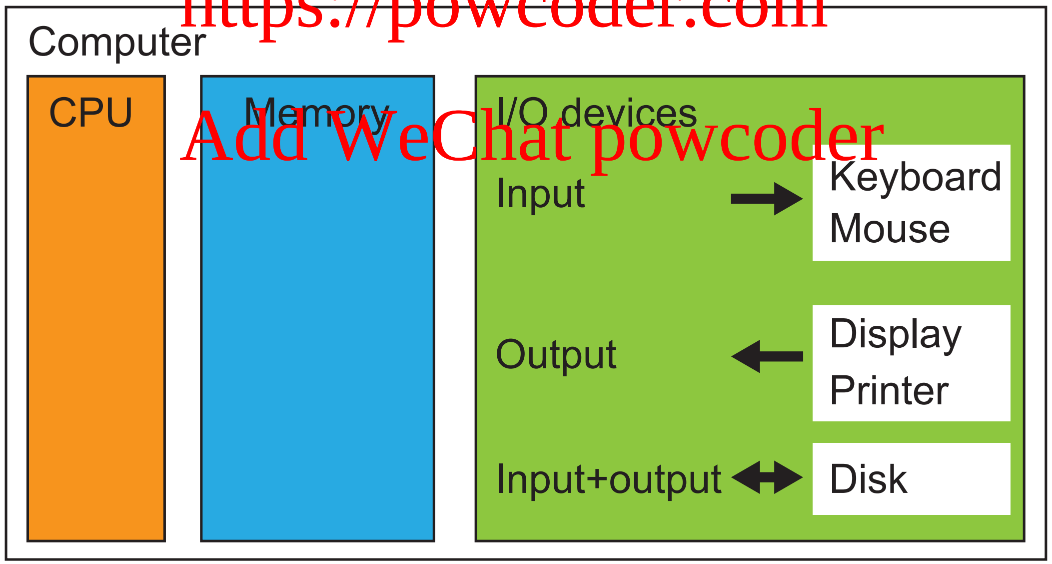
Function call housekeeping

Work

Function call housekeeping

Input / output background

- Why I/O devices?
  - Human interacting with computers.
  - Computer interacting with environment.



Input / output examples

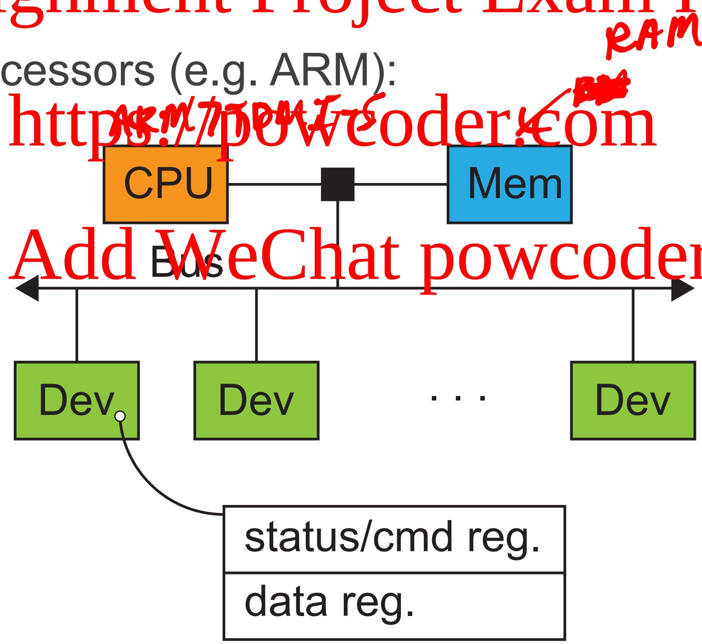
- I/O speed: bytes transferred per second.
- Wide range of data rates:

Device	Behaviour	Partner	Data rate (kB/s)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Line printer	Output	Human	1
Laser printer	Output	Human	100
Magnetic disk	Storage	Machine	100,000
Network-LAN	I or O	Machine	1,000,000
Graphics display	Output	Human	8,000,000

What’s required to make I/O work?

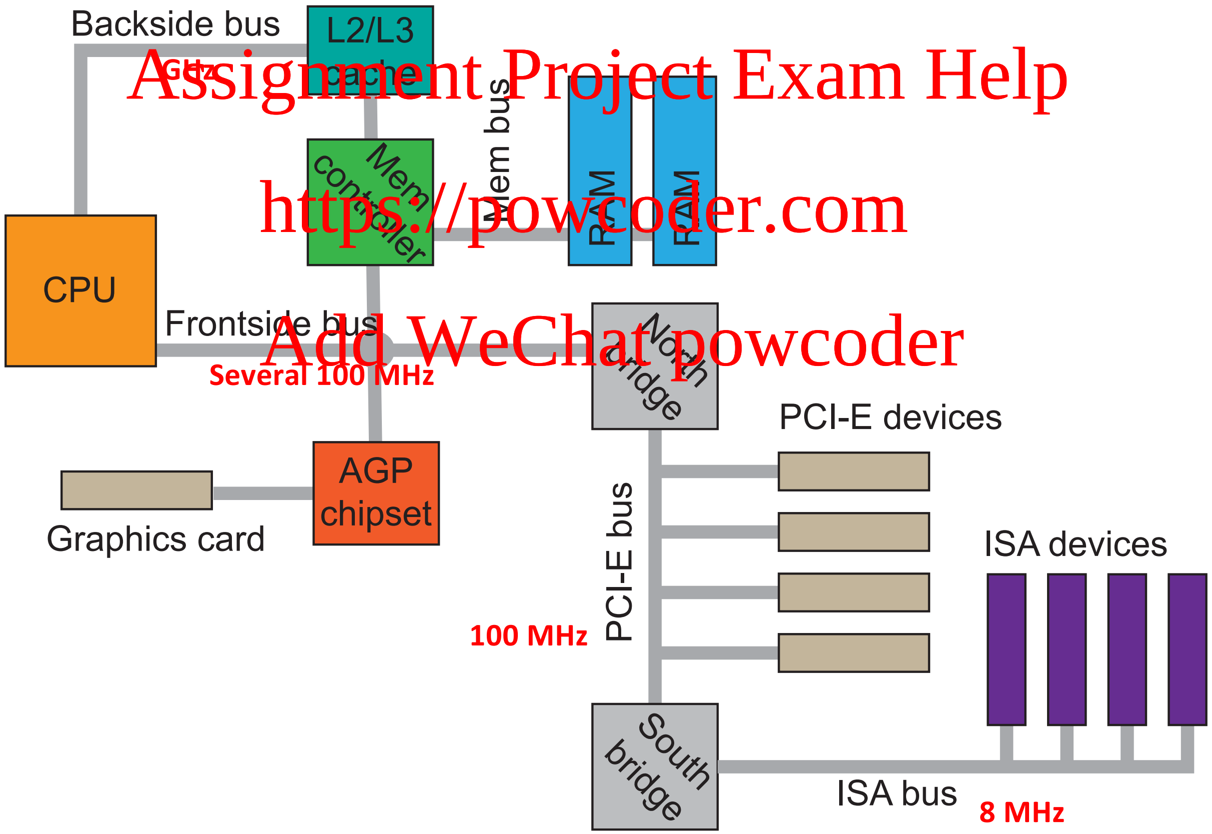
- A way to:
  - **connect** many device types to the processor and memory.
  - **control** these devices, respond to them, and transfer data.
  - **present** these devices to user programs.

- For embedded microprocessors (e.g. ARM):



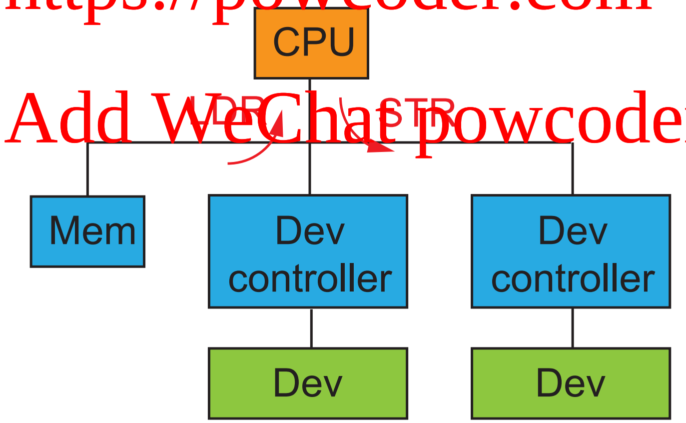
Buses in Intel x86 PC

- Use bus hierarchy – reduces latency.
  - Fastest ones closest to the CPU, slow ones are physically distant.
  - Similar devices clustered on the same bus.



Accessing devices from CPU

- Model 1: **dedicated I/O instructions**.
- Model 2: **memory mapped I/O** (used by ARM):
  - A portion of the address space is dedicated to I/O paths.
  - Input: read a sequence of bytes
  - Output: write a sequence of bytes



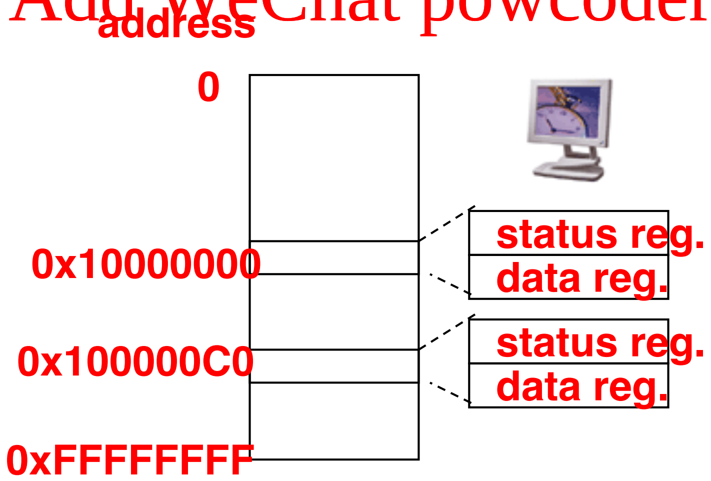
Memory mapped I/O

~4 GB

32 addr space

2^{32}

- I/O devices have registers for
  - Status / control
  - Data
- These registers have interfaces similar to memory and can be connected to the memory bus.
- Reading / writing "special" memory locations produces the desired change(s) in the I/O device controller.
- Typically, devices map to only a few bytes in memory.



Processor & I/O speed mismatch

- A 500 MHz microprocessor can execute 500 million load / store instructions per sec (2,000,000 kB/s data rate).
- I/O device might be 0.01 kB/s (e.g. keyboards).

- Input: device may not be ready to send data as fast as the processor loads it. E.g. waiting for human inputs.
- Output: device may not be ready to accept data as fast as processor stores it.

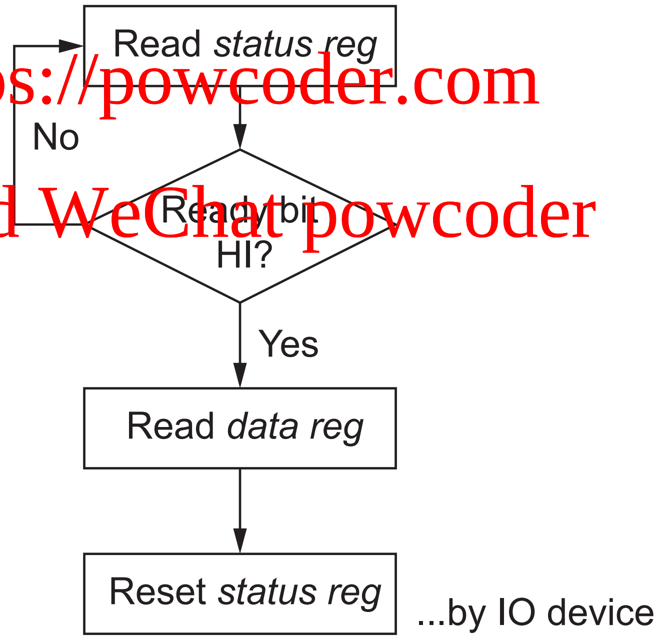
- Need to address the big speed mismatch.

  1. **Polling I/O**
  2. **Interrupt-driven I/O**

Accessing devices: polling

- Path to device generally has 2 registers:
  - **Status Register:** says it's OK to read/write (I/O ready).
  - **Data Register:** data resides here.

- Polling procedure:



Accessing devices: polling

- Polling can be expensive.
- Assuming a 500-MHz processor taking 400 clock cycles for a polling operation (calling poll routine, accessing the device and returning). Determine % of processor time for polling these devices:
  1. Mouse: polled 30 times/sec so as not to miss user movement
  2. Hard disk: transfers data in 16-byte chunks and can transfer at 8 MB/second. No transfer can be missed.

Accessing devices: polling

- Mouse
  - Polling clocks/sec = 400 clocks/sec × 30 = 12000 clocks/sec
  - % of processor time for servicing device: $12\times10^3 \div 500\times10^6$ = **0.002%**
  - **Small impact to processor** (indeed a common strategy). 500MHZ

- Hard disk
  - Times polling disk/sec = 8 MB/s ÷ 16B = 500K polls/sec

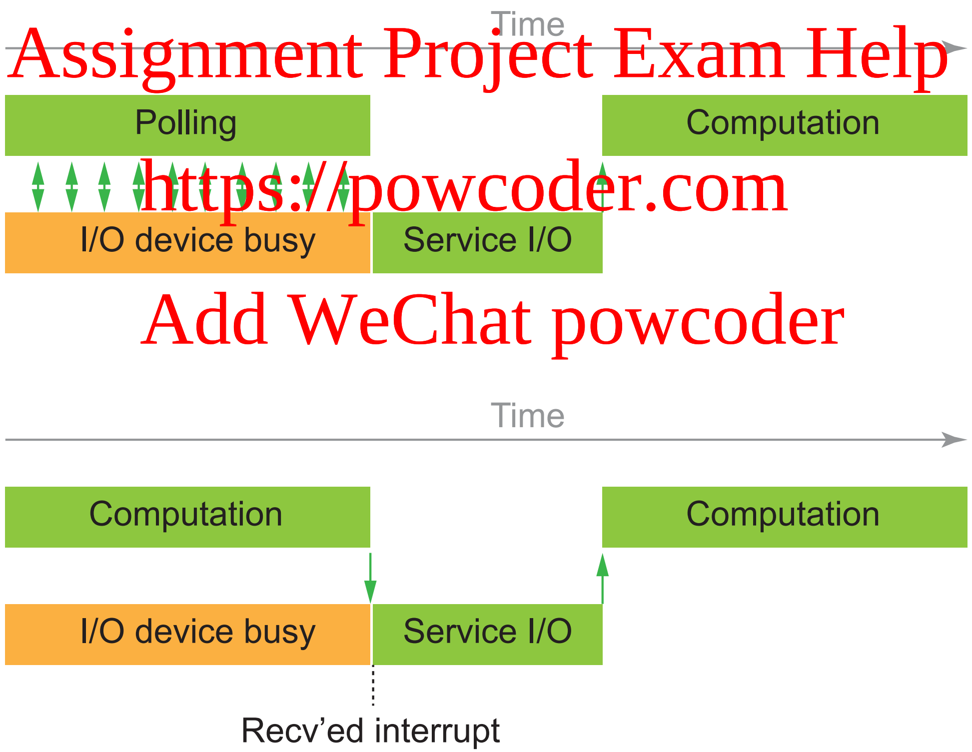
  - Polling clocks/sec = 400 clocks/sec × 500K = 200,000,000 clocks/sec
  - % of processor time for servicing device: $200\times10^6 \div 500\times10^6$ = **40%**

  - **Unacceptable**! 500MHZ

Accessing devices: interrupt-driven

- Wasteful spending so much time spin-waiting for I/O.
- Solution: use an interrupt mechanism – notify CPU only when I/O is ready. Freeing up the CPU to do work in parallel.

* Polling I/O



* interrupt I/O. parallelism.

*

Accessing devices: interrupt-driven

- Hard disk: transfers data in 16-byte chunks at 8 MB/second. No transfer can be missed … as before

1 500 clock cycle overhead per transfer, including interrupt (100 more than before, for interrupt mech). Find the % of processor consumed if the hard disk is **only active 5% of the time**. 2

- When disk is active: interrupt rate = polling rate
- Disk interrupts / sec = 8 MB/s ÷ 16B = 500K interrupts/sec

- Disk Polling Clocks/sec = 500 × 500K = 250,000,000 clocks/sec

100% HDD usage.

- % of processor time for servicing device **during transfer**: $250\times10^6 \div 500\times10^6$ = **50%**

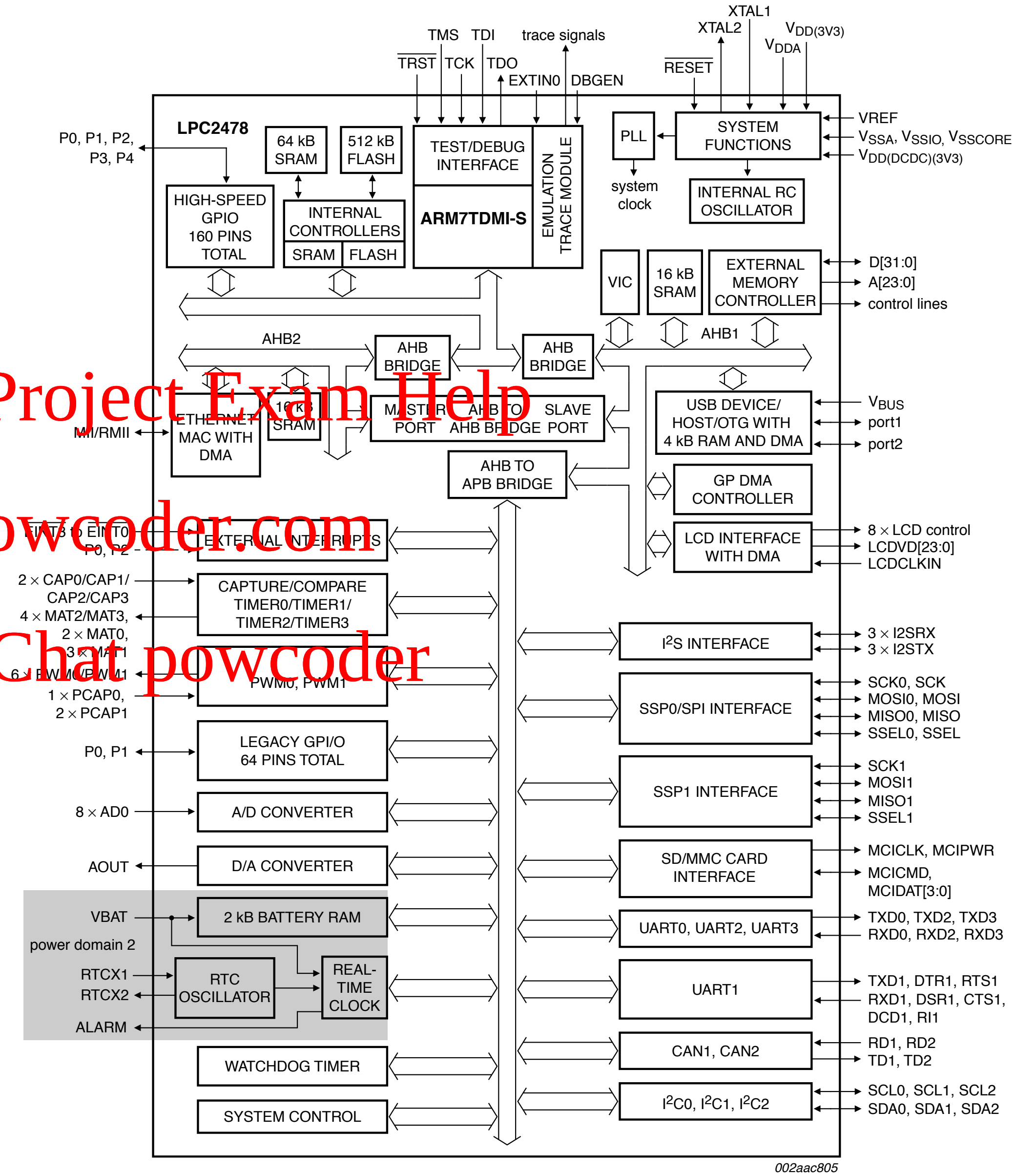
- **Average** % of processor time for servicing device: disk active 5% of time, so 5% × 50% = **2.5%**

Memory-mapped IO on LPC2478

- Specific memory addresses correspond to registers, which are responsible for driving actual I/O pins on the microcontroller.
  - Write to specific memory addresses to provide outputs
  - Read specific memory addresses to get inputs.
- The ARM architecture use memory-mapped I/O.

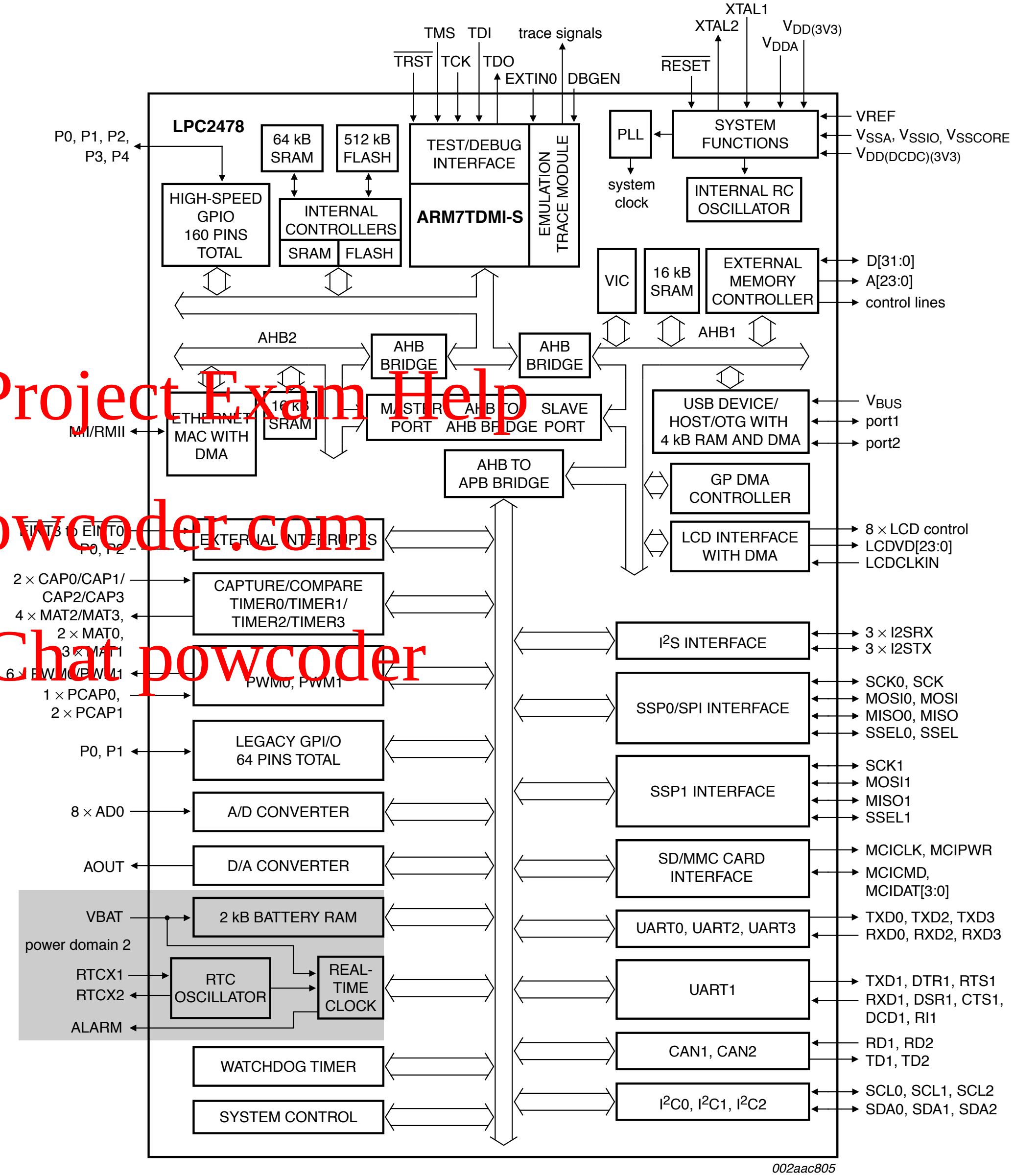
Memory-mapped IO on LPC2478

- ARM7TDMI-S processor.
- 72 MHz clock.
- 512kB on-chip flash memory.
- 32-bit ARM and 16-bit Thumb instructions.
- 10/100 Ethernet controller.
- USB2 device controller.



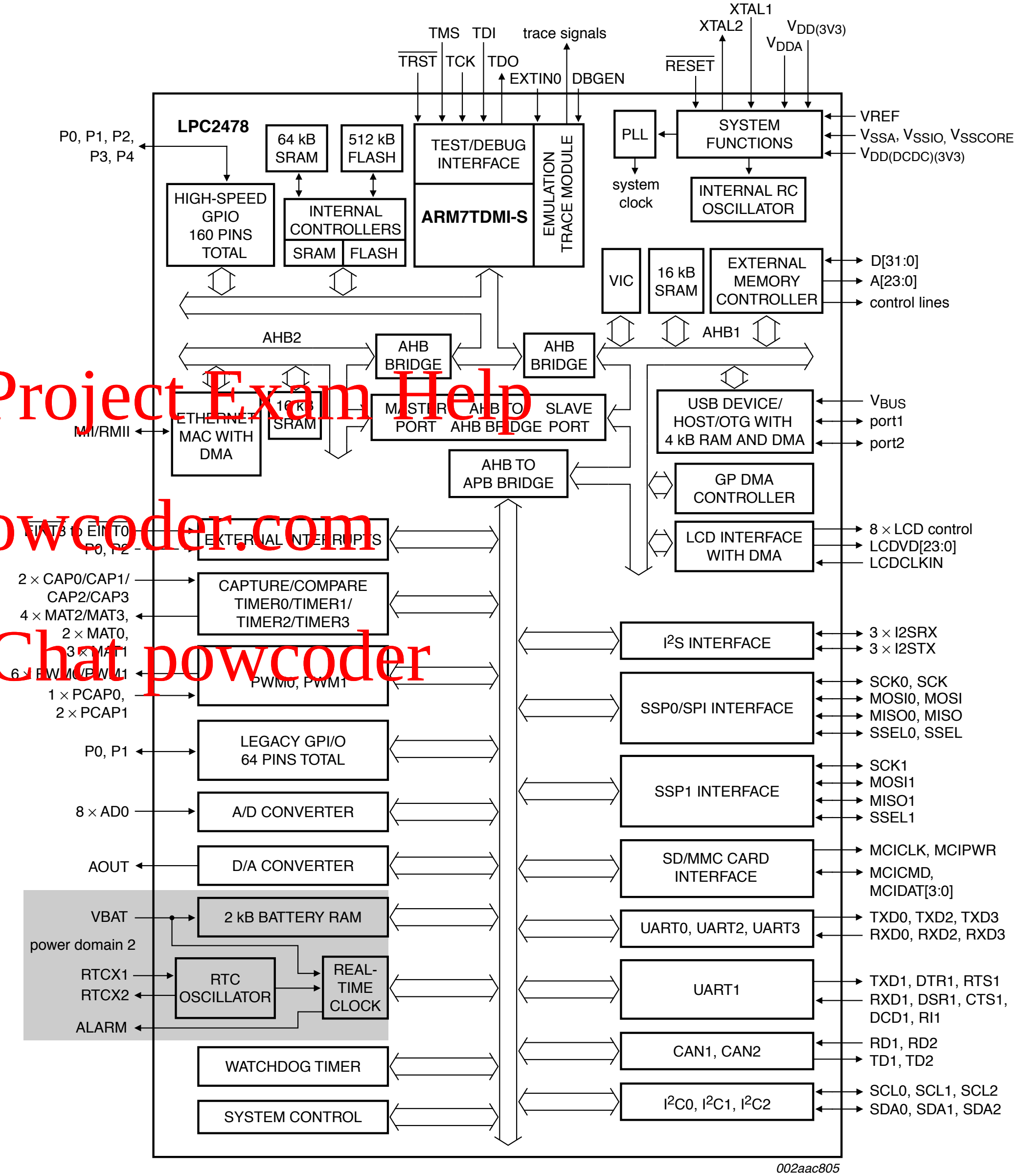
Memory-mapped IO on LPC2478

- 2 CAN (controller area network) channels.
- 2 PWM units.
- 3 I²C interfaces.
- I²S (inter-IC sound) interface.
- 2 SSP (synchronous serial ports).
- SPI (serial peripheral interface) port.



Memory-mapped IO on LPC2478

- 160 high-speed GPIO (general purpose IO) pins.
- 64 legacy GPIO pins.
- 4 UART (universal asynchronous receiver-transmitter).
- 10-bit D/A converter.
- 10-bit A/D converter.
- 4 timers.
- LCD interface.



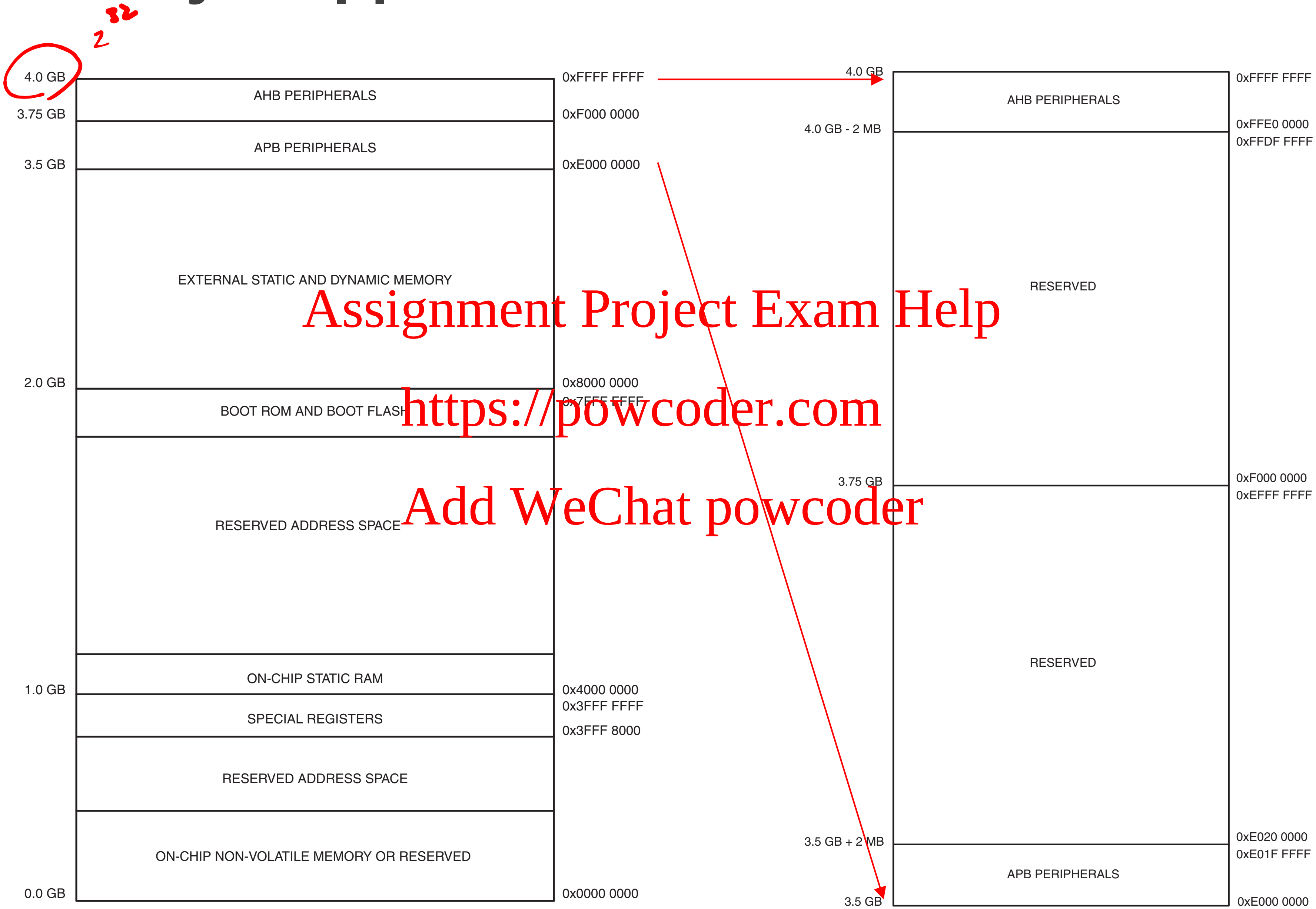
Memory-mapped IO on LPC2478

Table 16. LPC2468/78 memory usage and details

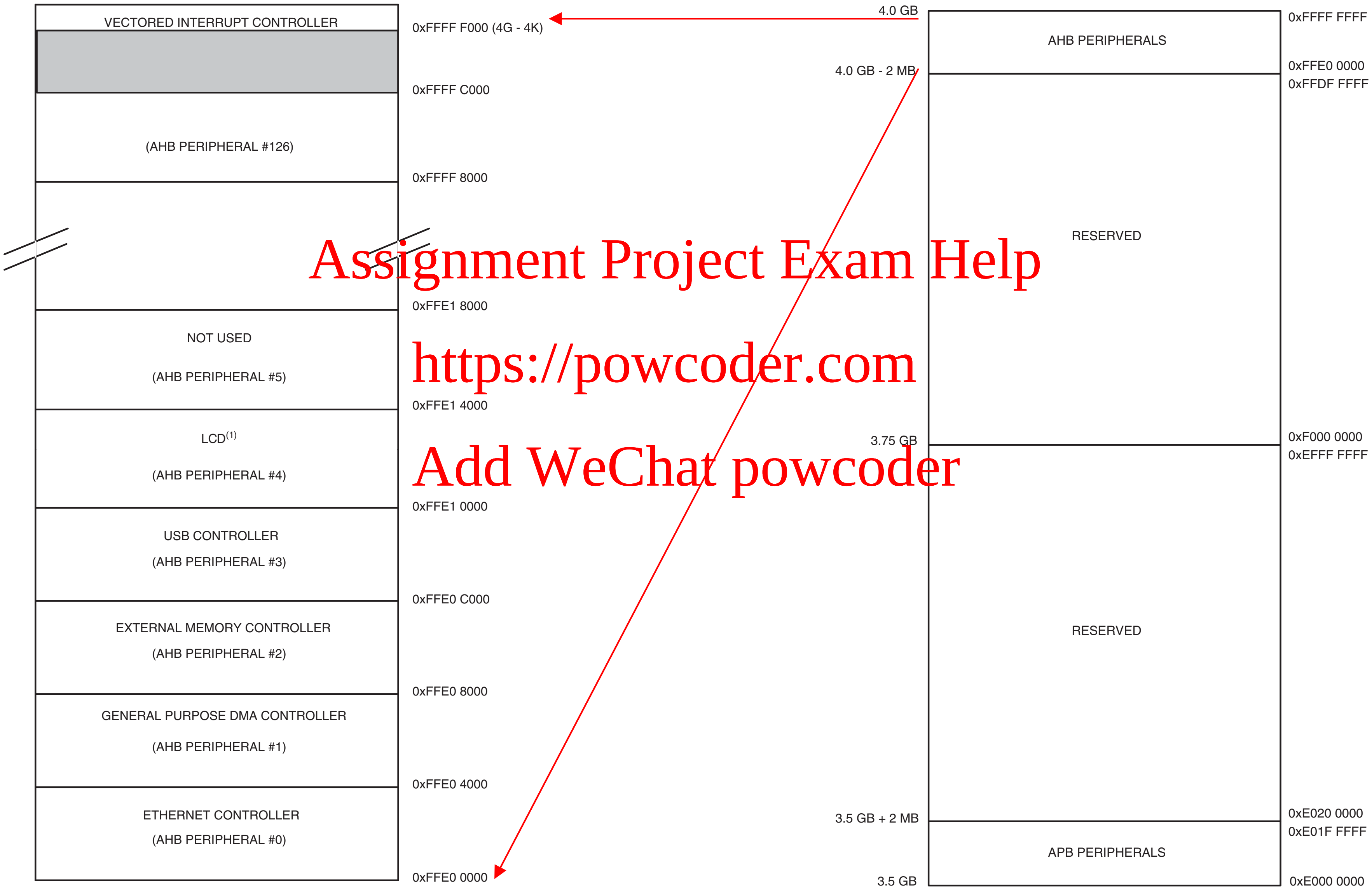
Address range	General use	Address range details and description	
0x0000 0000 to 0x3FFF FFFF	On-chip non-volatile memory and Fast I/O	0x0000 0000 - 0x0007 FFFF	Flash Memory (512 kB)
		0x3FFF C000 - 0x3FFF FFFF	Fast GPIO registers
0x4000 0000 to 0x7FFF FFFF	On-chip RAM	0x4000 0000 - 0x4000 FFFF	RAM (64 kB)
		0x7FE0 0000 - 0x7FE0 3FFF	Ethernet RAM (16 kB)
		0x7FD0 0000 - 0x7FD0 3FFF	USB RAM (16 kB)
0x8000 0000 to 0xDFFF FFFF	Off-Chip Memory	Four static memory banks, 16 MB each	
		0x8000 0000 - 0x80FF FFFF	Static memory bank 0
		0x8100 0000 - 0x81FF FFFF	Static memory bank 1
		0x8200 0000 - 0x82FF FFFF	Static memory bank 2
		0x8300 0000 - 0x83FF FFFF	Static memory bank 3
		Four dynamic memory banks, 256 MB each	
		0xA000 0000 - 0xAFFF FFFF	Dynamic memory bank 0
		0xB000 0000 - 0xBFFF FFFF	Dynamic memory bank 1
		0xC000 0000 - 0xCFFF FFFF	Dynamic memory bank 2
		0xD000 0000 - 0xDFFF FFFF	Dynamic memory bank 3
0xE000 0000 to 0xEFFF FFFF	APB Peripherals	36 peripheral blocks, 16 kB each	
0xF000 0000 to 0xFFFF FFFF	AHB peripherals		

LPC24XX User manual. Document No: UM10237

Memory-mapped IO on LPC2478



Memory-mapped IO on LPC2478

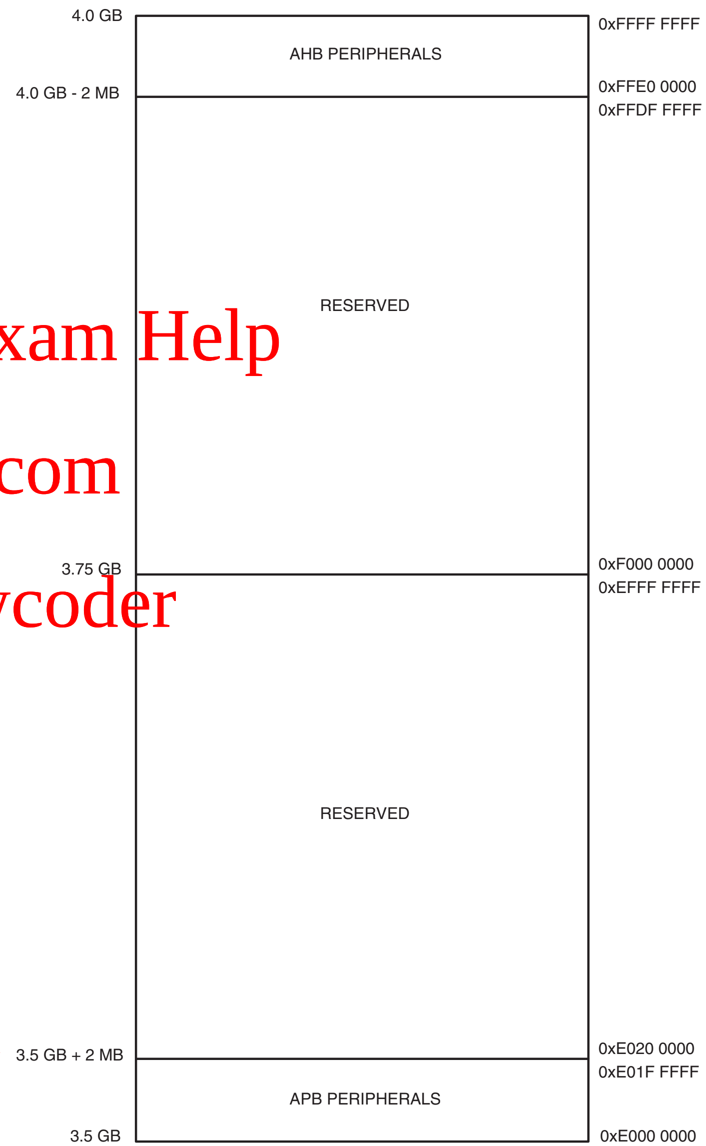


Memory-mapped IO on LPC2478

LPC24XX User manual. Document No: UM10237

Table 17. APB peripherals and base addresses

APB Peripheral	Base Address	Peripheral Name
0	0xE000 0000	Watchdog Timer
1	0xE000 4000	Timer 0
2	0xE000 8000	Timer 1
3	0xE000 C000	UART0
4	0xE001 0000	UART1
5	0xE001 4000	PWM0
6	0xE001 8000	PWM1
7	0xE001 C000	I^{2}C0
8	0xE002 0000	SPI
9	0xE002 4000	RTC
10	0xE002 8000	GPIO
11	0xE002 C000	Pin Connect Block
12	0xE003 0000	SSP1
13	0xE003 4000	ADC
14	0xE003 8000	CAN Acceptance Filter RAM
15	0xE003 C000	CAN Acceptance Filter Registers
16	0xE004 0000	CAN Common Registers
17	0xE004 4000	CAN Controller 1
18	0xE004 8000	CAN Controller 2
19 to 22	0xE004 C000 to 0xE005 8000	Not used
23	0xE005 C000	I^{2}C1
24	0xE006 0000	Not used
25	0xE006 4000	Not used
26	0xE006 8000	SSP0
27	0xE006 C000	DAC
28	0xE007 0000	Timer 2
29	0xE007 4000	Timer 3
30	0xE007 8000	UART2
31	0xE007 C000	UART3
32	0xE008 0000	I^{2}C2
33	0xE008 4000	Battery RAM
34	0xE008 8000	I^{2}S
35	0xE008 C000	SD/MMC Card Interface
36 to 126	0xE009 0000 to 0xE01F BFFF	Not used
127	0xE01F C000	System Control Block



This week

- Function call examples
- Input / output background
- Polling
- Interrupts
- LPC2478 microcontroller

In Moodle:

- **Start working on Lab (Due: end of your 3-hr lab)**
- **Start doing Week 7 exercise**
- **Put in EOI if your team is interested in connecting real sensor to the QVGA for the Design Project.**

References

[1] William Hohl, ARM Assembly Language: Fundamentals and Techniques, CRC Press, 2015 (2nd Edition).

[2] ARM Architecture Reference Manual.