HW3-1

1. Concept explanation

1. The Makefile already uses TARGET and VPATH = hal/$(TARGET) so we can support multiple CPUs by:

   setting TARGET = riscv,

   putting the RISC-V-specific HAL files in hal/riscv/.

2. For RISC-V we must change the **toolchain** and **march** options:

   CC = riscv32-unknown-elf-gcc

   AS = riscv32-unknown-elf-gcc (or ...-as)

   LD = riscv32-unknown-elf-gcc

   ARCH = rv32im (for example).

3. We must also provide **machine-dependent UART code** for RISC-V, not only the Makefile:

   a RISC-V UART register header (RiscvUart.h),

   hal/riscv/Uart.c that implements Hal_uart_init() and Hal_uart_put_char(),

   Regs.c that defines the UART pointer for that platform.

Lab03_UART

So the build will pick boot/Entry.s (RISC-V version) and hal/riscv/Uart.c and link them into rtos.axf.

To build an executable for riscv, I set TARGET=riscv and use a RISC-V cross compiler in the Makefile. Then I add hal/riscv/Uart.c, RiscvUart.h, and Regs.c so that Hal_uart_put_char() is implemented for the RISC-V UART.

2. Example Makefile changes

This is the ARM Makefile style from Slide 14, adapted to **riscv**.

Lab03_UART

```
# ----- architecture / platform -----
ARCH   = rv32im       # RISC-V 32-bit integer core (example)
TARGET = riscv        # directory name under hal/

# ----- toolchain for RISC-V -----
CC = riscv32-unknown-elf-gcc
AS = riscv32-unknown-elf-gcc
LD = riscv32-unknown-elf-gcc
OC = riscv32-unknown-elf-objcopy

LINKER_SCRIPT = ./rtos.ld
MAP_FILE     = build/rtos.map

# ----- sources -----
ASM_SRCS = $(wildcard boot/*.S)
```

```
ASM_OBJS = $(patsubst boot/%.S, build/%.so, $(ASM_SRCS))

VPATH = boot \
    hal/$(TARGET) \
    lib

C_SRCS  = $(notdir $(wildcard boot/*.c))
C_SRCS += $(notdir $(wildcard hal/$(TARGET)/*.c))
C_SRCS += $(notdir $(wildcard lib/*.c))
C_OBJS  = $(patsubst %.c, build/%.o, $(C_SRCS))

INC_DIRS = -I include \
      -I hal \
      -I hal/$(TARGET) \
      -I lib

CFLAGS = -c -g -std=c11 -march=$(ARCH)
LDFLAGS = -nostartfiles -nostdlib -nodefaultlibs -static -lgcc

rtos    = build/rtos.axf
rtos_bin = build/rtos.bin

all: $(rtos)

$(rtos): $(ASM_OBJS) $(C_OBJS) $(LINKER_SCRIPT)
            $(LD) -n -T $(LINKER_SCRIPT) -o $(rtos) \
                $(ASM_OBJS) $(C_OBJS) -Wl,-Map=$(MAP_FILE) $(LDFLAGS)
            $(OC) -O binary $(rtos) $(rtos_bin)

build/%.so: %.S
            mkdir -p $(shell dirname $@)
            $(CC) -march=$(ARCH) $(INC_DIRS) $(CFLAGS) -o $@ $<

build/%.o: %.c
            mkdir -p $(shell dirname $@)
            $(CC) -march=$(ARCH) $(INC_DIRS) $(CFLAGS) -o $@ $<
```

This Makefile builds rtos.axf for riscv using Entry.s (RISC-V version) and links in all C files, including hal/riscv/Uart.c, so the executable supports Hal_uart_put_char().

3. Example RISC-V UART HAL code (for Hal_uart_put_char())

Assume we run on a simple RISC-V board (like QEMU virt) where UART0 is a memory-mapped 16550-type UART at 0x10000000.

hal/riscv/RiscvUart.h

```
#ifndef HAL_RISCV_UART_H_

#define HAL_RISCV_UART_H_


#include <stdint.h>


typedef struct

{

  volatile uint32_t RBR_THR_DLL;   // 0x00 receive / transmit / divisor LSB

  volatile uint32_t IER_DLM;      // 0x04

  volatile uint32_t IIR_FCR;      // 0x08

  volatile uint32_t LCR;         // 0x0C

  volatile uint32_t MCR;         // 0x10

  volatile uint32_t LSR;         // 0x14 line status

  volatile uint32_t MSR;         // 0x18

  volatile uint32_t SCR;         // 0x1C

} RiscvUart_t;


#define RISCV_UART0_BASE   0x10000000UL


#endif
```

hal/riscv/Regs.c

```
#include "RiscvUart.h"


volatile RiscvUart_t* RiscvUart0 = (volatile RiscvUart_t*)RISCV_UART0_BASE;
```

hal/riscv/Uart.c

```
#include "stdint.h"

#include "RiscvUart.h"

#include "HalUart.h"


extern volatile RiscvUart_t* RiscvUart0;
```

```
void Hal_uart_init(void)
{
  // Minimal init – 8N1, no interrupts, baud assumed preconfigured
  // 8 bit word length: set LCR[1:0] = 3
  RiscvUart0->LCR = 0x03;
}

void Hal_uart_put_char(uint8_t ch)
{
  // Wait until transmitter holding register is empty (LSR bit 5 = THRE)
  while ((RiscvUart0->LSR & (1 << 5)) == 0)
    ;  // busy-wait

  // Write data to transmit register (THR)
  RiscvUart0->RBR_THR_DLL = ch;
}
```

This shows clearly *how* you would support Hal_uart_put_char() on RISC-V.

HW3-2

Implement Hal_uart_get_char(void)

Now we do it for the **ARM PL011** UART used in the slides. We already have Uart.h which defines PL011_t, UARTFR_t, UARTDR_t, etc.

We just need a blocking receive:

- **Wait** while the receive FIFO is empty: UARTFR.RXFE == 1.
- Then **read one byte** from UARTDR.DATA and return it.
- Ignore error bits as the homework says.

1. Code (add to hal/rvpb/Uart.c)

Assuming this file already has Hal_uart_init() and Hal_uart_put_char() from the slides.

```
#include "stdint.h"
#include "Uart.h"
#include "HalUart.h"

extern volatile PL011_t* Uart;

/*
 * Receive one character from UART (blocking).
 * This function waits until the receive FIFO has data,
```

```
 * then returns the 8-bit DATA field from UARTDR.
 */
uint8_t Hal_uart_get_char(void)
{
   // RXFE (bit 4) == 1 means "Receive FIFO empty".
   // Stay in the loop while there is no received data.
   while (Uart->uartfr.bits.RXFE)
   {
      // busy-wait until a character arrives
   }

   // When RXFE == 0 there is at least one byte in the FIFO.
   // UARTDR.DATA[7:0] holds the received character.
   return (uint8_t)(Uart->uartdr.bits.DATA);
}
```

You can also declare it in HalUart.h:

```
// HalUart.h
void Hal_uart_init(void);
void Hal_uart_put_char(uint8_t ch);
uint8_t Hal_uart_get_char(void);   // NEW: receive one character
```

2. Simple test program for screenshot

Use this main.c to show that Hal_uart_get_char() works (it **echoes** whatever you type):

```
#include "stdint.h"
#include "HalUart.h"
#include "stdio.h"

static void Hw_init(void);

void main(void)
{
   Hw_init();

   putstr("Echo test. Type keys...\n");

   while (1)
   {
```

```
      uint8_t ch = Hal_uart_get_char(); // wait for a key
      Hal_uart_put_char(ch);          // echo it back
    }
}

static void Hw_init(void)
{
   Hal_uart_init();
}
```

Build and run (same Makefile as Lab03):

```
make
make run
```