Learning RISC-V with embedded targets and QEMU

Loh Siu Yin
Beyond Broadcast LLP
siuyin@beyondbroadcast.com

1. Why this article?

The chips I use are ARM Cortex M0/M0+, MSP430G2, PIC16 and STM8.

I then came across RISC-V RV32EC in the form of the CH32V003. The 32 means 32-bit. `E` is for the embedded capability and `C` for compressed instructions to help reduce code size.

Each device had a steep learning curve and RISC-V was similar. The amount of resources was overwhelming. Where do I start? How do I create bare-metal code to blink an LED?

My starting point was: https://riscv.org/.

2. Running Linux on QEMU

https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html shows you how to *almost* get a QEMU environment up the hard way. You compile QEMU, Linux and BusyBox for RISC-V. I later found I could also use the QEMU packaged the linux distribution I used. For Ubuntu 22.04 it was the `qemu-system-misc` package.

When I got to the running stage at https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html#running, it crashed with:

```
[    1.200880] [<ffffffff80825ac4>] kernel_init+0x1e/0x10a
[    1.201294] [<ffffffff80003762>] ret_from_fork+0xa/0x1c
[    1.202683] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-blo
QEMU: Terminated
```

After searching the internet, I learned I needed to provide an initial ram filesystem. The invocation that finally worked for me was:

`runqemu.sh`:

```
qemu-system-riscv64 -nographic -machine virt \
  -kernel linux/arch/riscv/boot/Image -append "root=/dev/vda ro console=ttyS0" \
  -drive file=busybox/busybox,format=raw,id=hd0 \
  -device virtio-blk-device,drive=hd0 \
  -initrd initramfs.cpio.gz
```

And the output was a root linux prompt in QEMU:

```
[    1.368036] clk: Disabling unused clocks
[    1.448878] Freeing unused kernel image (initmem) memory: 2184K
[    1.450971] Run /init as init process

Boot took 1.71 seconds

Ctrl a x to exit QEMU
~ #
```

But how do you get that `initramfs.cpio.gz` file?

3. Using Busybox as the root filesystem in RISC-V QEMU

After more searching on the internet, I found:
https://gist.github.com/chrisdone/02e165a0004be33734ac2334f215380e

That lead me to write:

mkinitramfs.sh:

```
rm -rf initramfs
mkdir -p initramfs
cp init initramfs
chmod +x init

cd initramfs
mkdir -p bin sbin etc proc sys usr/bin usr/sbin
cp -a ../busybox/_install/* .

cp ../hello .

find . -print0 | cpio --null -ov --format=newc \
    | gzip -9 > ../initramfs.cpio.gz
```

And my init:

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mknod -m 666 /dev/ttyS0 c 4 64

echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
echo "Ctrl a x to exit QEMU"

setsid  cttyhack sh
exec /bin/sh
```

For reference here is a snapshot of my project root layout:

```
siuyin@ln03:~/riscv64-linux$ ls -F
about.article  busybox/  go.sum  hello.o  initramfs/         linux/           README.md     src/
build.sh*      go.mod    hello*  init*    initramfs.cpio.gz  mkinitramfs.sh*  runqemu.sh*
```

Ignore the go.sum and go.mod files as there are there only to support this about.article I am writing.

4. Running a RISC-V binary in QEMU running RISC-V Linux

I chose to learn RISC-V assembly first as that was closest to the bare metal. Below is my hello world in assembly.

You may find this quick reference card handy:
https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

src/linux/hello.s:

```
# Simple RISC-V hello world.

.global _start

.text
_start:
    addi    a0, x0, 1        # addi is add immediate. x0 is the zero-value register, 1+0 -> a0
    la    a1, helloworld       # la is load address
    addi    a2, x0, 13

    addi    a7, x0, 64       # 64 is the syscall to write
    ecall

_end:
    addi    a0, x0, 0        # return code is 0

    addi    a7, x0, 93       # 93 is the syscall to terminate
    ecall

.data
```

```
helloworld:
    .ascii    "Hello World.\n"
```

If you looked closely at my `mkinitramfs.sh` earlier, you will see that I copied the `hello` binary to the `initramfs.cpio.gz`.

This is how I built the `hello` binary:

`build.sh`:

```
#!/bin/bash

#riscv64-linux-gnu-gcc -o hello hello.s -nostdlib -static

riscv64-linux-gnu-as -march=rv64imac -o hello.o src/linux/hello.s
riscv64-linux-gnu-ld -o hello hello.o
```

The `-march=rv64imac` reads: assemble for RISC-V 64bit with integer, multiply (and divide), atomic instructions and compressed instructions.

Below is the output of a run in QEMU:

```
[    1.418066] Freeing unused kernel image (initmem) memory: 2184K
[    1.420248] Run /init as init process

Boot took 1.67 seconds

Ctrl a x to exit QEMU
~ # ls -F
bin/     etc/     init*    proc/    sbin/     usr/
dev/     hello*   linuxrc@ root/    sys/
~ # ./hello
Hello World.
~ #
```

5. Running RISC-V on QEMU 'bare-metal'

In the previous example I `syscall`'ed into Linux to print a string and to exit. Microcontrollers can be very small and usually will not run Linux. They may not run any operating system at all.

Thus my next RISC-V assembly program targets a sifive_e board that is emulated by QEMU.

```
siuyin@ln03:~/riscv64-linux$ qemu-system-riscv32 -machine help
Supported machines are:
none                 empty machine
opentitan            RISC-V Board compatible with OpenTitan
sifive_e             RISC-V Board compatible with SiFive E SDK
sifive_u             RISC-V Board compatible with SiFive U SDK
spike                RISC-V Spike board (default)
virt                 RISC-V VirtIO board
```

https://github.com/qemu/qemu/blob/792f77f376adef944f9a03e601f6ad90c2f891b2/hw/riscv/sifive_e.c#L15
gives more details about the emulated board.

5.1. Assembly code for the emulated red-v board

The sifive_e board emulates a serial port (UART) that allows strings to be printed onto the QEMU console.

`puts` in the assembly code below writes to the serial port.

`src/embed/embedhello.s`:

```
.align 2
.equ UART_REG_TXFIFO, 0
.equ UART_BASE, 0x10013000

.section .text
```

```
        .globl _start

    _start:
        # load the mhartid (machine hardware thread ID) control and status register into t0
        csrr    t0, mhartid
        # branch to halt if the hartid is not zero -- all other threads are halted
        bnez    t0, halt

        # load address stack_top (from linker script) into stack pointer
        la      sp, stack_top
        # load address of msg into a0 (argument 0) register, t0 above is temporary 0
        la      a0, msg
        jal     puts        # jump and link: call puts

        la      a0, msg2    # call puts again but now with msg2
        jal     puts

        j       halt        # end the program

    puts:
        li      t0, UART_BASE

    .puts_loop: lbu     t1, (a0)
        beqz    t1, .puts_leave

    .puts_wait: lw      t2, UART_REG_TXFIFO(t0)
        bltz    t2, .puts_wait

        sw      t1, UART_REG_TXFIFO(t0)
        add     a0, a0, 1

        j       .puts_loop

    .puts_leave:
        ret


    halt:       j       halt

    .section .rodata
    msg:
        .string "Hello risc-v!\n"

    msg2:
        .string "This is my second string.\n"
```

5.2. RISC-V privileged mode and multiple hardware threads

The csrr instruction surprised me. Normally embedded processes are single core with a single thread of execution. With some implementations, you may have multiple cores. This codes targets only the core with thread ID of zero.

```
        # load the mhartid (machine hardware thread ID) control and status register into t0
        csrr    t0, mhartid
        # branch to halt if the hartid is not zero -- all other threads are halted
        bnez    t0, halt
```

See the extract from the RISC-V manual volume 2 below:

3.1.5   Hart ID Register `mhartid`

The `mhartid` CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

MXLEN-1	0
Hart ID	

MXLEN

Figure 3.5: Hart ID register (`mhartid`).

In certain cases, we must ensure exactly one hart runs some code (e.g., at reset), and so require one hart to have a known hart ID of zero.

For efficiency, system implementers should aim to reduce the magnitude of the largest hart ID used in a system.

5.3. A customized linker script for the Red-V board

`stack_top` is defined in the linker script

`red-v.ld`:

```
OUTPUT_ARCH("riscv")
OUTPUT_FORMAT("elf32-littleriscv")

ENTRY( _start )
SECTIONS
{
    . = 0x20010000;
    .text : { *(.text) }
    .gnu_build_id : { *(.note.gnu.build-id) }
    .rodata : { * (.rodata) }

    . = 0x80000000;
    .data : { *(.data) }
    .sdata : { *(.sdata) }
    .debug : { *(.debug) }
    . += 0x1000;
    stack_top = .;

    _end = .;
}
```

After a red-v board is reset, its firmware points to address `0x20010000` to start executing code found there. This address is in the QSPI flash area.

However the stack must sit in RAM and that starts at address `0x80000000` in the red-v board.

0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4:   FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

is the GNU Binutils LD linker script reference.

5.4. Output of 'bare-metal' assembly code

`src/embed/run-qemu.sh:'

```
#!/bin/sh
echo "exit QEMU: Ctrl a x\n\n"

qemu-system-riscv32 -machine sifive_e \
        -nographic -bios none \
        -kernel embedhello-qemu
```

Note the machine type is 'sifive_e'.

```
$ ./run-qemu.sh
exit QEMU: Ctrl a x


Hello risc-v!
This is my second string.
```

6. C code for the emulated red-v board

Below is the startup `assembly` code to jump to the main C code.

src/c_embed/startup.s:

```
.align 2

.section .text
.globl _start

_start:
    csrr    t0, mhartid
    bnez    t0, halt

    la    sp, stack_top    # initialize stack

    j    main         # jump to main

halt:    j    halt
```

And below is the main C code.

src/c_embed/main.c:

```
typedef unsigned int size_t;
typedef unsigned char uint8_t;

#define UART_BASE ((size_t *)0x10013000)
#define TXDATA (UART_BASE + 0)

// Making putchar static optimizes space as it will not be 'exported'
// to be visible in other source files.
static void putchar(uint8_t ch) {
    while ((volatile int)*TXDATA < 0);
    *TXDATA = ch;
}

void puts(char *s) {
    // while char pointed to by s is non-zero, putchar and increment the pointer
    while (*s) putchar(*s++);
    putchar('\n');
}

void main() {
```

```
    puts("Hello RISC-V from C!");
    puts("bye");
}
```

Let's take a closer look at:

```
while ((volatile int)*TXDATA < 0);
```

We cast the `size_t` pointed to by UART_BASE to a `volatile int`.

`volatile` tells the compiler the value may be changed by the hardware and thus to not eliminate it when optimizing the code.

`int` makes it a signed integer. As the red-v manual extract below states, the most significant bit (MSB) will be set if the FIFO is full -- and thus not ready to take another byte. if the MSB is set, it is a negative number.

Thus the code reads:
wait for the TX FIFO to be not full then write `ch` to the memory pointed to by TXDATA.

```
while ((volatile int)*TXDATA < 0);
*TXDATA = ch;
```

18.4   Transmit Data Register (`txdata`)

Writing to the `txdata` register enqueues the character contained in the `data` field to the transmit FIFO if the FIFO is able to accept new entries. Reading from `txdata` returns the current value of the `full` flag and zero in the `data` field. The `full` flag indicates whether the transmit FIFO is able to accept new entries; when set, writes to `data` are ignored. A RISC-V `amoor.w` instruction can be used to both read the `full` status and attempt to enqueue data, with a non-zero return value indicating the character was not accepted.

Transmit Data Register (`txdata`)				
Register Offset		`0x0`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	`data`	RW	X	Transmit data
[30:8]	Reserved			
31	`full`	RO	X	Transmit FIFO full

Table 56:   Transmit Data Register

6.1. Makefile for embedded C

```
src/c_embed/Makefile:

  CC=riscv64-unknown-elf-gcc -O2 -g -nostdlib -Wno-builtin-declaration-mismatch -march=rv32ec -ma
  AS=riscv64-unknown-elf-as -march=rv32ec
  LD=riscv64-unknown-elf-ld -nostdlib -melf32lriscv -Tred-qemu.ld
  SIZE=riscv64-unknown-elf-size
  OBJDUMP=riscv64-unknown-elf-objdump

  OBJS=c_embed startup.o main.o main.s

  all: c_embed

  c_embed: startup.o main.o
      $(LD) -o c_embed startup.o main.o
      $(SIZE) c_embed

  %.o: %.s
      $(AS) -o $@ $<

  main.s: main.c
      $(CC) -o main.s -S main.c
```

```
    .PHONY: clean
    clean:
        rm -f $(OBJS)

    .PHONY: dump
    dump: c_embed
        $(OBJDUMP) -S --source-comment='#' -d c_embed
```

The line:

```
    CC=riscv64-unknown-elf-gcc -O2 -g -nostdlib -Wno-builtin-declaration-mismatch -march=rv32ec -ma
```

is needed because `gcc` already knows of a `putchar` function in libc and it is not a void function.

6.2. Embedded C code output

`src/c_embed/run-qemu.sh:'

```
    #!/bin/sh

    echo "exit QEMU: Ctrl a x\n\n"

    qemu-system-riscv32 -machine sifive_e \
            -nographic -bios none \
            -kernel c_embed
```

Note the machine type is 'sifive_e'.

```
    $ ./run-qemu.sh
    exit QEMU: Ctrl a x


    Hello RISC-V from C!
    bye
```

7. Really running RISC-V on the physical ch32v003 device

Nanjing Qinheng Microelectronics, makers of the ch32v003 device have good development system based on Eclipse. Their provided libraries were decent but it still took a lot of poring over source code to understand what was going on.

Futher, their linker script and start-up code was dense and hard to understand.

I belive this is why CNLohr wrote the lightweight ch32v003fun development environment. The environment consist of one C file, the associated header file, a linker script and software to drive the device programmer. The project is Open Source and I decided to use the linker script as the basis of my "hello world" into ch32v003 RISC-V assembly programming.

7.1. Code to blink an LED

On my development board -- just the ch32v003 part soldered to a breakout board -- I've connected an LED to PD0. I found this listing of RISC-V pseudo instructions useful. Pseudo instructions are human friendly instructions that are translated by the assembler to actual RISC-V instructions.

I also found this tutorial helpful.

CH32V003F4P6

```
 1   PD4/A7/UCK/T2CH1ETR/OPO/T1CH4ETR_          PD3/A4/T2CH2/AETR/UCTS/T1CH4_       20
 2   PD5/A5/UTX/T2CH4_/URX_                       PD2/A3/T1CH1/T2CH3_/T1CH2N_        19
 3   PD6/A6/URX/T2CH3_/UTX_               PD1/SWIO/AETR2/T1CH3N/SCL_/URX_           18
 4   PD7/NRST/T2CH4/OPP1/UCK_                 PC7/MISO/T1CH2_/T2CH2_/URTS_          17
 5   PA1/OSCI/A1/T1CH2/OPN0                  PC6/MOSI/T1CH1CH3N_/UCTS_/SDA_         16
 6   PA2/OSCO/A0/T1CH2N/OPP0/AETR2_   PC5/SCK/T1ETR/T2CH1ETR_/SCL_/UCK_/T1CH3_     15
 7   VSS                                    PC4/A2/T1CH4/MCO/T1CH1CH2N_            14
 8   PD0/T1CH1N/OPN1/SDA_/UTX_                    PC3/T1CH3/T1CH1N_/UCTS_           13
 9   VDD                       PC2/SCL/URTS/T1BKIN/AETR_/T2CH2_/T1ETR_             12
10   PC0/T2CH3/UTX_/NSS_/T1CH3_        PC1/SDA/NSS/T2CH4_/T2CH1ETR_/T1BKIN_/URX_   11
```

src/ch32v003/blink/blink.S:

```asm
    .align 2

    .equ RCC_APB2PCENR, 0x40021018
    .equ GPIOD_CFGLR, 0x40011400
    .equ GPIOD_BSHR, 0x40011410

    .section .text
    .globl _start

_start:
    # load address _eusrstack (from linker script) into stack pointer
    la sp, _eusrstack

gpiod_clk_en:
    # turn on clock to GPIOD to enable it
    li t0, 1<<5
    li t1, RCC_APB2PCENR
    sw t0, 0(t1)

gpiod_pd0_out:
    # configure GPIOD
    li t0, GPIOD_CFGLR
    lw t1, 0(t0)
    andi t1,t1,~(0xf)
    ori t1,t1,1
    sw t1, 0(t0)

pd0_on:
    # turn on LED connected to PD0
    li t0, GPIOD_BSHR
.pd0_on_1:
    li t1, 1
    sw t1, 0(t0)
    jal delay

pd0_off:
    li t1, 1<<16
    sw t1, 0(t0)
    jal delay

repeat:
    j .pd0_on_1

delay:
    li a0, 500000
.delay_1:
```

```
        addi a0,a0,-1
        beqz a0, .delay_end
        j .delay_1

    .delay_end:
        ret


        j halt

    halt:    j    halt
```

Let's break this down part by part. First consider:

```
    .align 2

    .equ RCC_APB2PCENR, 0x40021018
    .equ GPIOD_CFGLR, 0x40011400
    .equ GPIOD_BSHR, 0x40011410
```

`.align 2` tells the linker to place the code on a two-byte (16-bit) boundary. This is allowed as the part implements compressed instructions (rv32ec). If not code must be aligned on a word (32-bit) boundary.

What follows are `.equ` to equate symbols to memory-mapped register addresses. `RCC_APB2PCENR`, read, RCC APB2 P C EN R is the:
Reset and Clock Control
Advanced Peripheral Bus 2
Peripheral
Clock
Enable
Register.

Now let's proceed to the system initialisation.
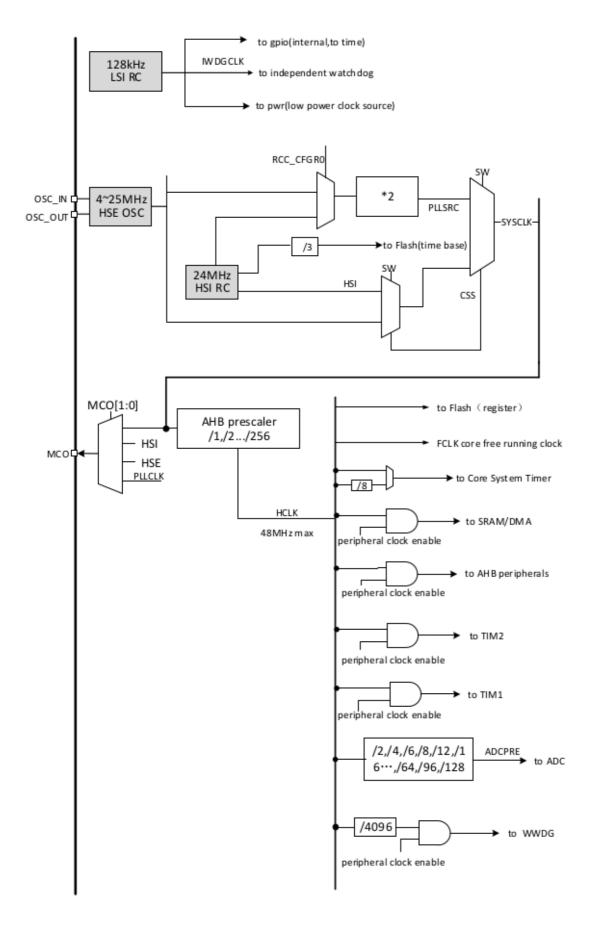
```
    .section .text
    .globl _start

    _start:
        # load address _eusrstack (from linker script) into stack pointer
        la sp, _eusrstack
```

The above code just loads the address of _e usr stack -- end of user stack to the stack pointer register. This is the bare essentials to get started.

Next, we enable GPIOD by routing the device's internal clock to the GPIO internal module. The device's reset condition is to use the 24MHz HSI RC oscillator with an AHB prescaler of /3 (actual and not /1 documented in the reference manual). This results in an HCLK of 8MHz being available to AHB and AHB2 peripherals.

But I don't see AHB2 ?! There is only one AHB bus on this device but on larger devices the Advanced High-performance Bus (a term I belive originating from ARM devices) are divided. Here on the ch32v003 AHB == AHB2 and vice-versa.

Here is how we actually enable GPIOD by feeding it the HCLK by setting bit 5 on RCC APB2 P C EN R.

3.4.7 APB2 Peripheral Clock Enable Register (RCC_APB2PCENR)

Offset address: 0x18

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
								Reserved							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	USART1 EN	Reserved	SPI1 EN	TIM1 EN	Reserved	ADC1 EN		Reserved		IOPD EN	IOPC EN	Reserved	IOPA EN	Reserved	AFIO EN

Bit	Name	Access	Description	Reset value
[31:15]	Reserved	RO	Reserved	0
14	USART1EN	RW	USART1 interface clock enable bit. 1: Module clock is on; 0: Module clock is off.	0
13	Reserved	RO	Reserved	0
12	SPI1EN	RW	SPI1 interface clock enable bit. 1: Module clock is on; 0: Module clock is off.	0
11	TIM1EN	RW	TIM1 module clock enable bit. 1: Module clock is on; 0: Module clock is off.	0
10	Reserved	RO	Reserved	0
9	ADC1EN	RW	ADC1 module clock enable bit. 1: Module clock is on; 0: Module clock is off.	0
[8:6]	Reserved	RO	Reserved	0
5	IOPDEN	RW	PD port module clock enable bit for I/O. 1: Module clock is on; 0: Module clock is off.	0
4	IOPCEN	RW	PC port module clock enable bit for I/O. 1: Module clock is on; 0: Module clock is off.	0
3	Reserved	RO	Reserved	0
2	IOPAEN	RW	PA port module clock enable bit for I/O. 1: Module clock is on; 0: Module clock is off.	0
1	Reserved	RO	Reserved	0
0	AFIOEN	RW	I/O auxiliary function module clock enable bit. 1: Module clock is on; 0: Module clock is off.	0

Note: When the peripheral clock is not enabled, the software cannot read out the peripheral register value and the value returned is always 0.

```
gpiod_clk_en:
    # turn on clock to GPIOD to enable it
    li t0, 1<<5
    li t1, RCC_APB2PCENR
    sw t0, 0(t1)
```

Study the above code:

`li t0, 1<<5` loads immediate into `t0` the binary value 100000. Note: the lowest bit is named bit-0.

What is `t0`? It is the ABI name for one of 16 CPU registers defined in RISC-V rv32e:

Table 1-2 RV32E registers

Register	ABI Name	Description	Storer
x0	zero	Hardcoded 0	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporary register	Caller
x8	s0/fp	Save register/frame pointer	Callee
x9	s1	Save register	Callee
x10-11	a0-1	Function parameters/return values	Caller
x12-15	a2-5	Function parameters	Caller

Similarly RCC_APB2PCENR is loaded into `t1`.

Next comes `sw t0, 0(t1)`, which reads:
store **w**ord in `t0` to the
memory address pointed to by `t1`,
offset by **0**.

When the above code executes, it stores `1<<5` to the RCC_ABP2PCENR register which then enables GPIOD.
The equivalent in C is: `RCC_APB2PCENR = 1<<5;`

The GPIO ports on the ch32v003 reset to a floating input condition, effectivly disconnecting the IO pins from whatever they were connected to. We want to make GPIO Port D pin 0 an output.

7.3.1.1 Port Configuration Register Low (GPIOx_CFGLR) (x=A/C/D)

Offset address: 0x00

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
CNF7[1:0]	MODE7[1:0]	CNF6[1:0]	MODE6[1:0]	CNF5[1:0]	MODE5[1:0]	CNF4[1:0]	MODE4[1:0]

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
CNF3[1:0]	MODE3[1:0]	CNF2[1:0]	MODE2[1:0]	CNF1[1:0]	MODE1[1:0]	CNF0[1:0]	MODE0[1:0]

Bit	Name	Access	Description	Reset value
[31:30] [27:26] [23:22] [19:18] [15:14] [11:10] [7:6] [3:2]	CNFy[1:0]	RW	(y=0-7), the configuration bits for port x, by which the corresponding port is configured. When in input mode (MODE=00b). 00: Analog input mode. 01: Floating input mode. 10: With pull-up and pull-down mode. 11: Reserved. In output mode (MODE>00b). 00: Universal push-pull output mode. 01: Universal open-drain output mode. 10: Multiplexed function push-pull output mode. 11: Multiplexing function open-drain output mode.	01b
[29:28] [25:24] [21:20] [17:16] [13:12] [9:8] [5:4] [1:0]	MODEy[1:0]	RW	(y=0-7), port x mode selection, configure the corresponding port by these bits. 00: Input mode. 01: Output mode, maximum speed 10MHz; 10: Output mode, maximum speed 2MHz. 11: Output mode, maximum speed 50MHz.	00b

The tricky part here is to configure only the bits relating to `PD0`, leaving the rest unmodified.

```
gpiod_pd0_out:
    # configure GPIOD
    li t0, GPIOD_CFGLR
    lw t1, 0(t0)
    andi t1,t1,~(0xf)
    ori t1,t1,1
    sw t1, 0(t0)
```

`lw t1, 0(t0)` : **l**oad **w**ord into `t1` from the memory address pointed to by `t0` offset by **0**.

`andi t1,t1,~(0xf)` : **and i**mmediate the binary value `1111..11110000` (32-bits)
with the value in `t1` (the *second* `t1`) and store the result in `t1` (the *first* `t1`).

When the above code is executed, binary value 0001 is stored into GPIOD_CFGLR and PD0 is now an output.

It is now time to turn on PD0. For that we need to modify yet another register: GPIOD_BSHR which reads,
GPIOD BS H R:
GPIOD
Bit Set High Reset .

7.3.1.4 Port Reset/Set Register (GPIOx_BSHR) (x=A/C/D)
Offset address: 0x10

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0

Bit	Name	Access	Description	Reset value
[31:24]	Reserved	R0	Reserved	0
[23:16]	BRy	WO	(y=0-7), the corresponding OUTDR bits are cleared for these location bits, and writing 0 has no effect. These bits can only be accessed in 16-bit form. If both BR and BS bits are set, the BS bit takes effect.	0
[15:8]	Reserved	RO	Reserved	0
[7:0]	BSy	WO	(y=0-7), for which the location bits will make the corresponding OUTDR location bits, writing 0 has no effect. These bits can only be accessed in 16-bit form. If both BR and BS bits are set, the BS bit takes effect.	0

Below is the code to assert PD0 to turn on the LED.

```
pd0_on:
    # turn on LED connected to PD0
    li t0, GPIOD_BSHR
.pd0_on_1:
    li t1, 1
    sw t1, 0(t0)
    jal delay
```

`jal delay` : **j**ump **a**nd **l**ink to delay, means call the delay routine.

```
delay:
    li a0, 500000
.delay_1:
    addi a0,a0,-1
    beqz a0, .delay_end
```

```
        j .delay_1

    .delay_end:
        ret
```

`beqz a0, .delay_end` : **b**ranch **eq**ual **z**ero to label `.delay_end` otherwise execute the following instruction. Which is `j .delay_1` : **j**ump to label `.delay_1`.

The code to turn off the LED is similar to the code to turning on the LED except that we write the `1` to the upper or higher half-word. This is why the regiter is named Reset High.

```
    pd0_off:
        li t1, 1<<16
        sw t1, 0(t0)
        jal delay
```

Finally at the end of the on-delay-off-delay sequence, a jump is executed to repeat the whole cycle. A tiny optimisation is made by jumping to label `.pd0_on_1` to skip reloading the address of GPIOD BS HR into `t0`.

```
    pd0_on:
        # turn on LED connected to PD0
        li t0, GPIOD_BSHR
    .pd0_on_1:
        li t1, 1
        sw t1, 0(t0)
        jal delay

    pd0_off:
        li t1, 1<<16
        sw t1, 0(t0)
        jal delay

    repeat:
        j .pd0_on_1
```

The LED connected to PD0 now blinks!

7.2. Makefile

The above was the assembler code, a `Makefile` is use by `make` to compile and link the code.

```
    all : flash

    TARGET:=blink
    TARGET_EXT:=S

    CFLAGS+=


    include ../ch32v003fun.mk

    flash : cv_flash
    clean : cv_clean
```

It is not very interesting as it mainly specifies that the `TARGET` is named blink and its extension is `.S` . By the way, .s assember files only allow `.include FILE` but .S files are processed by the C pre-processor to allow `#include "file"` and `#define ....` .

Here is the interesting `../ch32v003fun.mk` (credit CNLohr). It is long, so just give it a quick read-over and then I will explain the key sections of the file.

```
    PREFIX?=riscv64-unknown-elf

    CH32V003FUN?=..
    MINICHLINK?=/home/siuyin/ch32v003fun/minichlink

    CFLAGS+= \
        -g -Os -flto -ffunction-sections \
```

```
        -static-libgcc \
        -march=rv32ec \
        -mabi=ilp32e \
        -I/usr/include/newlib \
        -I$(CH32V003FUN)/../extralibs \
        -I$(CH32V003FUN) \
        -nostdlib \
        -I. -Wall $(EXTRA_CFLAGS)

    #LINKER_SCRIPT?=$(CH32V003FUN)/ch32v003fun.ld
    LINKER_SCRIPT?=$(CH32V003FUN)/simple.ld

    LDFLAGS+=-T $(LINKER_SCRIPT) -Wl,--gc-sections -L$(CH32V003FUN)/../misc -lgcc

    WRITE_SECTION?=flash
    SYSTEM_C?=$(CH32V003FUN)/ch32v003fun.c
    TARGET_EXT?=c

    #$(TARGET).elf : $(SYSTEM_C) $(TARGET).$(TARGET_EXT) $(ADDITIONAL_C_FILES)
    #    $(PREFIX)-gcc -o $@ $^ $(CFLAGS) $(LDFLAGS)

    $(TARGET).elf : $(TARGET).$(TARGET_EXT)
        $(PREFIX)-gcc -o $@ $^ $(CFLAGS) $(LDFLAGS)

    $(TARGET).bin : $(TARGET).elf
        $(PREFIX)-size $^
        $(PREFIX)-objdump -S $^ > $(TARGET).lst
        $(PREFIX)-objdump -t $^ > $(TARGET).map
        $(PREFIX)-objcopy -O binary $< $(TARGET).bin
        $(PREFIX)-objcopy -O ihex $< $(TARGET).hex

    ifeq ($(OS),Windows_NT)
    closechlink :
        -taskkill /F /IM minichlink.exe /T
    else
    closechlink :
        -killall minichlink
    endif

    terminal : monitor

    monitor :
        $(MINICHLINK)/minichlink -T

    gdbserver :
        -$(MINICHLINK)/minichlink -baG

    clangd :
        make clean
        bear -- make build

    clangd_clean :
        rm -f compile_commands.json

    FLASH_COMMAND?=$(MINICHLINK)/minichlink -w $< $(WRITE_SECTION) -b

    cv_flash : $(TARGET).bin
        make -C $(MINICHLINK) all
        $(FLASH_COMMAND)

    cv_clean :
        rm -rf $(TARGET).elf $(TARGET).bin $(TARGET).hex $(TARGET).lst $(TARGET).map $(TARGET).hex

    build : $(TARGET).bin
```

Let's first look at the step where the compiler/assember is invoked:

```
    $(TARGET).elf : $(TARGET).$(TARGET_EXT)
        $(PREFIX)-gcc -o $@ $^ $(CFLAGS) $(LDFLAGS)
```

The above reads:

```
to produced TARGET elf (executable and link format binary)
check if TARGET.EXT (in our case blink.S) has changed
if it has:
  run GCC and output TARGET.elf (blink.elf) by compiling TARGET.EXT (blink.S)
  with the following CFLAGS and LDFLAGS
if not, don't do anything.
```

Below are the CFLAGS:

```
CFLAGS+= \
    -g -Os -flto -ffunction-sections \
    -static-libgcc \
    -march=rv32ec \
    -mabi=ilp32e \
    -I/usr/include/newlib \
    -I$(CH32V003FUN)/../extralibs \
    -I$(CH32V003FUN) \
    -nostdlib \
    -I. -Wall $(EXTRA_CFLAGS)
```

The interesting parts are:

`-static-libgcc` : use the static version of libgcc

`-ffunction-sections` : from the gcc docs.
Place each function or data item into its own section in the output file if the target supports arbitrary sections.

`-mabi=ilp32e` : the machine application binary interface has integers,longs and pointer that are 32-bits wide, with function calls according to the embedded (as in rv32e) conventions.

And the LDFLAGS:

```
LDFLAGS+=-T $(LINKER_SCRIPT) -Wl,--gc-sections -L$(CH32V003FUN)/../misc -lgcc

#LINKER_SCRIPT?=$(CH32V003FUN)/ch32v003fun.ld
LINKER_SCRIPT?=$(CH32V003FUN)/simple.ld
```

I have substituted CNLohr's linker script with my own simplified one, `simple.ld`.

'--gc-sections' : garbage-collect unused sections. This reduces the size of the output binary.

Let's study `simple.ld` below.

```
ENTRY( _start )

MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 16K
    RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 2K
}

SECTIONS
{

    .text :
    {
      . = ALIGN(4);
     *(.text)
     *(.rodata)
      . = ALIGN(4);
    } >FLASH AT>FLASH

    .data :
    {
      . = ALIGN(4);
     *(.data .data.*)
```

```
        . = ALIGN(4);
          PROVIDE( _edata = .);
        } >RAM AT>FLASH

        PROVIDE( end = . );

        PROVIDE( _eusrstack = ORIGIN(RAM) + LENGTH(RAM));
    }
```

The script states the entry point of the executable is `_start`. `_start` is a label in our `blink.S` source file.

Also .text and .rodata sections should be placed in FLASH.
And .data sections should be placed in RAM.

Finally note the `_eusrstack` which defines the location of end of user stack. `_eusrstack` is used in `blink.S` to initialise the stack pointer.

7.3. Setting HCLK to run at 24MHz

According to the reference manual, the device should run at 24MHz immediately after reset. The internal RC oscillator does indeed run at 24MHz but I found the clock fed to internal peripherals was 8MHz. I later found the HPRE field in RCC_CFGR0 was 0b0010 after reset instead of 0b0000 as documented.

Below is the code in `blink24.S` to set HPRE to 0, a one-line change. The rest of the code remained the same as `blink.S`

src/ch32v003/blink24/blink24.S:

```
gpiod_clk_en:
    # turn on clock to GPIOD to enable it
    li t0, 1<<5
    li t1, RCC_BASE
    # set RCC CFG R0 to zero as documented in the reference manual to get 24MHz HSI to HCLK.
    sw zero, RCC_CFGR0(t1)
```

7.4. Setting HCLK to run at 48MHz

This required turning on the PLL, waiting for the PLL to become ready, then setting the clock source to be the PLL instead of HSI.

The flash access wait-state should also be set to 1 to ensure reliable operation.

I've adopted a convention in the source below where labels starting with a capitalized letter should be treated as a globally accessible or public label.

Labels starting with a lower case label are "private" and those starting with a . (dot) are "local".

src/ch32v003/blink48/blink48.S:

```
/* blink48.S sets the device to run at 48MHz.
 */

.align 2

.equ RCC_BASE, 0x40021000
.equ RCC_CTLR, 0x0
.equ RCC_CFGR0, 0x4
.equ RCC_INTR, 0x8
.equ RCC_APB2PCENR, 0x18

.equ FLASH_BASE, 0x40022000
.equ FLASH_ACTLR, 0x0

.equ GPIOD_BASE, 0x40011400
.equ GPIOD_CFGLR, 0x0
.equ GPIOD_BSHR, 0x10

.section .text
```

```
        .globl _start

_start:
        # load address _eusrstack (from linker script) into stack pointer
        la sp, _eusrstack

RCC_set_48MHz:
        li t0, RCC_BASE

        # set AHB divider to /1
        sw zero, RCC_CFGR0(t0)

        # turn on PLL
        lw t1, RCC_CTLR(t0)
        li t2, 1<<24
        or t1, t1, t2
        sw t1, RCC_CTLR(t0)

        # set flash latency to 1 cycle
        li t2, FLASH_BASE
        li t1, 0x01
        sw t1, FLASH_ACTLR(t2)

        # clear PLL, CSSC, HSE, HSI and LSI ready flags
        li t1, 0x009F0000
        sw t1, RCC_INTR(t0)

        # wait till PLL is ready
        li t2, 1<<25
.rcc_ctlr_pll_not_ready:
        lw t1, RCC_CTLR(t0)
        and t1, t1, t2
        beqz t1, .rcc_ctlr_pll_not_ready

        # set PLL as clock source
        lw t1, RCC_CFGR0(t0)
        ori t1, t1, 0b10
        sw t1, RCC_CFGR0(t0)

GPIOD_clk_en:
        # turn on clock to GPIOD to enable it
        li t0, RCC_BASE
        li t1, 1<<5
        sw t1, RCC_APB2PCENR(t0)

gpiod_pd0_out:
        # configure GPIOD
        li t0, GPIOD_BASE
        lw t1, GPIOD_CFGLR(t0)
        andi t1,t1,~(0xf)
        ori t1,t1,1
        sw t1, GPIOD_CFGLR(t0)

pd0_on:
        # turn on LED connected to PD0
        li t0, GPIOD_BASE
.pd0_on_1:
        li t1, 1
        sw t1, GPIOD_BSHR(t0)
        jal delay

pd0_off:
        li t1, 1<<16
        sw t1, GPIOD_BSHR(t0)
        jal delay

repeat:
        j .pd0_on_1
```

```
delay:
    li a0, 500000
.delay_1:
    addi a0, a0, -1
    beqz a0, .delay_end
    j .delay_1

.delay_end:
    ret

halt:    j halt
```

7.5. Getting the UART working

The USART (Universal Synchronous/Asynchronous Receiver Transmitter) on the ch32v003f4p6 is on PD5 (GPIO Port D, pin 5).

Below is the code to print "Hello RISC-V" to the uart.

This is a long source file, so give it a quick read-through and I will go through the parts step-by-step.

src/ch32v003/uart/uart.S:

```
/* uart.S prints hello world to the UART.
 * Connect PD5(uart TX) to the RX of your terminal.
 * 112500 8 N 1
 */

.align 2
.option rvc

.equ RCC_BASE, 0x40021000
.equ RCC_CTLR, 0x0
.equ RCC_CFGR0, 0x4
.equ RCC_INTR, 0x8
.equ RCC_APB2PCENR, 0x18

.equ FLASH_BASE, 0x40022000
.equ FLASH_ACTLR, 0x0

.equ GPIOD_BASE, 0x40011400
.equ GPIOD_CFGLR, 0x0
.equ GPIOD_BSHR, 0x10

.equ USART_BASE, 0x40013800
.equ USART_STATR, 0x0
.equ USART_DATAR, 0x04
.equ USART_BRR, 0x08
.equ USART_CTLR1, 0x0c

.section .text
.globl _start

_start:
    # load address _eusrstack (from linker script) into stack pointer
    la sp, _eusrstack

RCC_set_48MHz:
    li t0, RCC_BASE

    # set AHB divider to /1
    sw zero, RCC_CFGR0(t0)

    # turn on PLL
    lw t1, RCC_CTLR(t0)
    li t2, 1<<24
    or t1, t1, t2
    sw t1, RCC_CTLR(t0)
```

```asm
    # set flash latency to 1 cycle
    li t2, FLASH_BASE
    li t1, 0x01
    sw t1, FLASH_ACTLR(t2)

    # clear PLL, CSSC, HSE, HSI and LSI ready flags
    li t1, 0x009F0000
    sw t1, RCC_INTR(t0)

    # wait till PLL is ready
    li t2, 1<<25
.rcc_ctlr_pll_not_ready:
    lw t1, RCC_CTLR(t0)
    and t1, t1, t2
    beqz t1, .rcc_ctlr_pll_not_ready

    # set PLL as clock source
    lw t1, RCC_CFGR0(t0)
    ori t1, t1, 0b10
    sw t1, RCC_CFGR0(t0)

USART_115200:
    # send clock to to USART(TX=PDr) and GPIOD to enable it
    li t0, RCC_BASE
    lw t1, RCC_APB2PCENR(t0)
    li t2, 1<<14 | 1<<5
    or t1, t1, t2
    sw t1, RCC_APB2PCENR(t0)

    # make GPIOD PD5 alternate function (serial)
    li t0, GPIOD_BASE
    lw t1, GPIOD_CFGLR(t0)
    li t2, ~(0b1111<<(4*5))
    and t1,t1,t2
    li t2, 0b1001<<(4*5)
    or t1, t1, t2
    sw t1, GPIOD_CFGLR(t0)

    # enable RX,TX on USART_CTLR1
    li t0, USART_BASE
    lw t1, USART_CTLR1(t0)
    li t2, 1<<2 | 1<<3
    or t1, t1, t2
    sw t1, USART_CTLR1(t0)

    # set BR R (bit rate register) 26.042 = 48MHz/(16 * 115200); 0.667 = 0.042*16 => round to 1
    li t1, (26<<4) + 1
    sw t1, USART_BRR(t0)

    # enable uart
    lw t1, USART_CTLR1(t0)
    li t2, 1<<13
    or t1, t1, t2
    sw t1, USART_CTLR1(t0)

send_msg:
    la a0, msg
    jal Put_string

    j halt


Put_string:
    li t0, USART_BASE

.puts_loop:
    lbu t1, (a0)
    beqz t1, .puts_leave
```

```
.puts_wait:
    # check TX E (TX Empty) in status register
    lw t2, USART_STATR(t0)
    andi t2, t2,(1<<7)
    beqz t2, .puts_wait

    sw t1, USART_DATAR(t0)
    add a0,a0,1

    j .puts_loop

.puts_leave:
    ret


halt:    j halt

.section .rodata

msg:
    .string "Hello RISC-V!\n"
```

Below is the initialisation code to turn enable GPIOD and the USART.

```
USART_115200:
    # send clock to to USART(TX=PDr) and GPIOD to enable it
    li t0, RCC_BASE
    lw t1, RCC_APB2PCENR(t0)
    li t2, 1<<14 | 1<<5
    or t1, t1, t2
    sw t1, RCC_APB2PCENR(t0)
```

Now that we have the clock running to GPIOD and the USART, we configure GPIOD to be alternate function and a push-pull output.

```
    # make GPIOD PD5 alternate function (serial)
    li t0, GPIOD_BASE
    lw t1, GPIOD_CFGLR(t0)
    li t2, ~(0b1111<<(4*5))
    and t1,t1,t2
    li t2, 0b1001<<(4*5)
    or t1, t1, t2
    sw t1, GPIOD_CFGLR(t0)
```

We now get to configuring the USART. The code below turns on the USART transmitter and receiver sections. We have not actually enabled the USART yet, we do this later by setting the UE (USART Enable) bit.

```
    # enable RX,TX on USART_CTLR1
    li t0, USART_BASE
    lw t1, USART_CTLR1(t0)
    li t2, 1<<2 | 1<<3
    or t1, t1, t2
    sw t1, USART_CTLR1(t0)
```

Proceed to set the Bit Rate Register then enable the USART.

```
    # set BR R (bit rate register) 26.042 = 48MHz/(16 * 115200); 0.667 = 0.042*16 => round to 1
    li t1, (26<<4) + 1
    sw t1, USART_BRR(t0)

    # enable uart
    lw t1, USART_CTLR1(t0)
    li t2, 1<<13
    or t1, t1, t2
    sw t1, USART_CTLR1(t0)
```

With a working USART, we can send a pre-programmed message. This message is stored in the .rodata (read-only data) section on flash.

```
.section .rodata

msg:
    .string "Hello RISC-V!\n"
```

Refer again to the ABI, function call parameters are placed in a0 through a5 on rv32c. If there are more parameters, we place them on the stack. The stack also holds function-local variables. I will explore this later with C function calls.

Table 1-2 RV32E registers

Register	ABI Name	Description	Storer
x0	zero	Hardcoded 0	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporary register	Caller
x8	s0/fp	Save register/frame pointer	Callee
x9	s1	Save register	Callee
x10-11	a0-1	Function parameters/return values	Caller
x12-15	a2-5	Function parameters	Caller

Below is the function call to Put_string.

```
send_msg:
    la a0, msg
    jal Put_string

    j halt
```

`la a0, msg` : **l**oad **a**ddress into a0, the address of msg

And here is the definition of `Put_string`.

```
Put_string:
    li t0, USART_BASE

.puts_loop:
    lbu t1, (a0)
    beqz t1, .puts_leave

.puts_wait:
    # check TX E (TX Empty) in status register
    lw t2, USART_STATR(t0)
    andi t2, t2,(1<<7)
    beqz t2, .puts_wait

    sw t1, USART_DATAR(t0)
    add a0,a0,1

    j .puts_loop

.puts_leave:
    ret
```

Let's walk through the interesting parts.

`lbu t1, (a0)` : **l**oad **b**yte **u**nsigned with the byte pointed to by a0 into t1. The first time around the loop, this byte will be the first character of the `msg`.

`beqz t1, .puts_leave`: **b**ranch **eq**ual to **z**ero to label `.puts_leave` if t1 is zero. C strings are null (0, zero) terminated. Thus is we detect a zero, we are done and the Put_string function returns.

`USART STAT R` : the status register has a TX E (TX Empty) flag on bit 7. When this bit is set, the USART has sent the character in its buffer and is ready for the next character. If TX E is not set, we loop and check again.

`sw t1, USART_DATAR(t0)` : We get to this line when TX E is empty. This means we are clear to send a character. Do this by writing the byte stored in t1 to USART DATA R.

When are done writing, advance a0 to point to the next character in `msg`, then repeat.

7.6. ADC and UART

The goal is to use the ADC to measure the device's internal voltage reference (typically 1.2V).

The ADC's 10 bit output is then sent to the user's terminal in hexadecimal mode (vs ASCII).

Below is the full listing. Again only give this a quick read to get an overview and I will delve into the key areas after this.

src/ch32v003/adcvref/adcvref.S:

```
/* adcvref.S prints the value on the device's
 * internal voltage reference to the uart.
 * Connect PD5(uart TX) to the RX of your terminal.
 * 112500 8 N 1
 */

.align 2
.option rvc

.equ RCC_BASE, 0x40021000
.equ RCC_CTLR, 0x0
.equ RCC_CFGR0, 0x4
.equ RCC_INTR, 0x8
.equ RCC_APB2PCENR, 0x18

.equ FLASH_BASE, 0x40022000
.equ FLASH_ACTLR, 0x0

.equ GPIOD_BASE, 0x40011400
.equ GPIOD_CFGLR, 0x0
.equ GPIOD_BSHR, 0x10

.equ USART_BASE, 0x40013800
.equ USART_STATR, 0x0
.equ USART_DATAR, 0x04
.equ USART_BRR, 0x08
.equ USART_CTLR1, 0x0c

.equ ADC_BASE, 0x40012400
.equ ADC_STATR, 0x0
.equ ADC_CTLR1, 0x04
.equ ADC_CTLR2, 0x08
.equ ADC_SAMPTR1, 0x0c
.equ ADC_SAMPTR2, 0x10
.equ ADC_RSQR1, 0x2c
.equ ADC_RSQR2, 0x30
.equ ADC_RSQR3, 0x34
.equ ADC_RDATAR, 0x4c

.section .text
.globl _start

_start:
    # load address _eusrstack (from linker script) into stack pointer
    la sp, _eusrstack

RCC_set_48MHz:
    li t0, RCC_BASE
```

```
        # set AHB divider to /1
        sw zero, RCC_CFGR0(t0)

        # turn on PLL
        lw t1, RCC_CTLR(t0)
        li t2, 1<<24
        or t1, t1, t2
        sw t1, RCC_CTLR(t0)

        # set flash latency to 1 cycle
        li t2, FLASH_BASE
        li t1, 0x01
        sw t1, FLASH_ACTLR(t2)

        # clear PLL, CSSC, HSE, HSI and LSI ready flags
        li t1, 0x009F0000
        sw t1, RCC_INTR(t0)

        # wait till PLL is ready
        li t2, 1<<25
.rcc_ctlr_pll_not_ready:
        lw t1, RCC_CTLR(t0)
        and t1, t1, t2
        beqz t1, .rcc_ctlr_pll_not_ready

        # set PLL as clock source
        lw t1, RCC_CFGR0(t0)
        ori t1, t1, 0b10
        sw t1, RCC_CFGR0(t0)

USART_115200:
        # send clock to to USART(TX=PDr) and GPIOD to enable it
        li t0, RCC_BASE
        lw t1, RCC_APB2PCENR(t0)
        li t2, 1<<14 | 1<<5
        or t1, t1, t2
        sw t1, RCC_APB2PCENR(t0)

        # make GPIOD PD5 alternate function (serial)
        li t0, GPIOD_BASE
        lw t1, GPIOD_CFGLR(t0)
        li t2, ~(0b1111<<(4*5))
        and t1,t1,t2
        li t2, 0b1001<<(4*5)
        or t1, t1, t2
        sw t1, GPIOD_CFGLR(t0)

        # enable RX,TX on USART_CTLR1
        li t0, USART_BASE
        lw t1, USART_CTLR1(t0)
        li t2, 1<<2 | 1<<3
        or t1, t1, t2
        sw t1, USART_CTLR1(t0)

        # set BR R (bit rate register) 26.042 = 48MHz/(16 * 115200); 0.667 = 0.042*16 => round to 1
        li t1, (26<<4) + 1
        sw t1, USART_BRR(t0)

        # enable uart
        lw t1, USART_CTLR1(t0)
        li t2, 1<<13
        or t1, t1, t2
        sw t1, USART_CTLR1(t0)

send_startup_msg:
        la a0, msg0
        jal Put_string

enable_clock_to_adc:
```

```
        # set ADC PRE in RCC CFG R0 to /4 for 12Mhz ADC clock (max permited for 3.3V VDD)
        li t0, RCC_BASE
        lw t1, RCC_APB2PCENR(t0)
        li t2, 1<<9
        or t1, t1, t2
        sw t1, RCC_APB2PCENR(t0)

        # turn on ADC with AD ON and wait at least 2 ADC clock cycles (2*4 H CLK cycles)
        li t0, ADC_BASE
        lw t1, ADC_CTLR2(t0)
        ori t1, t1, 1<<0
        sw t1, ADC_CTLR2(t0)

        li t1, 2*2*4 // double the minimum
.adc_wait_init:
        addi t1,t1,-1
        bnez t1, .adc_wait_init

        # reset cal by setting RSTCAL in ADC_CTLR2, wait for bit to clear
        lw t1, ADC_CTLR2(t0)
        ori t1,t1,1<<3
        sw t1, ADC_CTLR2(t0)

        li t2, 1<<3
.adc_wait_cal_reset:
        lw t1, ADC_CTLR2(t0)
        and t1,t1,t2
        bnez t1, .adc_wait_cal_reset

        # start cal by setting CAL, wait for bit to clear
        lw t1, ADC_CTLR2(t0)
        ori t1,t1,1<<2
        sw t1, ADC_CTLR2(t0)

        li t2, 1<<2
.adc_wait_cal_done:
        lw t1, ADC_CTLR2(t0)
        and t1,t1,t2
        bnez t1, .adc_wait_cal_done

        # set the number of conversions = 1 in ADC R SQ R1 (regular sequence reg1)
        li t1, 0b0001<<20
        sw t1, ADC_RSQR1(t0)

        # set sampling time for channel 8 and 9
        lw t1, ADC_SAMPTR2(t0)
        li t2, (0b001<<8) | (0b001<<9)
        or t1,t1,t2
        sw t1, ADC_SAMPTR2(t0)

        # set the channel to be converted, Vref=ch8, for the 1st conversion, the field is in ADC_RS
        li t1, 8
        #li t1, 9
        sw t1, ADC_RSQR3(t0)

send_conversion_start_msg:
        la a0, msg1
        jal Put_string
.start_conv:
        # ADON to start measuring. Data will be in ADC_RDATAR
        li t0, ADC_BASE
        lw t1, ADC_CTLR2(t0)
        ori t1,t1, 1<<0
        sw t1, ADC_CTLR2(t0)

        # wait for end of conversion
        # EOC in ADC_STATR will be set after converson, cleared by reading ADC_RDATAR.
.wait_for_eoc:
        lw t1, ADC_STATR(t0)
```

```
        andi t1, t1, 1<<1
        beqz t1, .wait_for_eoc

        li t0, ADC_BASE
        lw a0, ADC_RDATAR(t0)
        jal Put_hex_half_word

        li a0, 4000000
        jal delay


        j .start_conv

        # data alignment in ADC_CTLR2. Default is right-aligned.
        # single channel conversion. following fields are reset: CONT,SCAN, RDISCEN/IDISCEN, IAUTO
        # EOC in ADC_STATR will be set after converson, cleared by reading ADC_RDATAR.
        # CALVOL in ADC_CTLR1 allows AD9 to be VDD/2 or 3/4*VDD
        # start conversion with SWSTART on ADC_CTLR2 ? may also need to set ADON in ADC_CTLR2.

        j halt

Put_string:
        li t0, USART_BASE

.puts_loop:
        lbu t1, (a0)
        beqz t1, .puts_leave

.puts_wait:
        # check TX E (TX Empty) in status register
        lw t2, USART_STATR(t0)
        andi t2, t2,(1<<7)
        beqz t2, .puts_wait

        sw t1, USART_DATAR(t0)
        add a0,a0,1

        j .puts_loop

.puts_leave:
        ret


Put_hex_half_word:
        mv a1,a0
        srli a0,a0,8
        li t0, USART_BASE
        jal t0, Put_char

        mv a0,a1
        jal t0, Put_char

        ret

Put_char:
        li t2, USART_BASE
.put_char_wait:
        lw t1, USART_STATR(t2)
        andi t1, t1, (1<<7)
        beqz t1, .put_char_wait
        sb a0, USART_DATAR(t2)
        jalr zero,t0,0

delay:
        addi a0,a0,-1
        bnez a0, delay
        ret

halt:     j halt
```

```
    .section .rodata

msg0:
    .string "reading Vref on analog in 7\n"
msg1:
    .string "ADC conversion started.\nPlease switch to hexadecimal mode on your terminal.\n"
```

The ADC is connected to GPIO pins and you will need to configure the GPIO ports accordingly to measure external signals.

I am however only measuring an internal signal on AN08 (Analog input 8). AN08 is not connected to GPIO pins and hence I've skipped the GPIO initialisation.

```
enable_clock_to_adc:
    # set ADC PRE in RCC CFG R0 to /4 for 12Mhz ADC clock (max permited for 3.3V VDD)
    li t0, RCC_BASE
    lw t1, RCC_APB2PCENR(t0)
    li t2, 1<<9
    or t1, t1, t2
    sw t1, RCC_APB2PCENR(t0)
```

The above code is pretty standard. It sends the clock to the ADC module. Below is where the ADC is actually turned on with the AD ON command. Note the requisite delay. I did not count cycles and chose to err to delay longer the necessary.

```
    # turn on ADC with AD ON and wait at least 2 ADC clock cycles (2*4 H CLK cycles)
    li t0, ADC_BASE
    lw t1, ADC_CTLR2(t0)
    ori t1, t1, 1<<0
    sw t1, ADC_CTLR2(t0)

    li t1, 2*2*4 // double the minimum
.adc_wait_init:
    addi t1,t1,-1
    bnez t1, .adc_wait_init
```

Next comes the calibration reset and calibration routine. This allows the ADC to run an internal calibration to improve accuracy.

```
    li t2, 1<<3
.adc_wait_cal_reset:
    lw t1, ADC_CTLR2(t0)
    and t1,t1,t2
    bnez t1, .adc_wait_cal_reset

    # start cal by setting CAL, wait for bit to clear
    lw t1, ADC_CTLR2(t0)
    ori t1,t1,1<<2
    sw t1, ADC_CTLR2(t0)

    li t2, 1<<2
.adc_wait_cal_done:
    lw t1, ADC_CTLR2(t0)
    and t1,t1,t2
    bnez t1, .adc_wait_cal_done
```

Next came the channel selection. This confused me, having to re-read the reference manual several times. My understanding is the ch32v003's ADC has external AN0 through AN7 and two internal channels AN8 and AN9. The ADC is able to sample AN0 through AN9 as a sequence of up to 16 measurements. The first in the sequence could be to measure, say, AN5, the second, AN3 and later on the 10th sequence AN5 again.

The first thing to do to program this sequence of sampling is to first decide the length of your sampling sequence. For this example, I chose the minimum of 1.

More confusingly, there are two sampling sequences. The "regular" sequence and the "injection" sequence. The regular sequence has access to DMA while the injection sequence does not. I will use only the regular sequence in this example.

```
        # set the number of conversions = 1 in ADC R SQ R1 (regular sequence reg1)
        li t1, 0b0001<<20
        sw t1, ADC_RSQR1(t0)

        # set sampling time for channel 8 and 9
        lw t1, ADC_SAMPTR2(t0)
        li t2, (0b001<<8) | (0b001<<9)
        or t1,t1,t2
        sw t1, ADC_SAMPTR2(t0)

        # set the channel to be converted, Vref=ch8, for the 1st conversion, the field is in ADC_RS
        li t1, 8
        #li t1, 9
        sw t1, ADC_RSQR3(t0)
```

I have commented out selection of channel 9 (V CAL -- connected to VDD / 2) so that I may easily switch between channel 9 (Vref) by commenting/uncommenting lines.

According to the data sheet the maximum ADC clock rate when powered with 3.3V is 12MHz. The minimum sampling time is 3 ADC clock cycles (0.25us or 250ns). I decided to go with 9 cycles (0.75us). This is acceptable as the sampling capacitor is only 3pF, even with in input impedance of 100 Ohms, the time constant is 0.3ns. 10 time constants is 3ns.

Let's move on the the actual ADC and data capture. In the conversion start message, I prompt the user to switch the terminal to hex mode. To avoid rewriting printf, I will be sending hex values and not ASCII characters. The ADC places the data after conversion in R DATA R (regular data register). That byte is then loaded into a0 before calling Put_hex_half_word. And that is when this code becomes interesting.

```
    send_conversion_start_msg:
        la a0, msg1
        jal Put_string
    .start_conv:
        # ADON to start measuring. Data will be in ADC_RDATAR
        li t0, ADC_BASE
        lw t1, ADC_CTLR2(t0)
        ori t1,t1, 1<<0
        sw t1, ADC_CTLR2(t0)

        # wait for end of conversion
        # EOC in ADC_STATR will be set after converson, cleared by reading ADC_RDATAR.
    .wait_for_eoc:
        lw t1, ADC_STATR(t0)
        andi t1, t1, 1<<1
        beqz t1, .wait_for_eoc

        li t0, ADC_BASE
        lw a0, ADC_RDATAR(t0)
        jal Put_hex_half_word

        li a0, 4000000
        jal delay


        j .start_conv
```

Please read the code below and note that I call a further subroutine Put_char within the subroutine Put_hex_half_word.

```
    Put_hex_half_word:
        mv a1,a0
        srli a0,a0,8
        li t0, USART_BASE
        jal t0, Put_char

        mv a0,a1
        jal t0, Put_char

        ret
```

```
Put_char:
    li t2, USART_BASE
.put_char_wait:
    lw t1, USART_STATR(t2)
    andi t1, t1, (1<<7)
    beqz t1, .put_char_wait
    sb a0, USART_DATAR(t2)
    jalr zero,t0,0
```

Put_char is called in an unusual way: `jal t0, Put_char` rather than just `jal Put_char`.

The regular form `jal Put_char` actually assembles to `jal ra, Put_char` and the `ret` psuedoinstruction assembles to `jalr x0,x1,0` or `jalr zero,ra,0` as x0 = zero and x1 = ra.

The call to Put_hex_half_word has already taken up ra. A nested call within Put_hex_half_word would replace the value in ra. Instead we use x5 (or t0) instead to call Put_char.

We then do not use `ret` but jalr zero,t0,0 instead.