

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Polling and interrupts are only relevant concepts for low level programming.

False. Similar concepts apply to almost all types of applications, including web apps, mobile apps, distributed systems, and so on.

1.2 Memory-mapped IO only works with polling.

False. The implementation backing the memory mapping can use interrupt-driven IO (for example, reading files).

## 2 Polling & Interrupts

2.1 Fill out this table that compares polling and interrupts.

Operation	Definition	Pro/Good for	Con
Polling	Forces the hardware to wait on ready bit (alternatively, if timing of device is known, the ready bit can be polled at the frequency of the device).	<ul style="list-style-type: none"> <li>• Low Latency</li> <li>• Low overhead when data is available</li> <li>• Good For: devices that are <u>always busy</u> or when you can't make progress until the device replies</li> </ul>	<ul style="list-style-type: none"> <li>• Can't do anything else while polling</li> <li>• Can't sleep while polling (CPU always at full speed)</li> </ul>
Interrupts	Hardware fires an "exception" when it becomes ready. CPU changes PC register to execute code in the interrupt handler when this occurs.	<ul style="list-style-type: none"> <li>• Can do useful work while waiting for response</li> <li>• Can wait on many things at once</li> <li>• Good for: Devices that <u>take a long time to respond, especially</u> if you can do other work while waiting.</li> </ul>	<ul style="list-style-type: none"> <li>• Nondeterministic when interrupt occurs</li> <li>• interrupt handler has some overhead (e.g. saves all registers, flush pipeline, etc.)</li> <li>• Higher latency per event</li> <li>• Worse throughput</li> </ul>

## 3 Memory Mapped I/O

3.1 For this question, the following addresses correspond to registers in some I/O devices and not regular user memory.

- 0xFFFF0000—Receiver Control: LSB is the ready bit (in the context of polling), there may be other bits set that we don't need right now.
- 0xFFFF0004—Receiver Data: Received data stored at lowest byte.
- 0xFFFF0008—Transmitter Control: LSB is the ready bit (in the context of polling), there may be other bit set that we don't need right now.
- 0xFFFF000C—Transmitter Data: Transmitted data stored at lowest byte.

Recall that receiver will only have data for us when the corresponding ready bit is 1, and that we can only write data to the transmitter when its ready bit is 1. Write RISC-V code that reads byte from the receiver (busy-waiting if necessary) and writes that byte to the transmitter (busy-waiting if necessary).

```

                                lui t0 0xffff0
receive_wait: lw t1 0(t0)
                                andi t1 t1 1           # poll on ready of receiver
                                beq t1 x0 receive_wait
                                lb t2 4(t0)             # load data
```

```
transmit_wait: lw t1 8(t0)          # poll on ready of transmitter
                andi t1 t1 1
                beq t1 x0 transmit_wait # write to transmitter
                sb t2 12(t0)
```