Recall from the lecture on I/O that there were generally two options: polling and interrupts. The approach we looked at then used polling: the CPU actively inspected the status register over and over until it was set, and then it could proceed.

Clearly this is wasteful: the CPU is not doing any “useful” work while waiting for the status register to be set. In practice, it cannot sit around waiting for:
- incoming network data
- mouse or keyboard events
- files to be ready for transfer from the disk

So interrupt-driven I/O allows the program to do work until the CPU gets a message (“interrupt”) from the I/O device, at which time a special piece of software (known as the interrupt handler) is automatically invoked. The interrupt handler may be responsible for:
- moving data from an external source into memory
- moving data from one part of memory to another
- moving data from a register into memory (or vice-versa)

For interrupt-driven I/O to work, each interrupt-capable device must have a special register that is part of the CPU. The register includes:
- a Ready flag, indicating whether or not the device wants to interrupt
- an Interrupt Enabled flag, indicating whether the CPU is currently willing to accept interrupts from this device
- a Priority; for instance, this could range from 0 (least important) to 7 (most important)
- an Interrupt Vector: the address in the interrupt vector table that holds the starting address of the interrupt handler (analogous to the trap vector table and service routines)

When the device is ready (i.e., it wants to interrupt), it sets its Ready flag to 1.

Before the start of the “fetch” phase, the CPU checks the Ready and Interrupt Enabled flags for all interrupt-capable devices. If any device has both flags set, and the Priority for that device is higher (more important) than the priority of the currently running program, then the interrupt with the highest priority will be handled.

For an interrupt to be handled:
1. the “state” of the program must be stored; this includes:
   - the value of all registers
   - the PC
   - the Program Status Register (PSR), which holds this program's mode (user or supervisor), priority, and condition codes
2. the interrupt vector is read from the interrupt register (e.g., this may be a number between x00 and xFF)
3. it is added to some value (e.g. 0x0100) which is the starting address of the interrupt vector table
4. that address is read to get the starting address of the interrupt handler
5. the PC is set to that value and the PSR is set accordingly
When the interrupt handler completes, it calls **RTI** (“return from interrupt”) to restore the registers, PC and PSR so that the interrupted program can continue.

Note that, like traps (service routines), the interrupt handlers are implemented in the operating system. Unlike traps, though:

- your program does not directly call an interrupt handler
- the interrupt handler mechanism requires hardware support

What if an interrupt handler gets interrupted? This can certainly happen if an interrupt arrives from a device with higher priority.

For instance, let's say your program (we'll call it A) is executing and it gets interrupted. Now the interrupt handler (we'll call it B) starts running. If that gets interrupted, and interrupt handler C starts, then what happens when C finishes? We should go back to B. And when B finishes, we should go back to A. So we need to make sure we:

- keep track of the order in which things were interrupted
- make sure we don't lose the state (registers, PC, PSR) of the programs that get interrupted

To do this, we use a data structure called a **stack**. A stack is a LIFO (“last in, first out”) data structure, and you can think of it like a stack of plates. The last one you add to the top of the stack (we call this “pushing” onto the stack) is the first one you use when you need to remove one (this is known as “popping”).

When an interrupt is to be handled, the state (registers, PC, PSR) of the interrupted program are pushed onto the **Supervisor Stack**, which starts at a specific address in memory and has a dedicated register (Supervisor Stack Pointer, or SSP) that indicates the “top” of the stack. The handling of subsequent interrupts pushes more things onto the stack.

When RTI is called to end the interrupt handler, the state on the top of the stack is popped and the most-recently-interrupted program is restored.