Interrupt Handling & Device Drivers

Computer Science 432 — Lecture 10 — Duane Bailey

March 7, 2022

Announcements

- Work continues on Lab 5, the COW optimization. Due on Friday.
- Small group D will meet at the usual time. Topic: Lab 3.
- Office Hours (this week, only): M1-2, T1-3, W1-4, F9-10:30
- O/S Conference topic contracts out on Wednesday.
- Early April: SEM Microscope lab during small group meetings (optional)

Interrupt Handling

- Recall: One type of trap is an *interrupt*, an asynchronous request
  - from a device, or
  - from software (e.g. from the timer unit)

- Device interrupts are dispatched through `devintr`, called from
  - `usertrap()` through `uservec` if user mode was interrupted, or
  - `kerneltrap()` through `kernelvec` if supervisor mode was interrupted, or
  - (or…through `timervec` if timer is involved; details later)

- Devices are identified by their IRQ (interrupt request) id's:
  - e.g. UART0 has IRQ 10

Device Drivers

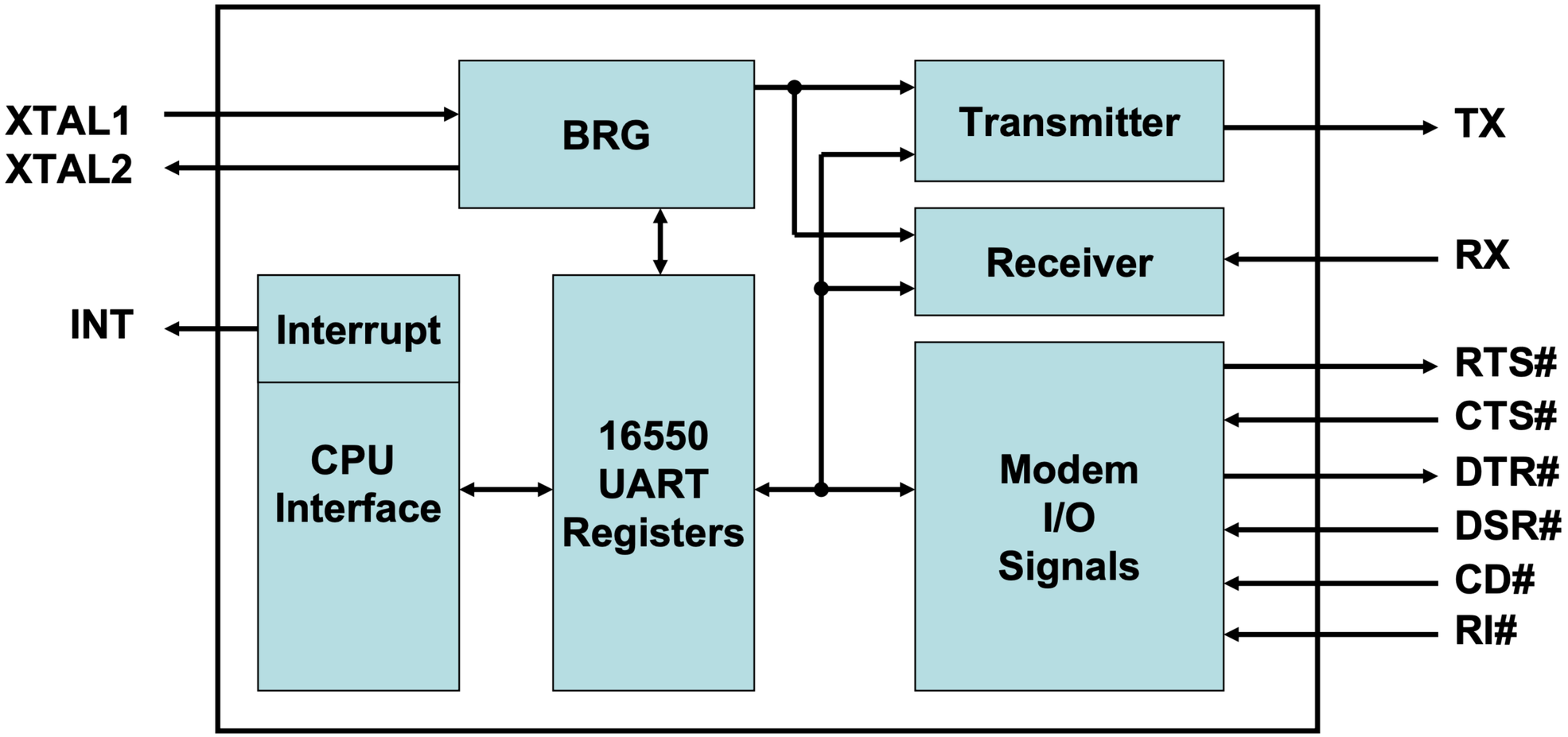
- Device drivers are modules that manage the interaction between devices and the O/S
  - The *top half* of the driver is the interface with the operating system, accessed by the kernel
    - Is called at convenience of the kernel.
    - Manages device-specific resources
  - The *bottom half* of the driver quickly manages the device at the time of interrupt
    - Could happen in any context
    - Lightweight design

Example: The Console Driver

- The *console* is the device that supports direct keyboard and display interactions with the user through a *serial port*.
  - Interfaces with a traditional UART (*universal asynchronous resceiver-transmitter*).
    A "16550" chip containing an internal FIFO of characters
  - Supports serial, full-duplex (bi-directional) transmission of bits over "*twisted pair*" (RS232).
  - Communication with UART appears through *memory mapped registers* (see `kernel/memlayout.h` and `kernel/uart.c`)

  - Gathers *lines* of input with minimal *editing* (backspace, EOF, newline, ^u, ^p)

UART Block Diagram



Primary 16550 UART Registers

- The UART contains several registers that are of interest to the console driver:
  - `IER`—the *interrupt identification register*—controls the interrupt generation
    - writing bits in this register activate/deactivate interrupts for receiving & transmission
  - `LSR`—the *line status register*—keeps track of the state of the UART
    - bits within the register identify UART state: ready for read, etc.
  - `RHR`—the *receive holding register*—the dequeue end of the receive FIFO
    - if this register is read, the character is dequeued from the FIFO
  - `THR`—the *transmit holding register*—the enqueue end of the transmit FIFO
    - if this register is written, the character will eventually be transmitted

Address A2-A0	Register Name	Read/Write	Register Function	Comment
0 0 0	DLL – Divisor LSB	Write-Only	Divisor (LSB) for BRG	LCR bit-7 = 1
0 0 1	DLM – Divisor MSB	Read-Only	Divisor (MSB) for BRG	LCR bit-7 = 1
0 0 0	THR – Transmit Holding Register	Write-Only	Loading data into TX FIFO	LCR bit-7 = 0
0 0 0	RHR – Receive Holding Register	Read-Only	Unloading data from RX FIFO	LCR bit-7 = 0
0 0 1	IER – Interrupt Enable Register	Read/Write	Enable interrupts	
0 1 0	FCR – FIFO Control Register	Write-Only	FIFO enable and reset	
0 1 0	ISR – Interrupt Status Register	Read-Only	Status of highest priority interrupt	
0 1 1	LCR – Line Control Register	Read/Write	Word length, stop bits, parity select, send break, select divisor registers	
1 0 0	MCR – Modem Control Register	Read/Write	RTS# and DTR# output control Interrupt output enable Internal Loopback enable	
1 0 1	LSR – Line Status Register	Read-Only	RX Errors/Status TX Status	
1 1 0	MSR – Modem Status Register	Read-Only	Modem Input Status	
1 1 1	SPR – Scratch Pad Register	Read/Write	General Purpose Register	

Table 5. Description of Internal Register Bits

Internal Register	Bit #	Access	Description
Receive Buffer Register (0x00)	7:0	R	Holds received data
Transmitter Holding Register (0x00)	7:0	W	Holds data to be transmitted
Interrupt Enable Register(0x01)	7:4	RW	Unimplemented
	3	RW	'1' enables Modem Status Interrupt
	2	RW	'1' enables Receiver Line Status Interrupt
	1	RW	'1' enables Transmitter Holding Register Empty Interrupt
	0	RW	'1' enables Received Data Available Interrupt
Interrupt Identification Register (0x02)	7:3	R	Unimplemented
	2:0	R	'001' None '110' Receiver Line Status Interrupt '100' Received data available '010' Transmitter Holding Register Empty '000' Modem Status Interrupt Refer to Table 3 for more details about IIR

Internal Register	Bit #	Access	Description
Line Status Register	7	R	Logic '0'
	6	R	'1' indicates Transmitter FIFO and Transmitter Shift Register are both empty
	5	R	'1' indicates Transmitter Holding Register empty
	4	R	'1' indicates break condition
	3	R	Framing Error Indicator '1' indicates received character did not have a valid stop bit
	2	R	Parity Error Indicator '1' indicates received character did not have a correct parity
	1	R	Overrun Error Indicator '1' occurs when RCVR FIFO is full
	0	R	'1' indicates Receiver Data ready

Console Overview

- The console driver code is found in `kernel/console.c`
  - `consoleinit`—initializes the UART to generate two different interrupts:
    - *receive* interrupt: whenever a character is received by the UART input FIFO.
    - a *transmit complete* interrupt: whenever a character is successfully sent from output FIFO

Console Reading

- Bottom Half: Lowest-level *receive interrupt* managed by `uartintr()` (`kernel/uart.c`)
  - hands off characters from UART to `consoleintr()`
  - `consoleintr()` performs basic interpretation of editing.
  - wakes any waiting readers on newline

- Top half: Input from the console:
  - triggered by a `read()` system call.
  - handled by `consoleread()`
  - reads from the internal buffer, `cons.buf`.
  - returns only when an entire line is read
  - sleeps, otherwise, to be woken by `consoleintr()`

Console Writing

- Bottom Half: Lowest-level *transmit complete* interrupt managed by `uartintr()` (`kernel/uart.c`)
  - calls `uartstart()` to have UART start transmitting next character

- Top half: Output to the console:
  - triggered by a `write()` system call.
  - handled by `uartputc()` (`kernel/uart.c`).
  - writes to the internal buffer, `uart_tx_buf`.
  - waits only if the buffer is full

Supporting Concurrency

- Some important things to note:
  - There is only one console device.
  - Buffers are used to communicate between bottom and top of the console drivers.
  - Locks (we'll see soon) are necessary to maintain critical structures.
  - Interrupt handling may happen in any context: in any process or the kernel.

Timer Interrupt Handling

- Scratch area set off for machine-mode interrupts in `start()` (`kernel/start.c`):
  - Saves only a few registers
  - Sets the machine interrupt vector to `timervec` (`kernel/kernelvec.c`)

- Periodically, a timer goes off, reminding the machine to swap tasks.
  - Interrupts are machine-level.
  - Very simple approach
    - Schedule another timer event
    - Raise a *software interrupt* (landing in trap.c's `devintr`).
  - Allows kernel to carefully control scheduling, through `yield()`