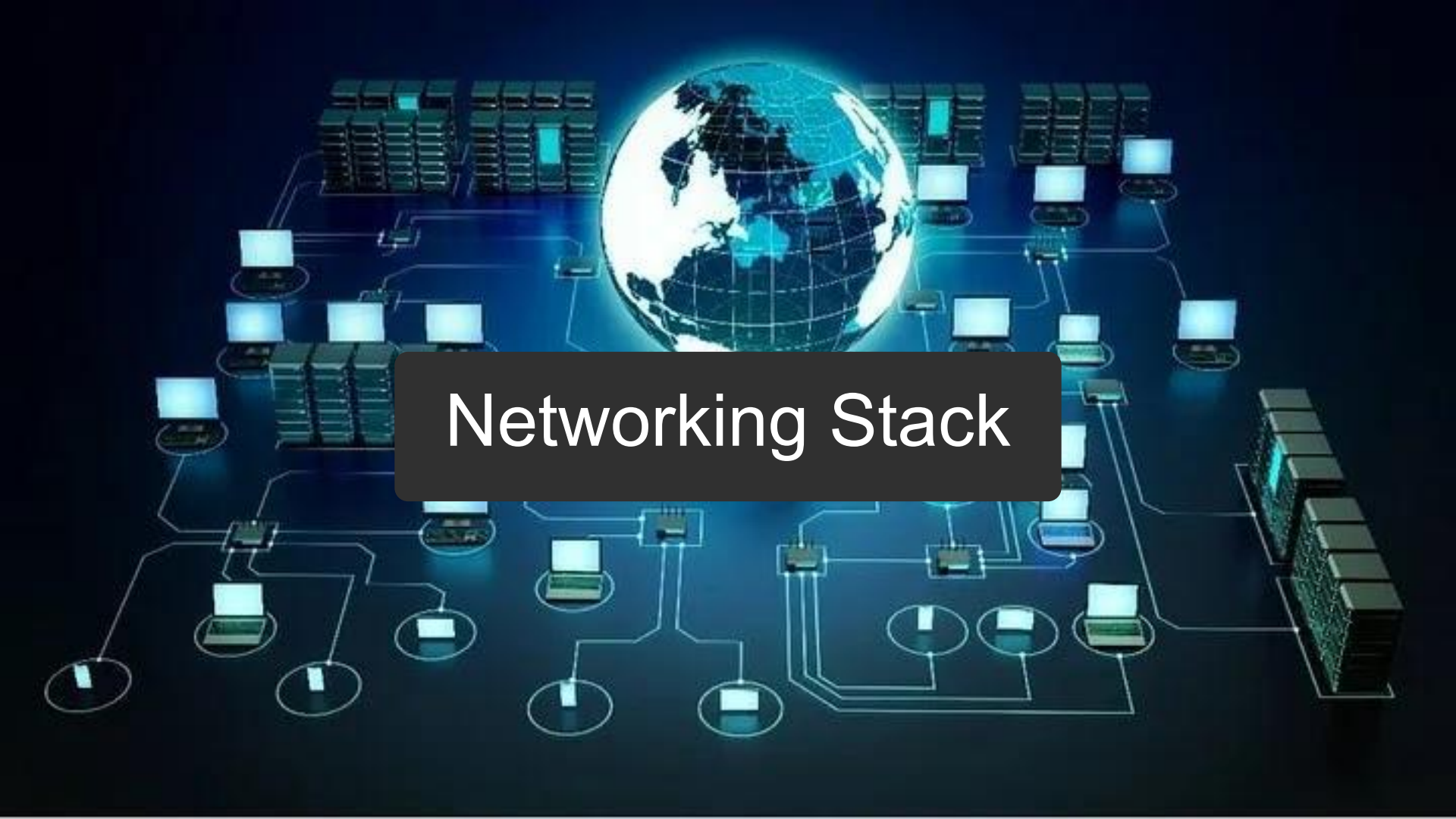
Team 3’s Operating System

Aaron Chan, Will Bicks, Nick Merante, John Arrandale

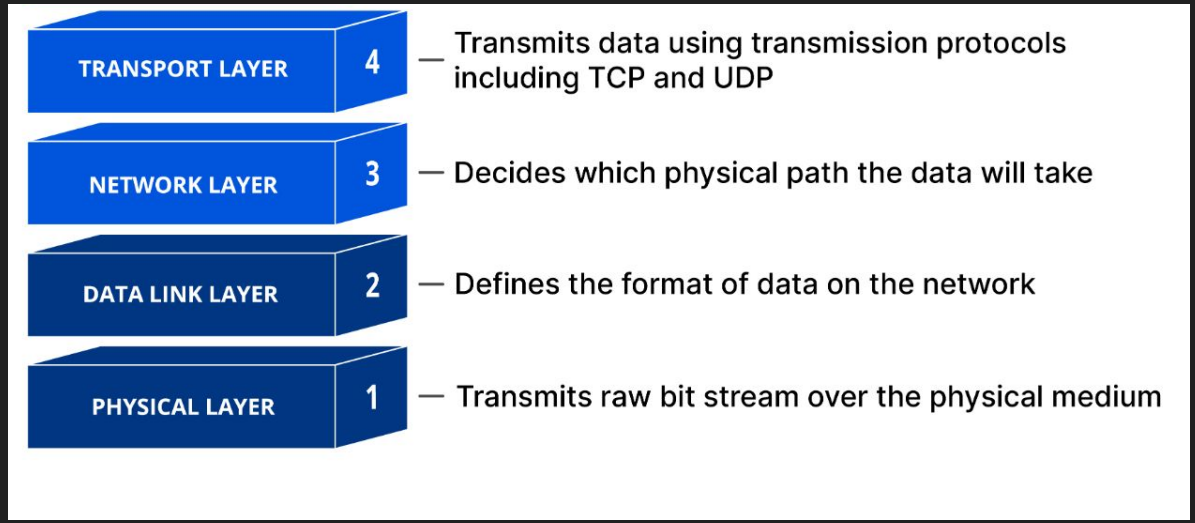
Project Overview

- Four projects
  - Networking Stack
    - Support communicating with other computers over a network
  - VGA Graphics Mode
    - Modular VGA driver for 256-color graphics with drawing and timing primitives
  - Audio Interface
    - Add a driver for the Intel High Definition Audio specification to generate and output audio
  - DMX512 Support
    - Add a driver supporting the DMX512 protocol to control stage and architectural lighting


Networking Stack

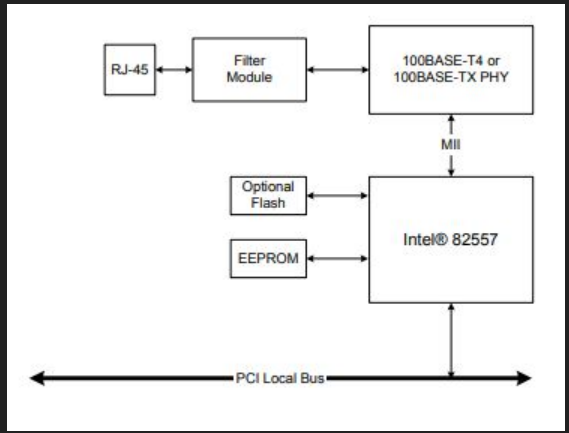
Network Stack Support

- Design philosophies
  - Platform independence
  - Testability
  - Modularity
- Implement driver support for an Ethernet chip
- Implements protocols from layers 2-4
  - RFCs used as reference



Physical Layer: Intel 8255x

- DSL Machines equipped with both an Intel 8255X and Intel I219-V Ethernet controller
  - Chose the Intel 8255X for simplicity and better references
- No socket offloading
  - OS required to provide physically contiguous buffers
  - Remaining network stack must be written in software
- MAC stored in EEPROM
  - Requires interfacing with Ethernet II software module
- QEMU offers simulation support (eepro100)
- Platform independence maintained by having function pointers for interfacing over PCI
  - Allowed for testing in xv6 while baseline OS was having issues




```
amc9897@sholto:~$ lspci | grep Ethernet
00:1f.6 Ethernet controller: Intel Corporation Ethernet Connection (2) I219-V
03:00.0 Ethernet controller: Intel Corporation 82557/8/9/0/1 Ethernet Pro 100 (rev 08)
amc9897@sholto:~$
```

Physical Layer: Intel 8255x

- Brute-force detect the device over the PCI bus
  - Searching for a device with vendor ID 0x8086 and device ID 0x1229
  - 0xCF8 for accessing the PCI Config Address Register
  - 0xCFC for accessing the PCI Data Register
- Issue a port reset command
- Grab BAR0, for doing MMIO access
- Read EEPROM for MAC address
  - Needed for initializing Ethernet II in software later…
- Allocate buffers and setup interrupts

```
uint32_t pci_read_config(uint8_t bus, uint8_t device, uint8_t func, uint8_t offset) {
    uint32_t address =
        (1 << PCI_CONFIG_ENABLE_BIT_OFFSET) |
        (bus << PCI_CONFIG_BUS_NUMBER_OFFSET) |
        (device << PCI_CONFIG_DEV_NUMBER_OFFSET) |
        (func << PCI_CONFIG_FUN_NUMBER_OFFSET) |
        (offset & 0xFC);  // Register number (must be 4-byte aligned)

    // Write address to PCI config space
    outl(PCI_CONFIG_ADDRESS_REGISTER, address);

    // Read results from PCI config space
    return inl(PCI_CONFIG_DATA_REGISTER);
}
```

6.3.3.1 PORT Software Reset

The Port Software Reset is synonymous with the software reset and is used to issue a complete reset to the device. Software must wait for ten system clocks and five transmit clocks before accessing the SCB registers again. (This may be a conservative 10 µs delay loop in software.) A software reset clears the device CSR and the PCI master block internal registers. It also requires the device to be completely re-initialized.

Link, Network and Transport Layers

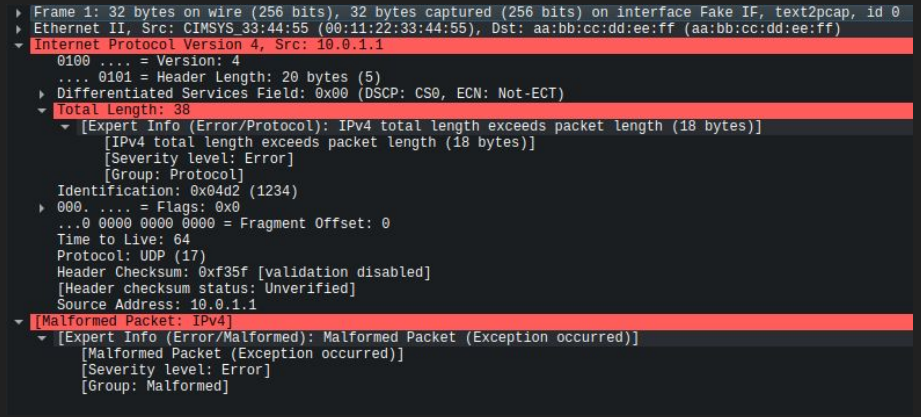
- Most protocols easy to implement in software
- Reference RFC for the specification for each protocol
  - Ethernet II: RFC 894
  - IPv4: RFC 791
  - UDP: RFC 768
  - ARP: RFC 826
- Function pointers for implementing general functionality
  - Initialization
  - Payload Set
  - Packet Send/Serialize
  - Packet Receive/Deserialize

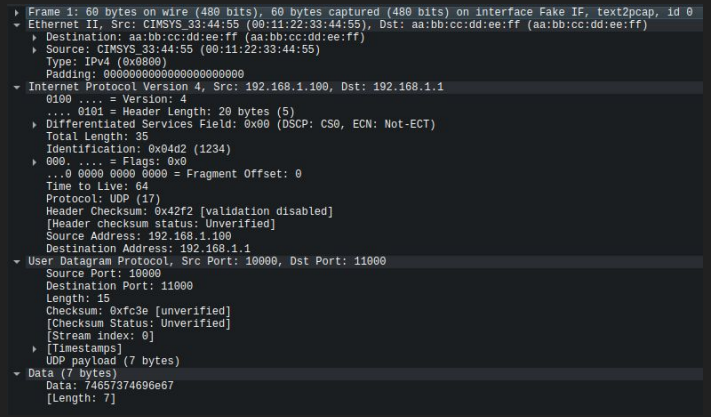
```
~/Development/RIT/CSCI452/452-OS
aa bb cc dd ee ff 00 11 22 33 44 55 08 00 45 00
00 23 04 d2 00 00 40 11 42 f2 c0 a8 01 64 c0 a8
01 01 27 10 2a f8 00 0f fc 3e 74 65 73 74 69 6e
67 00 00 00 00 00 00 00 00 00 00 00
```

```
typedef void (*init_fn)(void *packet, ...);
typedef void (*set_payload_fn)(void *packet, const uint8_t *payload,
                               uint32_t length);
typedef uint32_t (*transmit_fn)(const void *packet, uint8_t *buffer,
                                uint32_t buffer_size, ...);
typedef uint32_t (*receive_fn)(const void *packet, uint8_t *buffer,
                               uint32_t buffer_size, ...);
```

Network Stack Testing

- Layers 2-4 tested using Wireshark
- Build a test program that glues together the different protocols
  - Ethernet II - IPv4 - UDP
- Print out the resulting buffer, do a text2pcap and run it through Wireshark



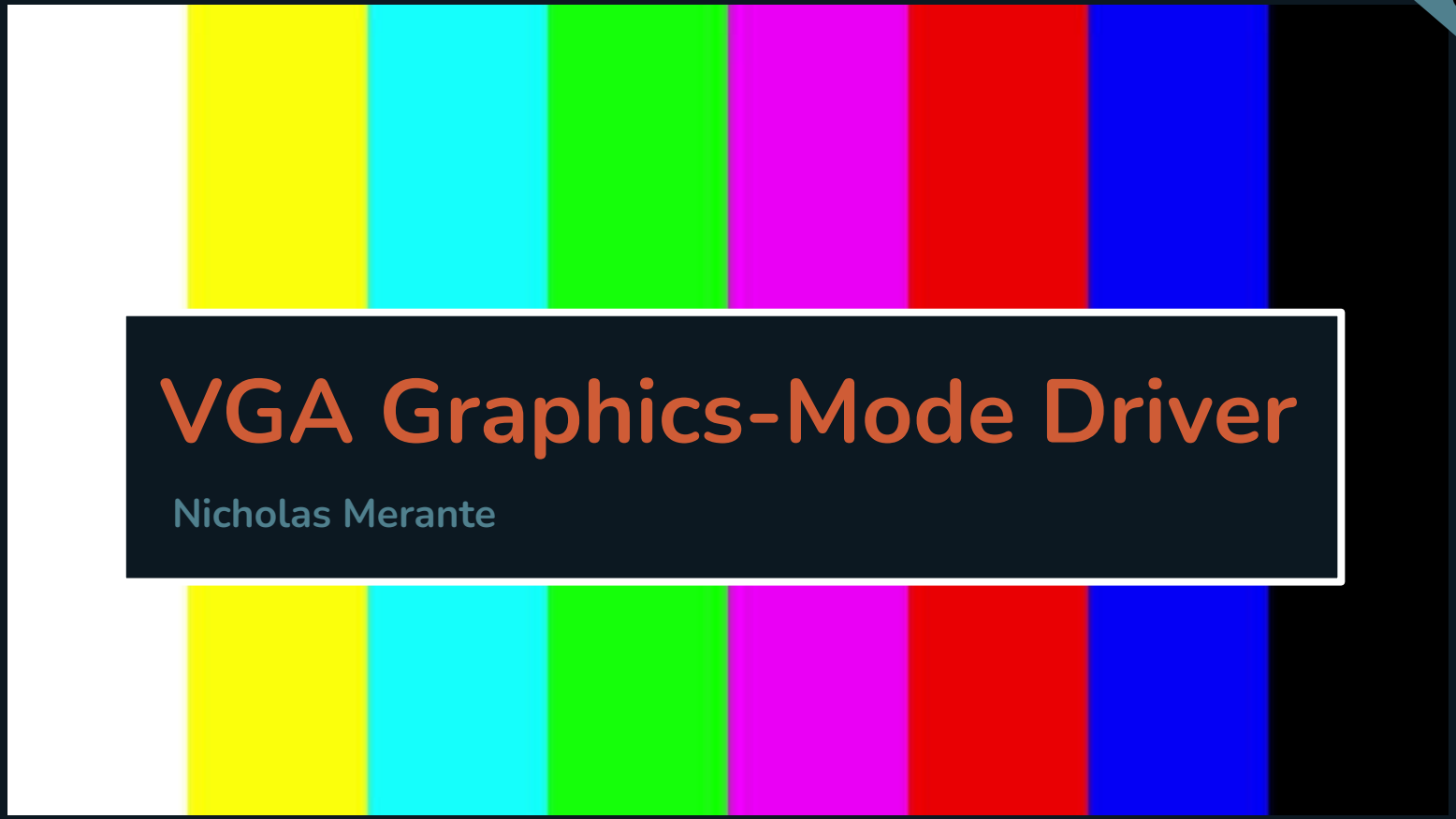


Integration

- Define platform-specific register access functions for the driver to use
- Initialize the 8255X and rest of the networking stack on startup
  - Issue a PORT RESET command
  - Read MAC address and initialize Ethernet II software module
  - Allocate buffers for Tx/Rx
  - Setup interrupts
- Interrupts triggered when the wire receives something
  - Read data from the buffer it points to
  - Once data is processed, recycle the buffer by clearing the status and making it available again
- System calls for sending and receiving packets
  - User applications can see what was received and send packets over a network

Challenges

- Interfacing with the Intel 8255X
  - Struggled to perform a few operations like getting the device initialized
  - No control over whether the NIC has to be interfaced with through I/O or MMIO
    - Bit 0 of PCI BAR indicates how the device should be interfaced with
  - The datasheet is ~200 pages long…
- Maintaining platform independence
  - Needed to see the network packet buffer easily during testing (i.e. printf calls)
  - Functions like memcpy where heavily used throughout the stack
  - Had to deal with compiling network stack like it was a normal C program and have it compile with the OS

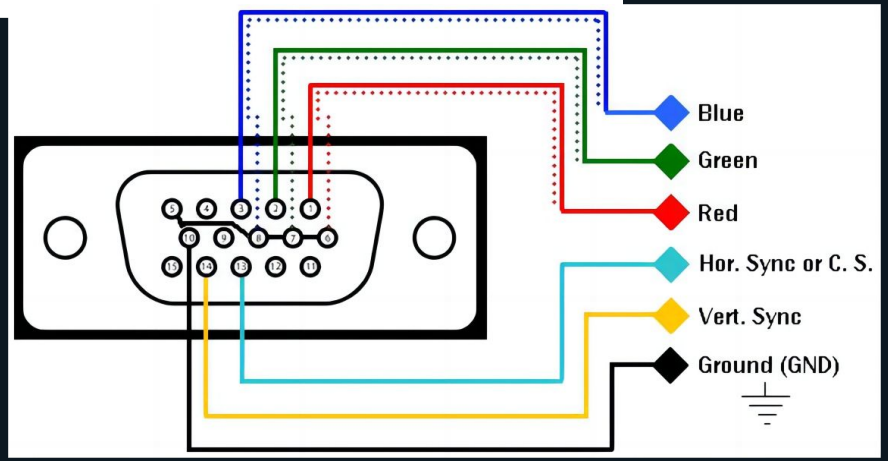


VGA Graphics-Mode Driver

Nicholas Merante

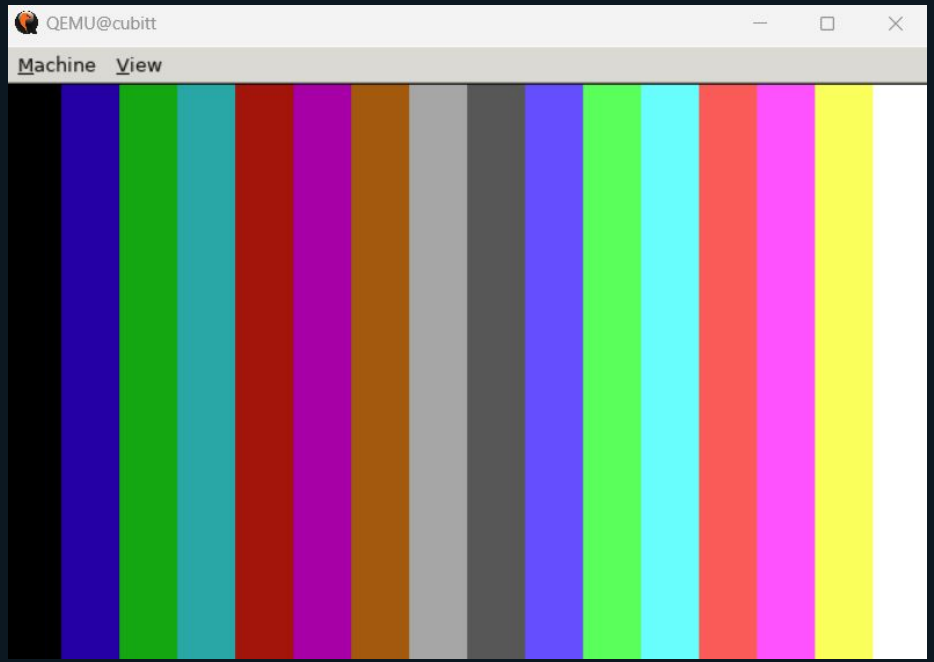
Driver Overview

- Implement VGA Mode 13h graphics driver for 320×200, 256 colors
- Provide primitives: pixels, lines, rectangles, circles, sprites
- Add double-buffering, palette control, vsync
- Demo applications: drawing tests, sprite demo, Snake game



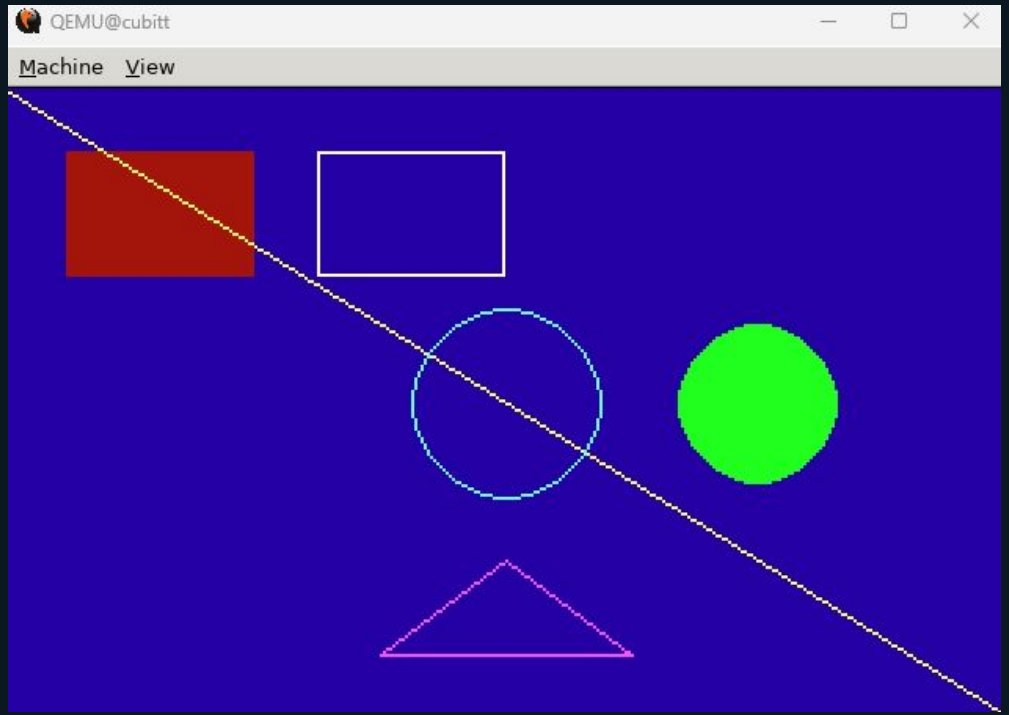
Hardware & Registers

- Frame Buffer (Mode 13h)
  - Physical address: 0xA0000–0xAFFFF (64 KiB)
  - Each byte = one pixel's 8-bit palette index
- Vertical-Retrace Status
  - Port 0x3DA – Input Status Register 1
  - Bit 3 = 1 during VBlank → use in vga_wait_vsync() to avoid tearing
- Mode Setting
  - BIOS INT 10h AH=0x00, AL=0x13 sets up CRTC and Sequencer registers (0x3C0–0x3DF) for 320×200×256 mode
  - Happens in boot.S before protected mode



Design Philosophy

- Keep API simple and modular
- Separate buffer vs. direct VRAM access
  - Primitive functions write directly to VRAM, e.g. for shapes demo here →
  - Buffered function calls for animated visuals
- Use Bresenham and Midpoint algorithms for efficiency

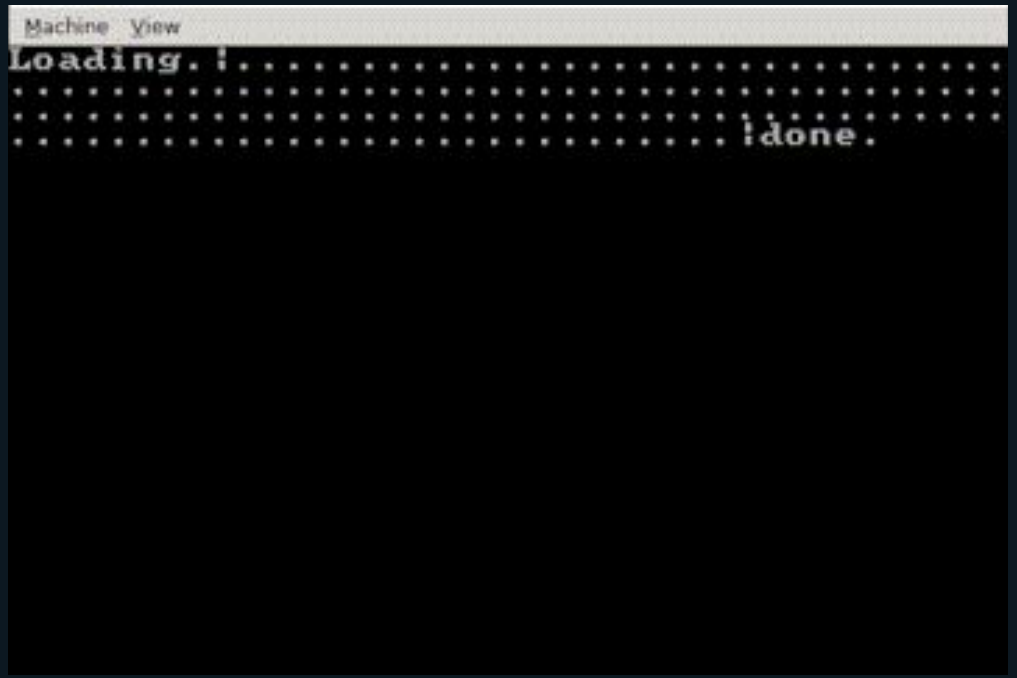


Sprite/Buffer Demo

- Drawing primitives test
- Sprite blitter test
- Double buffer test
  - Backbuffer is written to as data updates
  - Once VSYNC is resetting for a new frame, entire backbuffer is dumped to VRAM
  - Reduce flickering

Text Console Demo

- Basic character typing console
- Not as complete as text-mode one in cio.c, but a good starting point
- Proof of concept that basic font can be loaded in using the driver
- For now, all calls to write to console (normally text-mode VGA) are redirected to serial

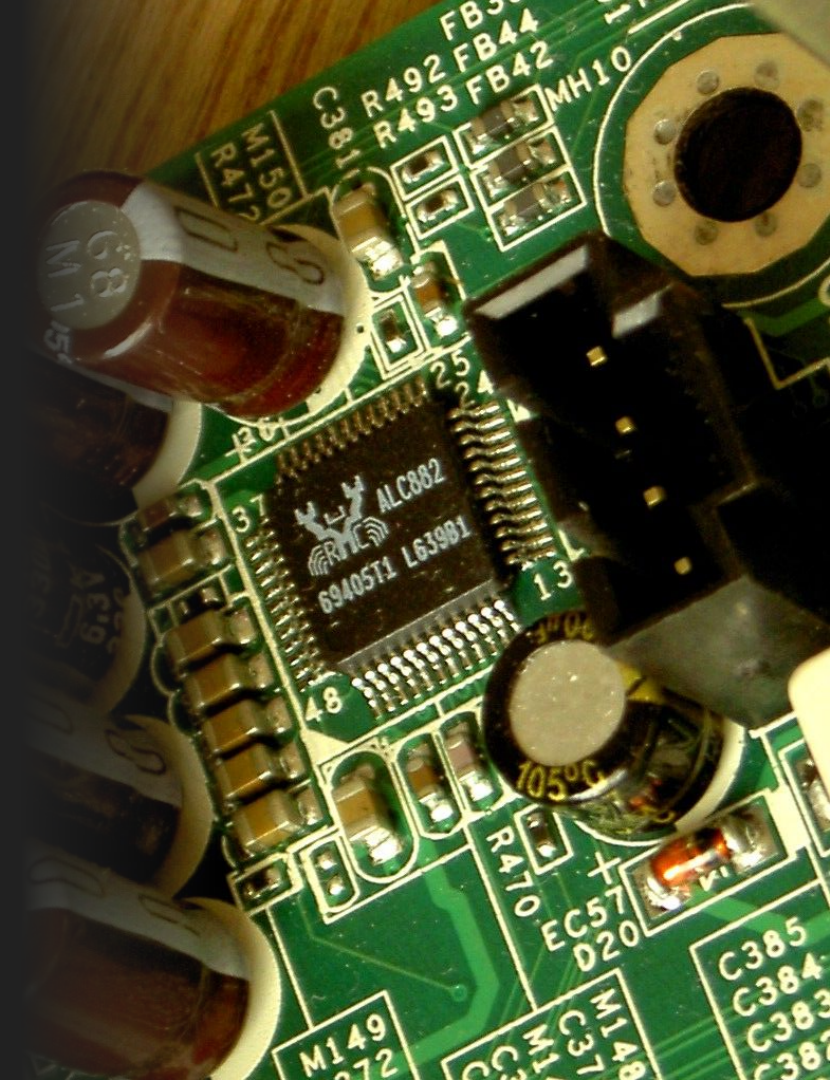


Conclusion

- Snake game showcasing full driver
- Modular VGA driver for 256-color graphics
- Robust drawing and timing primitives
- Extensible for future games and demos

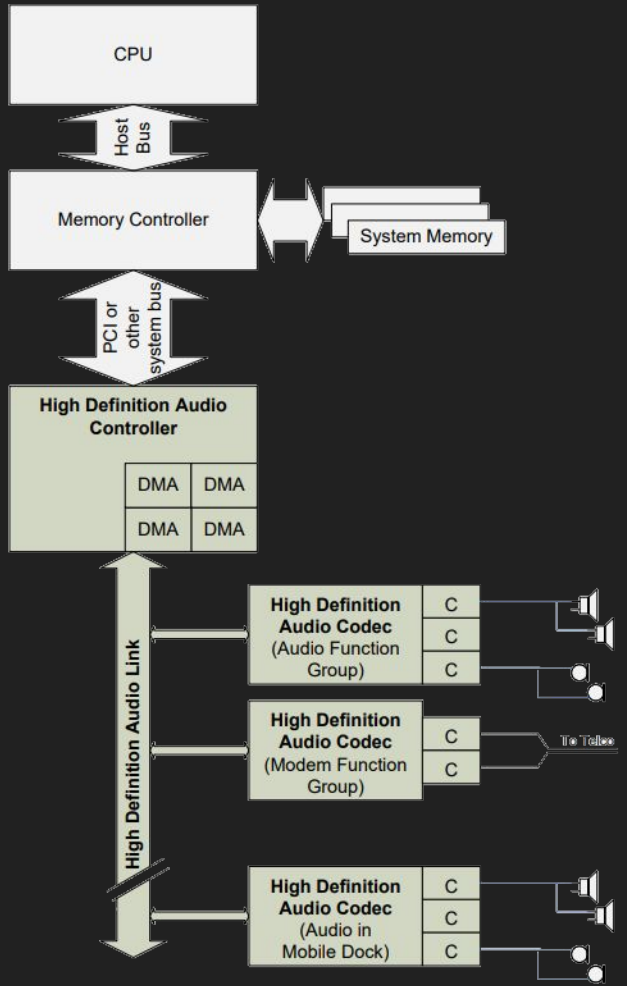
Audio Interface Driver

Will Bicks



Intel High Definition Audio

- Standard API implemented by multiple controller / code manufacturers.
- Major components:
  - Controller: PCI device that commands audio streams, access host audio data via DMAs data, transfers with codecs via HD Audio Link
  - HD Audio Link: connects controller to one or more codec
  - Codec: Converts HD Audio Link to / from standard audio interfaces (e.g. consumer line level, TOSLINK, built in speaker).



Courtesy of Intel High Definition Audio Specification

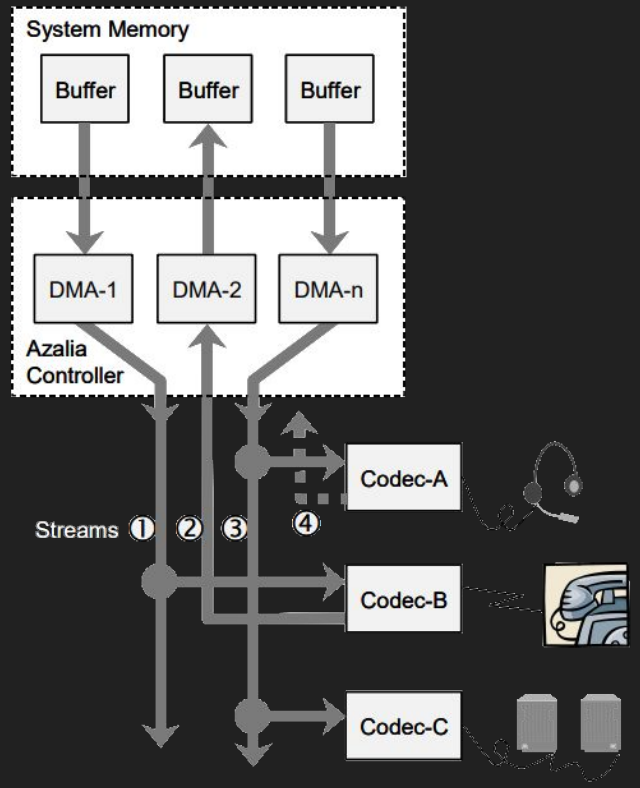
HD Audio Initialization Process

Communication via ring buffers
(host to controller commands, controller to host responses).

- Discover controller via PCI and reset and initialize.
- Using the controller to communicate with audio link, enumerate codecs on the link.
- Discover capabilities of each codec (sample rate, bit depth, inputs / outputs).

Getting Audio Flowing

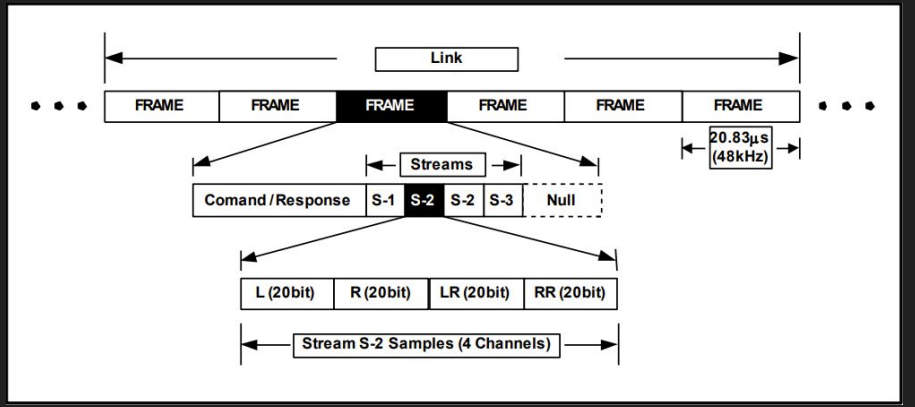
- Audio channels (e.g. left, right) are packed into streams between controller and a single codec in a single direction.
- Each stream is driven by a DMA engine in the controller.



Courtesy of Intel High Definition Audio Specification

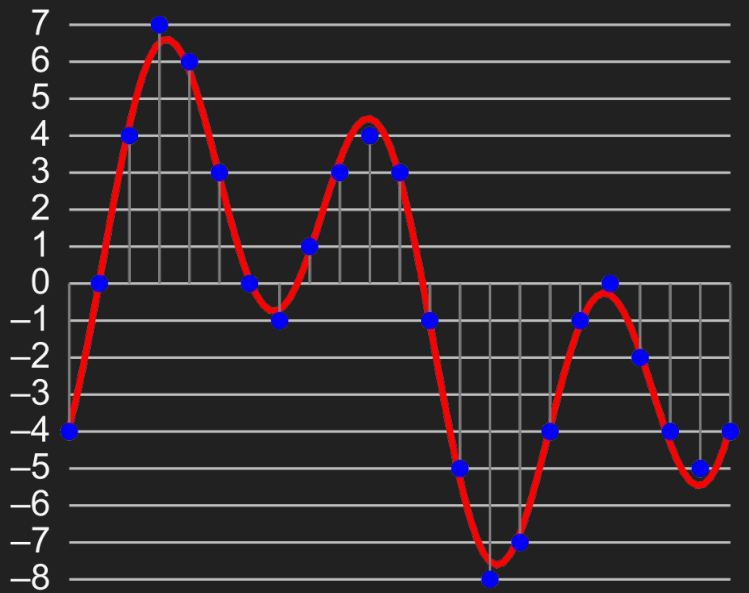
Getting Audio Flowing - Stream Contents

- HD Audio Link traffic is packed into frames every 20.83 µs (48 kHz)
- In this example there are three streams, and the S-2 stream has 4 channels (Left, Right, Left Rear, Right Rear).
- Sample rates higher than the frame rate are supported; in this example two S-2 stream sample sets are included in the frame (2x48 kHz = 96 kHz).



Courtesy of Intel High Definition Audio Specification

Getting Audio Flowing - Pulse Code Modulation (PCM)



Analog Signal (Red) Encoded in 4-bit PCM (Blue)

- Analog audio waveform represented as periodic digital samples.
- Recall Nyquist's theorem: max representable audio frequency is ½ sample rate.
- Higher bit depths introduce less quantization noise.

Driver / Application Architecture

Kernel Driver

User Program

- Detects and initializes single static HDA controller instance.
  (assumes only one controller, still allows multiple codecs / outputs)

- Gets handle to HDA controller
- Uses driver methods to detect codecs and capabilities on link
- Chooses codec and supported sample rate / bit depth combination
- Allocates memory region(s) for PCM samples, fills with some audio data
- Calls driver method to setup stream with audio parameters and pointer to audio buffer

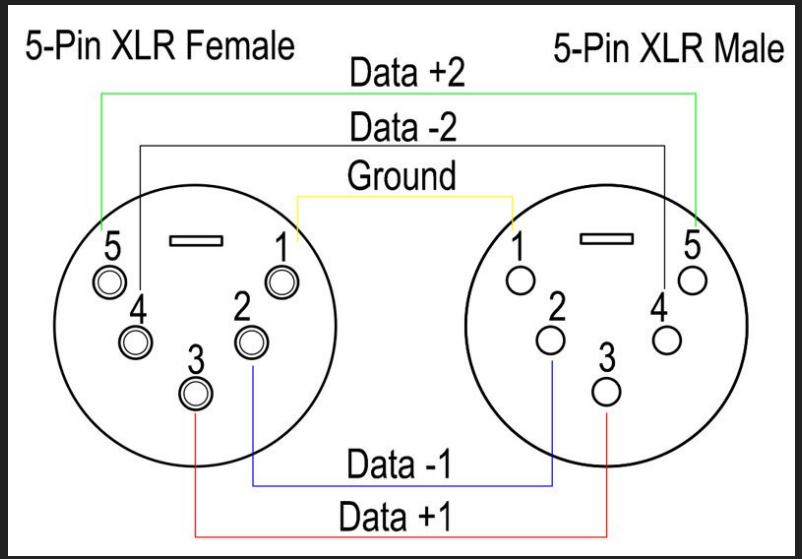
- Configures controller and codec to accept stream, sets up DMA engine to read audio samples.
- Writes data to codecs over HD Audio link, updates DMA position buffer and buffer descriptor list with progress

- Reads DMA position buffer or buffer descriptor list, update samples


DMX512 Driver

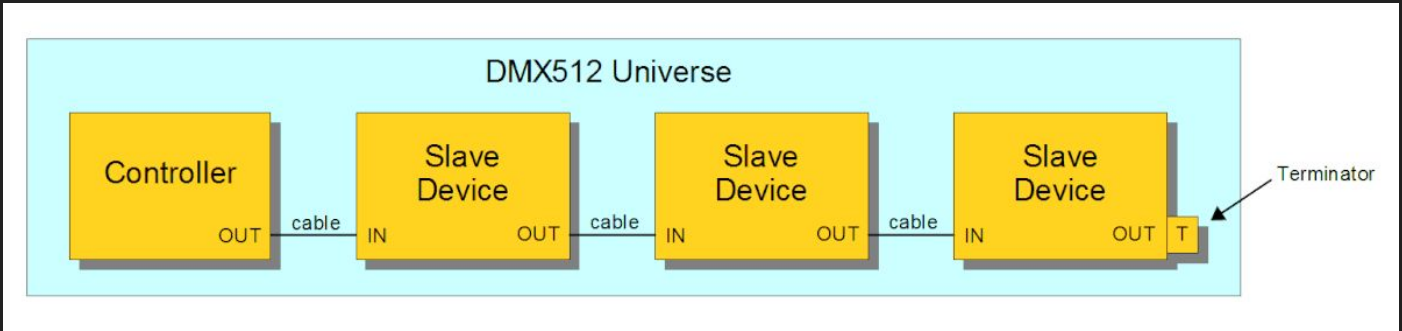
What is DMX512?

- A standard protocol used to digitally control stage lighting and effects
  - Popularly used in the live event industry (theater, concerts) and for architectural lighting
  - DMX → Digital Multiplex
- At the physical layer, it uses RS-485 and is wired with 3 conductors: data +/- and ground
- Asynchronous, half-duplex with a bitrate of 250 kbps
- RDM (Remote Device Management), is a newer addition to the DMX512 standard which allows for unidirectional communication

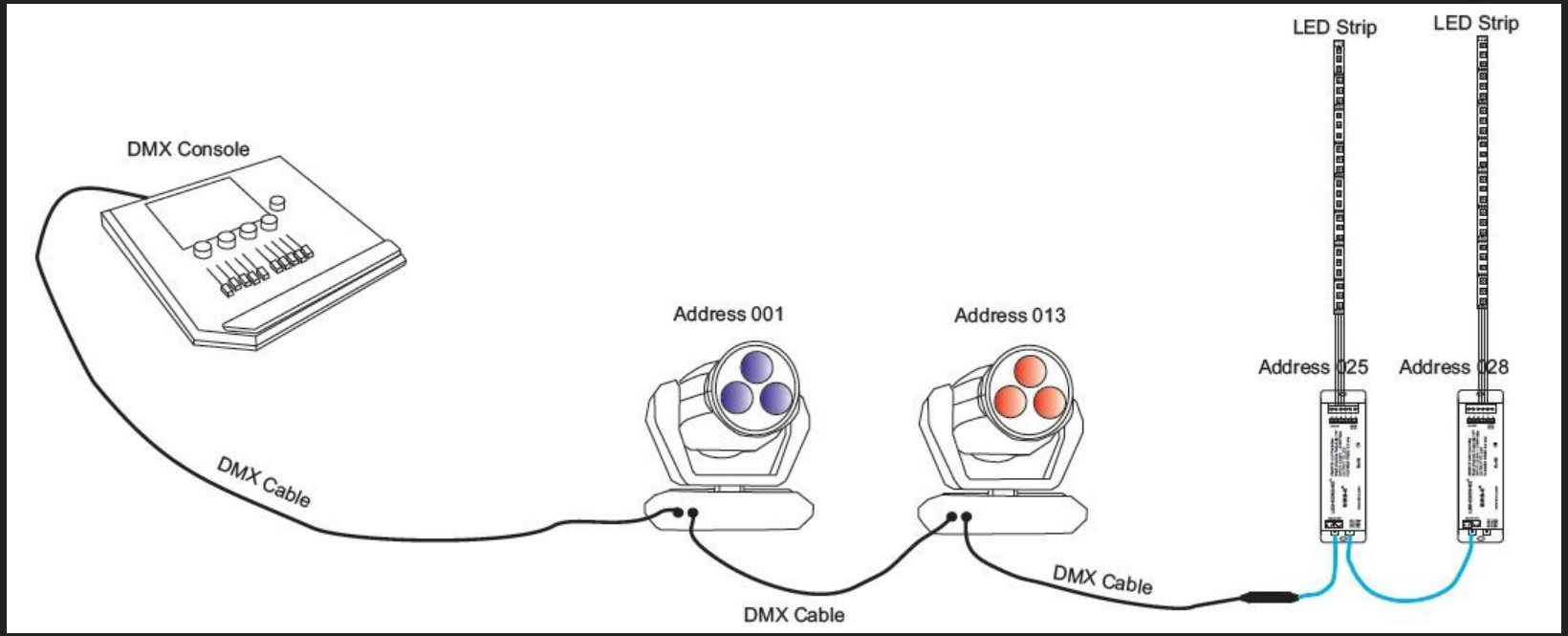


What is DMX512?

- A DMX “universe” consists of 512 channels with each channel carrying a value between 0-255
- Devices are daisy chained together in parallel
- Depending on the mode of a connected device, it will choose which channels to look at and how to interpret it



What is DMX512?



What is DMX512?

DMX Input Channel Profiles

DMX Profile	DMX Channels	Channel Assignments	Notes
5ch (Default)	5	1-Intensity 2-Red 3-Green 4-Blue 5-Strobe	Lime is mixed automatically.
RGB	3	1-Red 2-Green 3-Blue	
1ch	1	1-Intensity	This mode controls the intensity of Preset 1.
Dir	6	1-Intensity 2-Red 3-Green 4-Blue/Indigo* 5-Lime 6-Strobe	Original ColorSource PAR fixture uses blue in channel 4. Deep Blue ColorSource PAR fixture uses indigo in channel 4.

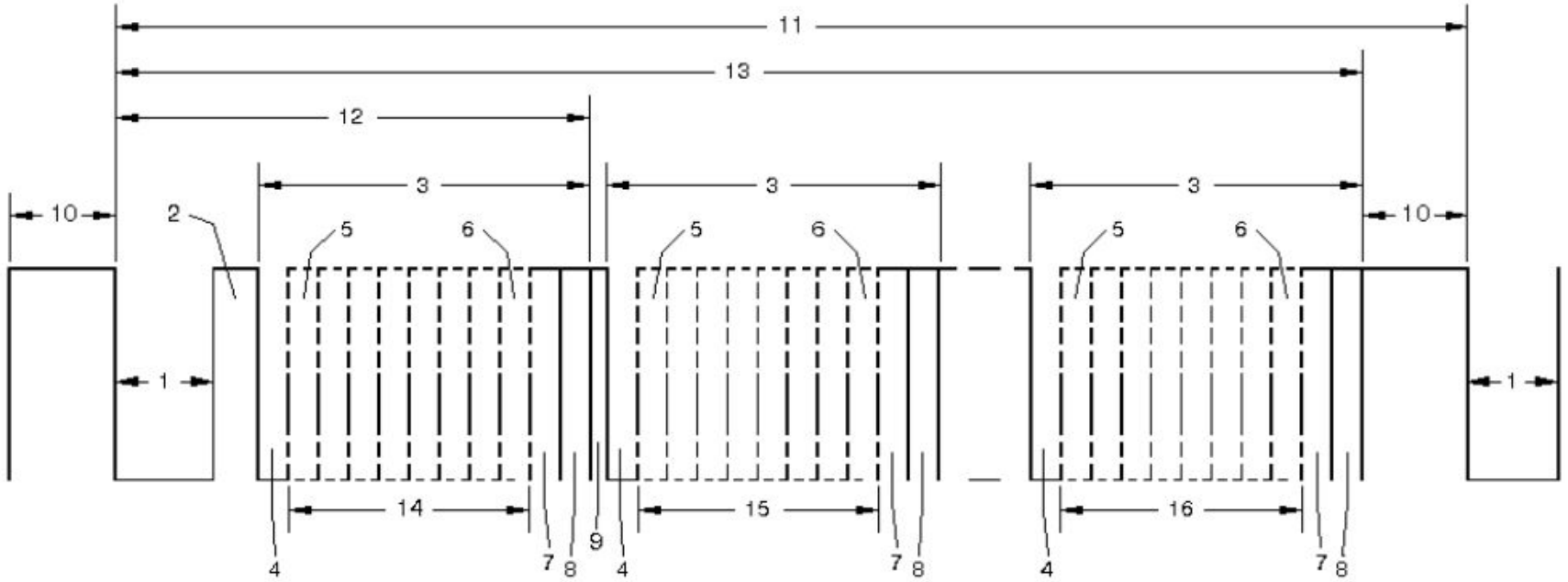
DMX512 Driver Support

- Design Philosophies
  - Adherence to the DMX512 standard where possible
  - Assume compatibility with 16540/16550 UART chips
    - Baseline provides definitions for these chips already
  - Ensure basic DMX512 functionality before implementing additional features (RDM)
- Implements the ANSI E1.11/USITT DMX512-A standard for transmitters
- Easy to use in user programs

DMX512 Standard Details

8.11 Timing Diagram - data+

Timings shall follow the requirements of the timing diagram (figure 5) and its associated tables 6 and 7.

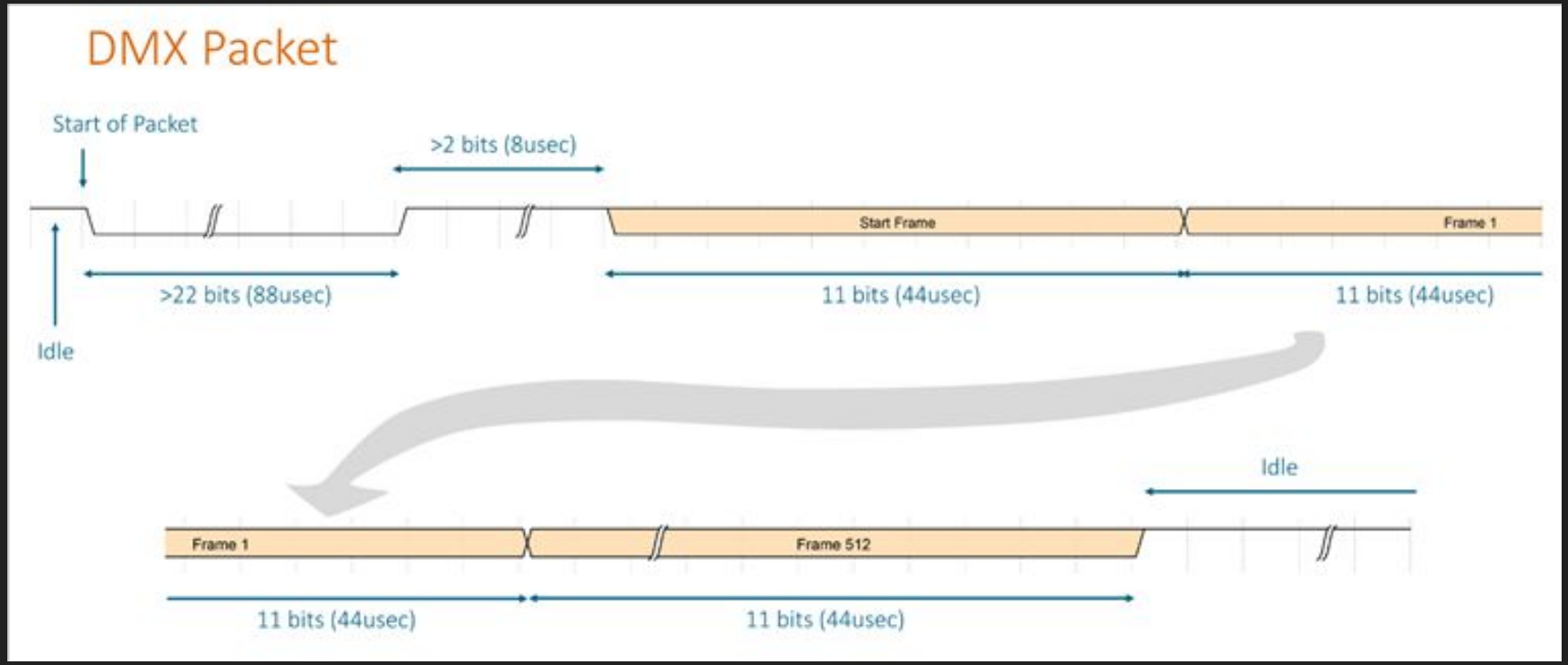


DMX512 Standard Details

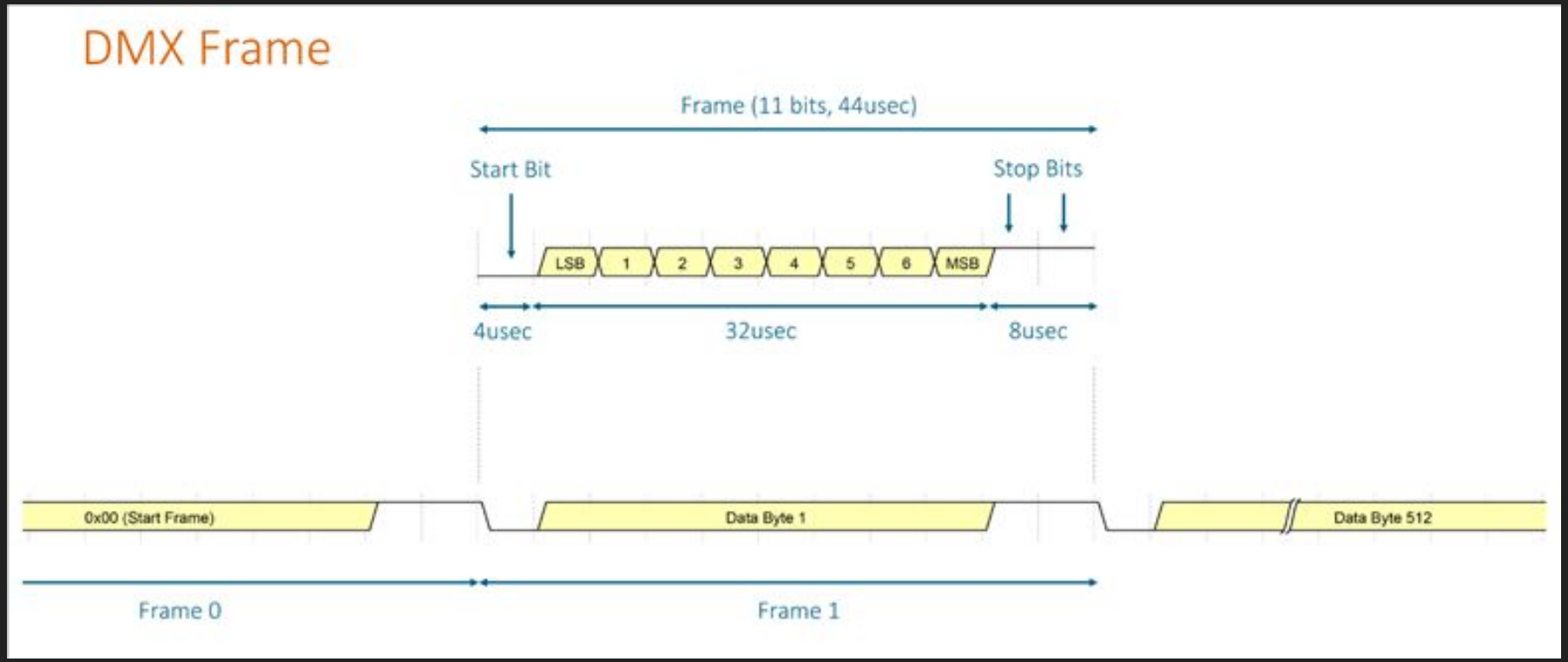
Table 6 - Timing Diagram Values - output of transmitting UART

Designation	Description	Min	Typical	Max	Unit
-	Bit Rate	245	250	255	kbit / s
-	Bit Time	3.92	4	4.08	µs
-	Minimum Update Time for 513 slots	–	22.7	–	ms
-	Maximum Refresh Rate for 513 slots	–	44	–	updates / s
1	"SPACE" for BREAK	92	176	–	µs
2	"MARK" After BREAK (MAB)	12	–	< 1.00	µs s
9	"MARK" Time between slots	0	–	< 1.00	s
10	"MARK" Before BREAK (MBB)	0	–	< 1.00	s
11	BREAK to BREAK Time	1204	–	– 1.00	µs s
13	DMX512 Packet	1204	–	– 1.00	µs s

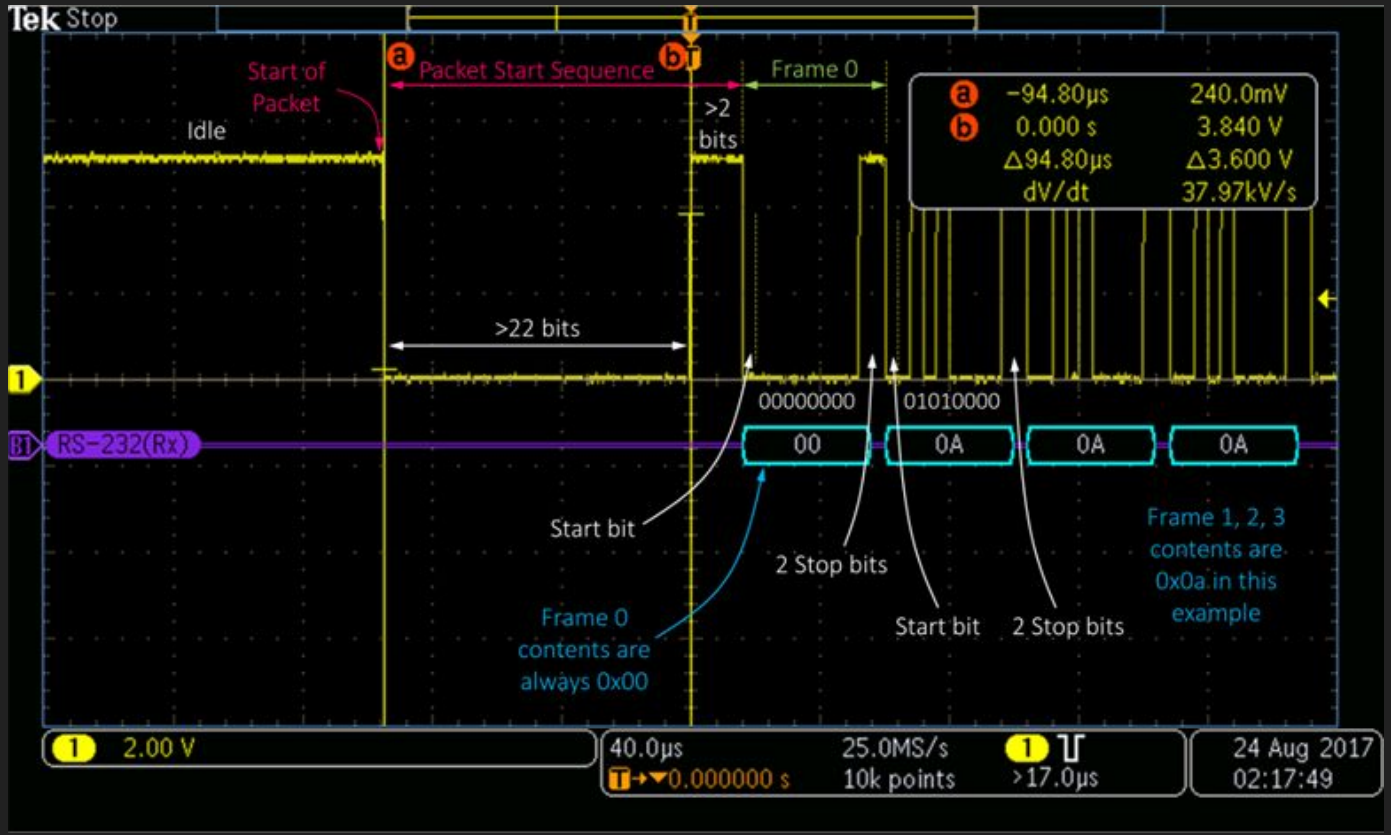
DMX512 Standard Details



DMX512 Standard Details



DMX512 Standard Details



Challenges

- Baud Rate Mismatch
  - DMX512 calls for a baud rate of 250 kbps
  - 16540/16550 UART chips can only handle up to 112.5 kbps (~ half)
    - Luckily, there's enough tolerance in some parts of the spec where this can be cheesed
      - 1 bit is roughly the same as 2 bits at 250 kbps
- During testing, differences between the standard and real-world implementations
  - Not all fixtures support alternate start codes
  - The standard gives a lot of leeway in timing, varies from device to device in terms of what is actually accepted
    - "SPACE" for BREAK
      - They state 176 usec is typical, in practice it's closer to ~94 usec
    - "MARK" After BREAK
      - Ranges from 12 usec to 1.00s, in practice it's ~16-24 usec
    - "MARK" Time between slots
      - Ranges from 0 to 1.00s, in practice not used
  - In the case of lost or corrupt data, the standard suggests devices should retain the last valid state for at least 1 minute or state "data handling procedures" in the product manual
    - Not followed, most typically keep the state for 1-3 seconds

Thank You!