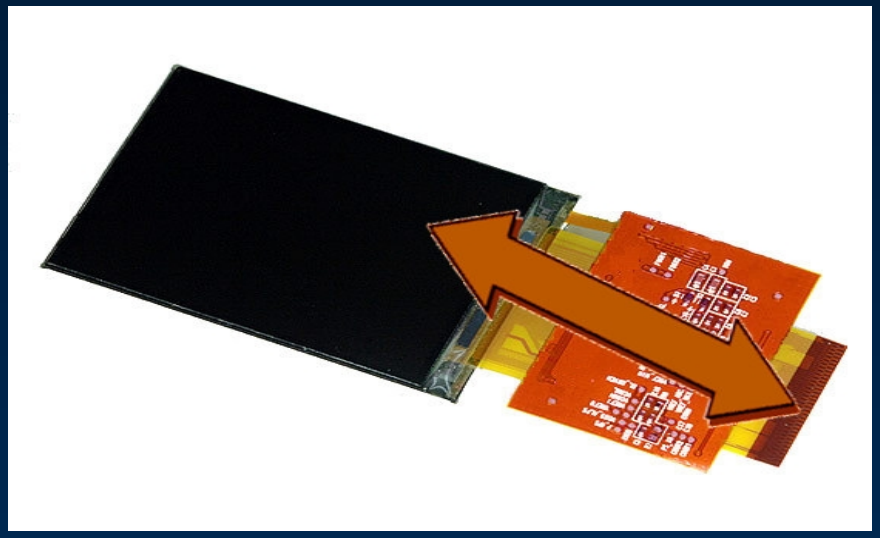
Using «dot clock» Displays in Embedded Linux Devices

Vitaly Wool
EmbeddedAlley Solutions Inc.

What is a «dot clock» display?

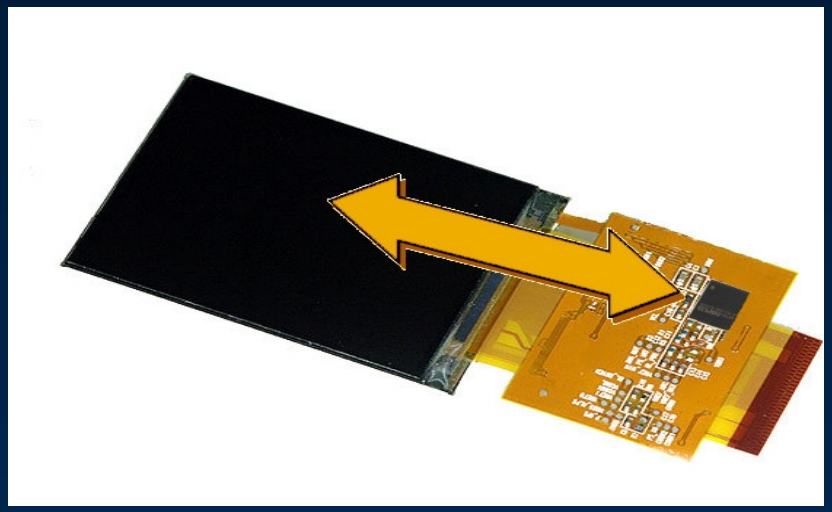
- «dot clock» display
  - Operates on 4 control signals
    - Vertical synchronization (VSYNC)
    - Horizontal synchronization (HSYNC)
    - Pixel clock (DOTCLK)
    - Data output enable (OE)
  - Exposes those signals to the outer world
    - RAM-less
  - This signal scheme is also called RGB

«Dot clock» display...



- No on-chip RAM for image storage
  - Exposes control signals to the outside
- Requires LCD interface capable to handle the control signals
- Requires careful programming

Typical LCDs for SoCs...



- ...are with internal RAM
  - So-called «frame buffer» type

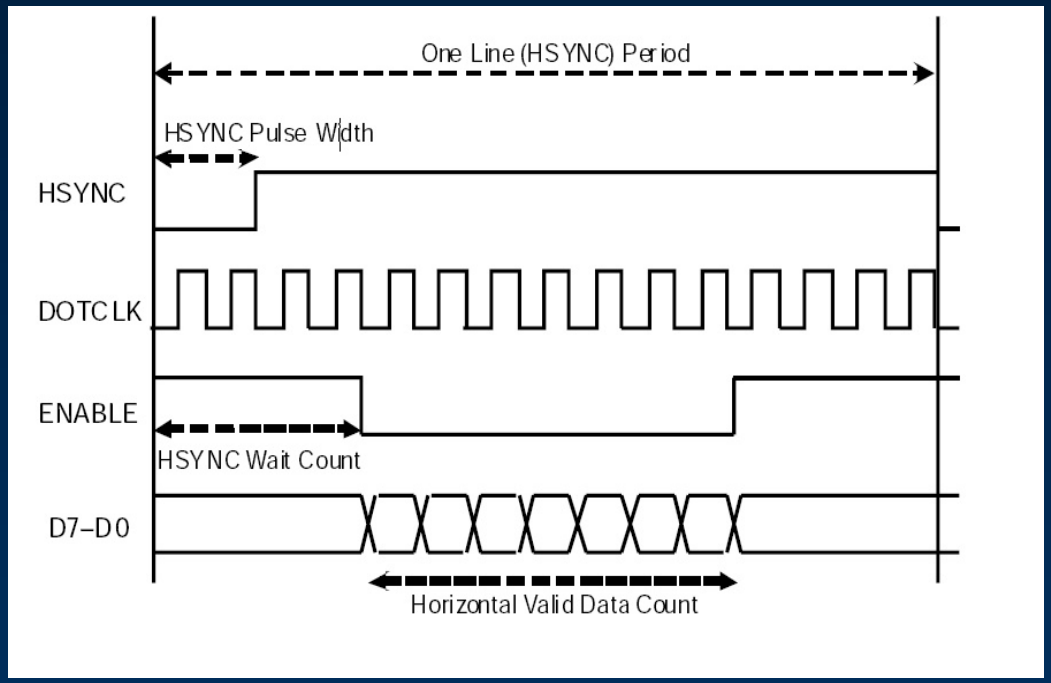
- No need to expose control signals
  - May expose VSYNC though
- Easy to program
  - No need to specify timings and such :-)
- No special requirements on the LCD interface

Frame buffer type LCDs

- Supports X×Y×BPP at max
  - Need to have X×Y×BPP / 8 bytes of RAM
    - That might be a lot
      - Expensive design
- The main data flow takes place internally
  - Stable image display
  - No requirements on the SoC throughput
- Display data update on demand
  - Data taken from user's buffer in main RAM

«Dot clock» displays

- Use RGB sginal scheme
  - VSYNC, HSYNC, DOTCLK, OE
- Use a piece of external RAM
  - Cheaper design
  - More flexibility
  - Constant load of the bus
  - Sophisticated hardware LCD interface



«Dot clock» hardware requirements

- LCD interface
  - Should support RGB signaling
  - Should be programmable for delays and timings
  - Should be accurate enough to handle these
  - Should support DMA transfers of course
  - NB: But we get TV-out support almost for free :-)
- DMA
  - Should support flow control from LCD interface
  - Should support linked-list transfers

Why linked lists?

- Single transfer
  - Interrupts flood
  - Hard requirements on interrupt latency
  - Discontiguous RAM framebuffer not allowed
- Linked list
  - Less interrupts
  - Interrupts may be served asynchronously
  - Discontiguous RAM framebuffer may be allowed

Sample ARM SoC

- Targets smart phone & PND market
  - SD/SDHC support
  - 512 MB MLC NAND
  - 64 MB RAM
  - USB OTG support
  - 320×240×24 BPP QVGA display
  - GSM/UMTS/GPRS
  - GPS
  - Dotclock- and TV-out capable LCDIF

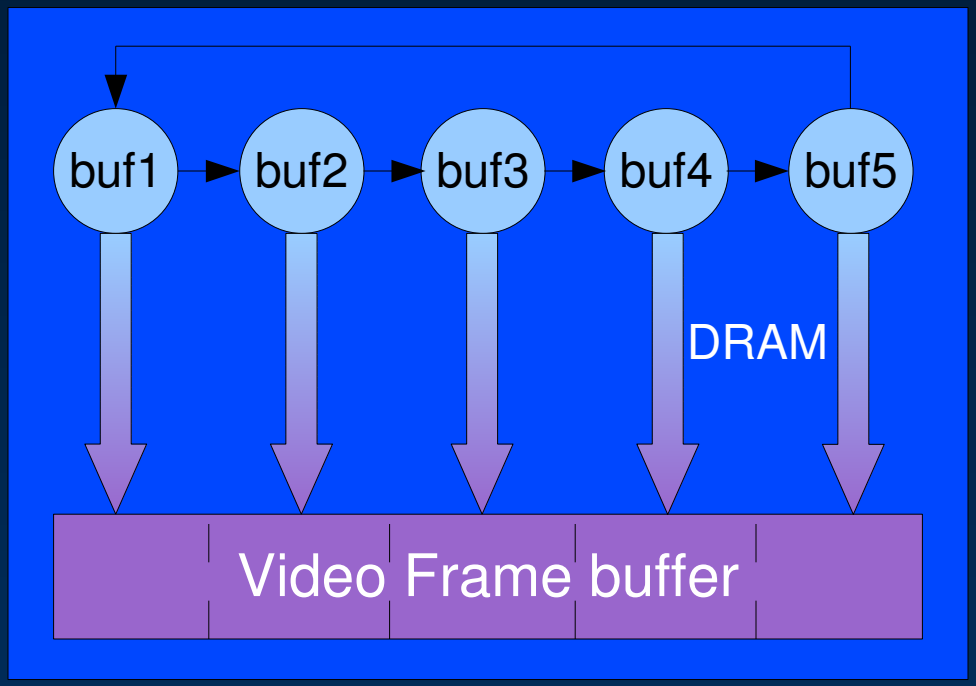
...more SoC characteristics

- Performance
  - CPU up to 320 MHz
  - AHB up to 160 MHz
  - mDDR up to 133 MHz
- DMA
  - M2M and M2P/P2M
  - Supports chaining
  - Operates 64K buffers at max

Linux framebuffer driver implementation

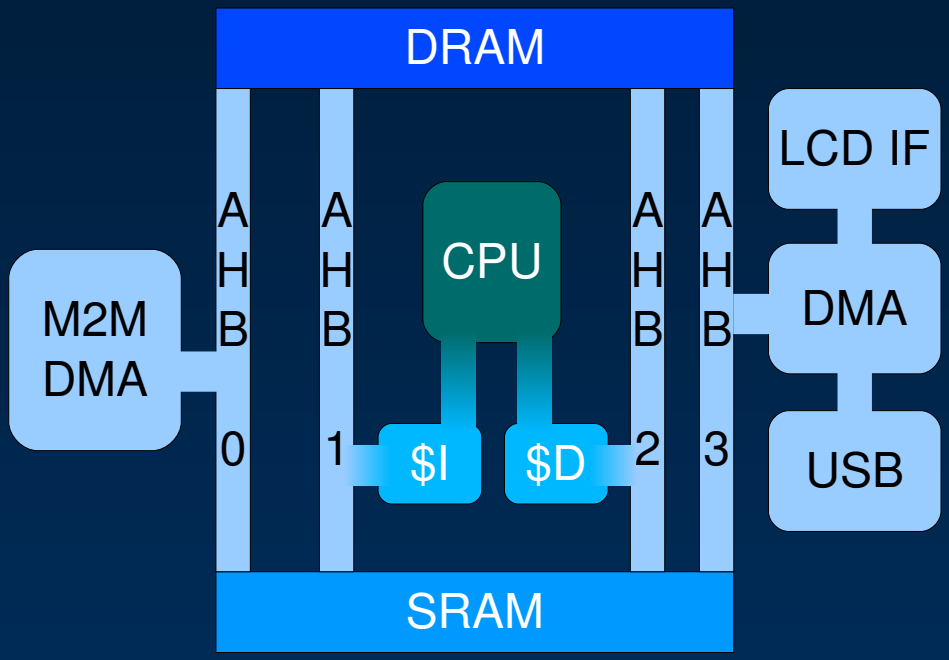
- Init sequence
  - Configure the timings and counters
    - Easily left out to platform code
  - Allocate channel and buffers for DMA
    - Line boundaries should be maintained
      - Take advantage of horizontal blanking period delay
      - So buffer size is multipe of 240*4
  - Chain the DMA buffers
  - Set the dot clock mode
  - Start the DMA channel

Software Design



- DMA chain
  - circular
  - 5×60K (=61440b) link buffers
  - Buffers may and may not be contiguos
- Interrupt-less mode
  - DMA suppors LCD IF flow control

Hardware Peculiarities



- DMA is on AHB3
  - USB/LCD share AHB3 via DMA
- I/D caches may block AHB1/AHB2
- M2M DMA operates on AHB0

Hardware challenges

- AHB buses are round-trip synchronous
  - A request to external RAM blocks AHB3
    - No other blocks may transfer data during the request
    - USB can consume AHB for up to ms
- RAM controller blocks all other transfers while handling the AHB
  - The I/D caches can block RAM for large time periods
- DMA controller can only handle one request round-trip at a time
  - if e. g. NAND is being serviced then LCD IF cannot be serviced

Some related calculations

- NAND at 10 MB/s
- LCD at 15 MB/s
- One RAM read operation takes ~10 bus clocks
- So 250/4 = 63 MHz

Problems experienced

- Underruns
  - On intensive NAND operations
  - On intensive USB operations
  - On some frequency scaling operations
- Flickering
  - On some frequency scaling operations
  - During the driver initialization from the early splashscreen

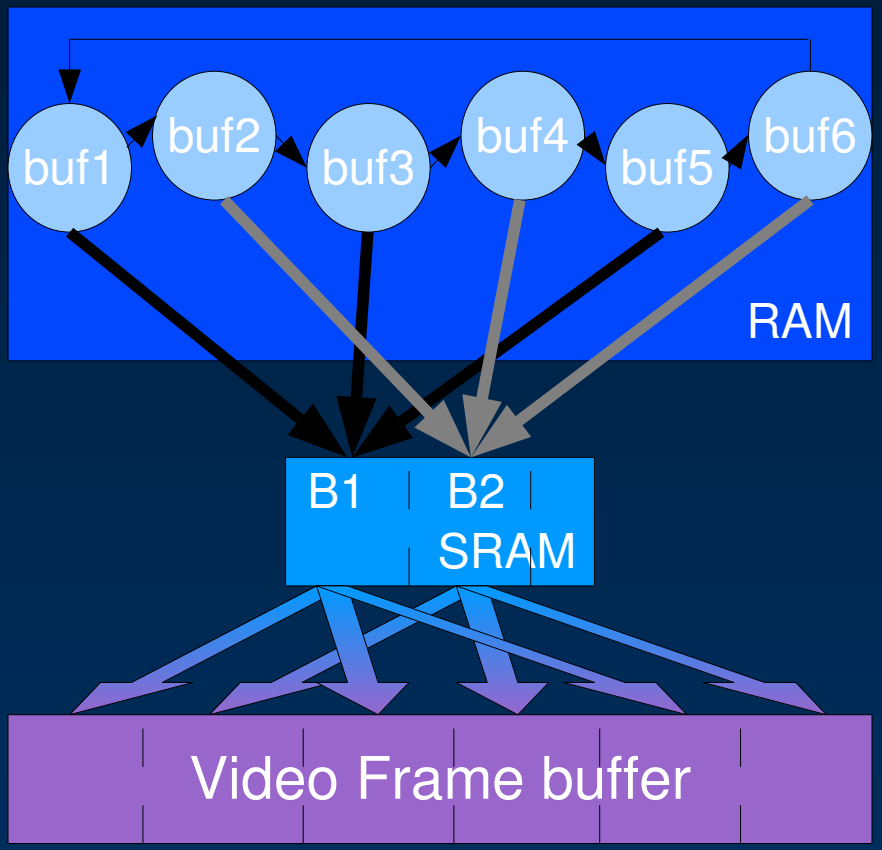
Diagnosis

This solution is totally inacceptible.

How to handle?

- The idea is to use SRAM
  - Kind of double buffering approach
- SRAM is only 64K
  - Will have to use a SRAM buffer for several DRAM ones
  - Tearing and wrong data display possible
- Problem mitigation
  - Use 2 SRAM buffers
    - Size to be carefully considered

SRAM buffering approach



- Even number of RAM buffers
  - Odd ones go to B1
  - Even ones go to B2
  - Buffer switch per LCD interrupt
- SRAM to LCD data flow
  - No underruns
  - Update delays possible
  - Tearing is possble
    - ...but may be avoided

SRAM buffer size considerations

- $t_{int} = L_c \times t_{hsync}$
  - L_c = lines per SRAM buffer
  - t_{int} = time between LCD interface interrupts
  - t_{hsync} = horizontal blanking period
- $B_c = L_c \times L_w$
  - B_c = bytes per SRAM buffer
  - $L_w = Y \times 4$ = bytes per line
- The number of initial buffers depend on B_c

SRAM buffer size tradeoffs

- Larger L_c
  - Less interrupts
    - Less CPU load
    - Less tearing risk
  - Consumes more SRAM
  - Smaller number of initial buffers
    - Easier to program
- Smaller L_c
  - All the opposite ;-)

Conclusions

- «Dot clock» LCDs are possible to employ in embedded systems
- Careful configuration examination and design are very important
  - Inaccurate hardware design may well lead to overcomplicated/fragile software implementation
- Linux kernel framebuffer framework is ready enough for dotclock displays' drivers

Questions?

mailto:vital@embeddedalley.com