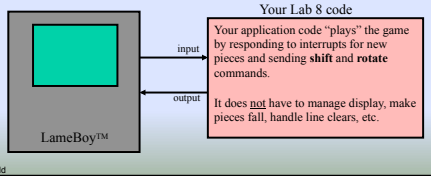
Lab 8: Fun and games

- Consider the ancient Game Boy
  - Non-trivial embedded, real-time system.
  - Had to balance computation and timely event handling.
  - Probably fun to design, develop, test.
- Inspiration for this lab
  - Focus: application software, not the RTOS, graphical display, or game engine.
  - Let's consider how it works and what you do.

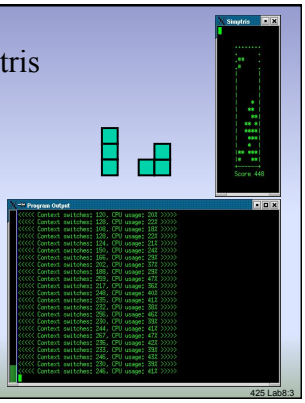
Conceptual model

- Imagine a small device that plays Tetris.
- Instead of having a mechanical interface (buttons), the game has an electronic interface.
- You build the system on the right to play the game.



Simptris

- Special mode in simulator
  - Type "simptris" to prompt
  - Special window created
- A simplified version of Tetris
  - Pieces consist of three blocks.
  - Just two shapes: corners, straight pieces.
  - Small board: 6 columns × 16 rows.
- Score is simply total lines cleared.
  - No bonus for clearing multiple lines at once.



Playing details

- Input to your system:
  - An interrupt occurs when each new piece appears.
  - Location, type of piece can be obtained from global variables.
- Output/control from your system:
  - Commands to rotate or shift specific piece; you call built-in functions.
  - New command cannot be sent until prior command received.
  - Transmission delay is fixed: ultimately a game-ending bottleneck.
- Pieces fall faster until your code breaks. High score wins.
  - For full credit: clear at least **200** lines at default tick frequency.

The interface

- The interrupts your system receives:
  - reset — priority 0
  - tick — priority 1
  - keypress — priority 2

  } Emu86 interrupts

  - game over — priority 3
  - new piece — priority 4
  - received command — priority 5
  - touchdown — priority 6
  - line clear — priority 7

  } Simptris interrupts

- Details are communicated via global variables
  - ID#, column, orientation of new piece.
  - ID of piece that touched down.
  - Screen bitmaps of pieces that have touched down.

Interface details

- The functions you can call (defined in simptris.s):
  - **void Slide_Piece(int id, int direction);** // 1=right, 0=left
  - **void Rotate_Piece(int id, int direction);** // 1=clockwise, 0=counterclockwise
  - **void Seed_Simptris(long seed);** // random number seed
  - **void Start_Simptris(void);**
- Dealing with transmission delay:
  - You can't call **Slide_Piece( )** or **Rotate_Piece( )** until "command received" interrupt received after most recent function call.
  - Interrupt indicates "clear to send" rather than last command completed okay.
    - Can't move piece into wall, for example.
  - Recommended: encapsulate communication details within task.

The simulator

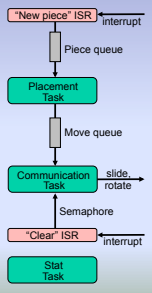
- Type “simptris” at Emu86> prompt and game display appears.
  - Normal text output from your code will appear in the program output window as before.
- You get reset, keypress, and timer ticks as before.
  - Simptris interrupts are added in simptris mode.
  - You decide which you want to pay attention to.
    - Write required ISRs and handlers.
    - Modify interrupt vector table.
  - **For each interrupt you want to ignore**: write a minimal ISR.
    - Contents: save ax, send EOI command, restore ax, iret.
    - Impractical in simulator to modify IMR.

Lab requirements

- Your application code must use your YAK kernel.
- Your code must accurately report CPU utilization and context switches every 20 ticks.
- Your code must clear 200 lines at default tick frequency.
  - You may use any seed you want to pass off lab.
  - You cannot change seed more than once per game.
  - Fairly straightforward placement algorithms are adequate if the overhead of your RTOS code is low.
  - Not an exercise in AI unless you make it one.

Suggested organization

- Here’s a starting point to consider:
  - Create three tasks:
    - One makes placement decisions, determines sequence of slide and rotate commands
    - One handles communication with Simptris
    - One handles statistics
  - Use two queues:
    - “Piece queue” buffers details about new pieces.
    - “Move queue” buffers details about move commands.
  - Use one semaphore:
    - Signal when next command can be sent.
- Choose your own design, but use good design principles.



Placement algorithms

- Very simple algorithm can clear 80+ lines (with right seed):
  - Straight pieces on one side, corners on the other.
- Slightly more complicated algorithms work much better. Example:
  - Divide area into two halves, play each piece on selected half
  - If both flat, place pieces on nearest side unless imbalance too great.
  - If not flat on one side, place corner piece to make it flat.
- Much more complicated algorithms are possible.
  - Use features, interrupts in any way you wish.
- Key measures to think about:
  - Worst case number of moves for any piece.
  - Worst case latency from new piece to first move command for piece.

The friendly competition

- We will jointly pick a seed (next to last day of class).
- Everyone will run their code on that seed, report results on last day of class.
- Twinkies awarded for
  - Highest score with that seed
  - Lowest score for code that cleared 200+ on another seed
  - A variety of noteworthy kernel “achievements” based on HW10
- For reference:
  - Minimum required: 200 lines
  - Maximum observed with current simulator: >450 lines.
  - My code has been beaten.