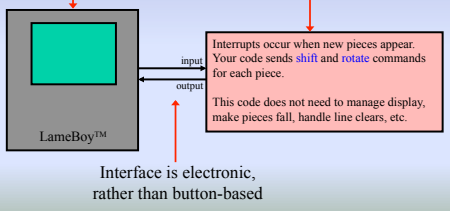
Lab 8: Time-critical application code

- Inspiration for this lab: the ancient Game Boy
  - Real-time responsiveness was critical
  - Design balanced computation and quick event handling
  - Probably fun to design, develop, and test
- Our focus for this lab: application software
  - *Not* the RTOS, graphical display, or game engine

Conceptual model

You are given a game-engine for Tetris

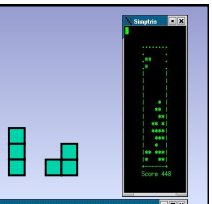
You create application code that plays game

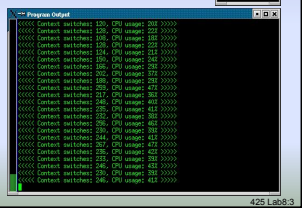


Interface is electronic, rather than button-based

Simptris

- 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
- Simptris is special mode in simulator
  - Type "simptris" to prompt
  - Window appears with game display
  - State represented with ASCII characters





Operational details

- Inputs to your system:
  - An interrupt signals the appearance of each new piece
  - Global variables have details: location, piece type
- Outputs from your system:
  - Movement commands: your code calls built-in rotate and shift functions
  - Constraint: next command cannot be sent until previous command completes
- Pieces fall faster until code can't keep up; high score wins
  - Limiting factor is fixed delay of movement commands
  - For full credit: clear at least **200** lines at default tick frequency

The interface: inputs

- 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 number, column, orientation of new piece
  - ID number of piece that touched down
  - Screen bitmaps of pieces that have touched down

The interface: outputs

- The functions available (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:
  - Must wait for "command received" interrupt for previous command before calling Slide_Piece( ) or Rotate_Piece( ), otherwise behavior is undefined
  - Interrupt indicates "clear to send" rather than last command completed successfully
    - Can't move piece into wall, for example
  - Recommendation: encapsulate communication details within a task

The simulator

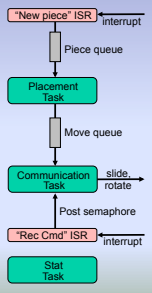
- Type "simptris" at Emu86> prompt and game display appears
  - Normal text output from your code will appear in the program output window as before
- You get reset, keypress, and timer ticks as before
  - Simptris interrupts are added in simptris mode
  - You decide *which* interrupts your code will 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
    - Conceptually cleaner to *mask* these interrupts, but impractical in simulator to modify IMR

Lab requirements

- Your application code must:
  - Use your YAK kernel
  - Accurately report CPU utilization and context switches every 20 ticks
  - Clear 200 lines at default tick frequency (with some seed)
  - Use just one random number seed per game
- Not an exercise in AI unless you choose to make it one
  - Fairly straightforward placement algorithms are adequate if the overhead of your RTOS code is low

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 correct seed):
  - Straight pieces on one side, corners on the other
- Slightly more complicated algorithms work much better. Example:
  - Divide area into two halves, choose which side to play each piece on
  - Track state of each side separately: flat or not flat
  - If both sides flat, place pieces on nearest side unless imbalance too great
  - If one side not flat, place next corner piece there to make it flat
- Much more complicated algorithms are possible
  - Can use features, interrupts in any way you wish
- Key measures to think about to achieve maximum scores:
  - Worst case number of moves for any piece
  - Worst case latency from new piece to sending of first move command for piece

The friendly competition

- We will jointly pick a seed (on next to last day of class)
- Everyone will run their code on that seed, report results in class on last day
- Special recognition will be given for
  - Highest scores achieved with that seed
  - Lowest score with that seed (for code that cleared 200+ with a different seed)
  - Various noteworthy kernel "achievements" based on results reported in HW9
- For reference:
  - Minimum required: 200 lines
  - Maximum observed with current simulator: >450 lines
  - My code has been beaten