Describing embedded system's processing behavior

Outline

- Models vs. Languages
- State Machine Model
  - FSM/FSMD
  - HCFSM and Statecharts Language
  - Program-State Machine (PSM) Model
- Concurrent Process Model
  - Communication
  - Synchronization
  - Implementation
- Real-Time Systems

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Introduction

- Describing embedded system's processing behavior
  - Can be extremely difficult
    - Complexity increasing with increasing IC capacity
      - Past: washing machines, small games, etc.
        - Hundreds of lines of code
      - Today: TV set-top boxes, Cell phone, etc.
        - Hundreds of thousands of lines of code
    - Desired behavior often not fully understood in beginning
      - Many implementation bugs due to description mistakes/ omissions
  - English (or other natural language) common starting point
    - Precise description difficult to impossible
    - Example: Motor Vehicle Code – thousands of pages long...

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

An example of trying to be precise in English

- California Vehicle Code
  - Right-of-way of crosswalks
    - 21950. (a) The driver of a vehicle shall yield the right-of-way to a pedestrian crossing the roadway within any marked crosswalk or within any unmarked crosswalk at an intersection, except as otherwise provided in this chapter.
    - (b) The provisions of this section shall not relieve a pedestrian from the duty of using due care for his or her safety. No pedestrian shall suddenly leave a curb or other place of safety and walk or run into the path of a vehicle which is so close as to constitute an immediate hazard. No pedestrian shall unnecessarily stop or delay traffic while in a marked or unmarked crosswalk.
    - (c) The provisions of subdivision (b) shall not relieve a driver of a vehicle from the duty of exercising due care for the safety of any pedestrian within any marked crosswalk or within any unmarked crosswalk at an intersection.
  - All that just for crossing the street (and there's much more)!

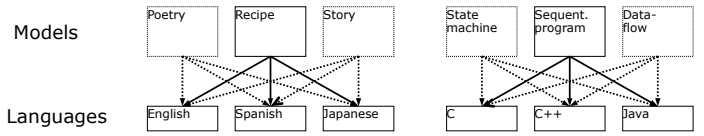
Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Models and languages

- How can we (precisely) capture behavior?
  - We may think of languages (C, C++), but *computation model* is the key
- Common computation models:
  - Sequential program model
    - Statements, rules for composing statements, semantics for executing them
  - Communicating process model
    - Multiple sequential programs running concurrently
  - State machine model
    - For control dominated systems, monitors control inputs, sets control outputs
  - Dataflow model
    - For data dominated systems, transforms input data streams into output streams
  - Object-oriented model
    - For breaking complex software into simpler, well-defined pieces

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Models vs. languages



Recipes vs. English **Sequential programs vs. C**

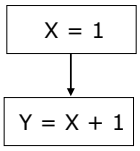
- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C
- Variety of languages can capture one model
  - E.g., sequential program model → C,C++, Java
- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model
- Certain languages better at capturing certain computation models

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Text versus Graphics

- Models versus languages not to be confused with text versus graphics
  - Text and graphics are just two types of languages
    - Text: letters, numbers
    - Graphics: circles, arrows (plus some letters, numbers)

```
X = 1;
Y = X + 1;
```

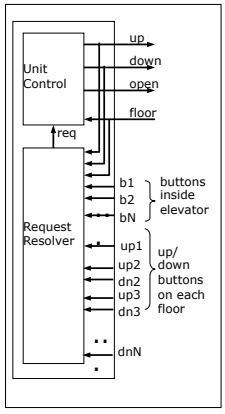


Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Introductory example: An elevator controller

- Simple elevator controller
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor
- Try capturing in C...

System interface



Partial English description

"Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down…"

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Elevator controller using a sequential program model

Sequential program model

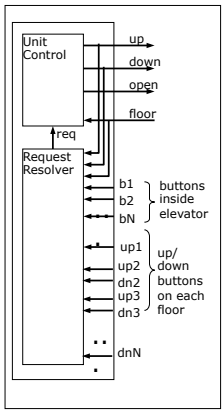
```
Inputs: int floor; bit b1..bN; up1..upN-1;
dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;
void UnitControl()
{
  up = down = 0; open = 1;
  while (1) {
    while (req == floor);
    open = 0;
    if (req > floor) { up = 1;}
    else {down = 1;}
    while (req != floor);
    up = down = 0;
    open = 1;
    delay(10);
  }
}

void RequestResolver()
{
  while (1)
  ...
    req = ...
  ...
}

void main()
{
  Call concurrently:
    UnitControl()
  and
  RequestResolver()
}
```

You might have come up with something having even more if statements.

Partial English description

"Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down..."

System interface



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

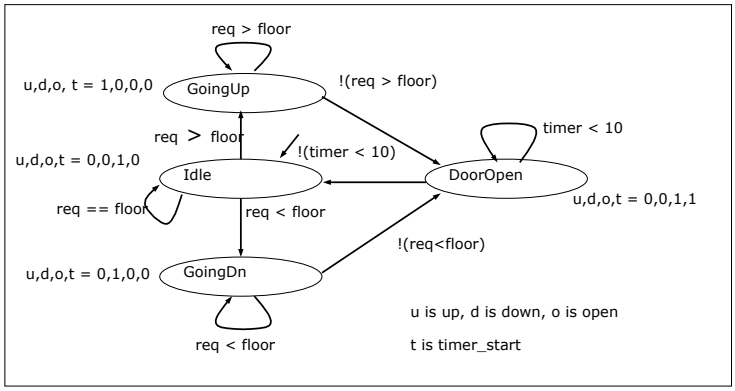
Finite-state machine (FSM) model

- Trying to capture this behavior as sequential program is a bit awkward
- Instead, we might consider an FSM model, describing the system as:
  - Possible states
    - E.g., *Idle*, *GoingUp*, *GoingDn*, *DoorOpen*
  - Possible transitions from one state to another based on input
    - E.g., req > floor
  - Actions that occur in each state
    - E.g., In the *GoingUp* state, u,d,o,t = 1,0,0,0 (up = 1, down, open, and timer_start = 0)
- Try it...

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Finite-state machine (FSM) model

***UnitControl* process using a state machine**



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Formal definition

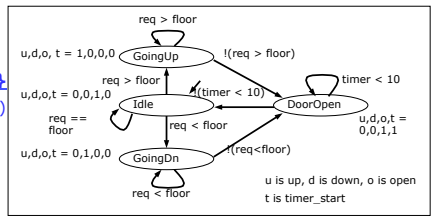
- An FSM is a 6-tuple $F<S, I, O, F, H, s_0>$
  - S is a set of all states $\{s_0, s_1, \ldots, s_l\}$
  - I is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - O is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - F is a next-state function ($S \times I \rightarrow S$)
  - H is an output function ($S \rightarrow O$)
  - s_0 is an initial state
- Moore-type
  - Associates outputs with states (as given above, H maps $S \rightarrow O$)
- Mealy-type
  - Associates outputs with transitions (H maps $S \times I \rightarrow O$)
- Shorthand notations to simplify descriptions
  - Implicitly assign 0 to all unassigned outputs in a state
  - Implicitly AND every transition condition with clock edge (FSM is synchronous)

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Finite-state machine with datapath model (FSMD)

- FSMD extends FSM: complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- FSMD: 7-tuple <*S*, *I* , *O*, ***V***, *F*, *H*, s_0>
  - *S* is a set of states $\{s_0, s_1, \ldots, s_l\}$
  - *I* is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - *O* is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - ***V* is a set of variables $\{v_0, v_1, \ldots, v_n\}$**
  - *F* is a next-state function ($S \times I \times V \rightarrow S$)
  - *H* is an **action** function ($S \rightarrow O + V$)
  - s_0 is an initial state

We described UnitControl as an FSMD

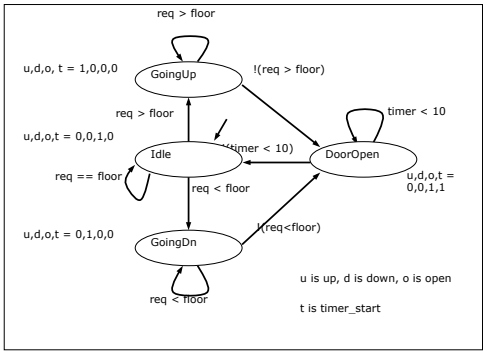


- *I*,*O*,*V* may represent complex data types (i.e., integers, floating point, etc.)
- *F*,*H* may include arithmetic operations
- *H* is an action function, not just an output function
  - Describes variable updates as well as outputs
- Complete system state now consists of current state, s_i, and values of all variables

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Describing a system as a state machine

1. List of possible states
2. Declare all variables (none in this example)
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

State machine vs. sequential program model

- Different thought process used with each model
- State machine:
  - Encourages designer to think of all possible states and transitions among states based on all possible input conditions
- Sequential program model:
  - Designed to transform data through series of instructions that may be iterated and conditionally executed
- State machine description excels in many cases
  - More natural means of computing in those cases
  - *Not* due to graphical representation (state diagram)
    - Would still have same benefits if textual language used (i.e., state table)
    - Besides, sequential program model could use graphical representation (i.e., flowchart)

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Try Capturing Other Behaviors with an FSM

- E.g., Answering machine blinking light when there are messages
- E.g., A simple telephone answering machine that answers after 4 rings when activated
- E.g., A simple crosswalk traffic control light
- Others

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Capturing state machines in sequential programming language

- Despite benefits of state machine model, most popular development tools use sequential programming language
  - C, C++, Java, Ada, VHDL, Verilog, etc.
  - Development tools are complex and expensive, therefore not easy to adapt or replace
    - Must protect investment
- Two approaches to capturing state machine model with sequential programming language
  - Front-end tool approach
    - Additional tool installed to support state machine language
      - Graphical and/or textual state machine languages
      - May support graphical simulation
      - Automatically generate code in sequential programming language that is input to main development tool
    - Drawback: must support additional tool (licensing costs, upgrades, training, etc.)
  - Language subset approach
    - Most common approach...

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Language subset approach

- Follow rules (template) for capturing state machine constructs in equivalent sequential language constructs
- Used with software (e.g.,C) and hardware languages (e.g.,VHDL)
- Capturing *UnitControl* state machine in C
  - Enumerate all states (#define)
  - Declare state variable initialized to initial state (IDLE)
  - Single switch statement branches to current state's case
  - Each case has actions
    - up, down, open, timer_start
  - Each case checks transition conditions to determine next state
    - if(…) {state = …;}

```
#define IDLE0
#define GOINGUP1
#define GOINGDN2
#define DOOROPEN3
void UnitControl() {
   int state = IDLE;
   while (1) {
      switch (state) {
         IDLE: up=0; down=0; open=1; timer_start=0;
            if   (req==floor) {state = IDLE;}
            if   (req > floor) {state = GOINGUP;}
            if   (req < floor) {state = GOINGDN;}
            break;
         GOINGUP: up=1; down=0; open=0; timer_start=0;
            if   (req > floor) {state = GOINGUP;}
            if   (!(req>floor)) {state = DOOROPEN;}
            break;
         GOINGDN: up=1; down=0; open=0; timer_start=0;
            if   (req < floor) {state = GOINGDN;}
            if   (!(req<floor)) {state = DOOROPEN;}
            break;
         DOOROPEN: up=0; down=0; open=1; timer_start=1;
            if (timer < 10) {state = DOOROPEN;}
            if (!(timer<10)){state = IDLE;}
            break;
      }
   }
}
```

***UnitControl* state machine in sequential programming language**

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

General template

```
#define S0  0
#define S1  1
...
#define SN  N
void StateMachine() {
   int state = S0; // or whatever is the initial state.
   while (1) {
      switch (state) {
         S0:
            // Insert S0's actions here & Insert transitions T_i leaving S0:
            if( T_0's condition is true ) {state = T_0's next state; /*actions*/ }
            if( T_1's condition is true ) {state = T_1's next state; /*actions*/ }
            ...
            if( T_m's condition is true ) {state = T_m's next state; /*actions*/ }
            break;
         S1:
            // Insert S1's actions here
            // Insert transitions T_i leaving S1
            break;
         ...
         SN:
            // Insert SN's actions here
            // Insert transitions T_i leaving SN
            break;
      }
   }
}
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Problems with Conventional FSM

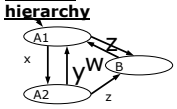
- Sometimes over-specify implementation
  - Sequencing is fully specified
- Scalability due to lack of metaphor for decomposition
  - Number of states can be unmanageable
- No concurrency support
- No support for orthogonal connections

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

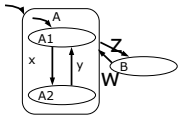
HCFSM and the Statecharts language

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - *With hierarchy* has identical functionality as *Without hierarchy*, but has one less transition (*z*)
    - Known as OR-decomposition
  - States can execute concurrently
    - Known as AND-decomposition
- Statecharts
  - Graphical language to capture HCFSM
  - *timeout*: transition with time limit as condition
  - *history*: remember last substate OR-decomposed state *A* was in before transitioning to another state *B*
    - Return to saved substate of A when returning from B instead of initial state

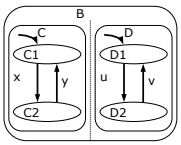
Without hierarchy



With hierarchy

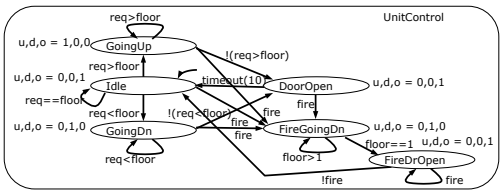


Concurrency



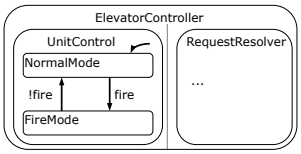
Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

UnitControl with *FireMode*



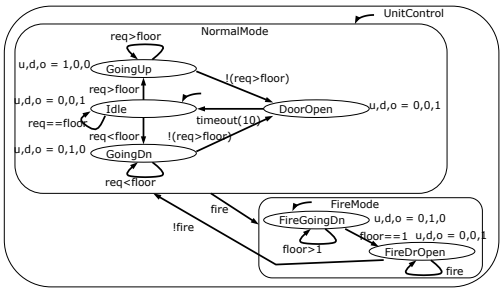
Without hierarchy

- FireMode
  - When *fire* is true, move elevator to 1st floor and open door
  - w/o hierarchy: Getting messy!
  - w/ hierarchy: Simple!



With concurrent *RequestResolver*

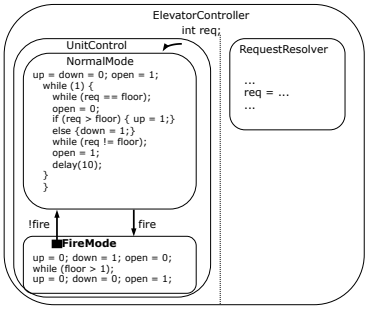
With hierarchy



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Program-state machine model (PSM): HCFSM plus sequential program model

- Program-state's actions can be FSM or sequential program
  - Designer can choose most appropriate
- Stricter hierarchy than HCFSM used in Statecharts
  - transition between sibling states only, single entry
  - Program-state may "complete"
    - Reaches end of sequential program code, OR
    - FSM transition to special *complete* substate
    - PSM has 2 types of transitions
      - Transition-immediately (TI): taken regardless of source program-state
      - Transition-on-completion (TOC): taken only if condition is true AND source program-state is complete
  - SpecCharts: extension of VHDL to capture PSM model
  - SpecC: extension of C to capture PSM model



- *NormalMode* and *FireMode* described as sequential programs
- Black square originating within *FireMode* indicates *!fire* is a TOC transition
  - Transition from *FireMode* to *NormalMode* only after *FireMode* completed

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

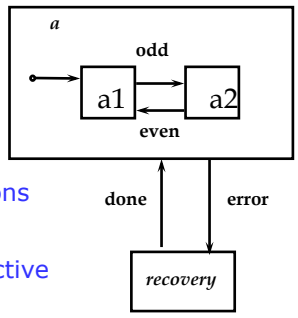
Harel's StateCharts: Extension of Conventional FSMs

- Conventional FSMs are inappropriate for the behavioral description of complex control
  - Flat and unstructured
  - Inherently sequential in nature
  - Give rise to an exponential blow-up in # of states
    - Small system extensions cause unacceptable growth in the number of states to be considered
- StateCharts support:
  - *Repeated decomposition* of states into AND/OR sub-states
    - Nested states, concurrency, orthogonal components
  - *Actions* (may have parameters)
  - *Activities* (functions executed as long as state is active)
  - *Guards*
  - History
  - A *synchronous* (instantaneous broadcast) comm. mechanism

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Hierarchical FSM models

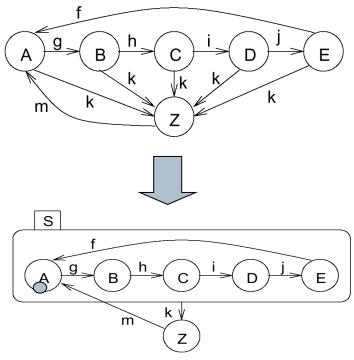
- Problem: how to reduce the size of the representation?
- Harel's classical papers on StateCharts (language) and bounded concurrency (model): 3 *orthogonal exponential reductions*
- Hierarchy:
  - State *a* "encloses" an FSM
  - Being in *a* means FSM in *a* is active
  - States of *a* are called OR states
  - Used to model pre-emption and exceptions
- Concurrency:
  - Two or more FSMs are simultaneously active
  - States are called AND states
- Non-determinism:
  - Used to abstract behavior



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Introducing hierarchy

- Classical automata not useful for complex systems (complex graphs cannot be understood by humans).
- ☞ Introduction of hierarchy ☞StateCharts [Harel, 1987]



FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

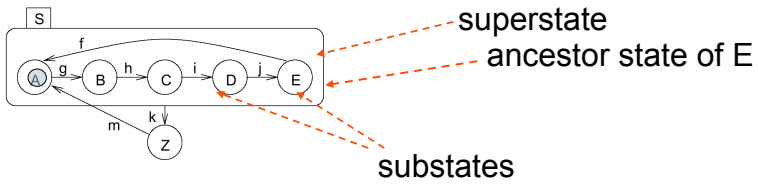
Features of StateCharts

- Nested states and hierarchy
  - Improves scalability and understandability
  - helps describing preemption
- Concurrency - two or more states can be viewed as simultaneously active
- Nondeterminism - there are properties which are irrelevant

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Definitions

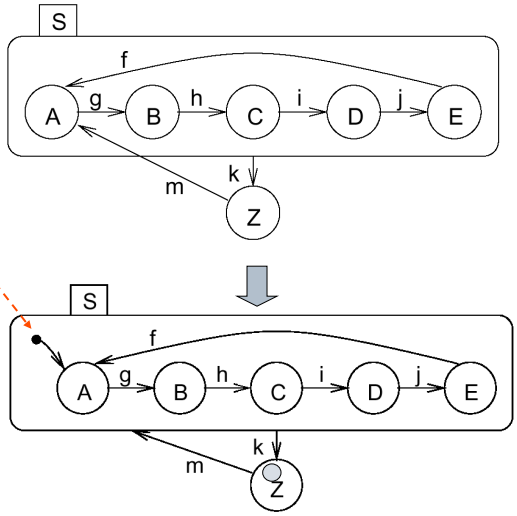
- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states.**
- States containing other states are called **super-states**.
- For each basic state s, the super-states containing s are called **ancestor states**.
- Super-states S are called **OR-super-states**, if exactly one of the sub-states of S is active whenever S is active.



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

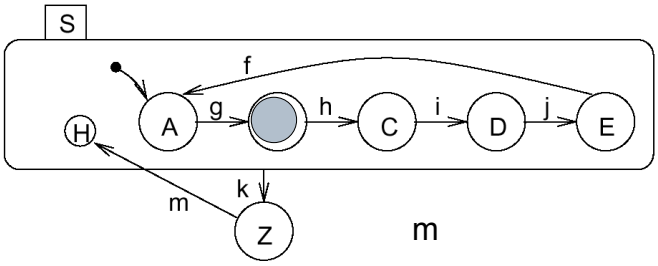
Default state mechanism

- Try to hide internal structure from outside world!
- ☞ Default state
- Filled circle indicates sub-state entered whenever super-state is entered.
- Not a state by itself!



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

History mechanism



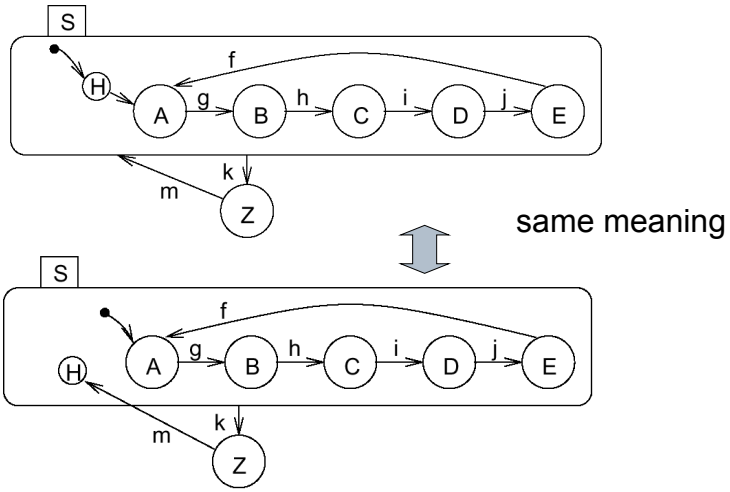
m

(behavior different from last slide)

- For input m, S enters the state it was in before S was left (can be A, B, C, D, or E). If S is entered for the very first time, the default mechanism applies.
- History and default mechanisms can be used hierarchically.

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

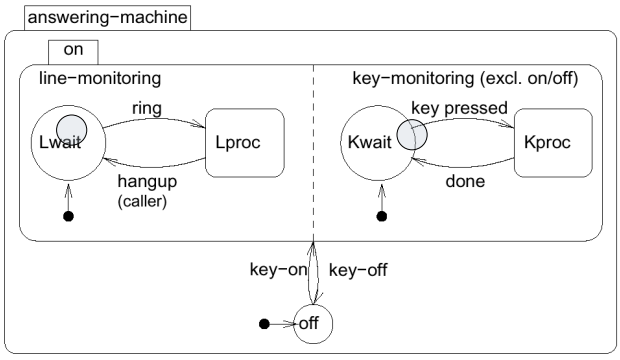
Combining history and default state mechanism



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

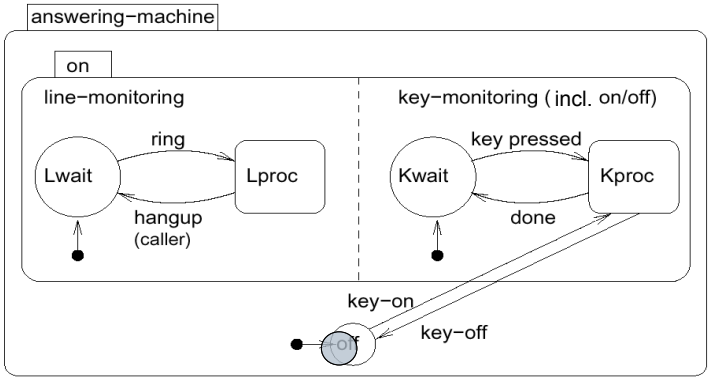
Concurrency

- Convenient ways of describing concurrency are required.
- **AND-super-states**: FSM is in **all** (immediate) sub-states of a super-state; Example:



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

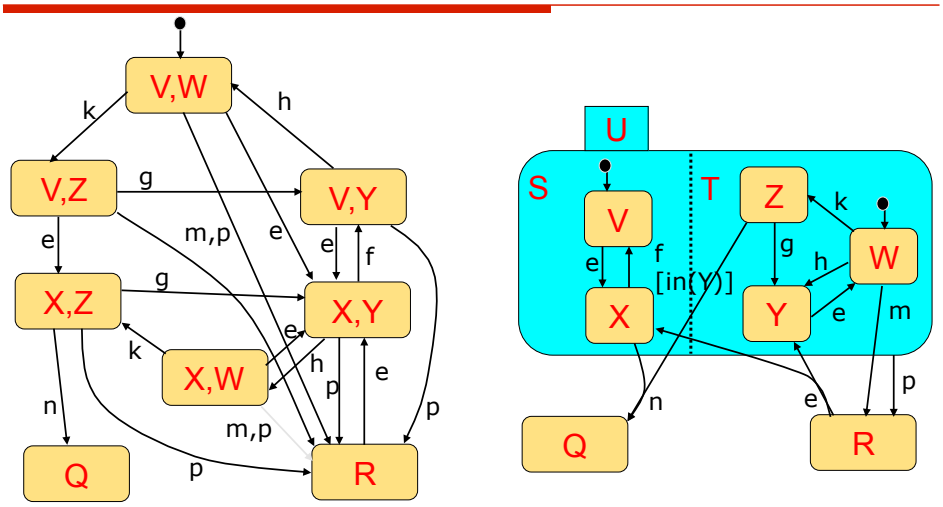
Entering and leaving AND-super-states



- Line-monitoring and key-monitoring are entered and left, when service switch is operated.

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

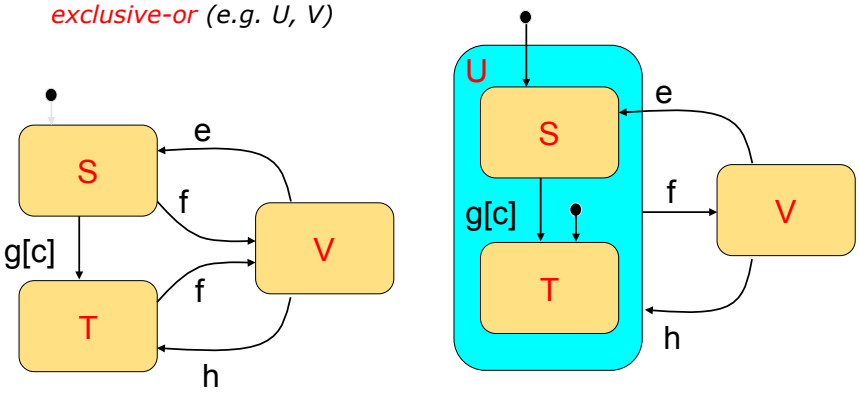
Benefits of AND-decomposition



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

AND/OR State Comparison

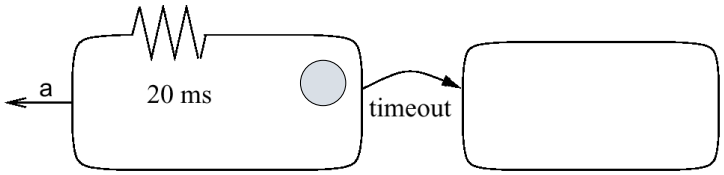
- *AND-states* have *orthogonal state components*
  - AND-decomposition can be carried out on any level of states
    - more convenient than allowing only one level of communicating FSMs
- *OR-states* have sub-states that are related to each other by *exclusive-or (e.g. U, V)*



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Timers

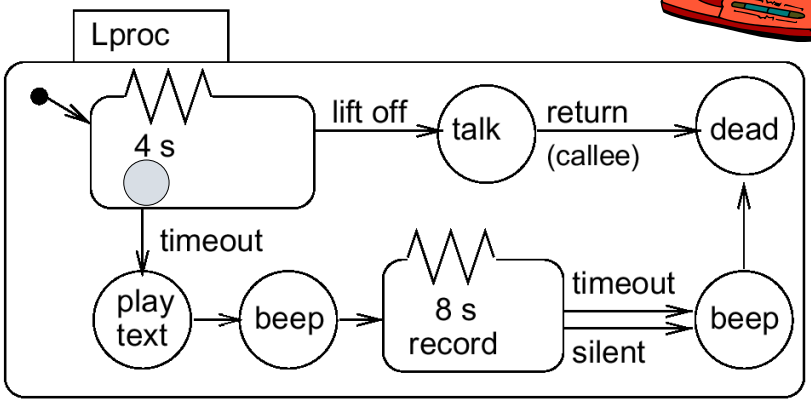
- Since time needs to be modeled in embedded systems,
- timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.



If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

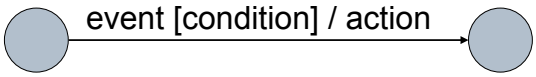
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Using timers in answering machine



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

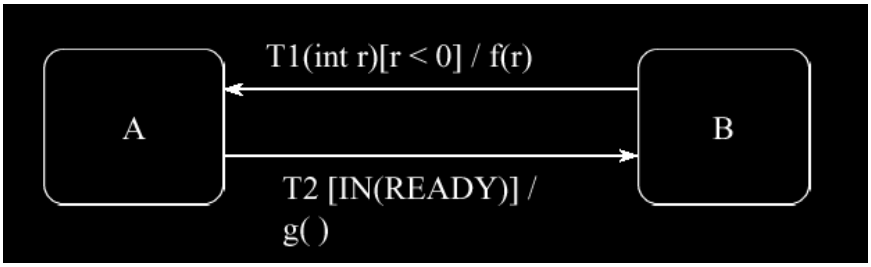
General form of edge labels



- The general syntax of an expression labeling a transition in a StateChart is *n[c]/a*, where
  - *n* is the *event* that triggers the transition
  - *c* is the *condition* that guards the transition (cannot be taken unless c is true when e occurs)
  - *a* is the *action* that is carried out if and when the transition is taken
- Alternative: *name(params)[guards]^event_list/action_list*
  - Event list, aka propagated transitions, is a list of transitions that occur in other concurrent state machines because of this transitions
- For each transition label, event condition and action are optional
  - an event can be the changing of a value
  - standard comparisons are allowed as conditions and assignment statements as actions

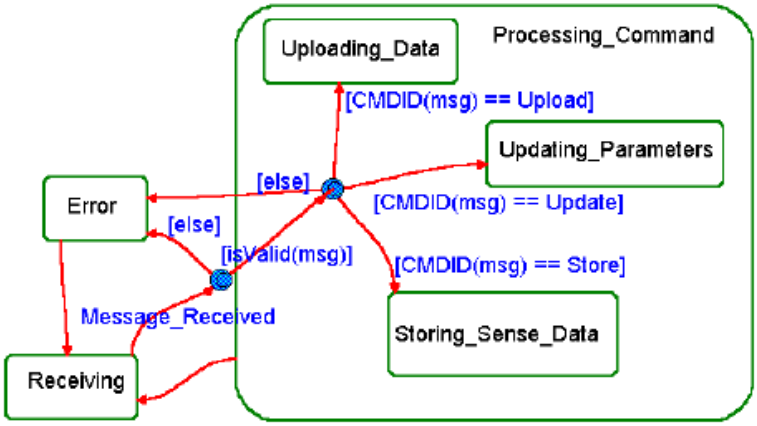
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Transitions



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Conditional Transitions



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Source: B. P. Douglass & iLogix

StateCharts Actions and Events

- An action *A* on the edge leaving a state may also appear as an event triggering a transition going into an orthogonal state
  - Executing the first transition will immediately cause the second transition to be taken simultaneously
- Actions and events may be associated to the execution of orthogonal components:
  - action *start(A)* causes activity A to start
  - event *stopped(B)* occurs when activity B stops
  - *entered(S), exited(S), in(S)* etc.

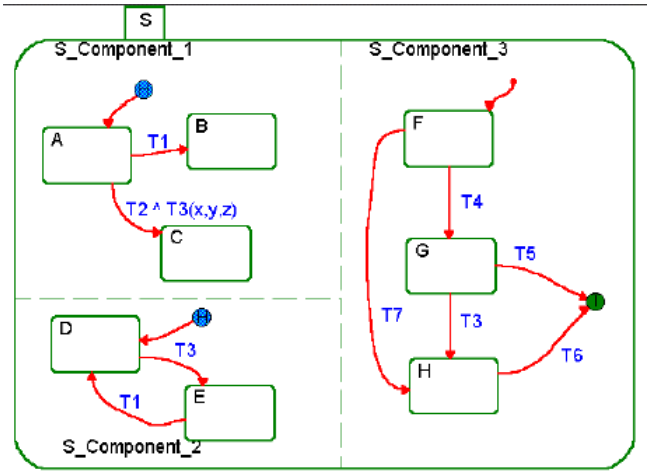
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Communication in Concurrent FSMs

- Broadcast events
  - Events are received by more than one concurrent FSM
  - Results in transitions of the same name in different FSM
- Propagated transitions
  - Transitions which are generated as a result of transitions in other FSMs

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

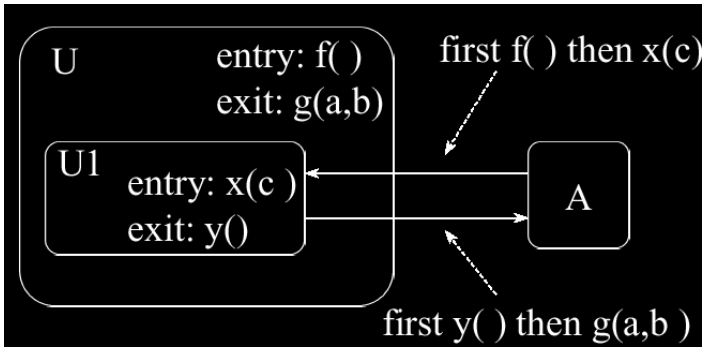
Propagations and Broadcasts



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Order of Nested Actions

- Executed from outermost – in on entry
- Executed from innermost – out on exit

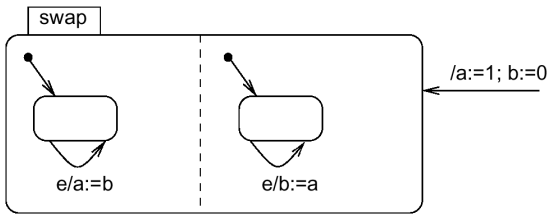


Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

The StateCharts simulation phases (StateMate Semantics)

- How are edge labels evaluated?
- Three phases:
  1. Effect of external changes on events and conditions is evaluated,
  2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
  3. Transitions become effective, variables obtain new values.
- Separation into phases 2 and 3 guarantees deterministic and reproducible behavior.

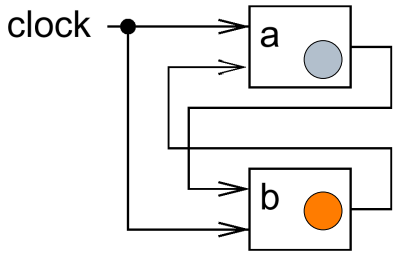
Example



- In phase 2, variables a and b are assigned to temporary variables. In phase 3, these are assigned to a and b. As a result, variables a and b are swapped.
- In a single phase environment, executing the left state first would assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The execution would be nondeterministic.

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Reflects model of clocked hardware



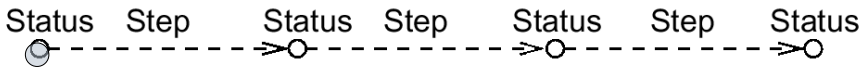
- In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

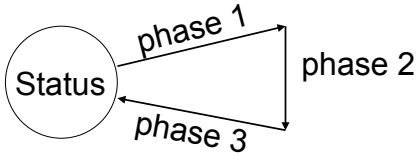
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Steps

- Execution of a StateChart model consists of a sequence of (status, step) pairs



Status= values of all variables + set of events + current time
Step = execution of the three phases (StateMate semantics)



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Evaluation of StateCharts

- **Pros:**
  - Hierarchy allows arbitrary nesting of AND- and OR-super states.
  - (StateMate-) Semantics defined in a follow-up paper to original paper.
  - Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
  - Available „back-ends" translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

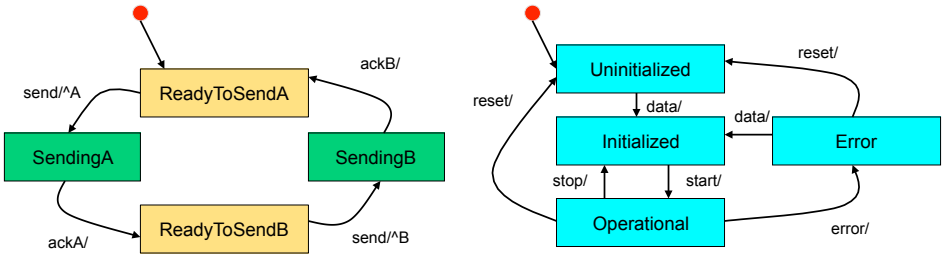
Evaluation of StateCharts

- **Cons:**
  - Generated C programs frequently inefficient,
  - Not useful for distributed applications,
  - No program constructs,
  - No description of non-functional behavior,
  - No object-orientation,
  - No description of structural hierarchy.

Extensions:
- Module charts for description of structural hierarchy.

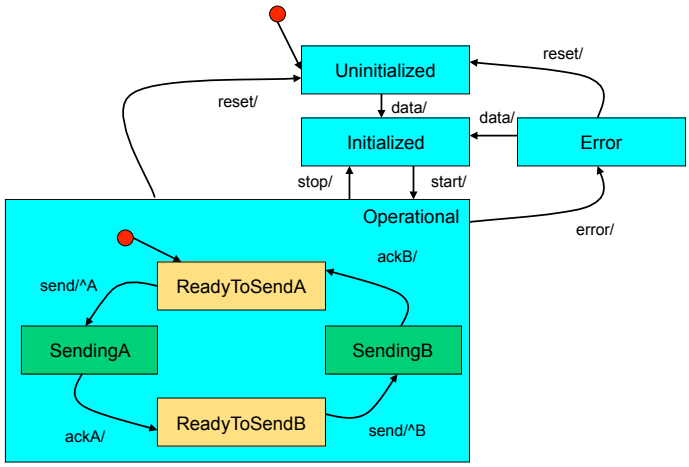
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Example of the Power of StateChart Formalism



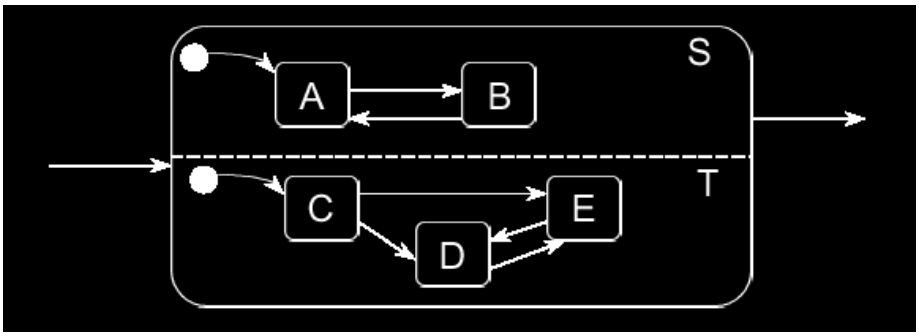
- Conflicting function & control behaviors
  - Function: primary service of the entity
  - Control: actions performed within the system context
- Solutions: single automaton, two peer concurrent states

The Combined State Machine in StateChart Formalism



Concurrent Statecharts

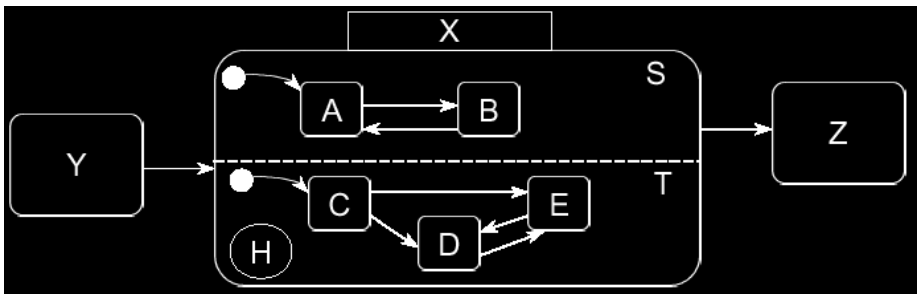
- Many embedded systems consist of multiple threads, each running an FSM
- State charts allow the modeling of these parallel threads



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Concurrent Statecharts

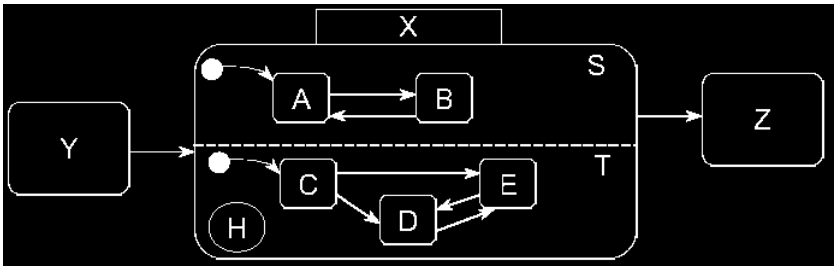
- States S and T are active at the same time as long as X is active
  - Either S.A or S.B must be active when S is active
  - Either T.C, T.D or T.E must be active when T is active



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

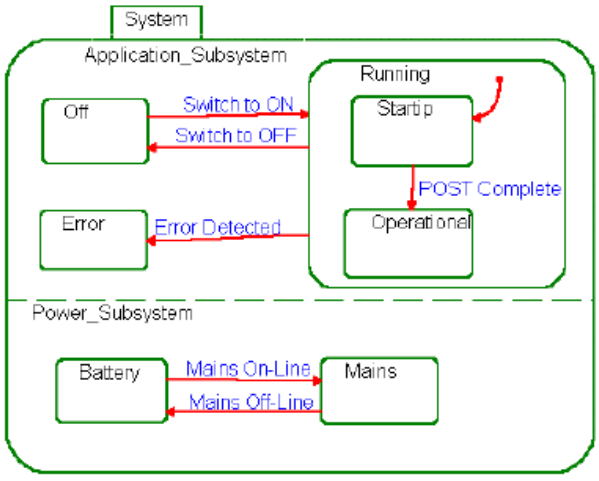
Concurrent Statecharts

- When X exits, both S and T exit
  - If S exits first, the FSM containing X must wait until T exits
  - If the two FSMs are always independent, then they must be enclosed at the highest scope



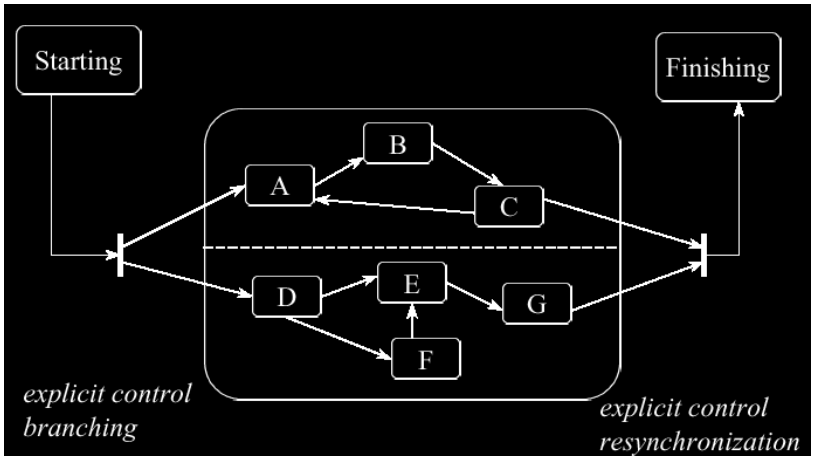
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Example Concurrent FSM



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Explicit Synchronization



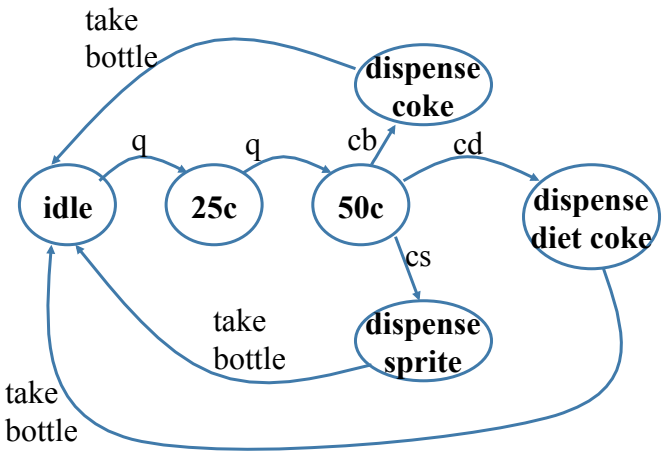
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Example: Coke Machine Version 1.0

- Suppose you have a soda machine:
  - When turned on, the machine waits for money
  - When a quarter is deposited, the machine waits for another quarter
  - When a second quarter is deposited, the machine waits for a selection
  - When the user presses "COKE," a coke is dispensed
  - When the user takes the bottle, the machine waits again
  - When the user presses either "SPRITE" or "DIET COKE," a Sprite or a diet Coke is dispensed
  - When the user takes the bottle, the machine waits again

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke Machine 1.0



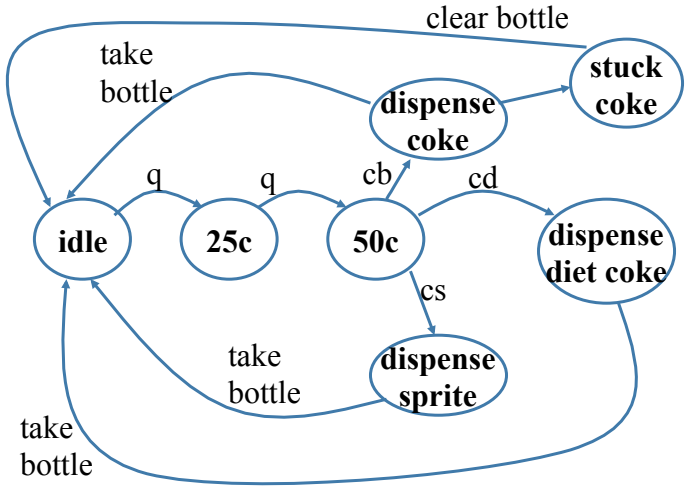
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke Machine, Version 2.0

- Bottles can get stuck in the machine
  - An automatic indicator will notify the system when a bottle is stuck
  - When this occurs, the machine will not accept any money or issue any bottles until the bottle is cleared
  - When the bottle is cleared, the machine will wait for money again
- State machine changes
  - How many new states are required?
  - How many new transitions?

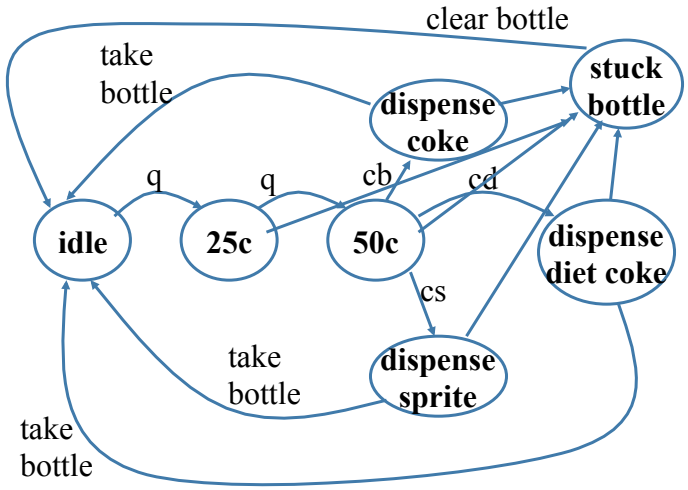
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke machine 2.0



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke machine 2.0



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke Machine, Version 2.1

- Bottles sometimes shake loose
  - An additional, automatic indicator will indicate that the bottle is cleared
  - When the bottles are cleared, the machine will return to the same state it was in before the bottle got stuck
- State machine changes
  - How many new states are required?
  - How many new transitions?

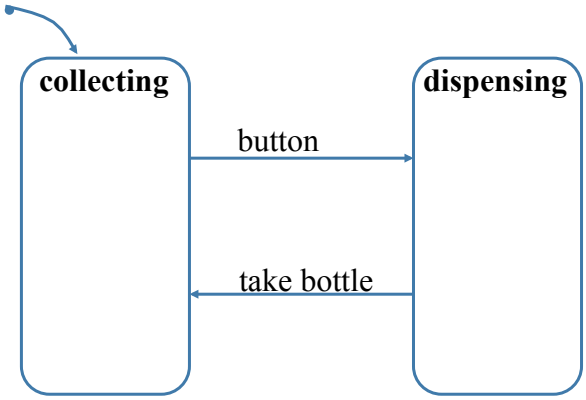
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Coke Machine, Version 3.0

- Automatic bottle filler
  - If a button is pressed, the machine will toggle between bottle filling and dispensing modes
  - When in bottle filling mode:
    - Bottles may be inserted if the Coke machine is ready
    - When a bottle is inserted, the machine will NOT be ready to accept another bottle and will check the bottle
    - If the bottle check finds a Coke was inserted, it will signal Coke_OK and return to ready
    - If the bottle check finds a Diet Coke was inserted, the coke machine will signal Diet_OK and return to ready
    - Otherwise, the bottle will be immediately dispensed
- State machine changes
  - How many new states are required?
  - How many new transitions?

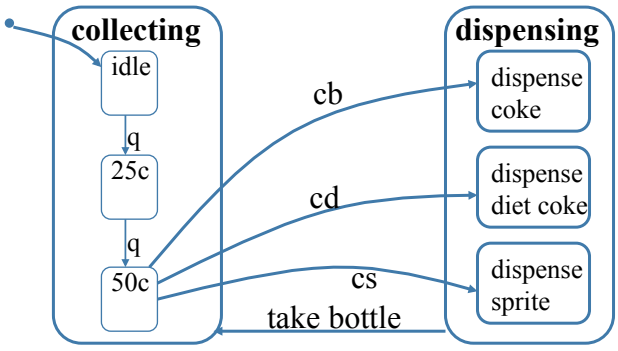
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Bottle Dispenser



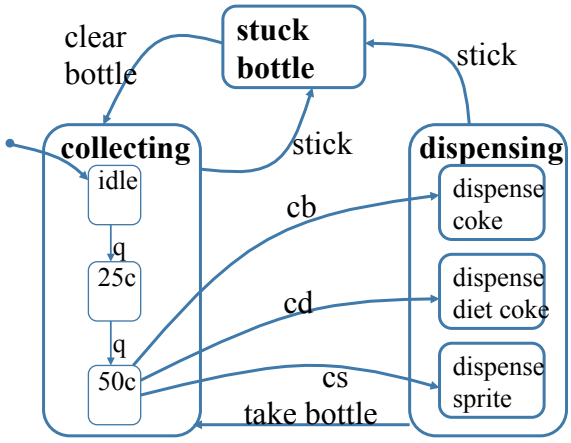
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Bottle dispenser



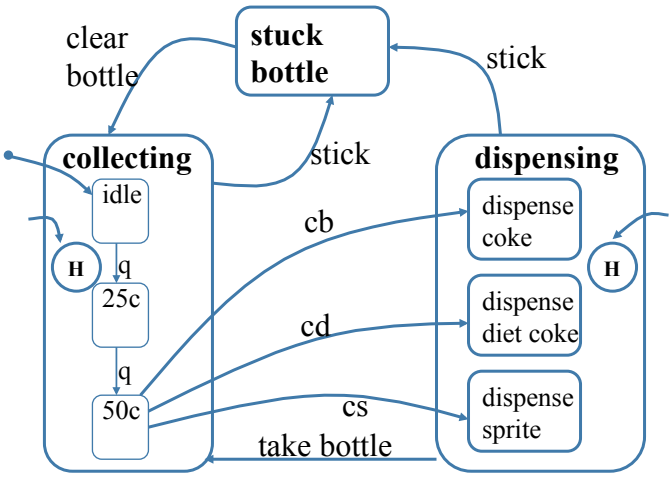
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Bottle Dispenser



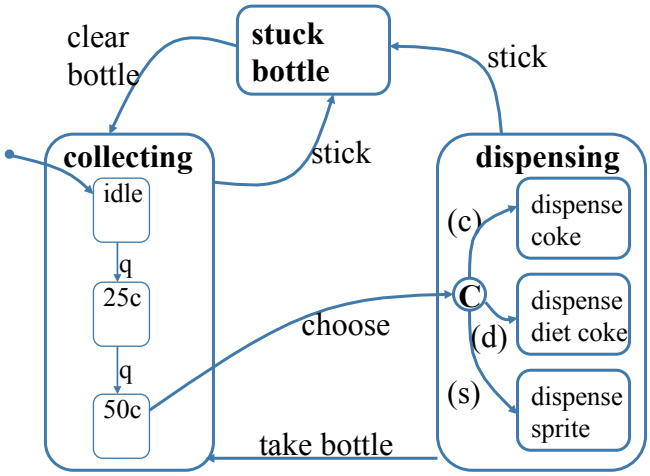
Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Adding History



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

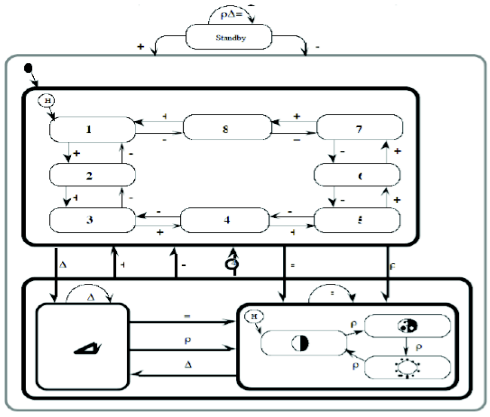
Adding Conditionals



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

StateChart Example – TV Controller

- 8 channels
- Buttons:
  - Standby
  - Channel: +/–
  - Volume: Δ+, Δ–
  - Contrast/Color/
  - Brightness: ρ
  - Pict. Adjust: ≡+, ≡–
- Missing details for:
  - volume /
  - brightness / color /
  - contrast selection

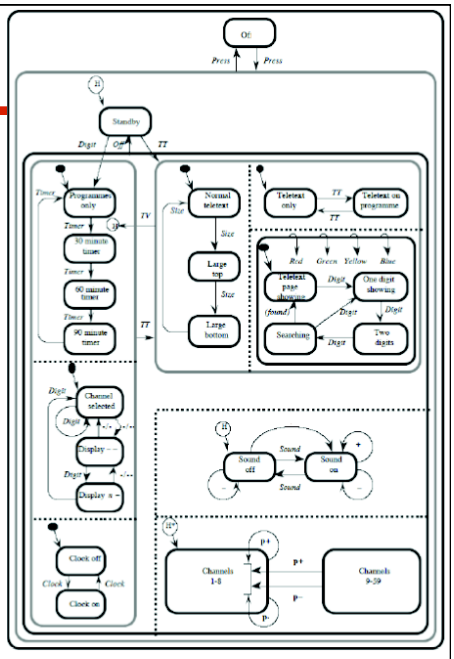


Note. ≡ means either of the ≡+ or ≡– buttons;
Δ means either of the Δ+ or Δ– buttons

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Example – TV Remote

- Independent states
  - Timer
  - Channel – split into two states
  - Sound
  - Clock



Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Conclusions

- Finite State machines
  - Simple
  - Many formal properties
  - Problem – too many states => Hierarchy
- StateChart
  - StateMate – Design environment for reactive systems
  - Behavioral modeling language for StateMate
  - Extends FSMs to better handle concurrency
  - Adds conditional statements, memory to FSM
- Further Reading:
  - D. Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, vol. 8 (1987) 231 - 274.
  - D. Harel et al., STATEMATE: A working environment for the development of complex reactive systems, IEEE Transactions on Software Eng.,vol. 16 (1990), no. 4, 403 - 414.

Source: ECE 253 Embedded System Design. Ryan Kastner, January 24, 2007. ExPRESS Lab at UC Santa Barbara

Role of appropriate model and language

- Finding appropriate model to capture embedded system is an important step
  - Model shapes the way we think of the system
    - Originally thought of sequence of actions, wrote sequential program
      - First wait for requested floor to differ from target floor
      - Then, we close the door
      - Then, we move up or down to the desired floor
      - Then, we open the door
      - Then, we repeat this sequence
    - To create state machine, we thought in terms of states and transitions among states
      - When system must react to changing inputs, state machine might be best model
        - HCFSM described *FireMode* easily, clearly
- Language should capture model easily
  - Ideally should have features that directly capture constructs of model
  - *FireMode* would be very complex in sequential program
    - Checks inserted throughout code
  - Other factors may force choice of different model
    - Structured techniques can be used instead
      - E.g., Template for state machine capture in sequential program language

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

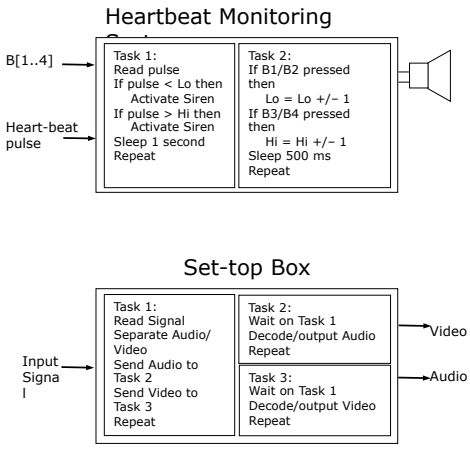
Concurrent processes and real-time systems

Universidad de Alcalá

Departamento de Electrónica

Concurrent processes or tasks

- Consider two examples having separate tasks running independently but sharing data
- Difficult to write system using sequential program model
- Concurrent process model easier
  - Separate sequential programs (processes) for each task
  - Programs communicate with each other



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

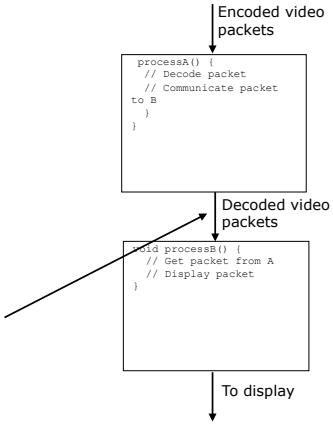
Process (or Tasks)

- A sequential program, typically an infinite loop
  - Executes concurrently with other processes
  - We are about to enter the world of "concurrent programming"
- Basic operations on tasks
  - Create and terminate
    - Create is like a procedure call but caller doesn't wait
      - Created process can itself create new processes
    - Terminate kills a process, destroying all data
  - Suspend and resume
    - Suspend puts a process on hold, saving state for later execution
    - Resume starts the process again where it left off
  - Join
    - A process suspends until a particular child process finishes execution

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Communication among processes

- Processes need to communicate data and signals to solve their computation problem
  - Processes that don't communicate are just independent programs solving separate problems
- Basic example: producer/consumer
  - Process A produces data items, Process B consumes them
  - E.g., A decodes video packets, B display decoded packets on a screen
- How do we achieve this communication?
  - Two basic methods
    - Shared memory
    - Message passing



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Shared Memory

- Processes read and write shared variables
  - No time overhead, easy to implement
  - But, hard to use – mistakes are common
- Example: Producer/consumer with a mistake
  - Share *buffer*[*N*], *count*
    - *count* = # of valid data items in *buffer*
  - *processA* produces data items and stores in *buffer*
    - If *buffer* is full, must wait
  - *processB* consumes data items from *buffer*
    - If *buffer* is empty, must wait
  - Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs. Say "count" is 3.
    - *A* loads *count* (*count* = 3) from memory into register R1 (R1 = 3)
    - *A* increments R1 (R1 = 4)
    - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
    - *B* decrements R2 (R2 = 2)
    - *A* stores R1 back to *count* in memory (*count* = 4)
    - *B* stores R2 back to *count* in memory (*count* = 2)
  - *count* now has incorrect value of 2

```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N );/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Message Passing

- Message passing
  - Data explicitly sent from one process to another
    - Sending process performs special operation, *send*
    - Receiving process must perform special operation, *receive*, to receive the data
    - Both operations must explicitly specify which process it is sending to or receiving from
    - Receive is blocking, send may or may not be blocking
  - Safer model, but less flexible

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Back to Shared Memory: Mutual Exclusion

- Certain sections of code should not be performed concurrently
  - Critical section
    - Possibly noncontiguous section of code where simultaneous updates, by multiple processes to a shared memory location, can occur
- When a process enters the critical section, all other processes must be locked out until it leaves the critical section
  - Mutex
    - A shared object used for locking and unlocking segment of shared data
    - Disallows read/write access to memory it guards
    - Multiple processes can perform lock operation simultaneously, but only one process will acquire lock
    - All other processes trying to obtain lock will be put in blocked state until unlock operation performed by acquiring process when it exits critical section
    - These processes will then be placed in runnable state and will compete for lock again

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Correct Shared Memory Solution to the Consumer-Producer Problem

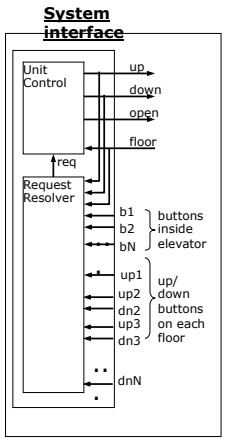
- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other
- Following the same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be put in blocked state
  - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *B* stores R2 back to *count* in memory (*count* = 2)
  - *B* executes *unlock* operation
    - *A* is placed in runnable state again
  - *A* loads *count* (*count* = 2) from memory into register R1 (R1 = 2)
  - *A* increments R1 (R1 = 3)
  - *A* stores R1 back to *count* in memory (*count* = 3)
- *Count* now has correct value of 3

```
01: data_type buffer[N];
02: int count = 0;
03: mutex count_mutex;
04: void processA() {
05:   int i;
06:   while( 1 ) {
07:     produce(&data);
08:     while( count == N );/*loop*/
09:     buffer[i] = data;
10:     i = (i + 1) % N;
11:     count_mutex.lock();
12:     count = count + 1;
13:     count_mutex.unlock();
14:   }
15: }
16: void processB() {
17:   int i;
18:   while( 1 ) {
19:     while( count == 0 );/*loop*/
20:     data = buffer[i];
21:     i = (i + 1) % N;
22:     count_mutex.lock();
23:     count = count - 1;
24:     count_mutex.unlock();
25:     consume(&data);
26:   }
27: }
28: void main() {
29:   create_process(processA);
30:   create_process(processB);
31: }
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Process Communication

- Try modeling "req" value of our elevator controller
  - Using shared memory
  - Using shared memory and mutexes
  - Using message passing



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

A Common Problem in Concurrent Programming: Deadlock

- Deadlock: A condition where 2 or more processes are blocked waiting for the other to unlock critical sections of code
  - Both processes are then in blocked state
  - Cannot execute unlock operation so will wait forever
- Example code has 2 different critical sections of code that can be accessed simultaneously
  - 2 locks needed (mutex1, mutex2)
  - Following execution sequence produces deadlock
    - *A* executes lock operation on *mutex1* (and acquires it)
    - *B* executes lock operation on *mutex2*(and acquires it)
    - *A*/*B* both execute in critical sections 1 and 2, respectively
    - *A* executes lock operation on *mutex2*
      - *A* blocked until *B* unlocks *mutex2*
    - *B* executes lock operation on *mutex1*
      - *B* blocked until *A* unlocks *mutex1*
    - DEADLOCK!
- One deadlock elimination protocol requires locking of numbered mutexes in increasing order and two-phase locking (2PL)
  - Acquire locks in 1st phase only, release locks in 2nd phase

```
01: mutex mutex1, mutex2;
02: void processA() {
03:   while( 1 ) {
04:     …
05:     mutex1.lock();
06:     /* critical section 1 */
07:     mutex2.lock();
08:     /* critical section 2 */
09:     mutex2.unlock();
10:     /* critical section 1 */
11:     mutex1.unlock();
12:   }
13: }
14: void processB() {
15:   while( 1 ) {
16:     …
17:     mutex2.lock();
18:     /* critical section 2 */
19:     mutex1.lock();
20:     /* critical section 1 */
21:     mutex1.unlock();
22:     /* critical section 2 */
23:     mutex2.unlock();
24:   }
25: }
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Synchronization among processes

- Sometimes concurrently running processes must synchronize their execution
  - When a process must wait for:
    - another process to compute some value
    - reach a known point in their execution
    - signal some condition
- Recall producer-consumer problem
  - *processA* must wait if *buffer* is full
  - *processB* must wait if *buffer* is empty
  - This is called busy-waiting
    - Process executing loops instead of being blocked
    - CPU time wasted
- More efficient methods
  - Join operation, and blocking send and receive discussed earlier
    - Both block the process so it doesn't waste CPU time
  - Condition variables and monitors

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Condition variables

- Condition variable is an object that has 2 operations, signal and wait
- When process performs a wait on a condition variable, the process is blocked until another process performs a signal on the same condition variable
- How is this done?
  - Process *A* acquires lock on a mutex
  - Process *A* performs wait, passing this mutex
    - Causes mutex to be unlocked
  - Process *B* can now acquire lock on same mutex
  - Process *B* enters critical section
    - Computes some value and/or make condition true
  - Process *B* performs signal when condition true
    - Causes process *A* to implicitly reacquire mutex lock
    - Process *A* becomes runnable

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Condition variable example: consumer-producer

- 2 condition variables
  - *buffer_empty*
    - Signals at least 1 free location available in *buffer*
  - *buffer_full*
    - Signals at least 1 valid data item in *buffer*
- *processA*:
  - produces data item
  - acquires lock (*cs_mutex*) for critical section
  - checks value of *count*
  - if *count* = *N*, *buffer* is full
    - performs wait operation on *buffer_empty*
    - this releases the lock on *cs_mutex* allowing *processB* to enter critical section, consume data item and free location in *buffer*
    - *processB* then performs signal
  - if *count* < *N*, *buffer* is not full
    - *processA* inserts data into *buffer*
    - increments *count*
    - signals *processB* making it runnable if it has performed a wait operation on *buffer_full*

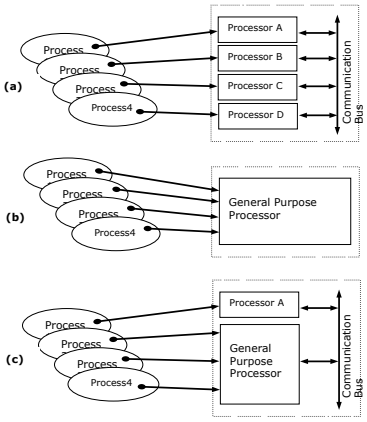
Consumer-producer using condition variables

```
01: data_type buffer[N];
02: int count = 0;
03: mutex cs_mutex;
04: condition buffer_empty, buffer_full;
06: void processA() {
07:   int i;
08:   while( 1 ) {
09:     produce(&data);
10:     cs_mutex.lock();
11:     if( count == N ) buffer_empty.wait(cs_mutex);
13:     buffer[i] = data;
14:     i = (i + 1) % N;
15:     count = count + 1;
16:     cs_mutex.unlock();
17:     buffer_full.signal();
18:   }
19: }
20: void processB() {
21:   int i;
22:   while( 1 ) {
23:     cs_mutex.lock();
24:     if( count == 0 ) buffer_full.wait(cs_mutex);
26:     data = buffer[i];
27:     i = (i + 1) % N;
28:     count = count - 1;
29:     cs_mutex.unlock();
30:     buffer_empty.signal();
31:     consume(&data);
32:   }
33: }
34: void main() {
35:   create_process(processA); create_process(processB);
37: }
```

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Concurrent process model: implementation

- Can use single and/or general-purpose processors
- (a) Multiple processors, each executing one process
  - True multitasking (parallel processing)
  - General-purpose processors
    - Use programming language like C and compile to instructions of processor
    - Expensive and in most cases not necessary
  - Custom single-purpose processors
    - More common
- (b) One general-purpose processor running all processes
  - Most processes don't use 100% of processor time
  - Can share processor time and still achieve necessary execution rates
- (c) Combination of (a) and (b)
  - Multiple processes run on one general-purpose processor while one or more processes run on own single_purpose processor



Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Implementation: multiple processes sharing single processor

- Can manually rewrite processes as a single sequential program
  - Ok for simple examples, but extremely difficult for complex examples
  - Automated techniques have evolved but not common
  - E.g., simple Hello World concurrent program from before would look like:

```
I = 1; T = 0;
while (1) {
    Delay(I); T = T + 1;
    if X modulo T is 0 then call PrintHelloWorld
    if Y modulo T is 0 then call PrintHowAreYou
}
```

- Can use multitasking operating system
  - Much more common
  - Operating system schedules processes, allocates storage, and interfaces to peripherals, etc.
  - Real-time operating system (RTOS) can guarantee execution rate constraints are met
  - Describe concurrent processes with languages having built-in processes (Java, Ada, etc.) or a sequential programming language with library support for concurrent processes (C, C++, etc. using POSIX threads for example)
- Can convert processes to sequential program with process scheduling right in code
  - Less overhead (no operating system)
  - More complex/harder to maintain

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Processes vs. threads

- Different meanings when operating system terminology
- Regular processes
  - Heavyweight process
  - Own virtual address space (stack, data, code)
  - System resources (e.g., open files)
- Threads
  - Lightweight process
  - Subprocess within process
  - Only program counter, stack, and registers
  - Shares address space, system resources with other threads
    - Allows quicker communication between threads
  - Small compared to heavyweight processes
    - Can be created quickly
    - Low cost switching between threads

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Implementation: suspending, resuming, and joining

- Multiple processes mapped to single-purpose processors
  - Built into processor's implementation
  - Could be extra input signal that is asserted when process suspended
  - Additional logic needed for determining process completion
    - Extra output signals indicating process done
- Multiple processes mapped to single general-purpose processor
  - Built into programming language or special multitasking library like POSIX
  - Language or library may rely on operating system to handle

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Implementation: process scheduling

- Must meet timing requirements when multiple concurrent processes implemented on single general-purpose processor
  - Not true multitasking
- Scheduler
  - Special process that decides when and for how long each process is executed
  - Implemented as preemptive or nonpreemptive scheduler
  - Preemptive
    - Determines how long a process executes before preempting to allow another process to execute
      - Time quantum: predetermined amount of execution time preemptive scheduler allows each process (may be 10 to 100s of milliseconds long)
    - Determines which process will be next to run
  - Nonpreemptive
    - Only determines which process is next after current process finishes execution

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Scheduling: priority

- Process with highest priority always selected first by scheduler
  - Typically determined statically during creation and dynamically during execution
- FIFO
  - Runnable processes added to end of FIFO as created or become runnable
  - Front process removed from FIFO when time quantum of current process is up or process is blocked
- Priority queue
  - Runnable processes again added as created or become runnable
  - Process with highest priority chosen when new process needed
  - If multiple processes with same highest priority value then selects from them using first-come first-served
  - Called priority scheduling when nonpreemptive
  - Called round-robin when preemptive

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Priority assignment

- Period of process
  - Repeating time interval the process must complete one execution within
    - E.g., period = 100 ms
    - Process must execute once every 100 ms
  - Usually determined by the description of the system
    - E.g., refresh rate of display is 27 times/sec
    - Period = 37 ms
- Execution deadline
  - Amount of time process must be completed by after it has started
    - E.g., execution time = 5 ms, deadline = 20 ms, period = 100 ms
    - Process must complete execution within 20 ms after it has begun regardless of its period
    - Process begins at start of period, runs for 4 ms then is preempted
    - Process suspended for 14 ms, then runs for the remaining 1 ms
    - Completed within 4 + 14 + 1 = 19 ms which meets deadline of 20 ms
    - Without deadline process could be suspended for much longer
- Rate monotonic scheduling
  - Processes with shorter periods have higher priority
  - Typically used when execution deadline = period
- Deadline monotonic scheduling
  - Processes with shorter deadlines have higher priority
  - Typically used when execution deadline < period

Rate monotonic

Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	100 ms	6
D	12 ms	1
E	40 ms	4
F	75 ms	2

Deadline monotonic

Process	Deadline	Priority
G	17 ms	5
H	50 ms	2
I	32 ms	3
J	10 ms	6
K	140 ms	1
L	32 ms	4

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Real-time systems

- Systems composed of 2 or more cooperating, concurrent processes with stringent execution time constraints
  - E.g., set-top boxes have separate processes that read or decode video and/or sound concurrently and must decode 20 frames/sec for output to appear continuous
  - Other examples with stringent time constraints are:
    - digital cell phones
    - navigation and process control systems
    - assembly line monitoring systems
    - multimedia and networking systems
    - etc.
  - Communication and synchronization between processes for these systems is critical
  - Therefore, concurrent process model best suited for describing these systems

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

Real-time operating systems (RTOS)

- Provide mechanisms, primitives, and guidelines for building real-time embedded systems
- Windows CE
  - Built specifically for embedded systems and appliance market
  - Scalable real-time 32-bit platform
  - Supports Windows API
  - Perfect for systems designed to interface with Internet
  - Preemptive priority scheduling with 256 priority levels per process
  - Kernel is 400 Kbytes
- QNX
  - Real-time microkernel surrounded by optional processes (resource managers) that provide POSIX and UNIX compatibility
    - Microkernels typically support only the most basic services
    - Optional resource managers allow scalability from small ROM-based systems to huge multiprocessor systems connected by various networking and communication technologies
  - Preemptive process scheduling using FIFO, round-robin, adaptive, or priority-driven scheduling
  - 32 priority levels per process
  - Microkernel < 10 Kbytes and complies with POSIX real-time standard

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

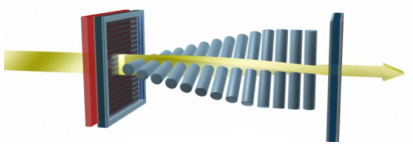
Summary

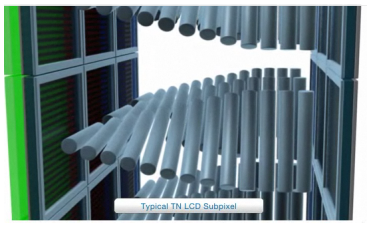
- Computation models are distinct from languages
- Sequential program model is popular
  - Most common languages like C support it directly
- State machine models good for control
  - Extensions like HCFSM provide additional power
  - PSM combines state machines and sequential programs
- Concurrent process model for multi-task systems
  - Communication and synchronization methods exist
  - Scheduling is critical

Source: Embedded Systems Design: A Unified Hardware/Software Introduction, Vahid/Givargis

LCD (Liquid Crystal Display)

- LCD Panel is based on
  - A light valve for each pixel that turn the light on, off, or an intermediate level.
- Grid of such light valve for the LCD display panel.
- A back light and display enhancement films create the illumination.

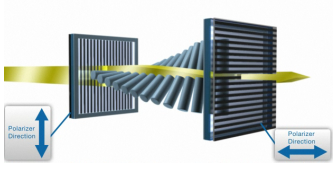


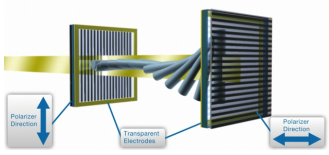


Source: Computer Graphics Course. Department of Computer Science , Ben-Gurion University of the Negev, Israel

About Liquid Crystal

- Liquid crystal molecules can move freely while maintaining their orientation.
- It align itself to a polyimide film to the inside of a panel glass.
- When the two glass panels are not aligned the liquid crystal twists accordingly.
- The liquid crystal will also align to electric field.





Source: Computer Graphics Course. Department of Computer Science , Ben-Gurion University of the Negev, Israel

What Does TFT Stand For?

T = thin

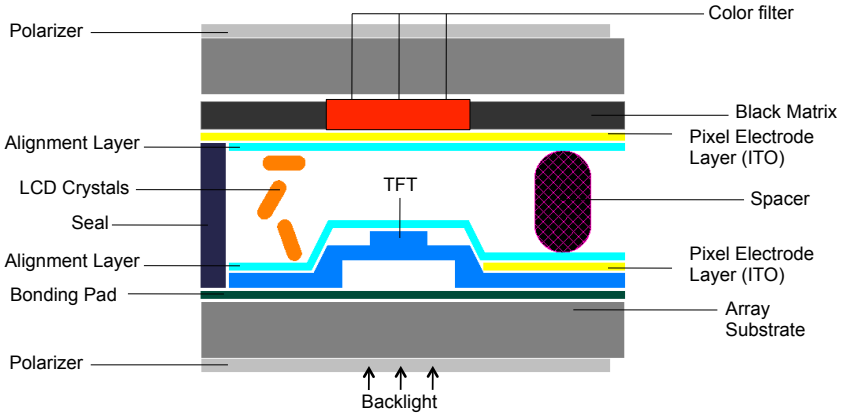
F = film

T = transistor

TFT stands for thin film transistor. A TFT is actually a component of a an LCD designed to improve the quality and control of the LCD display. It is basically a tiny transistor linked to each individual pixel on the screen. In today's marketplace, TFT technology provides the best resolution of all the flat-panel techniques. TFT screens are sometimes called active-matrix LCDs.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

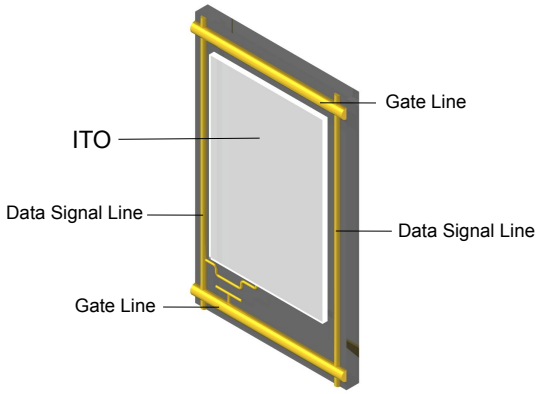
How TFT Technology Works



A TFT uses liquid crystal to control the passage of light. The basic structure of a TFT-LCD panel may be thought of as two pieces of glass with a layer of liquid crystal between them. The front glass is fitted with a color filter, while the back glass has transistors on it. When voltage is applied to a transistor, the liquid crystal is bent, allowing light to pass through to form a pixel. A light source, in many cases an LED, is located at the back of the panel and is what ,makes up the backlight. The front glass is fitted with a color filter, which gives each pixel its own color. The combination of these pixels in different colors forms the image on the panel.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

How TFT Technology Works



A TFT panel array contains a specific number of pixels, often known as subpixels. Thousands or millions of these unit pixels together create an image on the display. This diagram shows the simple structure of a sub-pixel. Each unit pixel contains a TFT, a pixel electrode or ITO and microscopic storage capacitors. Each unit pixel is connected to one of the gate bus lines and one of the data bus lines in a matrix format. This allows for easy individual pixel addressing. TFT devices are switching devices, which function to turn each individual pixel on or off thereby controlling the number of electrons that flow into the ITO zone. As the number of electrons reaches the expected value, TFT turns off and these electrons can be kept within the ITO zone.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

How Do TFT's Generate Color?

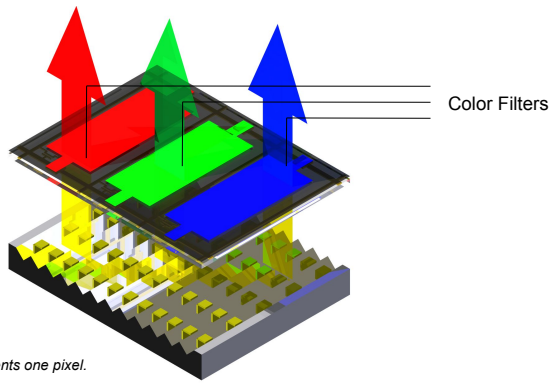
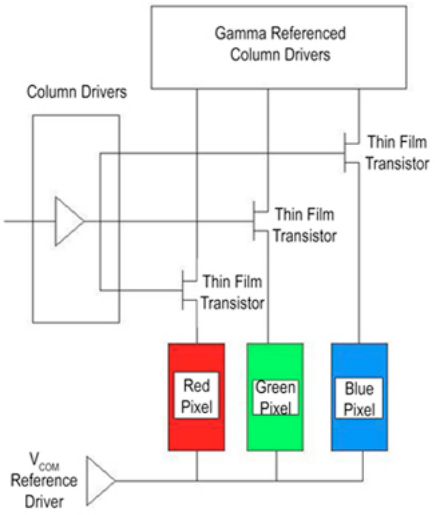


Illustration represents one pixel.

When power is applied to bend the liquid crystal, light passes through from the backlight into the color filter. How much light that passes through depends on the amount of power applied to the pixel. If there were no color filter, the output would be in the form of a grayscale. The color filter is an RGB (red, green and blue) stripe. One set of three subpixels makes up one unit pixel. The white light from the backlight passes through the color filter and outputs all three colors; the intensity of which depends on how far the liquid crystal gets bent. The human eye cannot resolve each color from a tiny pixel; instead the brain mixes the 3 colors together to give the appearance of the combined color (such as mixing red and blue to make purple).

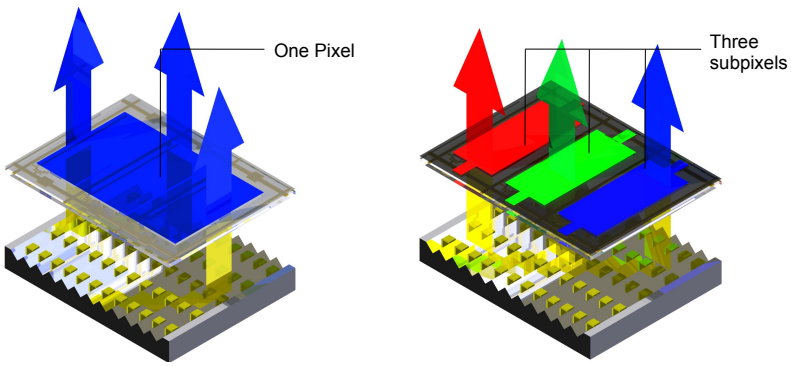
Source: Lumex (http://www.lumex.com) - LED and LCD technology

How Do TFT's Generate Color?



Source: Lumex (http://www.lumex.com) - LED and LCD technology

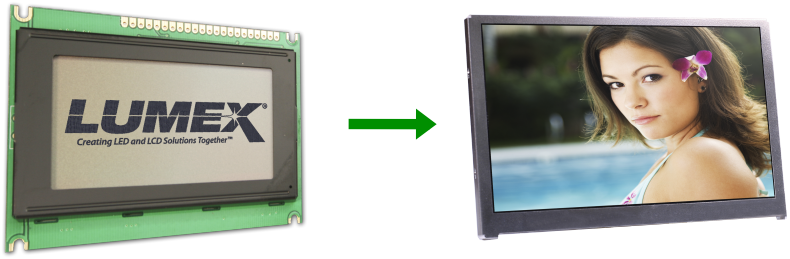
Difference Between Monochromatic and TFT



Monochromatic displays consist of a passive-matrix structure utilizing super-twisted nematic fluid with no switching devices. Most of the monochromatic displays offer black and white images except for the color STN types which offers 16 colors only. Slow response time and less contrast are typical of passive-matrix addressed LCDs. TFTs consist of an active matrix structure utilizing a layer of transistors for addressing each pixel. TFT offers full color capability, high pixel resolution and good contrast.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

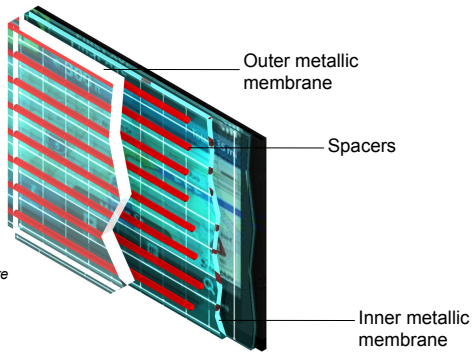
Transitioning from Monochromatic to TFT



Due to the simplicity in construction of a monochromatic LCD, they are ideal for text and static image on the screen with no color. TFTs are a bit more complex in construction compared to a monochromatic display, therefore TFT require more data input in order to display full color dynamic video on the screen.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

Capacitive vs. Resistive TFT Technology



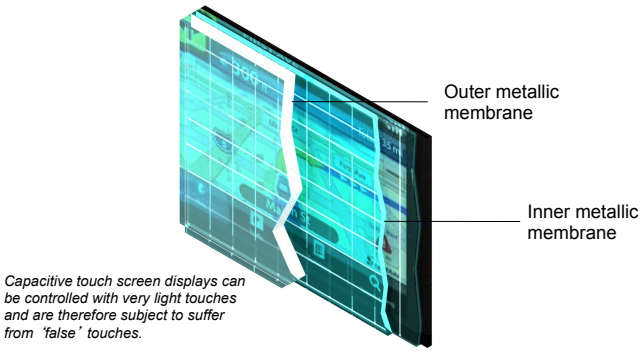
Resistive touch screen displays have multiple layers that are separated by thin spacers.

Resistive type touch screens require more pressure to activate than capacitive touch screens.

TFT applications are including touch screen capability in order to make the user interface more friendly. There are two primary types of touch screens: resistive and capacitive. Simply, resistive touch screens use two thin layers of a metallic membrane with a gap in between the two. A person touching the screen at a specific point compresses the outer layer until it touches the other layer. This technology is relatively inexpensive, however it can also be fragile. Environments, such as medical equipment, require resistive touch screens because they are easy to clean, maintain and do no register false readings.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

Capacitive vs. Resistive TFT Technology



Capacitive touch screen displays can be controlled with very light touches and are therefore subject to suffer from 'false' touches.

Capacitive touch screens are similar to resistive touch screens in that they have multiple layers. With capacitive touch screen technology, the outer layer is an insulator and the inner layer is conductive. When the finger touched the outer layer, it changes the capacitance and registers a touch. Capacitive touch screens, due to their nature, require the bare finger and can register false touches, but are more impact resistant. Capacitive touch screens are generally more expensive than resistive touch screens due to their relative robustness.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

Advantages of TFT

- Space savings
- Finer imaging quality
- Less glare and flicker
- More vibrant color
- Increased response time

There are several advantages to TFT technology, including space savings, enhanced resolution and finer quality. Of all the flat panel technologies available, TFT displays offer tremendous space savings. A Lumex InfoVue TFT module, for instance, starts at an industry-leading 3mm in width. In addition, TFT displays provide a finer imaging quality with less glare and flicker for a reduction in eye strain to the end user. TFT displays also offer a more vibrant color and response time than other color LCD technologies.

Source: Lumex (http://www.lumex.com) - LED and LCD technology

Controlador TFT SPFD5408B

SPFD5408B

720-channel 6-bit Source Driver with System-on-chip for Color Amorphous TFT-LCDs

- Caracerísticas del controlador de TFT
  - Resolución de 320 x 240 pixels con 18 bits por pixel (256K colores)
  - Diferentes modos de comunicación (18 bits, 16 bits, 8 bits, SPI)
    - En el módulo HY28A - LCDA fijado el modo 16 bits por hardware
- Registros de control
  - Dispone de más de 50 registros de control
- Memoria CGRAM (Memoria gráfica)
  - Contiene la información de los 320x240 pixels con 18 bits de información por pixel
  - Representa la información que es presentada en el display.

Controlador TFT SPFD5408B

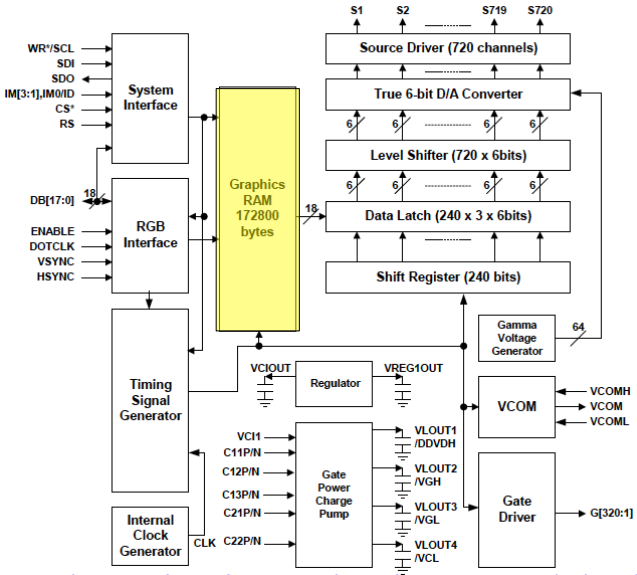
SPFD5408B

720-channel 6-bit Source Driver with System-on-chip for Color Amorphous TFT-LCDs

2. FEATURE

- One-chip solution for amorphous TFT-LCD.
- Supports resolution up to 240xRGBx320, incorporating a 720-channel source driver and a 320-channel gate driver
- Outputs 64 γ -corrected values using an internal true 6-bit resolution D/A converter to achieve 262K colors
- Built-in 172800 bytes internal RAM
- Line Inversion AC drive / frame inversion AC drive
- System interfaces
  - High-speed interfaces to 8-, 9-, 16-, and 18-bit parallel ports
  - Serial Peripheral Interface (SPI)
- Interfaces for moving picture display
  - 6-, 16-, and 18-bit RGB interfaces
- Diverse RAM accessing for functional display
  - Window address function to display at any area on the screen via a moving picture display interface
  - Window address function to limit the data rewriting area and reduce data transfer
  - Moving and still picture can display at the same time
  - Vertical scrolling function
  - Partial screen display

Controlador TFT SPFD5408B



Graphics RAM (GRAM): 172800 bytes (240 x 320 x 18 / 8 bytes)

Controlador TFT SPFD5408B

- Líneas de control
  - RS indica si se escribe en el puntero de registro o en el registro apuntado por el puntero.

Interface input Signals				
/CS	1	I	MPU	Chip select signal. Low: the SPFD5408B is accessible High: the SPFD5408B is not accessible Must connect to the GND or IOVCC level when not used. This pin has weak pull high/low resistors and can be modified to high / low by metal layer change for customer's request.
RS	1	I	MPU	Register select signal. Low: Index register or internal status is selected. High: Control register is selected. Must connect to the GND or IOVCC level when not used. This pin has weak pull high/low resistors and can be modified to high / low by metal layer change for customer's request.
(/WR) / (SCL)	1	I	MPU	(A) In 80-system interface mode, a write strobe signal can be input via this pin and initializes a write operation when the signal is low.
/RD	1	I	MPU	In 80-system interface mode, a read strobe signal can be input via this pin and initializes a read operation when the signal is low. Must connect to the GND or IOVCC level when not in use. This pin has weak pull high/low resistors and can be modified to high / low by metal layer change for customer's request.

Controlador TFT SPFD5408B

- Líneas de control
  - RS indica si se escribe en el puntero de registro o en el registro apuntado por el puntero.

DB0-DB17	1	I/O	MPU	Served as an 18-bit parallel bi-directional data bus. Data bus pin assignment corresponding to different modes are summarized in the table:

Mode	Pin Assignment
8-bit system interface	DB17-DB10
9-bit system interface	DB17-DB9
16-bit system interface	DB17-DB10, DB8-DB1
18-bit system interface	DB17-DB0
6-bit External (RGB) interface	DB17-DB12
16-bit External (RGB) interface	DB17-13, DB11-DB1
18-bit External (RGB) interface	DB17-DB0

Must connect to the GND or IOVCC level when not in use.
These pins have weak pull high/low resistors and can be modified to high / low by metal layer change for customer's request.

Controlador TFT SPFD5408B

- Configuración del modo de comunicación

IM3	IM2	IM1	IM0/ ID	Interface Mode	DB Pin	Colors	
0	0	0	0	Setting disabled	-	-	
0	0	0	1	Setting disabled	-	-	
0	0	1	0	80-system 16-bit interface	DB17-10, DB8-1	262,144 see Note 1	16 bits
0	0	1	1	80-system 8-bit interface	DB17-10	262,144 see Note 2	8 bits
0	1	0	*(ID)	Clock synchronous serial interface	-	65,536	SPI
0	1	1	0	Setting disabled	-	-	
0	1	1	1	Setting disabled	-	-	
1	0	0	0	Setting disabled	-	-	
1	0	0	1	Setting disabled	-	-	
1	0	1	0	80-system 18-bit interface	DB17-0	262,144	18 bits
1	0	1	1	80-system 9-bit interface	DB17-9	262,144	9 bits
1	1	0	0	Setting disabled	-	-	
1	1	0	1	Setting disabled	-	-	
1	1	1	0	Setting disabled	-	-	
1	1	1	1	Setting disabled	-	-	

Notes: 1. 65,536 colors in one transfer mode
2. 65,536 colors in two transfers mode

Controlador TFT SPFD5408B

- Acceso los Registros de Configuración
  - Hay más de 50 registros de configuración de 16 bits
  - Para escribir en un registro primero hay que escribir un puntero que apunte al registro
  - La escritura en el puntero al registro (Index Register) o en el registro apuntado por el puntero se controla con el pin RS
    - RS = 0 → El dato se escribe en el Index Register (Puntero)
    - RS = 1 → El dato se escribe en el registro apuntado por el Index Register

6.2.1. Index Register (IR)

R/W	RS	CB15	CCB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	0	*	*	*	*	*	*	*	*	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0

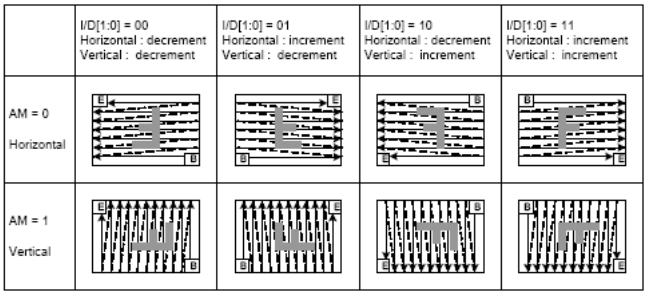
The index register specifies the index (R00h ~ RFFh) of a control register. The index range is from "000_0000" to "111_1111" in binary format.

6.2.5. Entry Mode (R03h)

R/W	RS	CB15	CB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	1	TRIREG	DFM	0	BGR	0	0	0	0	ORG	0	I/D1	I/D0	AM	0	0	0

Controlador TFT SPFD5408B

- Ejemplo de registro de control: R03h – Entry Mode
  - Entre otras cosas permite configurar la orientación de la visualización.



6.2.5. Entry Mode (R03h)

R/W	RS	CB15	CB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	1	TRIREG	DFM	0	BGR	0	0	0	0	ORG	0	I/D1	I/D0	AM	0	0	0

Controlador TFT SPFD5408B

- Acceso a la CGRAM
  - Se debe especificar el puntero de acceso a la CGRAM (R20h y R21h) y luego escribir o leer
  - Con cada lectura o escritura se autoincrementa el puntero
  - Cada fila sólo tiene útiles 240 (0x0000-0x00EF) (0x0100-0x01EF)…

6.2.18. GRAM Address Set (Horizontal Address) (R20h)

R/W	RS	CB15	CB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	1	0	0	0	0	0	0	0	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0

6.2.19. GRAM Address Set (Vertical Address) (R21h)

R/W	RS	CB15	CB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	1	0	0	0	0	0	0	0	AD16	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8

6.2.20. Write Data to GRAM (R22h)

R/W	RS	CB15 CB14 CB13 CB12 CB11 CB10 CB9 CB8 CB7 CB6 CB5 CB4 CB3 CB2 CB1 CB0
W	1	RAM write data (WD17-0) The DB17-0 pin assignment is different in different interface modes.

6.2.21. Read Data Read from GRAM (R22h)

R/W	RS	CB15 CB14 CB13 CB12 CB11 CB10 CB9 CB8 CB7 CB6 CB5 CB4 CB3 CB2 CB1 CB0
R	1	RAM Read data (RD17-0) The DB17-0 pin assignment is different in different interface modes.

Controlador TFT SPFD5408B

- Acceso mediante un bus paralelo de 16 bits
  - Modo 64K colores → 16 bits por pixel
    - Cada pixel necesita una palabra de 16 bits
    - Se pierde la resolución del bit menos significativo del R y del B
  - El modo se controla con la información del registro "Entry Mode (R03H)

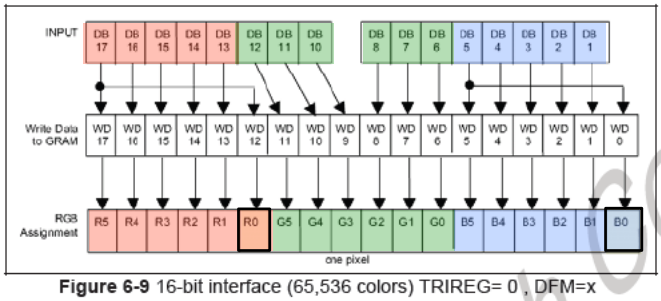


Figure 6-9 16-bit interface (65,536 colors) TRIREG= 0 , DFM=x

6.2.5. Entry Mode (R03h)

R/W	RS	CB15	CB14	CB13	CB12	CB11	CB10	CB9	CB8	CB7	CB6	CB5	CB4	CB3	CB2	CB1	CB0
W	1	TRIREG	DFM	0	BGR	0	0	0	0	ORG	0	I/D1	I/D0	AM	0	0	0

Controlador TFT SPFD5408B

- Acceso mediante un bus paralelo de 16 bits
  - Modo 256K colores → 18 bits por pixel
    - Cada pixel necesita dos transferencias de 16 bits
    - Hay dos modos de distribuir la información en las dos palabras

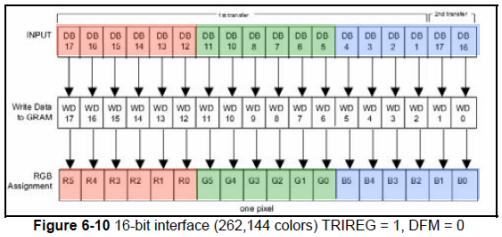


Figure 6-10 16-bit interface (262,144 colors) TRIREG = 1, DFM = 0

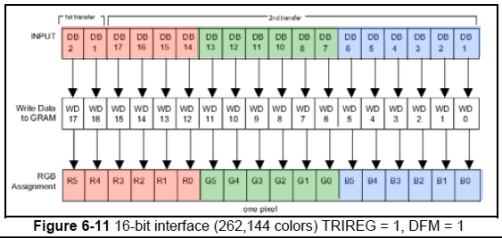
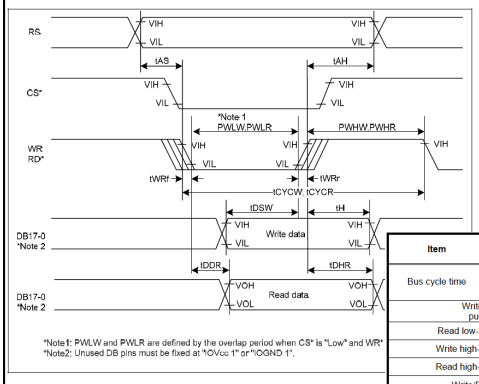


Figure 6-11 16-bit interface (262,144 colors) TRIREG = 1, DFM = 1

Conexión Controlador TFT SPFD5408B

- Temporización de lectura y escritura del bus paralelo

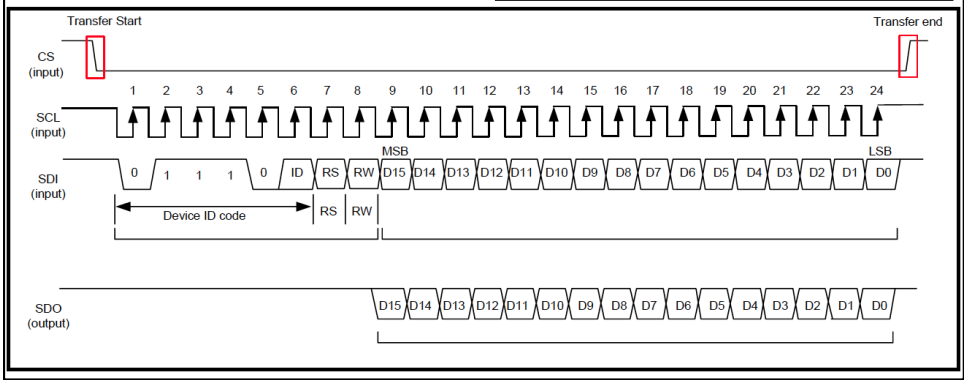


Item		Symbol	Unit	Min.	Typ.	Max.
Bus cycle time	Write	tCYCW	ns	125	-	-
	Read	tCYCR	ns	450	-	-
Write low-level pulse width		PWLW	ns	45	-	-
Read low-level pulse width		PWLR	ns	170	-	-
Write high-level pulse width		PWHW	ns	70	-	-
Read high-level pulse width		PWHR	ns	250	-	-
Write/Read rise/ fall time		tWRr, WRf	ns	-	-	25
Setup time	Write (RS to CS*,WR*)	tAS	ns	0	-	-
	Read (RS to CS*, RD*)		ns	10	-	-
Address Hold Time		tAH	ns	2	-	-
Write data setup time		tDSW	ns	25	-	-
Write data hold time		tH	ns	10	-	-
Read data delay time		tDDR	ns	-	-	150
Read data hold time		tDHR	ns	5	-	-

Controlador TFT SPFD5408B

- Acceso mediante SPI
  - Transferencia de 24 bits (3 bytes)
    - (6 bits) Identificador de dispositivo
    - Bit RS
    - Bit RW
    - Dato de 16 bits

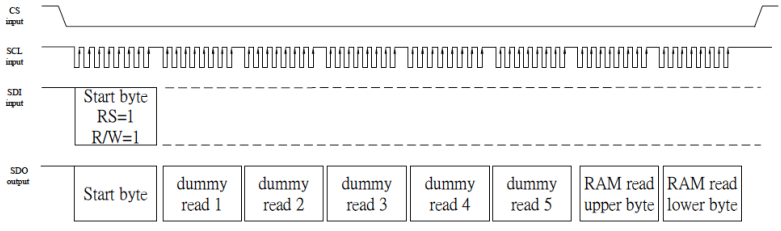
RS	R/W	Function
0	0	Set an index register
0	1	Read a status
1	0	Write an instruction or RAM data
1	1	Read an instruction or RAM data



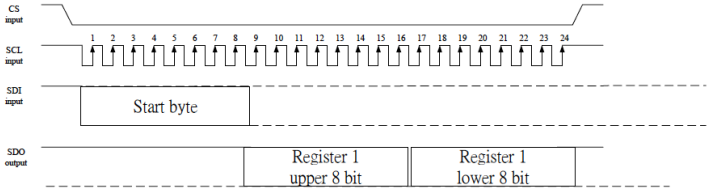
Controlador TFT SPFD5408B

- Acceso mediante SPI
  - La lectura de los registros está disponible en el segundo byte
  - La lectura de la RAM está disponible en el séptimo byte

RAM resd data transfer



Status / register read transmission



Controlador TFT SPFD5408B

- Acceso mediante SPI
  - Formato de los bits de Datos

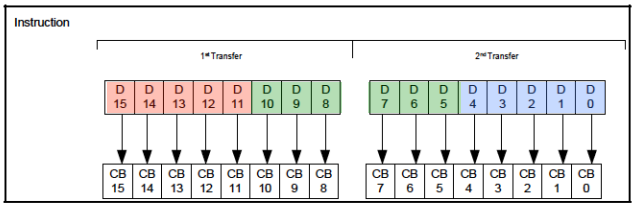


Figure 8-10

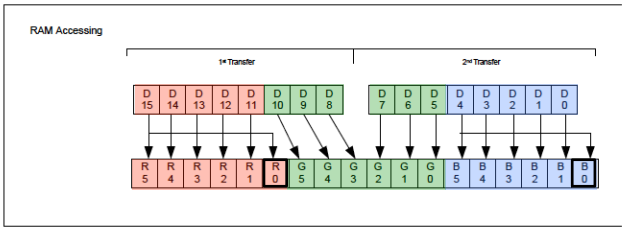


Figure 8-11

Controlador TFT SPFD5408B

- Acceso mediante SPI
  - Escritura y Lectura de un Byte

```
unsigned char LPC17xx_SPI_SendRecvByte (unsigned char byte_s)
{
  /* wait for current SSP activity complete */
  while (SSP_GetStatus(LPC_SSP0, SSP_STAT_BUSY) ==  SET);

  SSP_SendData(LPC_SSP0, (unsigned short) byte_s);

  while (SSP_GetStatus(LPC_SSP0, SSP_STAT_RXFIFO_NOTEMPTY) == RESET);
  return (SSP_ReceiveData(LPC_SSP0));
}
```

Controlador TFT SPFD5408B

- Acceso mediante SPI
  - Escritura del "Index Register" (RS=0, RW=0)
    - El "Index Register" es de 8 bits

```
void LCD_WriteIndex(unsigned char index)
{
  SPI_CS_LOW;

       /* SPI write data */
  LPC17xx_SPI_SendRecvByte(SPI_START | SPI_WR | SPI_INDEX);   /* Write : RS = 0, RW = 0       */
  LPC17xx_SPI_SendRecvByte(0);
  LPC17xx_SPI_SendRecvByte(index);

  SPI_CS_HIGH;
}
```

  - Escritura del un datos donde apunta "Index Register" (RS=1, RW=0)
    - El contenido de los registros es de 16 bits

```
void LCD_WriteData( unsigned short data)
{
  SPI_CS_LOW;

  LPC17xx_SPI_SendRecvByte(SPI_START | SPI_WR | SPI_DATA);    /* Write : RS = 1, RW = 0       */
  LPC17xx_SPI_SendRecvByte((data >>   8));                    /* Write D8..D15                */
  LPC17xx_SPI_SendRecvByte((data & 0xFF));                    /* Write D0..D7                 */

  SPI_CS_HIGH;
}
```

Controlador TFT SPFD5408B

- Acceso mediante SPI
  - Escritura en un registro

```
void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue)
{
  /* Write 16-bit Index, then Write Reg */
  LCD_WriteIndex(LCD_Reg);
  /* Write 16-bit Reg */
  LCD_WriteData(LCD_RegValue);
}
```

  - Lectura de un registro (RS=1, RW=0)
    - Los registros son de 16 bits

```
unsigned short LCD_ReadData(void)
{
  unsigned short value;

  SPI_CS_LOW;

  LPC17xx_SPI_SendRecvByte(SPI_START | SPI_RD | SPI_DATA);    /* Read: RS = 1, RW = 1         */
  LPC17xx_SPI_SendRecvByte(0);                                /* Dummy read 1                 */
  value   = LPC17xx_SPI_SendRecvByte(0);                      /* Read D8..D15                 */
  value <<= 8;
  value  |= LPC17xx_SPI_SendRecvByte(0);                      /* Read D0..D7                  */

  SPI_CS_HIGH;

  return value;
}
```