Embedded Application

COMP595EA
Basic Design using RTOS

Embedded Design

- Has as many exceptions as it does rules
  - ingenuity and experience are important factors for the success of embedded designs.
- More questions need to be addressed:
  - What must the system do?
  - How fast must it do it?
  - How should failures be handled?
  - How critical are timing issues?

Effort Management

- Timing issues?
  - Can we tolerate decent timing 99% of the time?
  - Or do we need rapid response 100% of the time?
- Important because it may not be worth the heroic design efforts to achieve the latter is it is not necessary.

RTOS types

- Hard real-time systems
  - Systems that require that absolute deadlines must be met.
- Soft real-time systems
  - Allow some fudge in deadlines.
- Both benefit from similar design techniques.

Extensive Knowledge

- To design effectively requires knowledge of the hardware.
- Knowing which computations will take long enough to affect other deadlines is a necessary design consideration
- Experience and experimentation go a long way towards guiding designs in these areas.

Classic Resources

- Debuggers, Interactive development environments, compilers
  - These and all other classic tools to assist in application software designs are just as useful in assisting with embedded designs
- The techniques for embedded design are in addition to these classical tools.
  - “Learn to love your debugger and editor” - J.W.

Testing and Debugging

- Many application software designs view testing and debugging as a nuisance left until the last minute.
- When designing embedded applications make sure that your designs include testing and debugging as part of the overall design, process and methodology.
  - You do not want to explain to your boss that you neglected to fully test the firmware upload routine on the rover.

Hurry Up and Wait

- Most embedded systems actually have nothing to do until the passage of time generates an external event upon which to act.
  - Interrupts tend to be the driving force
  - Tasks spend most of their time blocked; waiting for an interrupt routine to unblock them.
  - Each interrupt can create a cascade of signals and task activity.

Write Short Interrupt Routines

- For two reasons:
  - Even the lowest priority interrupt routine is executed in preference to the highest priority task.
    - Long interrupt routines translate directly to slow responding task code.
  - Interrupt routines tend to be more bug-prone and harder to debug than task code.
- Interrupt routines should handle immediate activities but should signal task code to do the rest

Interrupt Routine Trade-off

- Although interrupt routines should be short they shouldn't be too short.
  - Interrupt routines usually cause a call to some RTOS routines to pass messages or activate semaphores.
  - RTOS routines are generally costly to execute.
  - Exceedingly small interrupt routines can therefor consume too much computational power by causing too many RTOS calls.
- Try for the balance of this trade-off

How many tasks should I have?

- Create a large number of tasks has the following advantages:
  - more tasks yields better control of response times
  - more tasks allows a design to be more modular
  - more tasks allow for data to be encapsulated more effectively.

- The disadvantages are:
  - more tasks are likely to have more shared data
  - more tasks require more message passing
  - more tasks will require more memory (stacks)
  - more tasks require more context switching
  - more tasks mean more RTOS routine calls
    - RTOS routines don't do anything the customer cares about
    - Systems run faster WITHOUT RTOS calls

Moral of the story

- Other things being equal-
  - Use as few tasks as you can get away with.
  - Add more tasks only for clear reasons.

Reasons to Add Tasks

- To establish priorities for activities
- To encapsulate hardware shared by other tasks
- To encapsulate data structures shared by multiple tasks.
- Adding tasks can simplify the design
- Separate tasks make sense for responding differently to different stimulus.

State Machines

- Tasks in RTOS designs are often modeled as state machines.

Creating and Deleting Tasks

- Some RTOS will allow you to create and destroy tasks during execution.
- This should be avoided.
  - create/destroy RTOS routines are typically expensive
  - It can be difficult to destroy tasks without leaving things in a mess.
- Create all the tasks at startup. Tasks that want to be “Destroyed” should simply block and never wake up.

Time Slicing

- Many RTOS will time-slice between tasks of equal priority.
- This should also be avoided and turned off if possible
  - context switches waste time.
  - Fair is not an issue in embedded systems
    - tasks are typically not of equal urgency
    - If they are you typically don't care which finishes first.
- if you can't pinpoint a reason why, don't use it.

Encapsulating Semaphores

- The act of putting Give and Get semaphore routines throughout your code causes an increased probability of mistakes being made.
- Encapsulate Semaphore routines so that tasks that want to execute some behavior involving the semaphore make a call to a single routine
  - The Get and Give calls are inside this encapsulated routine.

Encapsulating Queues

- Similarly mistakes dealing with queues can be reduced by encapsulation.
  - queues typically have a protocol or format associated with.
  - multiple message passing calls increase the probability of a communication mistake.
  - encapsulating such calls into functions that take specific arguments reduces these errors.

Encapsulating Consideration

- When encapsulating semaphores or queues:
  - routines are shared by many tasks
  - Care must be taken to avoid shared data problems
  - routines must be reentrant.

Hard Real-Time Scheduling

- Theory exists for guaranteeing absolute deadlines
    - Based on n tasks, worst case task times Cn, and Tn units of time.
    - To more complex analysis depending on jitter and deadlines
- Having predictable execution times is almost as important as being fast.
- It is important to write routines that always execute in the same amount of time or have clear worst case times.

Saving Memory Space

- Embedded systems have limited memory
- ROM (code) and RAM(data) are separate.
  - Packing data tightly and neatly usually consumes more code space.
- It is often necessary to save space to get everything to fit.
- How much space do you need?

Two Methods

- Two methods for determining space requirements:
  - trace the deepest functional calls possible. Add all the local variables, parameters and interrupt routine needs.
    - accurate but very difficult to carry out.
  - Fill stacks with a known pattern. Run the program for a while see what got used
    - easy but not so accurate. Might have missed the worst case.

Methods for Reducing Space

- Make sure you aren't using two functions to do the same thing
- Check that development tools aren't sabotaging you but pulling in large libraries or other garbage.
- Configure the RTOS to contain only those functions that you need.
- Look at the assembly language to verify the compiler is producing optimized code
- Consider using static variables instead of locals.
- Consider using char instead of int
- Write it in assembly (ugh)

Saving Power

- Many embedded systems run on battery power
- Battery life is a big issue
- method for conserving power including turning off components, including the microprocessor
- Most embedded microprocessors have at least one power saving mode

Turn off the Microprocessor

- Some systems conserve power by turning of the microprocessor completely.
  - Saves a lot of power
  - Requires the hardware engineer to provide some means for resetting the microprocessor
  - code starts from the beginning when the microprocessor is reset.
    - state can be stored and recovered in static RAM

On-board peripherals

- Some systems conserve power by halting execution instruction but leave on-board peripherals (timers, interrupts, etc) running
  - Interrupt causes execution to resume
  - code continues where it left off.
  - Saves less power
  - Doesn't require special hardware

Turn it all off

- Some devices turn everything off
  - Saves the most power
  - requires hardware
  - Must be manually turned on again
  - not suitable for some situations