



Moving up to an 32-bit processor

Your First Operating System

Quadros[™]
Systems Inc.



Making a Move

- **There is a strong migration trend from 8- and 16-bit processors to the 16/32-bit ARM architectures**
 - ◆ Easier use of available address space
 - ◆ More on-chip integration of I/O controllers
 - ◆ Faster processing better able to handle increasingly complex applications
- **Planning such a migration is full of complex issues**
- **One of those issues is the selection of an operating system that best fits the application's computational requirements**



Where Are You Now?

Quadros[™]
Systems Inc.



What is Driving the Move?

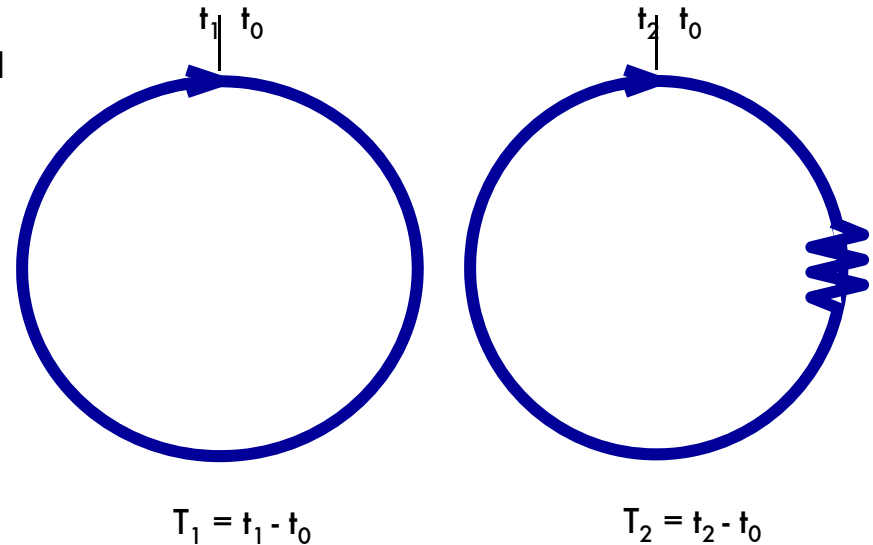
- **Using an 8- or 16-bit microcontroller**
 - ◆ Availability of current processor
 - ◆ Need more memory
 - ◆ Need more I/O ports
 - ◆ Not enough on-chip integration
- **Feature creep — not creeping but rampant**
 - ◆ Marketing wants more features to keep up or stay ahead of competition
 - ◆ Connectivity demands (Ethernet, USB, WiFi)
 - ◆ Standards requirements
- **Application complexity continues to grow**
 - ◆ Need a solid roadmap for future growth that older processors just can't give

Structure of Your Application

- **Software architecture of current application may be insufficient to meet needs of growth**
- **Application code structure in many 8- and 16-bit applications is usually one of three forms:**
 - ◆ Super Loop and Round-Robin
 - ◆ Simple scheduler
 - ◆ Multitasking kernel or RTOS
- **Each structure presents a set of issues for a successful migration**
 - ◆ Structure is independent of processor architecture
 - ◆ Mitigation of risk and cost
 - ◆ Preservation of investment

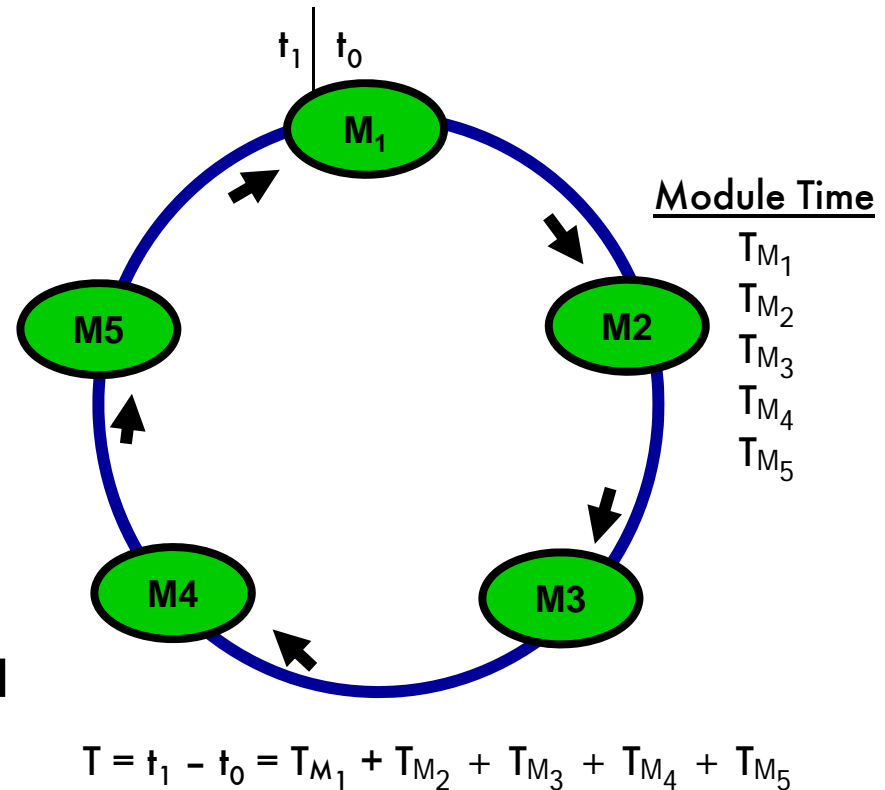
Super Loop

- **Frequently found in small systems**
 - ◆ Remember: about half of embedded systems use no operating system
- **Ad hoc design**
 - ◆ No formal structure
 - ◆ It's not even a Scheduler
- **Single task, executes in one big loop**
- **Code often becomes convoluted (spaghetti code)**
- **Fine for simple applications**



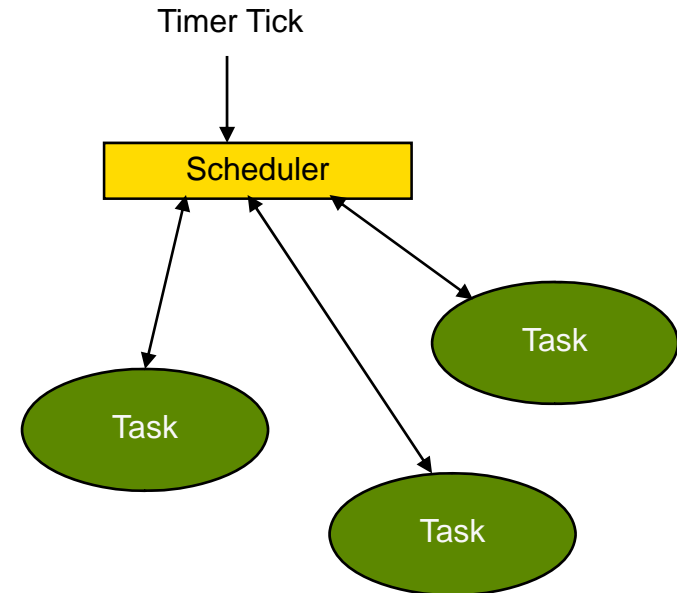
Round Robin

- Widely used; simple to create
- Single task, modules execute sequentially, with or without time limit, in one loop
- Application has close to 100% of CPU
- Maintainability is improved with modular design.
- Unstable timing between successive passes of loop
- Event response time non-predictable.



Simple Schedulers

- Found in many forms
- Usually tailored to a particular application
- Application code (tasks) formed as functions
 - ◆ Equal priority
- Timer tick is often basic scheduling element
- Tasks called by a simple scheduler
 - ◆ Round-Robin
 - ◆ Time limited execution





Moving Up



Benefits of 32-bit Architectures

- **Full 32-bit addressing**
 - ◆ Eliminates need for multiple memory models in compilers
 - ◆ Makes code design and debugging easier
- **High levels of integration with popular device controllers**
 - ◆ USB host/device/OTG
 - ◆ Ethernet
 - ◆ LCD
 - ◆ CAN
- **Many low-cost options**
 - ◆ ARM (Atmel, ST, NXP et. Al)
 - ◆ Freescale ColdFire
 - ◆ ADI Blackfin

What to Expect with the Move

- **A multitasking RTOS is probably in your future**
 - ◆ Prioritized, preemptive multitasking
 - ◆ Cooperative multitasking
- **Dealing with legacy application code**
 - ◆ Legacy code is always a factor when one contemplates a processor change or system upgrade
 - ◆ It could be C, but there's a lot of assembly language out there in the 8- and 16-bit world
 - ◆ Issue #1 is how to preserve investment already made in legacy code
- **Which RTOS model is best?**
 - ◆ The structure of your application will determine the RTOS requirements

Reasons to use an RTOS

- **Improve programmer productivity**
- **Shorten development time**
- **Improve system performance**
- **Optimize use of system resources**
- **Enhance product reliability, maintainability and quality**
- **Promote product evolution**



RTOS Models

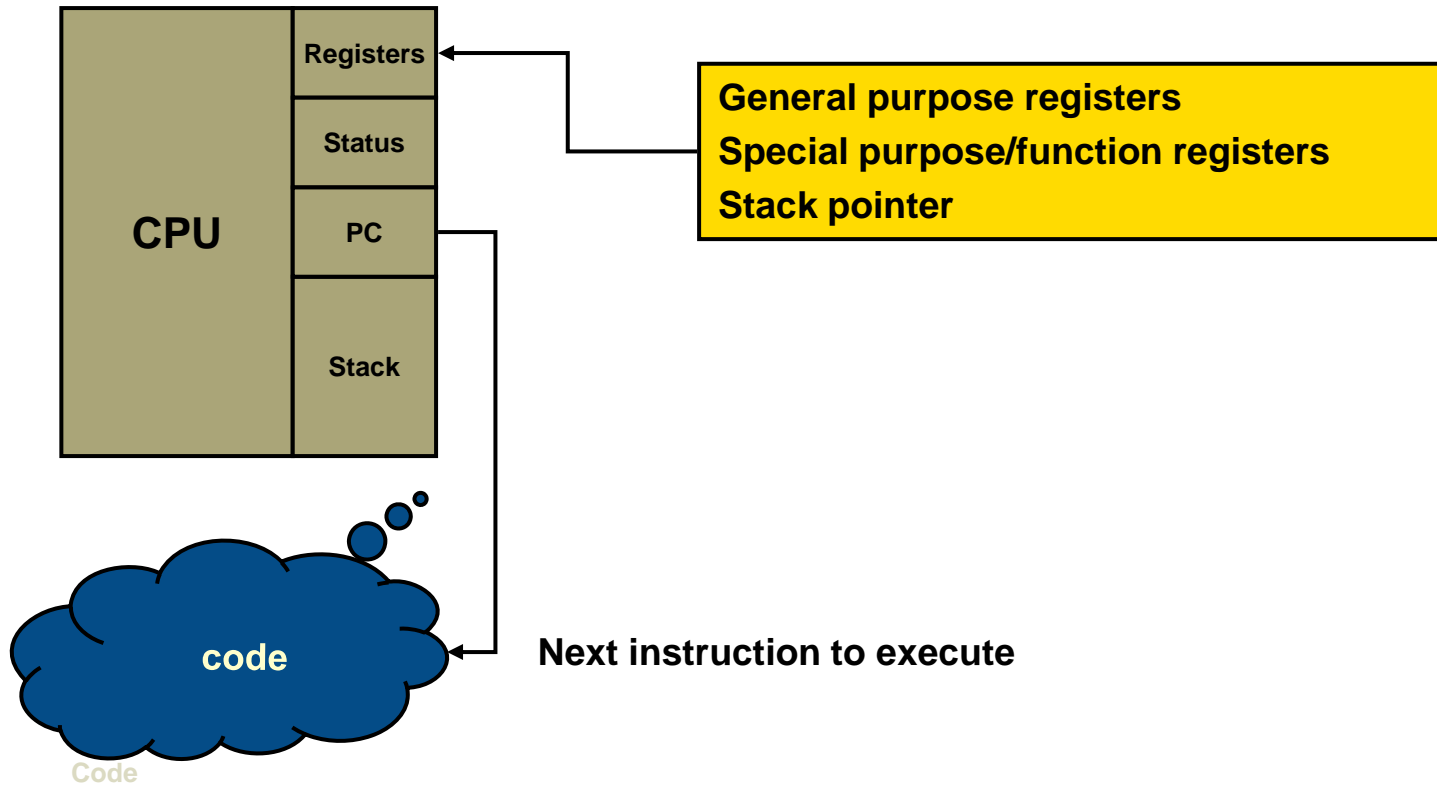


Multitasking RTOS Models

- **Single Stack (all code shares one stack)**
 - ◆ Cooperative multitasking
 - ◆ Cyclic Scheduler
- **Multiple Stack (each task has its own stack)**
 - ◆ Preemptive multitasking
 - ◆ Round robin multitasking
 - ◆ Time-sliced multitasking
- **Dual Mode**
 - ◆ Combination of Single and Multiple Stack kernels

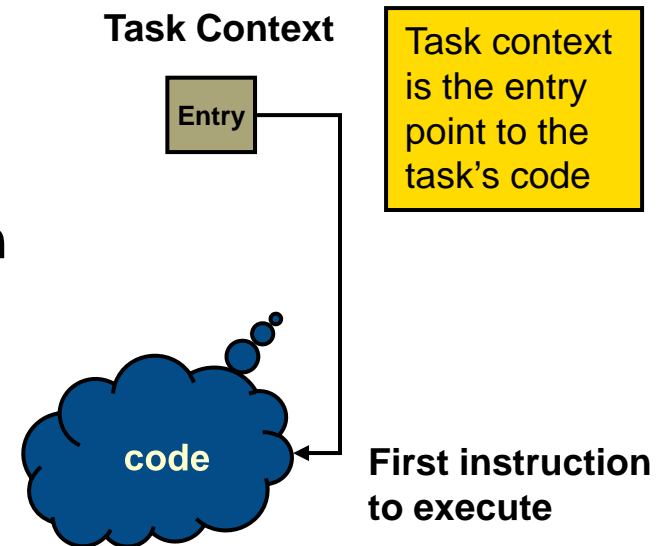
The Processor Model

Physical Hardware



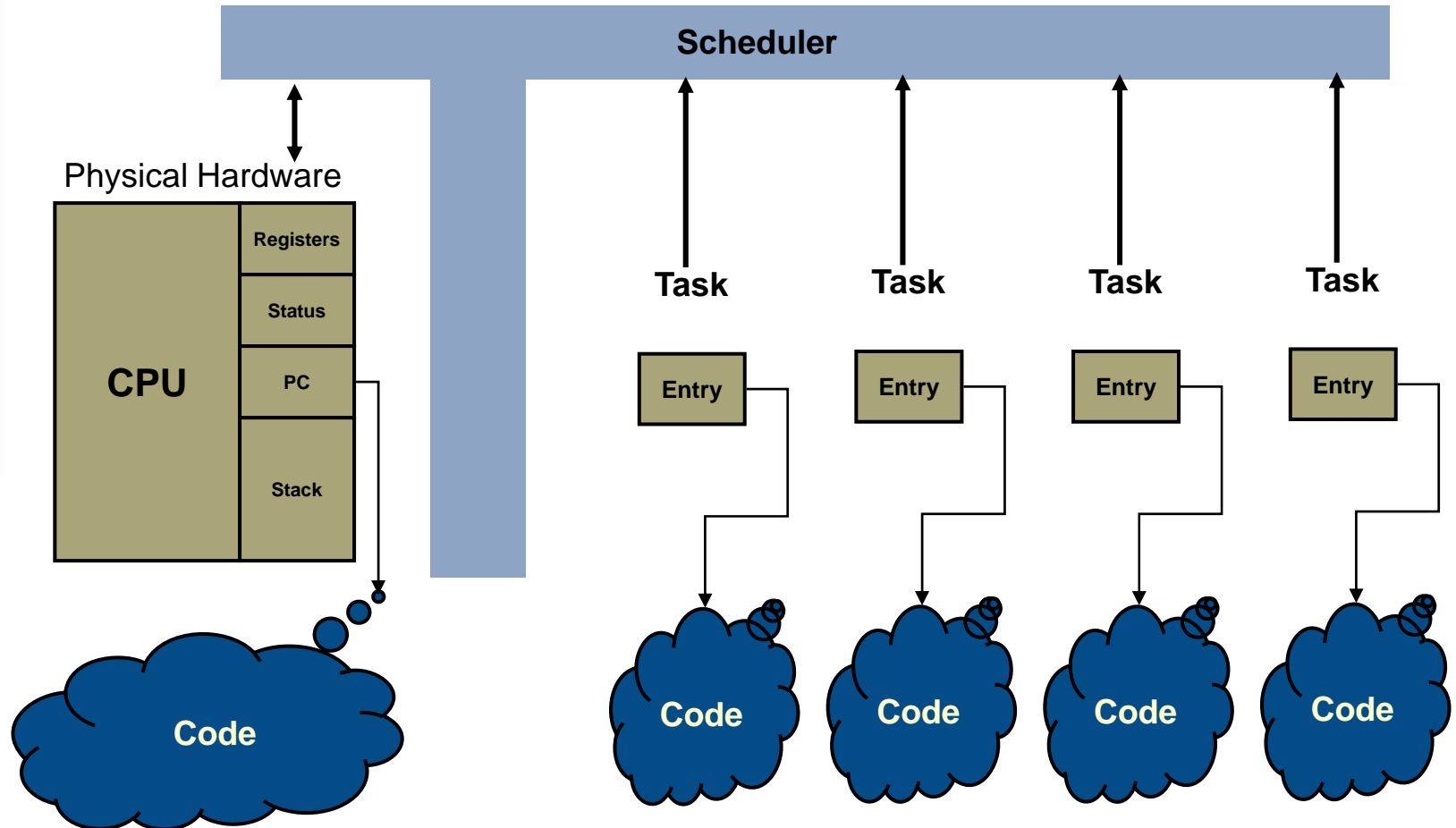
Single Stack Multitasking

- All execution entities (Tasks, kernel and ISRs) use same stack
- Scheduler selects next task only when the current task completes
- Tasks have minimal context
- Tasks start and run to completion without blocking
- Simple design
 - ◆ Very small code footprint, small RAM use
 - ◆ Lack of context makes it very fast when switching tasks
- Easily adapted to state machine software design or to data flow processing such as DSP algorithms



Single Stack Multitasking

Cooperative or Cyclic Multitasking Kernel



Single Stack Scheduler

- **Runs on points of rescheduling**
 - ◆ When current task completes
- **Task flow controlled via a Ready Table**
 - ◆ Indicates which tasks are ready to run
 - ◆ Task priority indicated by position in Ready Table
 - ◆ Scheduler uses Ready Table to determine next task to run
 - ◆ Gives control to task by calling it as a function
 - ◆ Regains control when current task completes (returns)
- **Scheduler can implement a scheduling policy**
 - ◆ Most common is Round Robin
 - ◆ Preemptive scheduling is possible but more complex
- **Background processes run when all application code is complete (empty Ready Table)**

Single Stack Multitasking Summary

- **Advantages:**

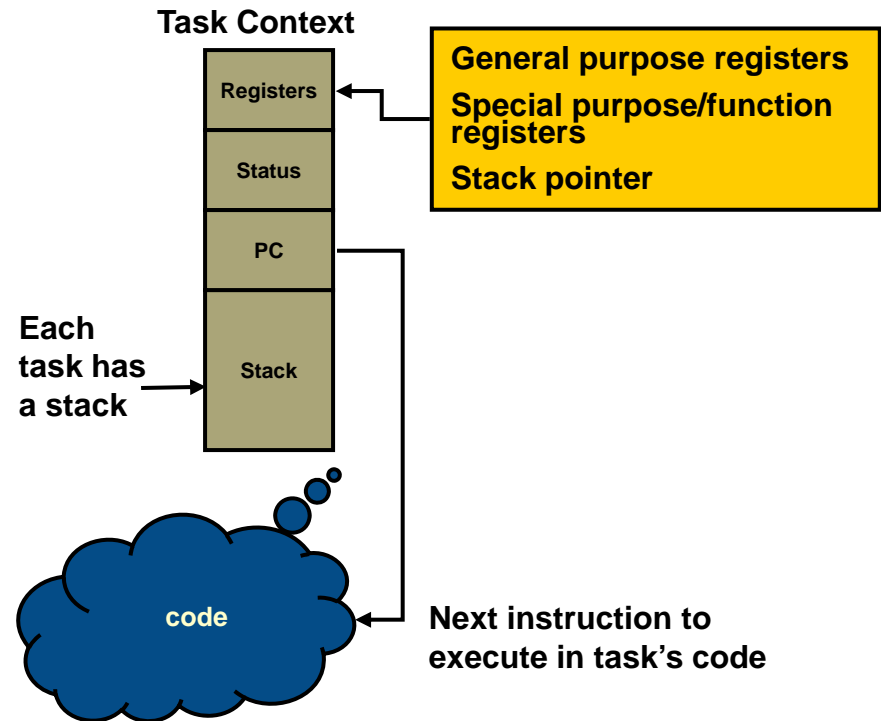
- ◆ Low overhead
- ◆ Small and fast
- ◆ No context to save or restore during a normal context switch
- ◆ Single stack minimizes RAM usage
- ◆ Ideal for State Machine software architecture

- **Drawbacks:**

- ◆ Tasks can't wait for events, making synchronization complex
- ◆ No preemption between tasks in the robin
- ◆ Difficult for communication protocol stacks
- ◆ Background operation often performed ad-hoc or without formal structure

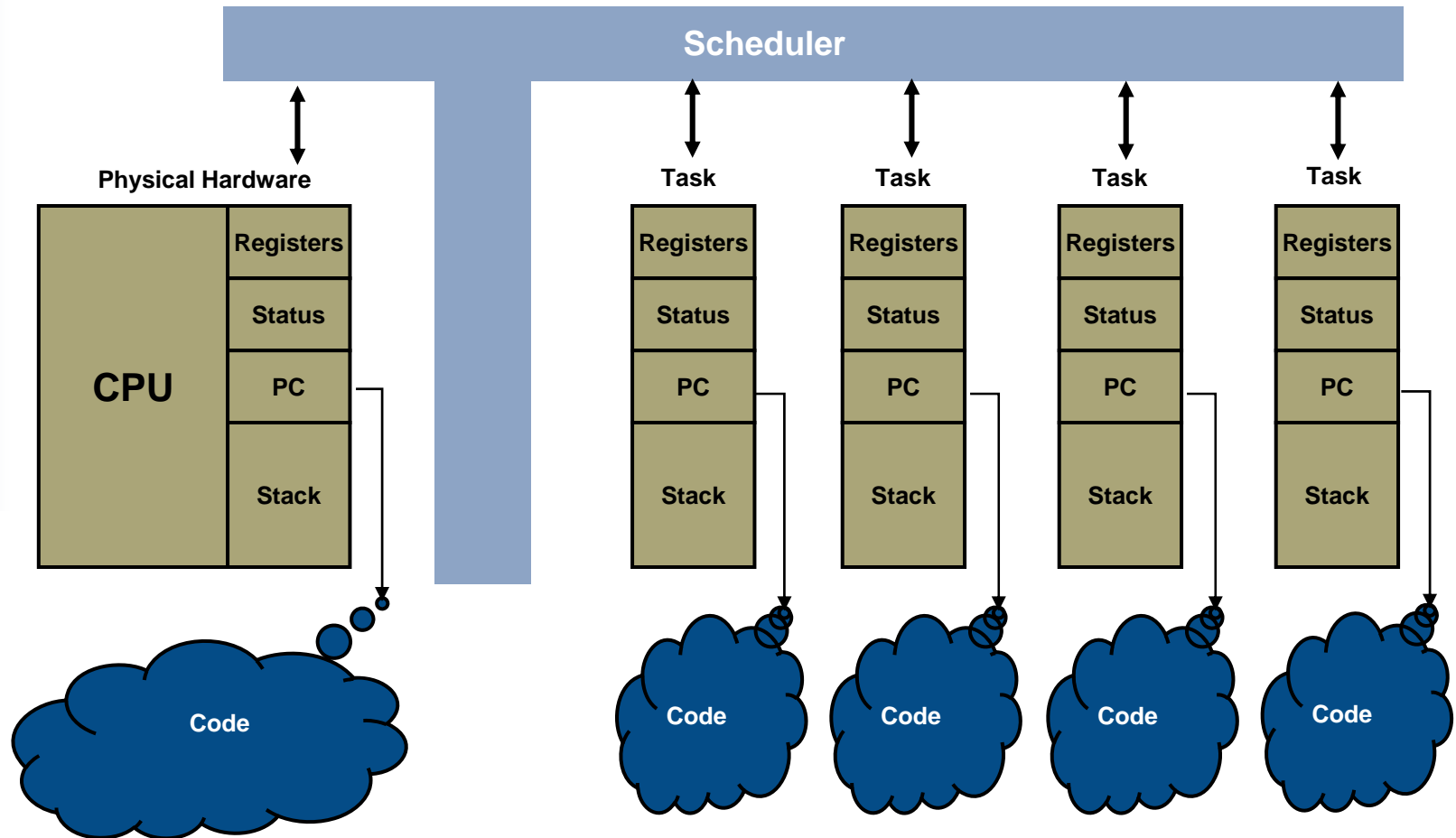
Multiple Stack Multitasking

- Each task has its own stack
- Tasks can wait for events or otherwise block
- Tasks are interruptible and may be preempted
- Scheduler policy selects which task gets CPU control
 - ◆ Scheduler runs as a result of a kernel service or interrupt
- Tasks context is maintained – even when task is not running



Multiple Stack Multitasking

Multiple Stack Multitasking Kernel



Multiple Stack Multitasking – Scheduler

- **Scheduling policy determines the nature of the kernel design**
 - ◆ Influences the type of application for which the kernel is best suited
- **Scheduler runs as the result of some action**
 - ◆ A task can make a kernel call that causes it to block
 - ◆ An interrupt can cause a task to be made ready
- **Scheduler policy identifies task that is ready to receive CPU control, according to a policy**
 - ◆ Preemptive
 - ◆ Round Robin
 - ◆ Time-Sliced

Dual Mode Multitasking

- **Combination of single stack and multi-stack multitasking models into a unified RTOS**
- **Single stack RTOS runs application code that uses data flow or state machine designs**
 - ◆ Helps preserve legacy code
 - ◆ Can yield better performance than running the code under a multi-stack RTOS
- **Multi-stack RTOS with prioritized, preemptive scheduling**
 - ◆ Easily handles sporadic scheduling usually found in control applications
 - ◆ Ideal for the connectivity stacks and related applications

Application Code Structure Summary

- **Super Loops, Round Robin Schedulers and Simple Schedulers**
 - ◆ Commonly used
 - ◆ No structure, hard to maintain
 - ◆ Can't handle time and event synchronization in a deterministic manner
 - ◆ Ideal for State Machine software architecture
 - ◆ Not good fits for communication protocol stacks
- **Multitasking kernels**
 - ◆ Good fit for soft real-time requirements
 - ◆ Best fit for hard real-time requirements
 - ◆ Structured organization of application
 - ◆ Promotes ease of maintenance of application code
 - ◆ Ideal for communication protocol stacks



RTOS Purchase Considerations



Price vs. Cost

- **Price (money paid to supplier) includes:**
 - ◆ Development kit license fees
 - ◆ Production licenses
 - ◆ Technical support
 - ◆ Ongoing maintenance
 - ◆ Royalties
- **Cost (includes price)**
 - ◆ Time (cost of labor including overhead)
 - ◆ Lost opportunity
 - ◆ Integration
 - ◆ Debugging

Make or Buy the RTOS?

- **“Make” is no longer a viable choice except when there is no product to “Buy”**
- **Commercial RTOS packages readily available for 32-bit processors targeted at the migration from 8- and 16-bit CPU applications**
 - ◆ Chip support packages and integrated software available with RTOS
 - USB stacks and controller drivers
 - TCP/IP stacks and Ethernet drivers
 - GUI development tools (HMI) and LCD drivers
 - CAN stacks and drivers
- **Developer can focus on the application**

Getting Started with an RTOS

- **How can I learn to use an RTOS**
 - ◆ Training?
 - ◆ Courses?
 - ◆ Vendor-supplied development/configuration tools



Conclusion



Summary

- **System complexity and connectivity requirements move applications toward 16/32-bit platforms**
- **Applications of all kinds can benefit from an operating system**
 - ◆ Manage connectivity
 - ◆ Speed development
- **“Make” is no longer a viable option**
 - ◆ Off-the-shelf commercial RTOSes are readily available
 - ◆ System complexity demands more complex OS
- **It is easy to get started**
- **There is an RTOS licensing model for every need**