ECE3411 – Fall 2016

Lecture 6b.

Real Time Operating Systems Cont'd
Bus and Communication Interfaces

Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

Copied from Lecture 6a and Lecture 7b, ECE3411 – Fall 2015,
by Marten van Dijk and Syed Kamran Haider

Realtime Kernel Design Strategies

- Polled Loop Systems
- Interrupt Driven Systems
- Multi-Tasking
- Foreground/Background Systems

Polled Loops

- Simplest RT kernel
- A single and repetitive instruction tests a flag that indicates whether or not an event has occurred
  - Examples: Non-blocking LCD instructions, Non-blocking “get string” over the UART channel
- No intertask communication or scheduling needed. Only single tasks exist
- Excellent for handling high-speed data channels, especially when
  - Events occur at widely spaced intervals and
  - Processor is dedicated to handling the data channel

Examples interrupt-driven system

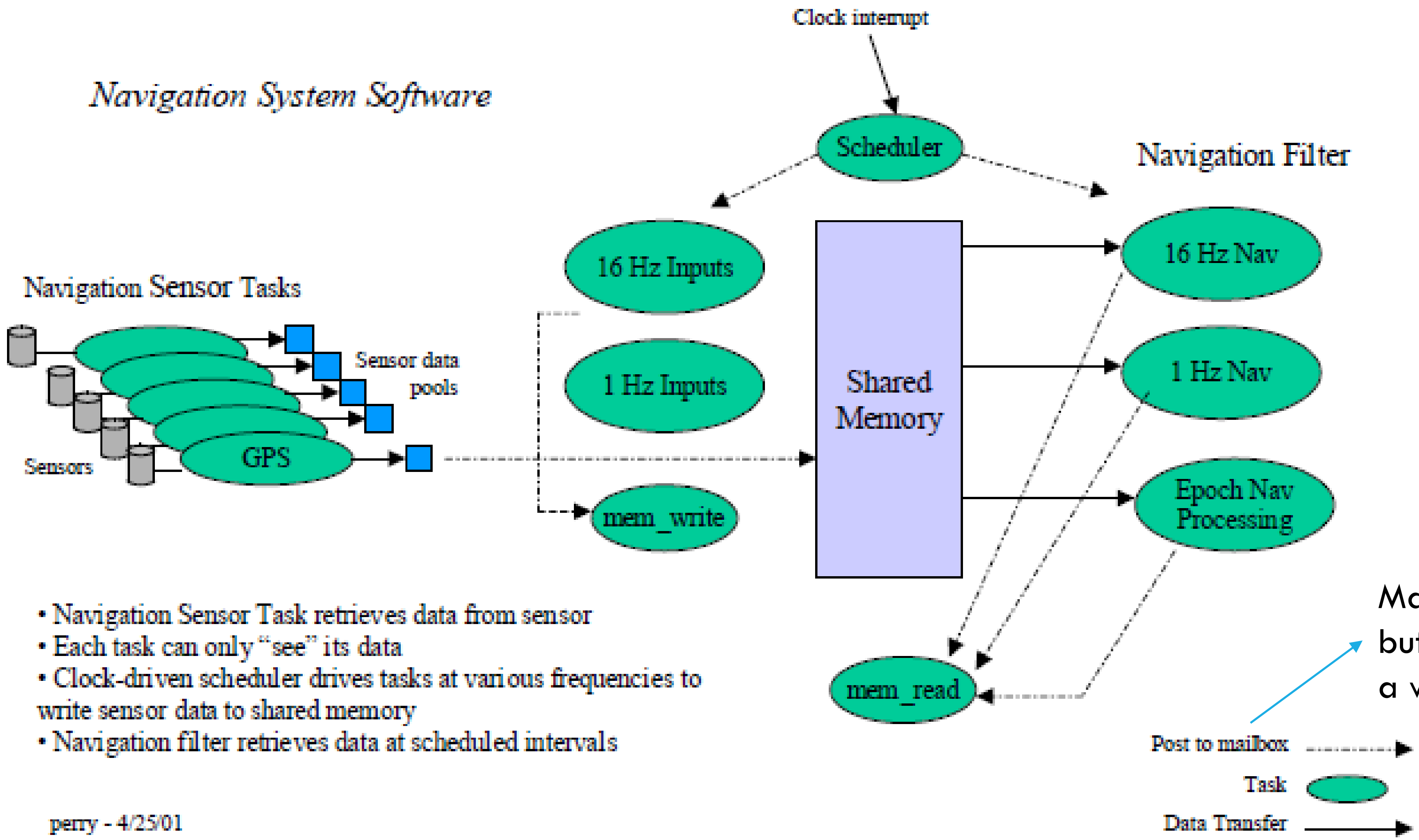
Interrupt Driven Software Examples

- IFF receiver sees a threat and interrupts an aircraft mission computer to sound a cockpit alarm
- Inertial Navigation Unit data (Δ velocities in north/east/up coordinates) is available at 32 Hz and interrupts the navigation software with new data when it is ready
- Sonar contact data interrupts signal processing software when new data is available
- Low altitude indicator triggers a fly-up command for a pilot

Multitasking

- Separate tasks that share one processor (or processors)
- Each task executes within its own context
  - Owns processor
  - Sees its own variables
  - May be interrupted
- Tasks may interact to execute as a whole program

Example



- Navigation Sensor Task retrieves data from sensor
- Each task can only "see" its data
- Clock-driven scheduler drives tasks at various frequencies to write sensor data to shared memory
- Navigation filter retrieves data at scheduled intervals

Mailbox is a circular buffer with a read and a write pointer

Context Switching

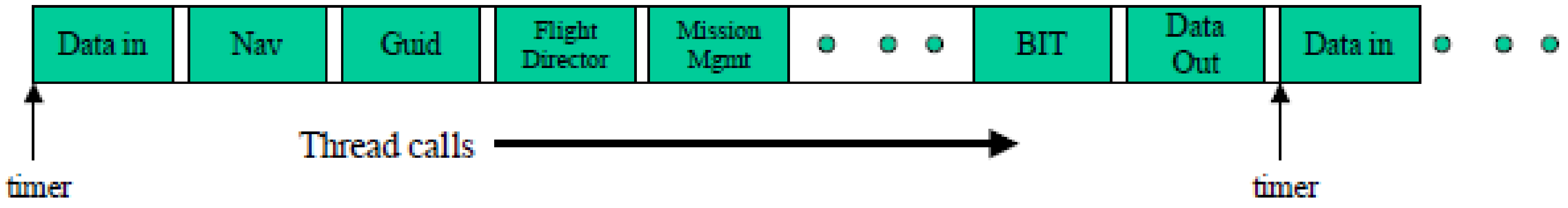
- When the CPU switches from one task to running another, its is said to have *switched contexts*
- Save the minimum needed to restore the interrupted process
  - Contents of registers
  - Contents of the program counter
  - Contents of coprocessor registers (if applicable)
  - Memory page registers
  - Memory-mapped I/O
  - Special variables
- During context switching, interrupts are often disabled
- Real time systems require minimal times for context switches

Multitasking

- How do many tasks share the same CPU?
  - Cyclic executive systems
  - Round robin systems
  - Pre-emptive priority systems

Cyclic Executive Systems

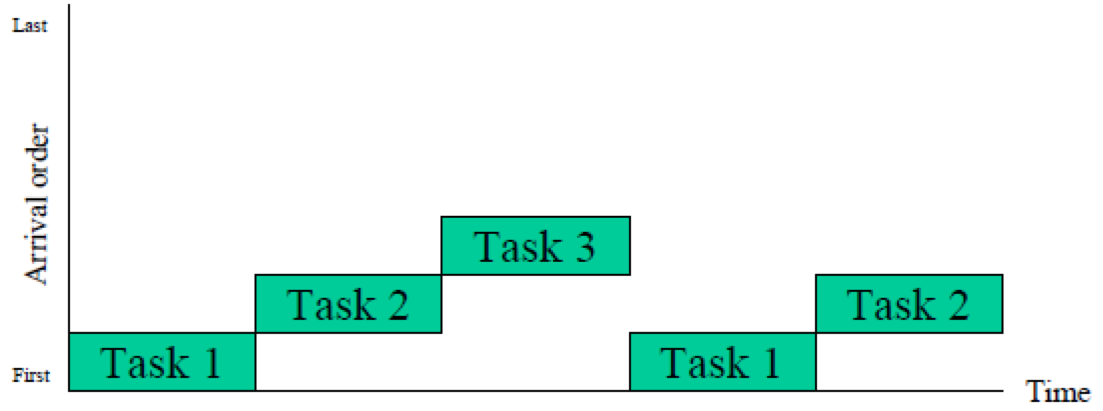
- Calls to statically ordered threads



- Pros
  - Easy to implement (used extensively in complex safety critical systems)
- Cons
  - Not efficient in overall usage of CPU processing
  - Does not provide optimal response time

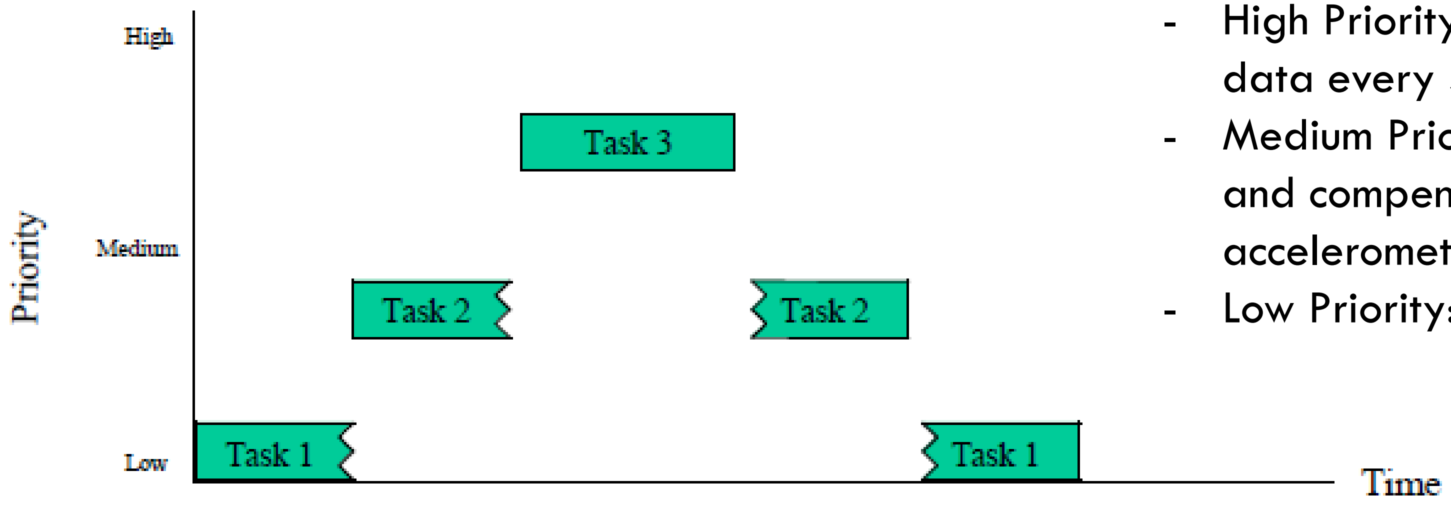
Round Robin Systems

- Several processes execute sequentially to completion
- Often in conjunction with a cyclic executive
- Each task is assigned a fixed time slice
- Fixed rate clock initiates an interrupt at a rate corresponding to the time slice
  - Task executes until it completes or its execution time expires
  - Context saved if task does not complete
- Just like our task-based programming without fixed times slices per task



Pre-emptive Priority Systems

- Higher priority task can preempt a lower priority task if it interrupts the lower-priority task
- Priorities assigned to each interrupt are based upon the urgency of the task associated with the interrupt
- Priorities can be fixed or dynamic
  - Round Robin Systems - Preemptive Scheduling of 3 Tasks



Example: Aircraft Navigation System

- High Priority: Task that checks accelerometer data every 5ms
- Medium Priority: Task that collects gyro data and compensates this data and the accelerometer data every 40ms
- Low Priority: Display update, Built-in-Test (BIT)

Problems Multitasking

- High priority tasks hog resources and starve low priority tasks
- Low priority tasks share a resource with high priority tasks and block high priority tasks

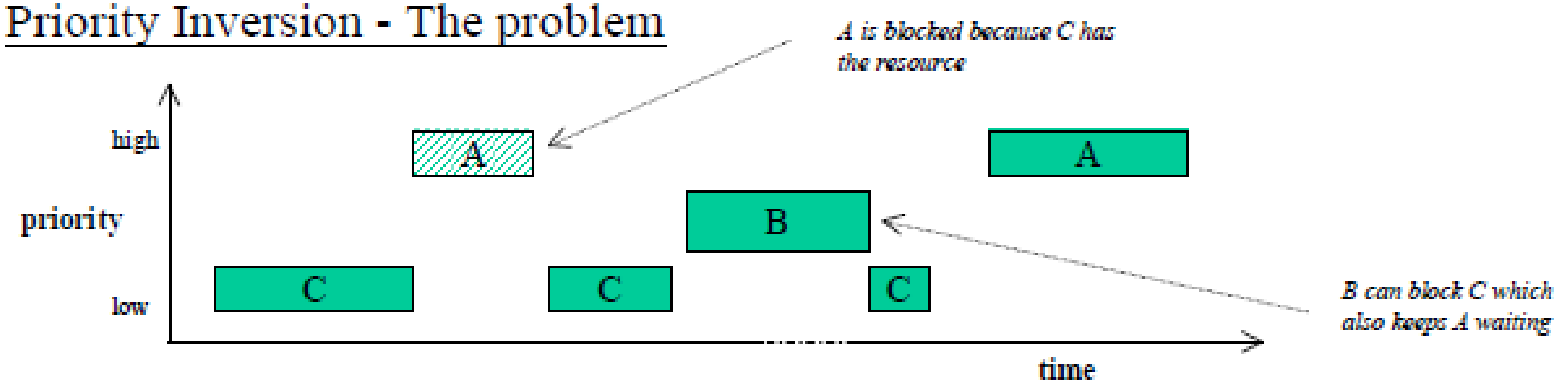
- How does a RTOS deal with some of these issues?
  - Rate Monotonic Systems (higher execution frequency = higher priority)
  - Priority Inheritance

Priority Inversion / Priority Inheritance

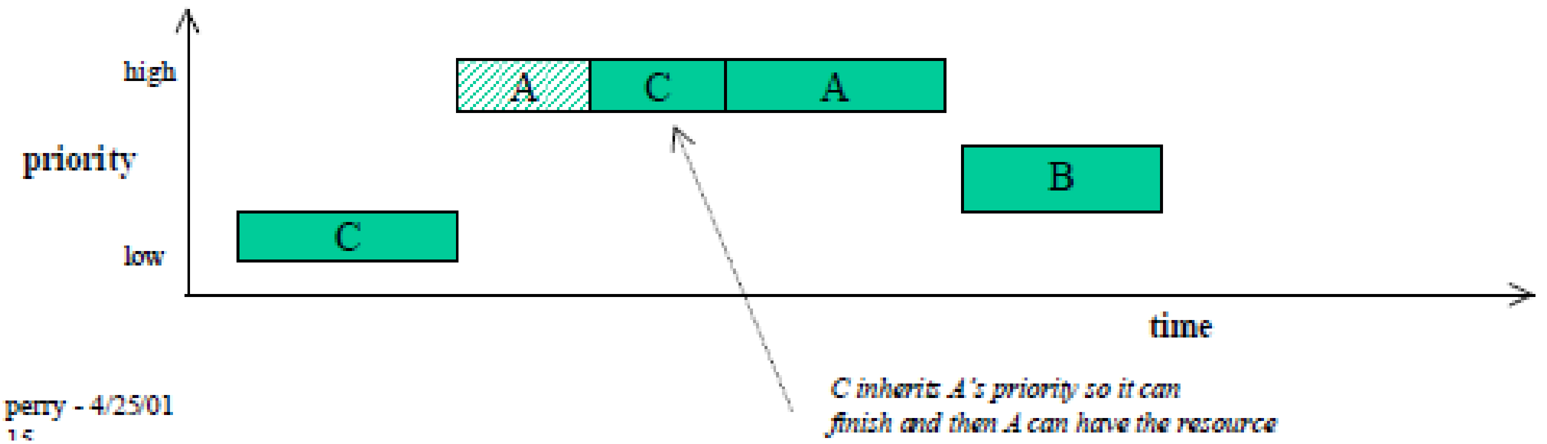
- Task A and Task C share a resource
- Task A is high priority
- Task C is low priority
- Task A is blocked when Task C runs (effectively assigning A to C's priority, hence priority inversion)
- Task A will be blocked for longer, if Task B of medium priority comes along to keep Task C from finishing
- A good RTOS would sense this condition and temporarily promote Task C to the high priority of Task A (Priority Inheritance)

Priority Inversion / Priority Inheritance

Priority Inversion - The problem


A is blocked because C has the resource
high
priority
low
A
B
C
C
C
A
B can block C which also keeps A waiting
time

Priority Inheritance - A solution


high
priority
low
A
C
A
B
C
time
C inherits A's priority so it can finish and then A can have the resource

perry - 4/25/01
15

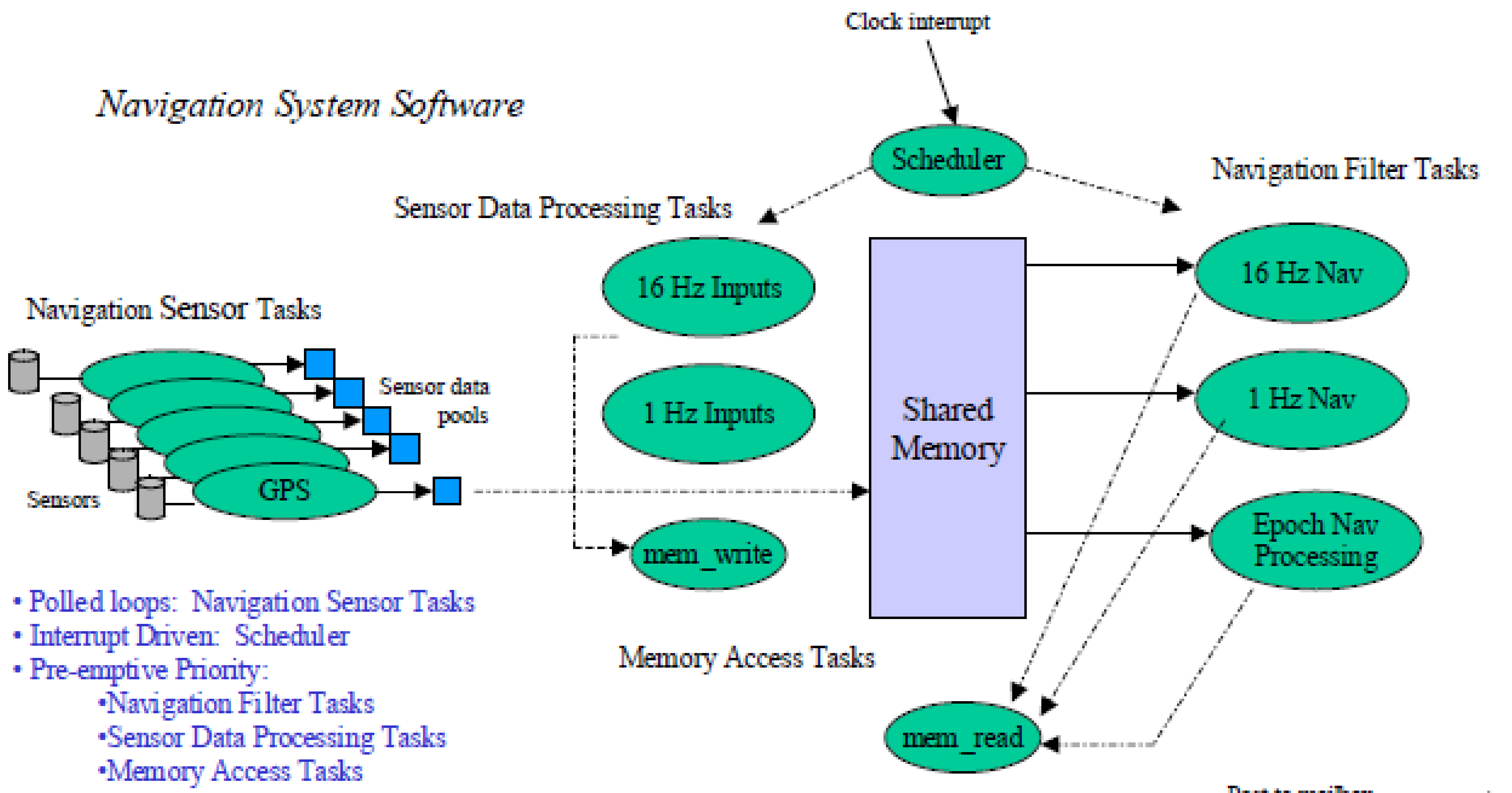
Foreground/Background Systems

- Most common hybrid solution for embedded applications
- Involve interrupt driven (foreground) AND noninterruptive driven (background) processes
- All realtime solutions are just a special case of foreground/background systems
  - Polled loops = background only system
  - Interrupt-only systems = foreground only system
- Anything not time-critical should be in background
  - Background is process with lowest priority

Foreground/Background Systems

- Gives hybrid systems = combining what we have seen so far
  - Polled loops
  - Interrupt-driven systems
  - Multi-tasking
    - Pre-emptive priority or
    - Round robin or
    - Cyclic executive

Back to the multitasking example



Navigation System Software

- Polled loops: Navigation Sensor Tasks
- Interrupt Driven: Scheduler
- Pre-emptive Priority:
  - Navigation Filter Tasks
  - Sensor Data Processing Tasks
  - Memory Access Tasks

perry - 4/25/01

Multitasking Pros & Cons

- Pros
  - Segments the problem into small, manageable piece (modular computer system design principle)
  - Makes more modular software (can reuse portions more easily)
  - Allows software designer to prioritize certain tasks over others
- Cons
  - Depending upon implementation, timing may not be deterministic (jitter caused by variations in timing of incoming data)
  - Context switching adds overhead

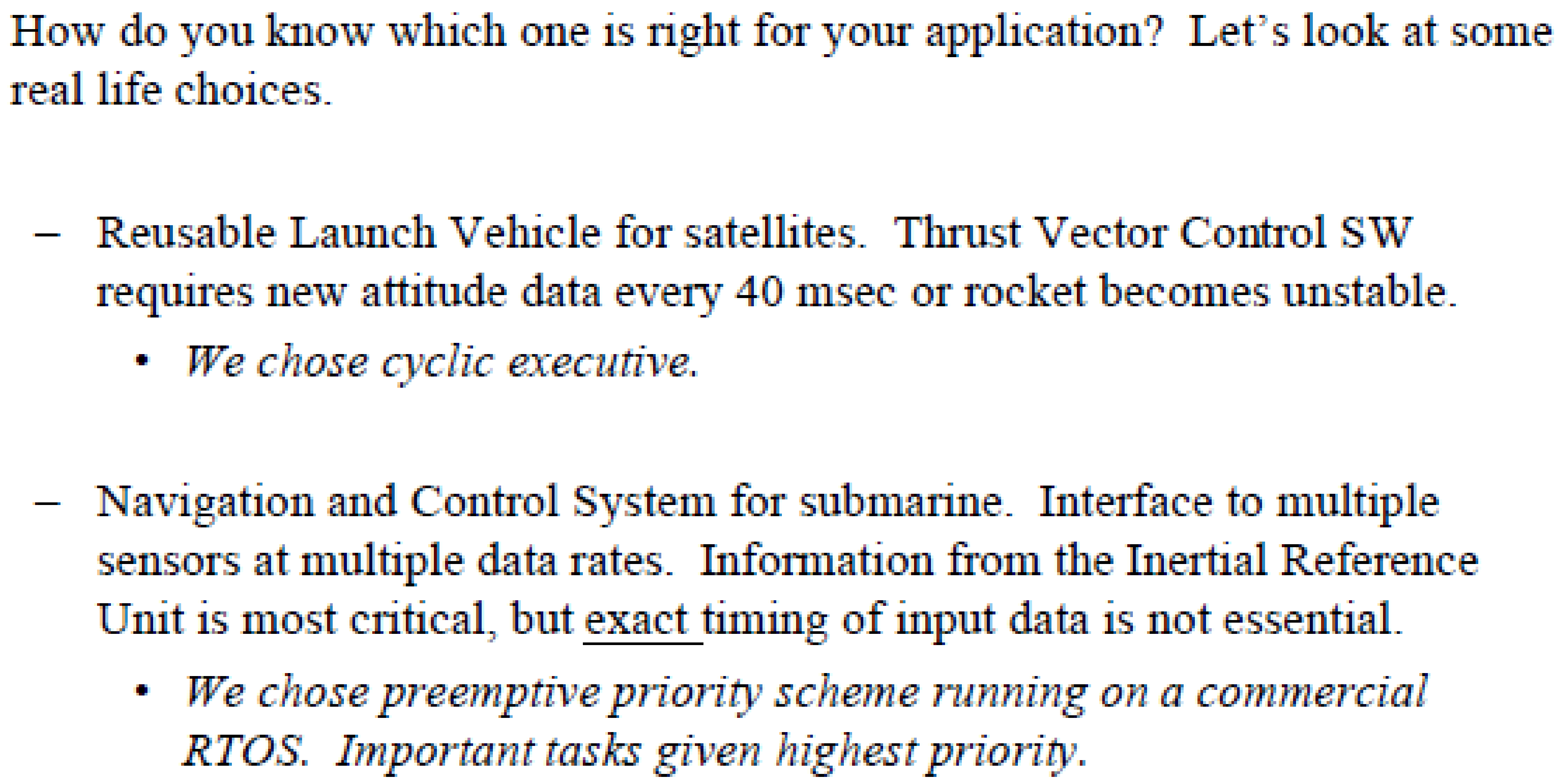
Full Featured RTOS

- Expand foreground/background solution
  - Add network interfaces
  - Add device drivers
  - Add complex debugging tools
- Most common choice for complex systems
- Many commercial operating systems available

Choosing a RTOS approach

- How do you know which one is right for your application?
- Look at what is driving your system (arrival pattern of data)
  - Irregular (known but varying sequence of intervals between events)
  - Bursty (arbitrary sequence with bound on number of events)
  - Bounded (minimum interarrival interval)
  - Bounded with average rate (unpredictable event times, but cluster around mean)
  - Unbounded (statistical prediction only)
- What is the critical I/O?
- Are there absolute hard deadlines?

Choosing a RTOS approach

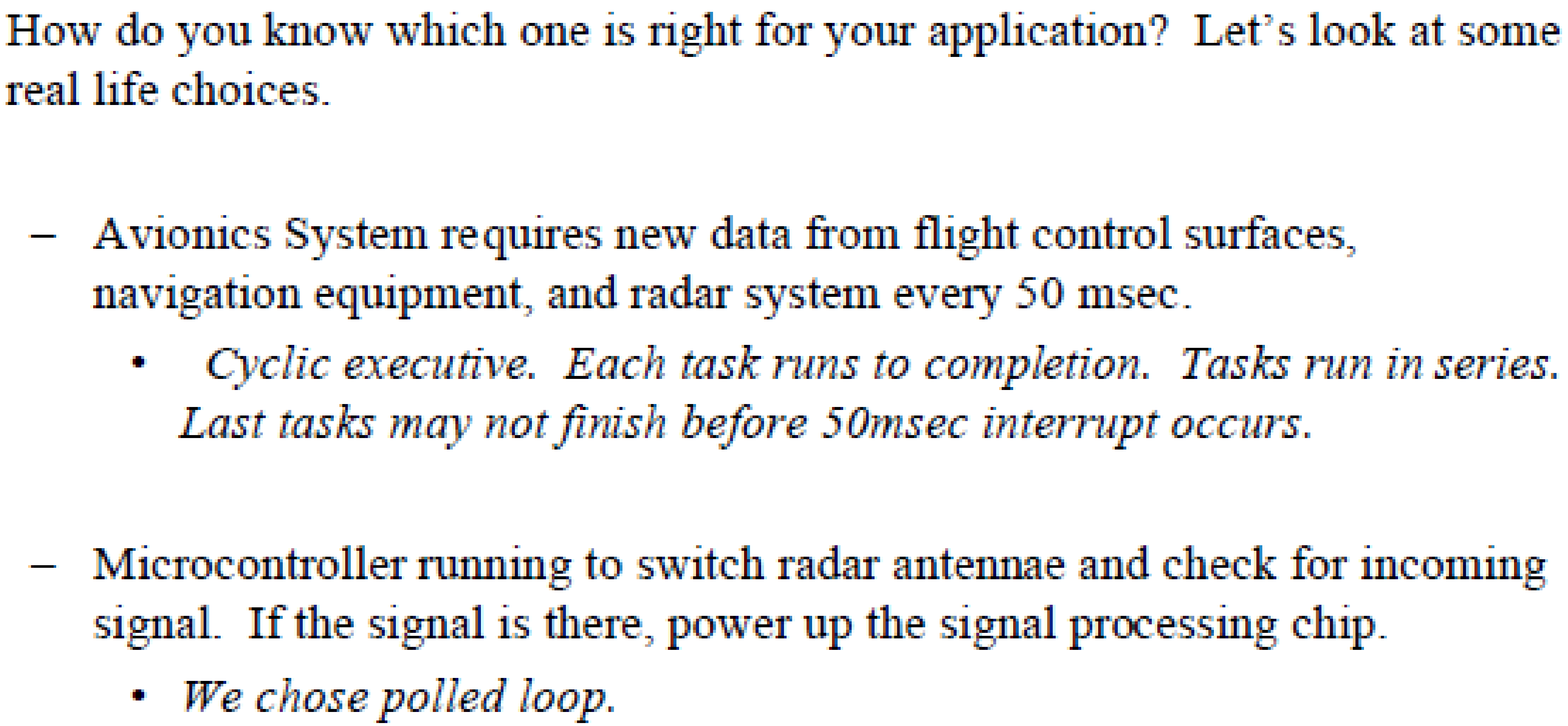


How do you know which one is right for your application? Let's look at some real life choices.

- Reusable Launch Vehicle for satellites. Thrust Vector Control SW requires new attitude data every 40 msec or rocket becomes unstable.
  - *We chose cyclic executive.*

- Navigation and Control System for submarine. Interface to multiple sensors at multiple data rates. Information from the Inertial Reference Unit is most critical, but exact timing of input data is not essential.
  - *We chose preemptive priority scheme running on a commercial RTOS. Important tasks given highest priority.*

Choosing a RTOS approach



How do you know which one is right for your application? Let's look at some real life choices.

– Avionics System requires new data from flight control surfaces, navigation equipment, and radar system every 50 msec.
  - *Cyclic executive. Each task runs to completion. Tasks run in series. Last tasks may not finish before 50msec interrupt occurs.*

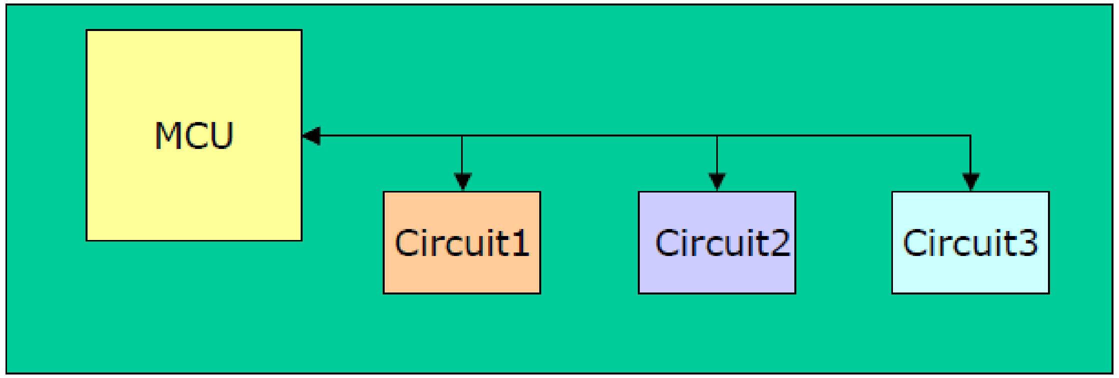
– Microcontroller running to switch radar antennae and check for incoming signal. If the signal is there, power up the signal processing chip.
  - *We chose polled loop.*

Bus and Communication Interfaces

- Parallel Bus Systems
  - Processor Buses – AVR etc.
  - Industrial Buses
    - VMEbus
    - CompactPCI
    - PC/104
    - …
- Serial Local Buses
  - SPI
  - MicroWire
  - I2C
  - 1-Wire
- Serial Lines (1 to 1, 1 to N)
  - UART
  - RS-232C
  - RS-422
  - USB
- Networks (N to M)
  - CAN
  - RS-485
  - LAN/Ethernet
- Wireless Communication
  - IR/IrDA
  - ISM
  - WiFi
  - Bluetooth
  - Zigbee

Serial synchronous interfaces

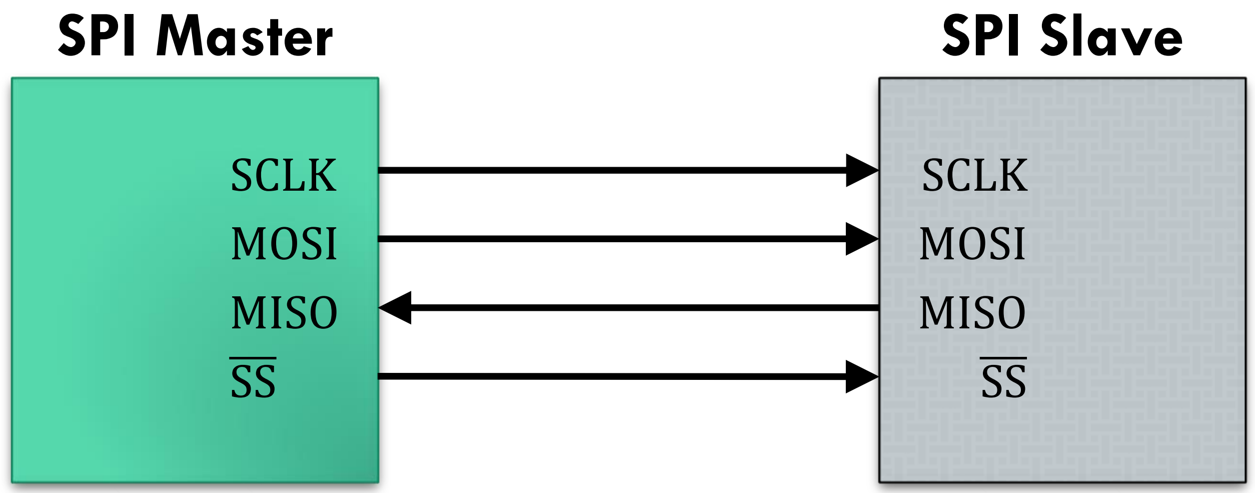
- Local serial interconnection of microcontrollers and peripheral circuits/functions
- Required features:
  - Low complexity
  - Low to medium data rate
  - Small physical footprint/few pins
  - Short distances
  - Low cost

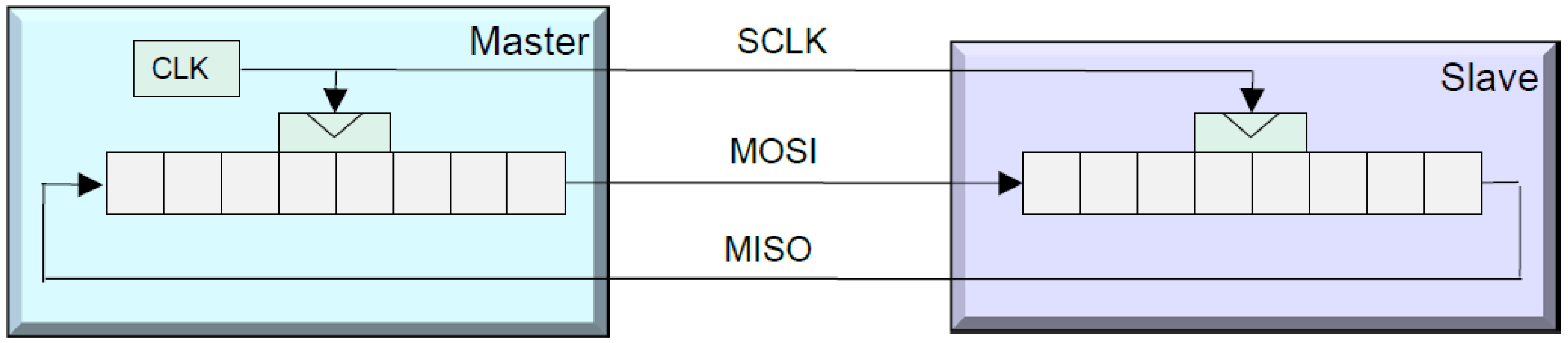


- Most MCUs have built-in peripheral units for communicating with external circuits, e.g. ATmegaAVR (SPI and TWI (I2C))
- Great abundance of different types of peripheral circuits that implements synchronous serial interfaces (Flash, EEPROM, AD, DA, RTC, Display drivers, sensors etc.)

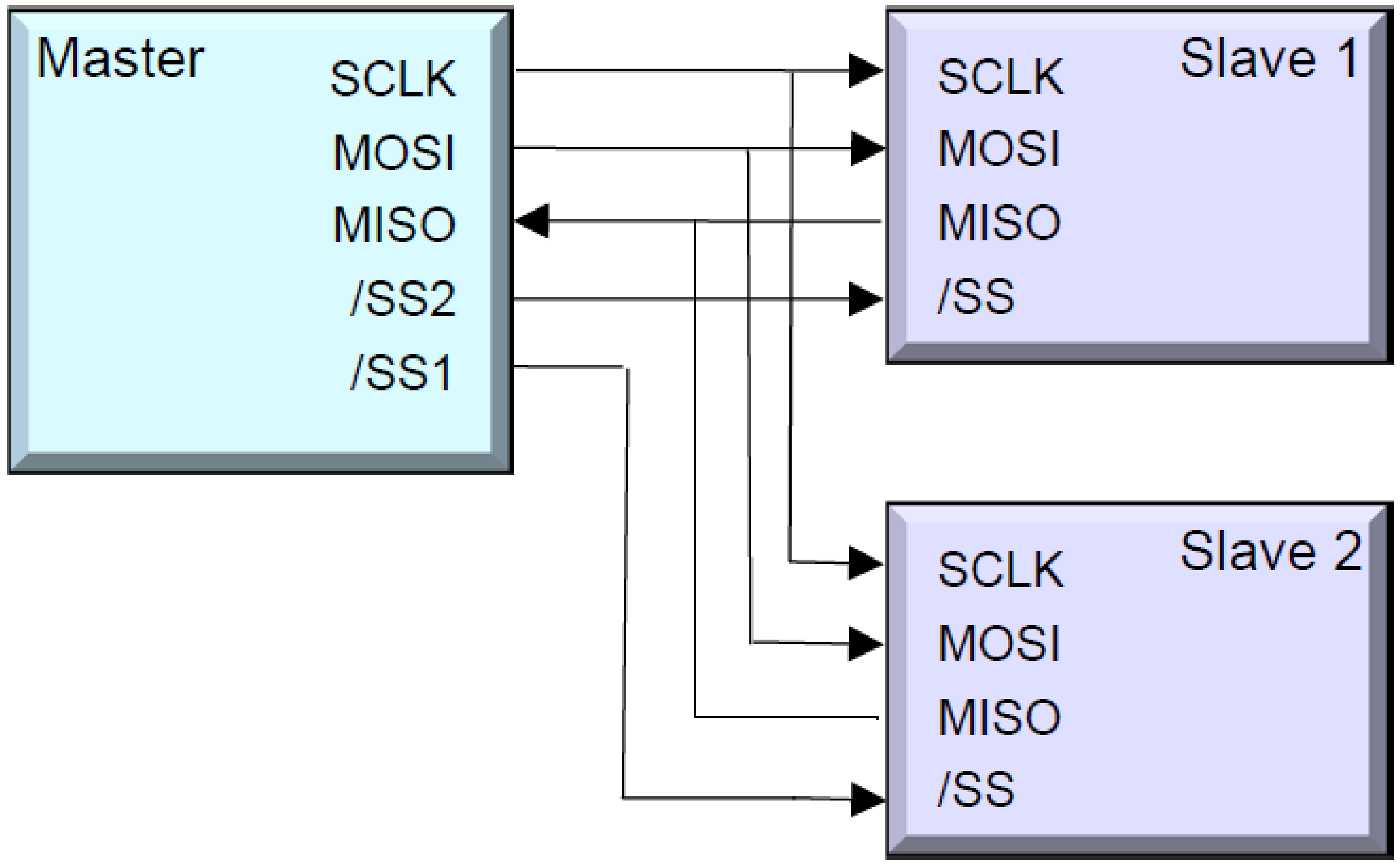
SPI: Serial Peripheral Interface

- Synchronous Data Transfer
- Master/Slave configuration
- 4-Line Bus
- Full Duplex operation



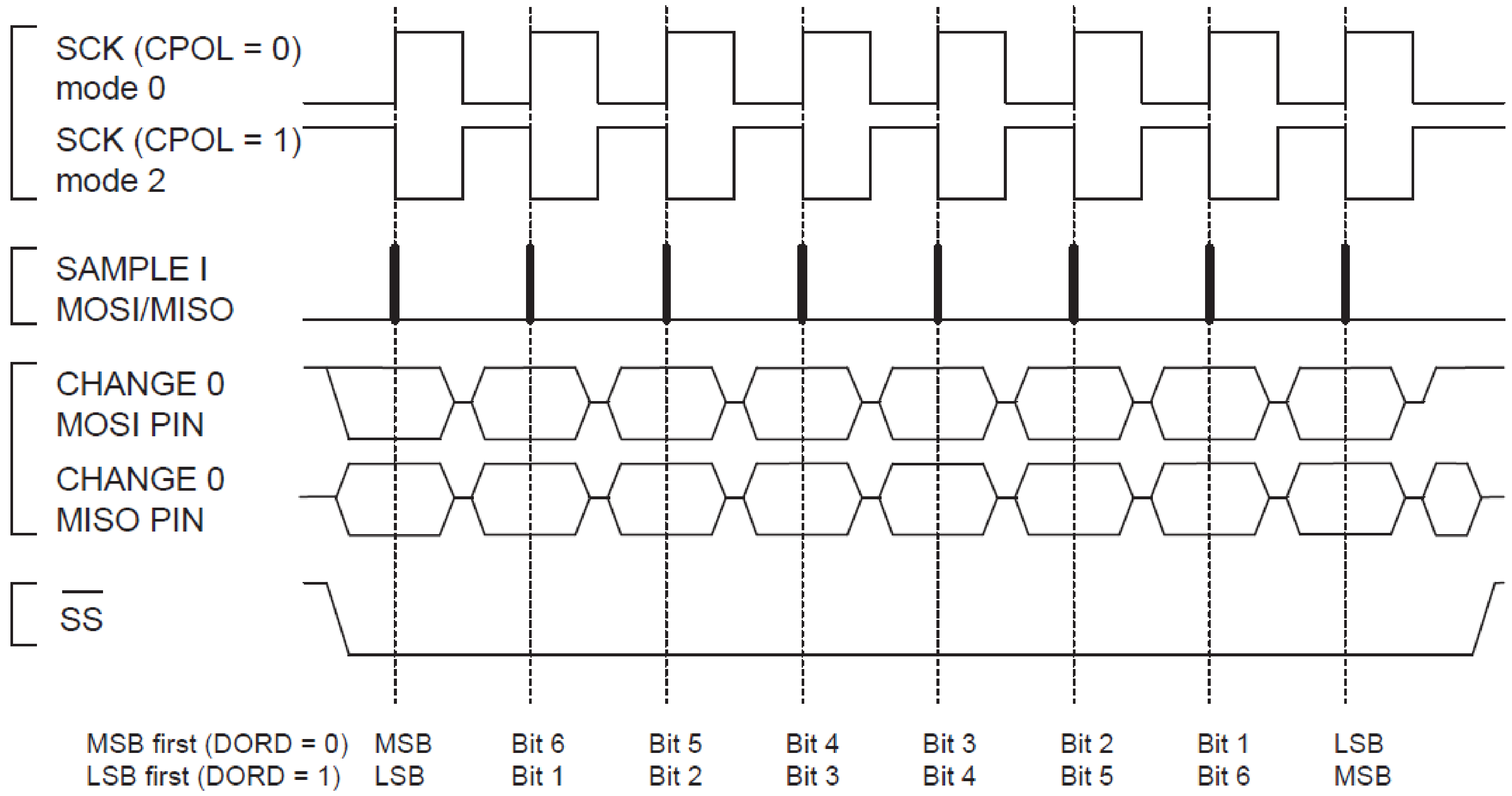


SPI Master with Multiple Slaves



SPI Frame Transfer

Figure 18-3. SPI Transfer Format with CPHA = 0



MicroWire (μWire)

- Essentially a subset of SPI
- SPI mode 0 → (CPOL, CPHA) = (0, 0)
- Often found in half duplex “three-wire mode”
- Common bi-directional serial data line → only three wires needed (SIO, SCLK, CS)
- Used in e.g. RTCs (real-time clocks) and serial EEPROMs