Software issues

- Compilers
  - assemblers
  - high-level language compilers
  - loaders/linkers
- Layers of abstraction
  - subroutines
  - device drivers
  - run-time systems: concurrency and communication
  - operating systems
  - application programmer interfaces (APIs)
- Debugging
  - emulation
  - monitors
  - agents

Writing code

- Machine code
- Assembler
  - macro capability
  - symbolic variables
  - direct access to microcontroller's special features
- Compilers
  - high-level language (PL/M, C, etc.)
  - escape to assembly language when needed
  - API for special features (I/O ports, special registers, etc.)
- Program loader/linker
  - stores program into ROM/RAM
  - combines multiple programs (e.g., user code with device drivers)

Abstraction

- Subroutines
  - encapsulate frequently used functions
  - save system state on entering, restore on exit
  - parameter passing through the stack or registers
- Device drivers
  - special subroutines for accessing peripheral devices
  - may or may not include state
  - interactions with other activities (e.g., interrupts)

Very simple device driver

- Turn LED on/off
- Parameters:
  - port pin
- API:
  - on(port_pin)
    - specifies the port pin (e.g., port D pin 3)
  - off(port_pin)
- Interactions:
  - only if other devices want to use the same port

Simple device driver

- Turning an LED on and off at a fixed rate
- Parameters:
  - port pin
  - rate at which to blink LED
- API:
  - on(port_pin, rate)
    - specifies the port pin (e.g., port D pin 3)
    - specifies the rate to use in setting up the timer (what scale?)
  - off(port_pin)
- Internal state and functions:
  - keep track of state (on or off for a particular pin) of each pin
  - interrupt service routine to handle timer interrupt
  - set up interrupt service routine address

Interesting interactions

- What if other devices also need to use timer (e.g., PWM device)?
  - timer interrupts now need to be handled differently depending on which device's alarm is going off
- Benefits of output compare
  - output compare pins used exclusively for one device
  - each output compare has a separate interrupt handling routine
- What if we don't have output compare capability?

Sharing timers

- Create a new device driver for the timer unit
  - allow other devices to ask for timer services
  - manage timer independently so that it can service multiple requests
- Parameters:
  - time to wait
  - address to call when timer reaches that value
- API:
  - set_timer(time_to_wait, call_back_address)
    - set call_back_address to correspond to time+time_to_wait
    - compute next alarm to sound and set up timer for that
    - update in interrupt service routine for next alarm
- Internal state and functions:
  - how many alarms can the driver keep track of?
  - how are they organized? FIFO? priority queue?

Concurrency

- Multiple programs being interleaved to give appearance of parallelism
- Each program can request access to devices and services (e.g., timers)
- Exclusive or concurrent access to devices
  - allow only one program at a time to access a device (e.g., serial port)
  - arbitrate multiple accesses (e.g., timer)
- State and arbitration needed
  - keep track of state of devices and concurrent programs using resource
  - arbitrate their accesses (order, fairness, exclusivity)
  - monitors/locks (supported by primitive operations in ISA - test-and-set)
- Interrupts
  - disabling may effect timing of programs
  - keeping enabled may cause unwanted interactions

Handling concurrency

- Traditional operating system
  - multiple threads or processes
  - file system
  - virtual memory and paging
  - input/output (buffering between CPU, memory, and I/O devices)
  - interrupt handling (mostly with I/O devices)
  - resource allocation and arbitration
  - command interface (execution of programs)
- Embedded operating system
  - lightweight threads
  - input/output
  - interrupt handling
  - real-time guarantees

Embedded operating systems

- Lightweight threads
  - basic locks
  - fast context-switches
- Input/output
  - API for talking to devices
  - buffering
- Interrupt handling (with I/O devices and user interface)
  - translate interrupts into events to be handled by user code
  - trigger new tasks to run (reactive)
- Real-time issues
  - guarantee task is called at a certain rate
  - guarantee an interrupt will be handled within a certain time
  - priority or deadline driven scheduling of tasks

Case studies

- Palm OS
  - US Robotics Palm Pilot
  - Motorola 68328 microcontroller (Dragonball)
  - simple OS for PDAs
  - only supports single threads
- Windows CE
  - PDA operating system
  - spin-off of Windows '95
  - portable to a wide variety of processors
  - full-featured OS but modularized so uses only include features as needed
- Wind River Systems VxWorks
  - one of the most popular embedded OS kernels
  - highly portable to an even wider variety of processors
  - modularized even further

embedded operating systems typically reside in ROM

Palm OS

- Interface management
  - basic user interface model
  - event dispatch loop paradigm
- System management
  - alarm and time (timers and real-time clocks)
  - sound, pen, key, serial port (I/O devices)
  - string (libraries of routines)
  - system (errors, power, application start/stop)
  - event (interrupt->event translation, dispatch loop)
  - access to registers of microcontroller (access special features not in API)
- Memory management
  - structures for data in memory (heaps, records, databases)
- Communication
  - serial port (layered with TCP/IP protocol)

Palm OS Application Model

- One application is "running"
  - running application has control of the screen and user input
  - others may be active in the background and receive other I/O events
  - when application starts, "Main" is called
- Main
  - runs "StartApplication" to set up user interface of program
  - runs "StopApplication" before it stops execution
  - all data in RAM
  - permanent storage in memory heaps organized in "databases"
  - "EventLoop" is run between start and stop

Events

- Generated by system (timers, I/O, user input, etc.)
- Applications polls (in a loop) for new events
- Many different types of events (29 in all)
  - application events (e.g., stop)
  - user interface events
    - menu
    - selection
    - pen
    - key
- Interpret events
  - specific to application
  - default behavior provided by system

Event dispatch loop

- Get event
- Handle event
  - system event handler
  - menu event handler
  - application event handler
  - user interface element event handler
- Leverage default behaviors
  - most user code goes into the application event handler

Storage system

- Everything is stored in RAM
  - dynamic (stacks, heaps, global vars of application)
    - when application stops, data may be lost
  - storage (databases)
    - analagous to a file and provides static storage across invocation of the application
- Databases
  - 32K chunks of data
  - each entry is associated with a type (linked to an application)

Conduits

- Mechanism for transferring databases across serial port to PC
- Hot-Sync
  - synchronizes RAM databases with PC copies
  - changes on PC propagate to RAM and vice-versa
- Also used to transfer new applications to RAM
- Can be extended to bring arbitrary data into the Pilot
  - e.g., collect e-mail for later reading
- Pilot was envisioned as extension of desktop
  - user input at both ends (PC or Pilot)
  - same data (synchronized periodically)

Development environment

- Elements
  - user interface constructor
  - source code editor
  - compiler
  - debugger
- SDKs: system development kits
  - Metrowerks CodeWarrior for Palm Pilot
  - GNU for Palm Pilot
- Conduit development kits for PC