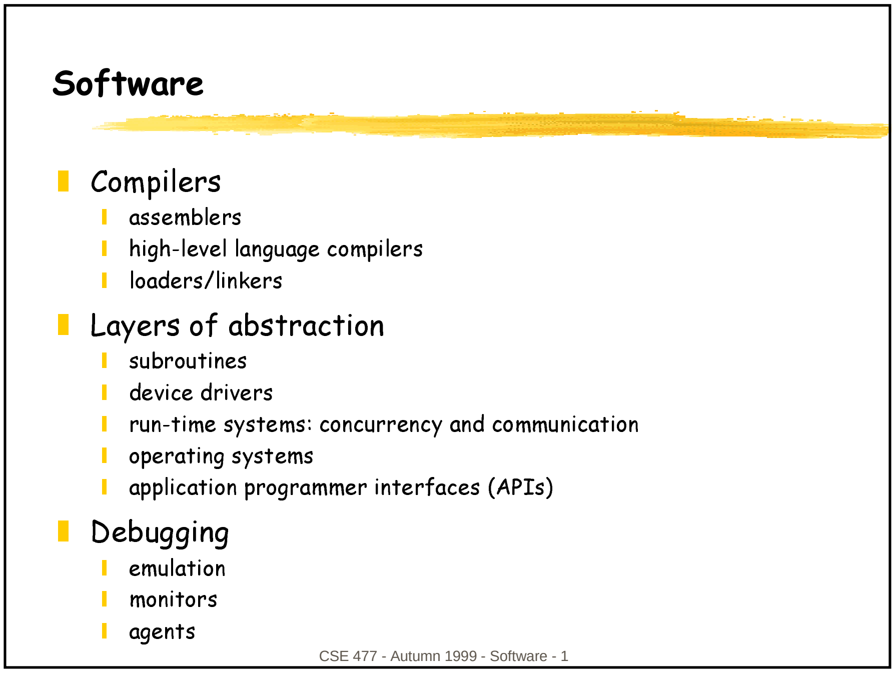
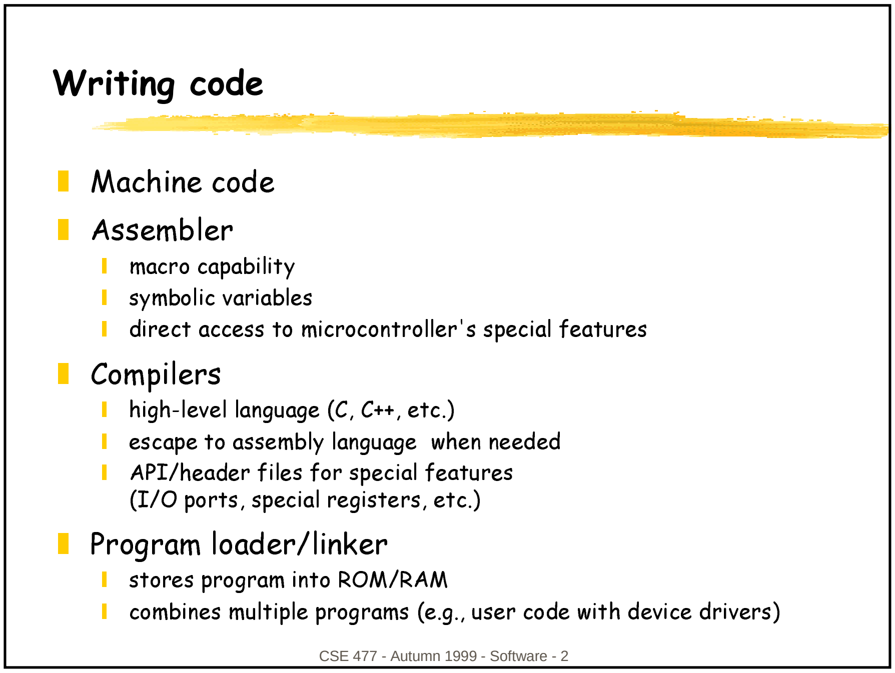


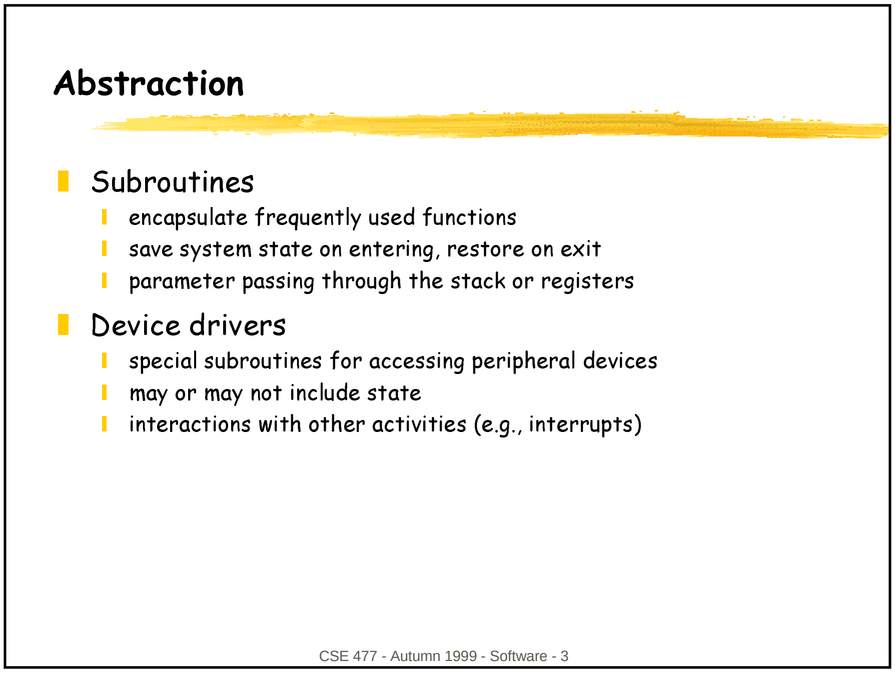
Software

- 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



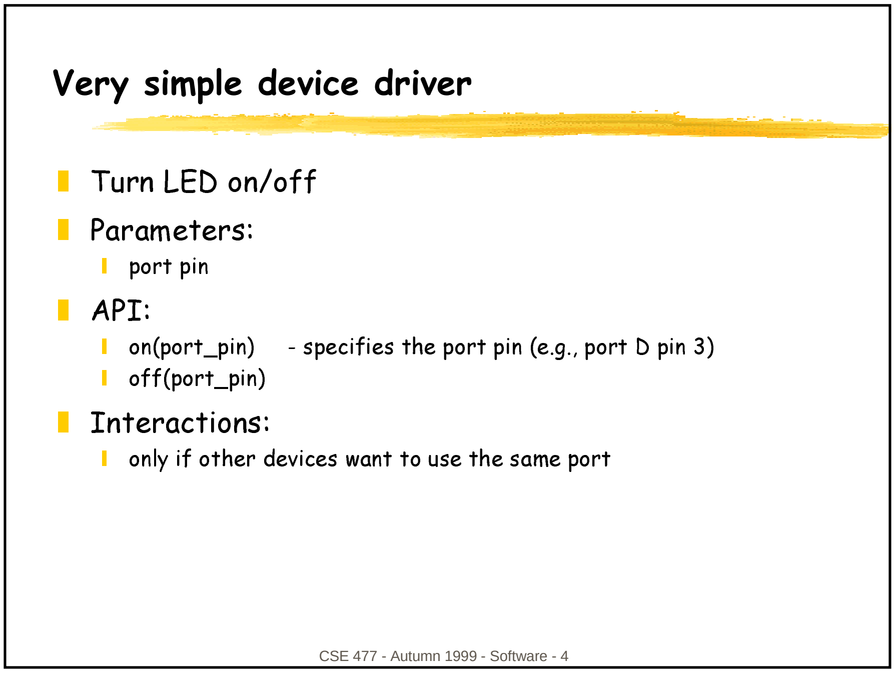
Writing code

- Machine code
- Assembler
  - macro capability
  - symbolic variables
  - direct access to microcontroller's special features
- Compilers
  - high-level language (C, C++, etc.)
  - escape to assembly language when needed
  - API/header files 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)



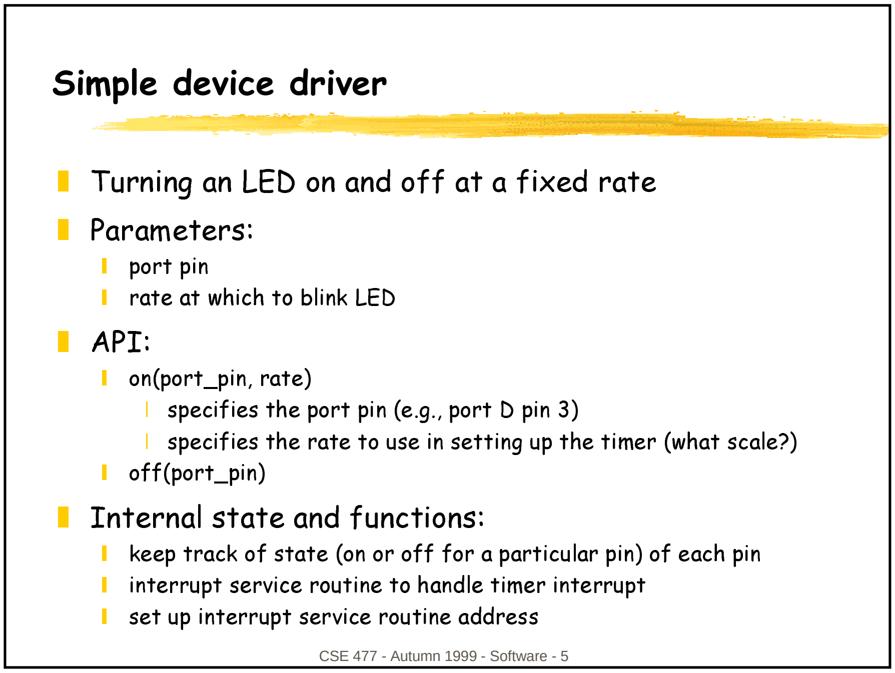
Abstraction

- 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)



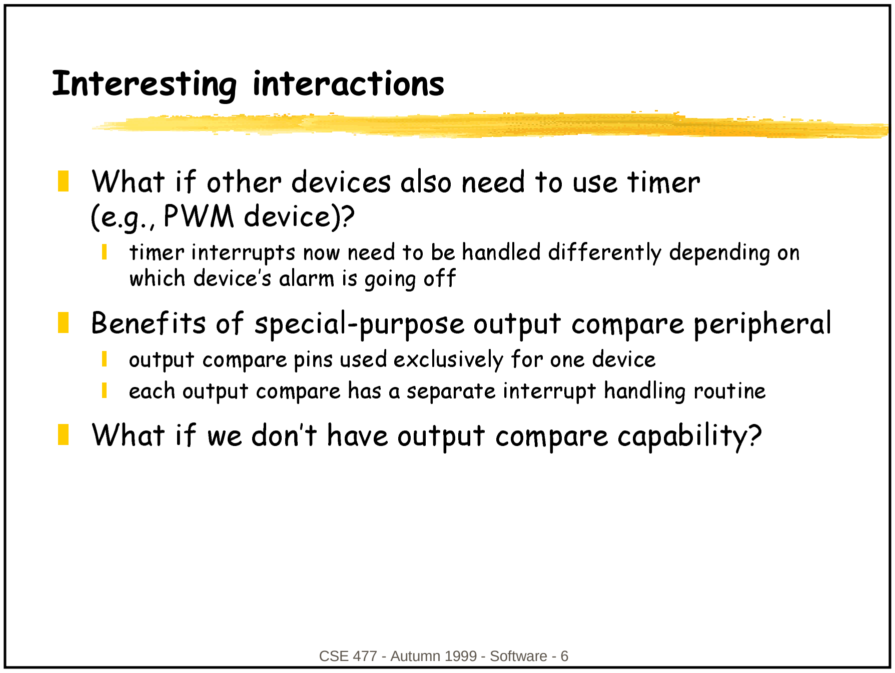
Very simple device driver

- 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



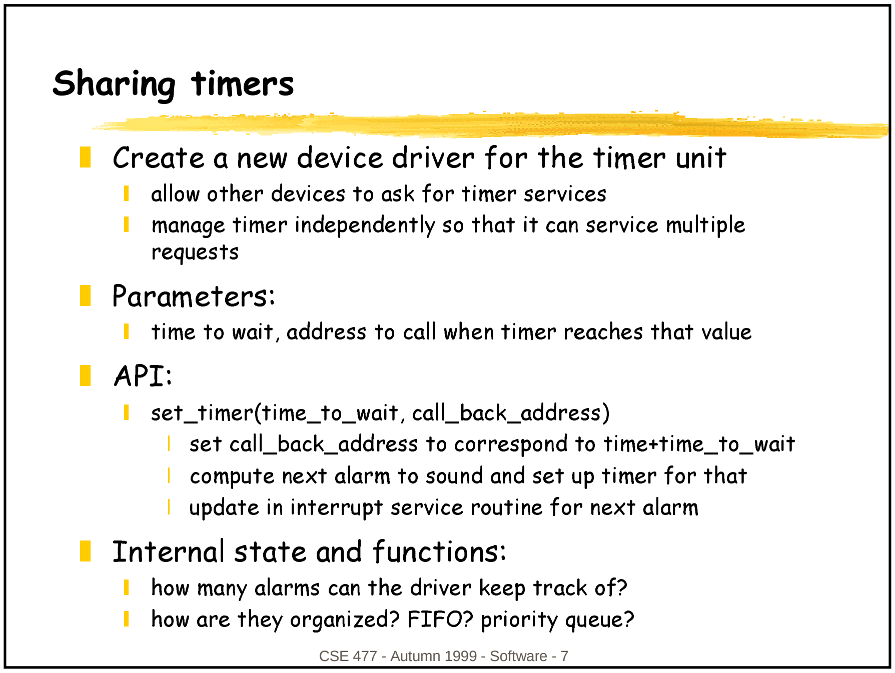
Simple device driver

- 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



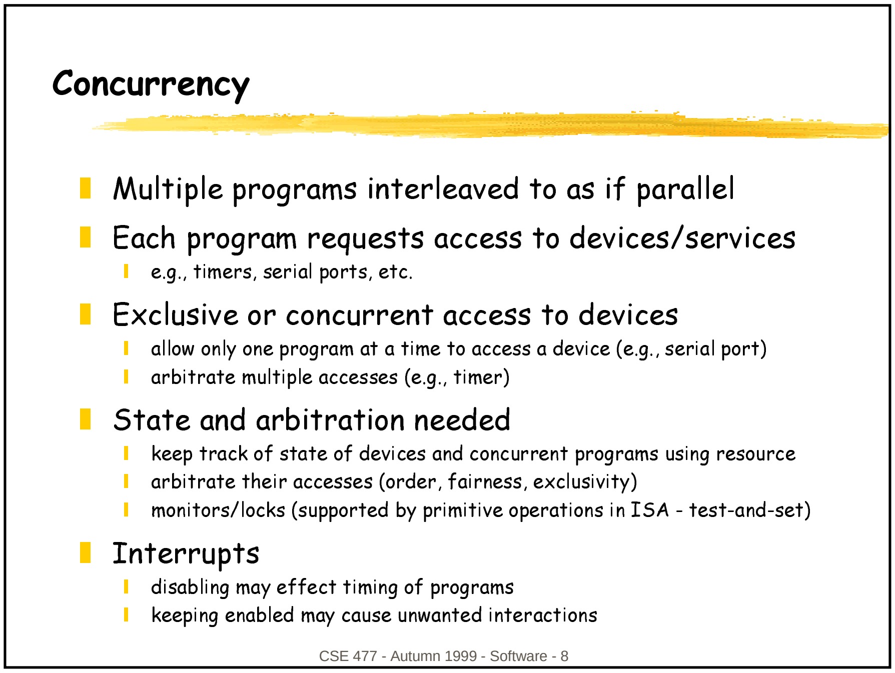
Interesting interactions

- 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 special-purpose output compare peripheral
  - 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?



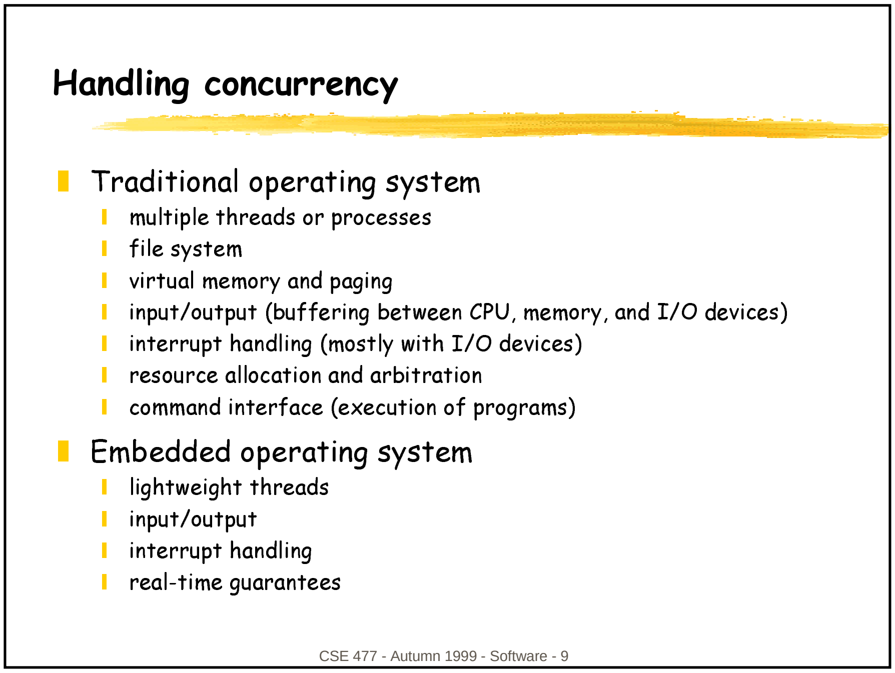
Sharing timers

- 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?



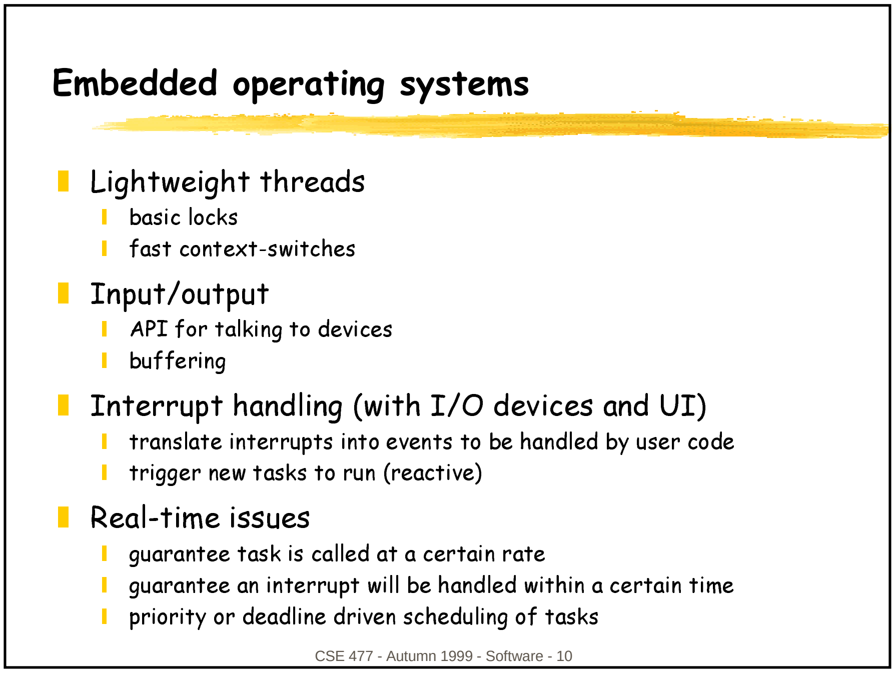
Concurrency

- Multiple programs interleaved to as if parallel
- Each program requests access to devices/services
  - e.g., timers, serial ports, etc.
- 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 UI)
  - 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

Examples

- Palm OS (e.g., IBM Workpad)
  - US Robotics Palm Pilot
  - Motorola microcontrollers (68328 - Dragonball)
  - simple OS for PDAs
  - only supports single threads
- Windows CE (e.g., Nino)
  - PDA operating system
  - spin-off of Windows '95
  - portable to a wide variety of processors
  - full-featured OS modularized to 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 (tiny to huge)
  - modularized even further

embedded operating systems typically reside in ROM (flash)

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)
- Memory management
  - structures for data in memory (heaps, records, databases)
- Communication
  - serial port (layered with TCP/IP protocol)

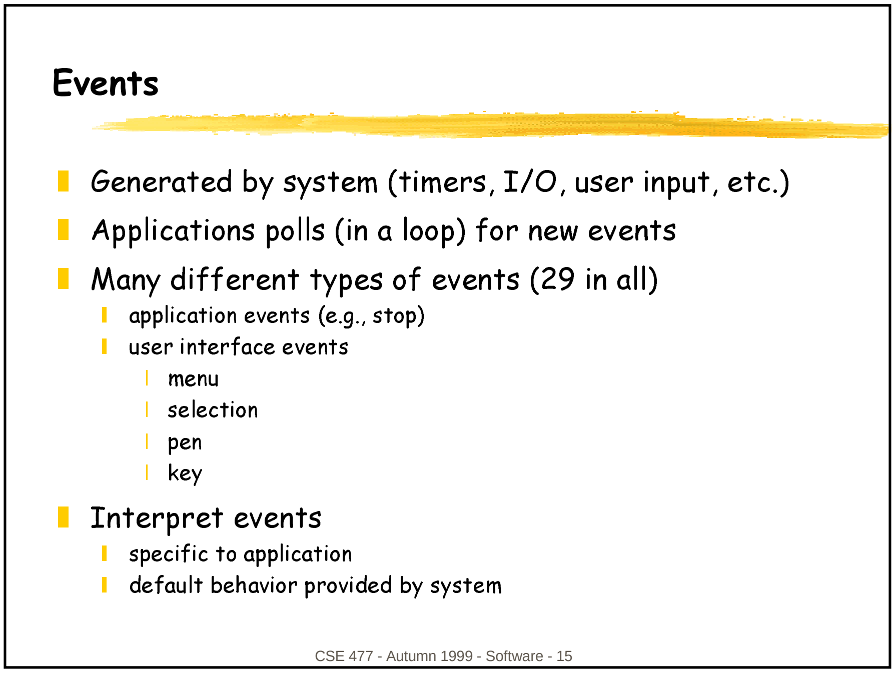
Palm OS Application Model

- 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, “PilotMain” is called
- Main
  - runs “AppStart” to set up user interface of program
  - runs “AppStop” before it stops execution
  - all data in RAM
  - permanent storage in memory heaps organized in “databases”
  - “AppEventLoop” is run between start and stop

PilotMain

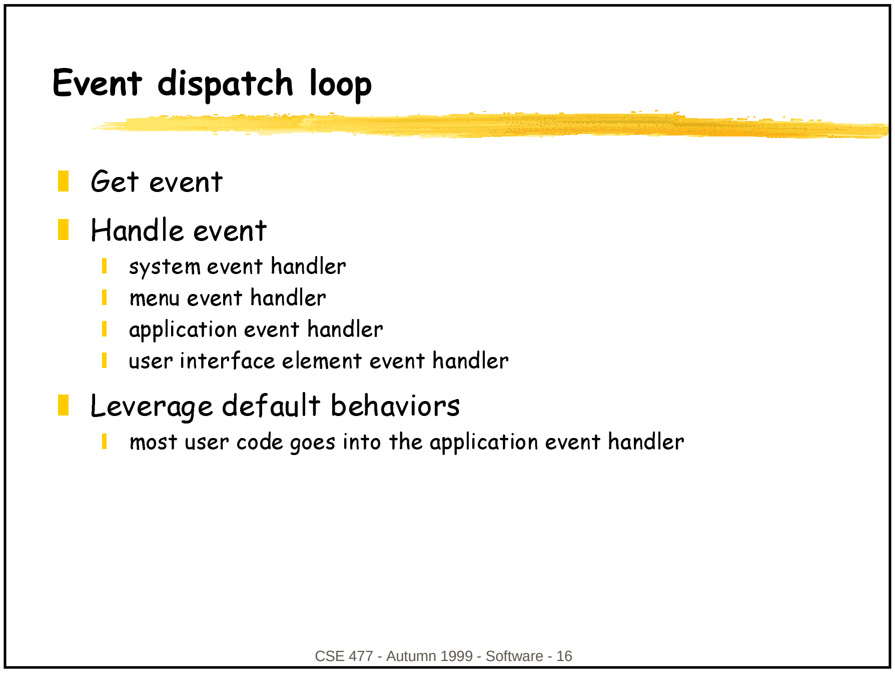
```
static DWord StarterPilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{
        Err error;
        error = RomVersionCompatible(ourMinVersion, launchFlags);
        if (error)
                return error;
        switch (cmd) {
        case sysAppLaunchCmdNormalLaunch:
                error = AppStart();
                if (error)
                        return error;

                FrmGotoForm(MainForm);
                AppEventLoop();
                AppStop();
                break;
        default:
                break;
        }
        return 0;
}
```



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

AppEventLoop

```
static void AppEventLoop(void)
{
        Word error;
        EventType event;

        do {
                EvtGetEvent(&event, evtWaitForever);

                if (! SysHandleEvent(&event))
                        if (! MenuHandleEvent(0, &event, &error))
                                if (! AppHandleEvent(&event))
                                        FrmDispatchEvent(&event);

                /*
                ** do other stuff here
                */

        } while (event.eType != appStopEvent);
}
```

Serial Line

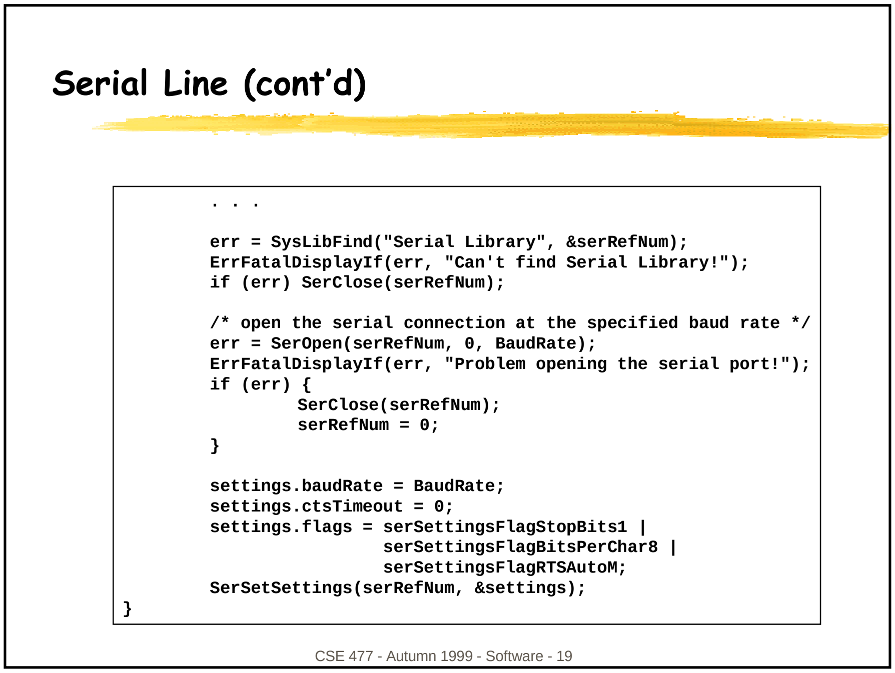
```
static void EVBconnect()
{
        int             i;
        Err             err = 0;
        SerSettingsType settings;
        FormPtr frmP;

        /* already connected? */
        if (serRefNum != 0) {
                frmP = FrmInitForm(ConnectForm);
                FrmDoDialog(frmP);
                FrmDeleteForm(frmP);
                return;
        }

        /* initialize variables */
        gotFramingByte = FALSE;
        numBytesRcvd = 0;

        for (i=0; i < SerRcvQueueSize; i++)
                serRcvQueue[i] = '\0';

        . . .
```



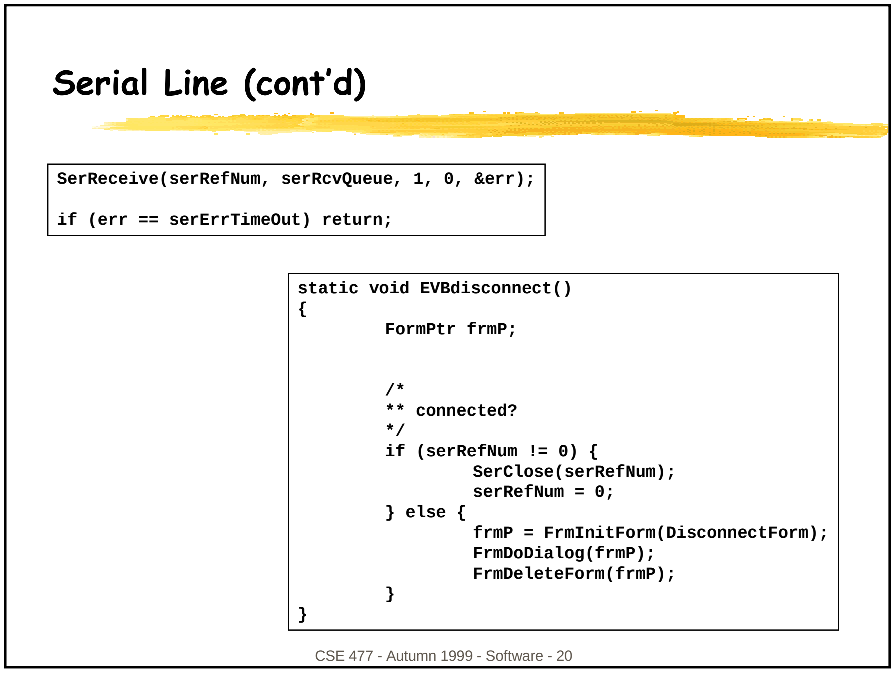
Serial Line (cont'd)

```
        . . .

        err = SysLibFind("Serial Library", &serRefNum);
        ErrFatalDisplayIf(err, "Can't find Serial Library!");
        if (err) SerClose(serRefNum);

        /* open the serial connection at the specified baud rate */
        err = SerOpen(serRefNum, 0, BaudRate);
        ErrFatalDisplayIf(err, "Problem opening the serial port!");
        if (err) {
                SerClose(serRefNum);
                serRefNum = 0;
        }

        settings.baudRate = BaudRate;
        settings.ctsTimeout = 0;
        settings.flags = serSettingsFlagStopBits1 |
                         serSettingsFlagBitsPerChar8 |
                         serSettingsFlagRTSAutoM;
        SerSetSettings(serRefNum, &settings);
}
```



Serial Line (cont'd)

```
SerReceive(serRefNum, serRcvQueue, 1, 0, &err);

if (err == serErrTimeOut) return;
```

```
static void EVBdisconnect()
{
        FormPtr frmP;


        /*
        ** connected?
        */
        if (serRefNum != 0) {
                SerClose(serRefNum);
                serRefNum = 0;
        } else {
                frmP = FrmInitForm(DisconnectForm);
                FrmDoDialog(frmP);
                FrmDeleteForm(frmP);
        }
}
```

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
  - chunks of data
  - each entry is associated with a type (linked to an application)

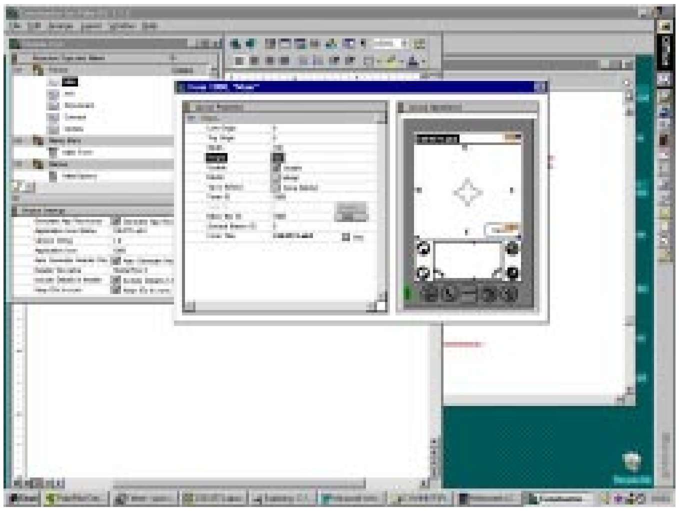
Conduits

- Mechanism for transferring databases across serial ports of device (RS232 or IrDA) 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)

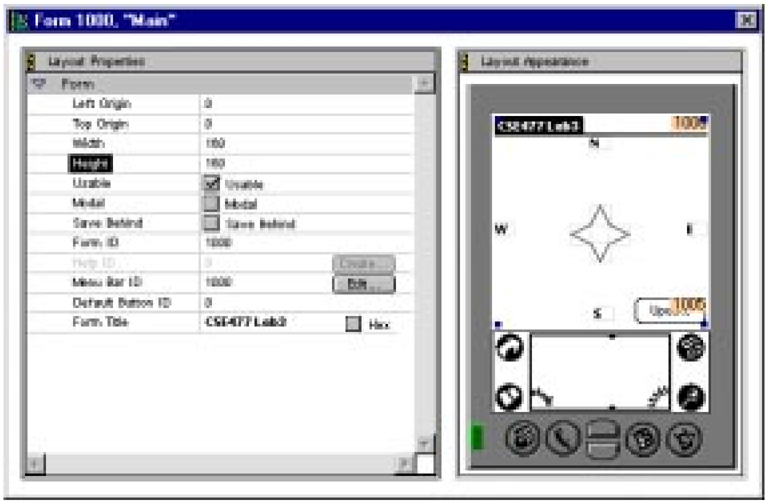
Development environment

- 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

User Interfaces in PalmOS



Form Layout



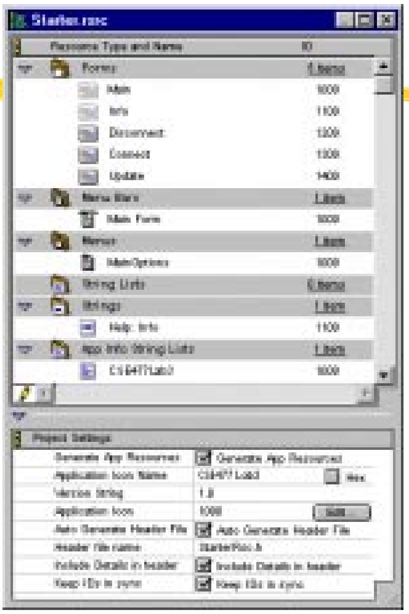
Form 1000, "Main"

Layout Properties

▽ Form

Left Origin	0
Top Origin	0
Width	160
Height	160
Usable	☑ Usable
Modal	□ Modal
Save Behind	□ Save Behind
Form ID	1000
Help ID	0
Menu Bar ID	1000
Default Button ID	0
Form Title	CSE477 Lab3 □ Hex

UI Resources



Starter.rsrc

Resource Type and Name	ID
▽ Forms	5 items
Main	1000
Info	1100
Disconnect	1200
Connect	1300
Update	1400
▽ Menu Bars	1 item
Main Form	1000
▽ Menus	1 item
MainOptions	1000
String Lists	0 items
▽ Strings	1 item
Help: Info	1100
▽ App Info String Lists	1 item
CSE477Lab3	1000

Project Settings

Generate App Resources	☑ Generate App Resources
Application Icon Name	CSE477 Lab3 □ Hex
Version String	1.0
Application Icon	1000
Auto Generate Header File	☑ Auto Generate Header File
Header File name	StarterRsc.h
Include Details in header	☑ Include Details in header
Keep IDs in sync	☑ Keep IDs in sync

Resources header file

```
//  Resource: tFRM 1000
#define MainForm                  1000  //(Left Origin = 0, Top Origin = 0, Width =
#define MainUpdateButtonButton    1005  //(Left Origin = 107, Top Origin = 138, Widt
#define MainUnnamed1001BitMap     1000  //(Left Origin = 72, Top Origin = 67, Bitmap
#define MainUnnamed1002BitMap     1200  //(Left Origin = 86, Top Origin = 82, Bitmap
#define MainUnnamed1003BitMap     1100  //(Left Origin = 72, Top Origin = 95, Bitmap
#define MainUnnamed1004BitMap     1300  //(Left Origin = 57, Top Origin = 82, Bitmap
#define MainUnnamed1006Label      1006  //(Left Origin = 72, Top Origin = 15, Usable
#define MainUnnamed1007Label      1007  //(Left Origin = 75, Top Origin = 144, Usabl
#define MainUnnamed1008Label      1008  //(Left Origin = 145, Top Origin = 80, Usabl
#define MainUnnamed1009Label      1009  //(Left Origin = 0, Top Origin = 80, Usable

//  Resource: tFRM 1100
#define InfoForm                  1100  //(Left Origin = 2, Top Origin = 46, Width =
#define InfoUnnamed1101Button     1101  //(Left Origin = 60, Top Origin = 75, Width
#define InfoUnnamed1102Label      1102  //(Left Origin = 45, Top Origin = 29, Usable

//  Resource: MENU 1000
#define MainOptionsMenu                                 1000
#define MainOptionsConnect                              1000
#define MainOptionsDisconnect                           1001
#define MainOptionsUpdate                               1002
#define MainOptionsAboutCSE477Lab3                      1004
```

Debugging code

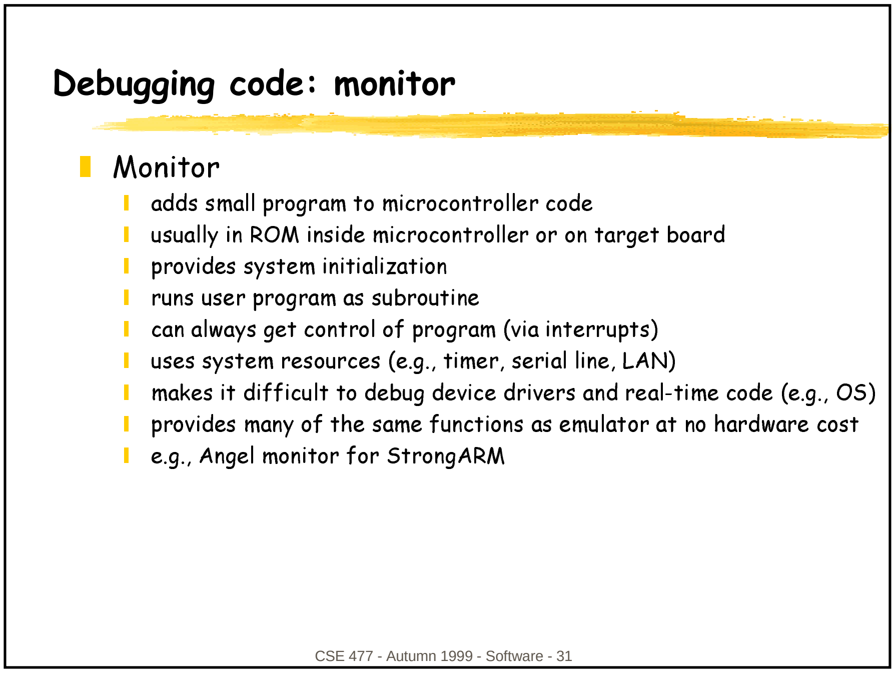
- Emulators
  - replaces microcontroller in system
  - "debuggable" version of microcontroller
- Monitors
  - add code to microcontroller that can always take control
  - requires resources
- Agents
  - smaller amount of code called by program being debugged
  - does not provide complete control

Debugging code: direct

- Real microcontroller
  - hard limits (e.g., program must fit in available memory of target)
  - load ROM or dual-ported RAM with program
  - initializations must be just right (e.g., stack pointer and comm port)
  - debugged via logic analyzer on pins
  - modern microcontrollers include special pins to permit access to internal state without disrupting running program

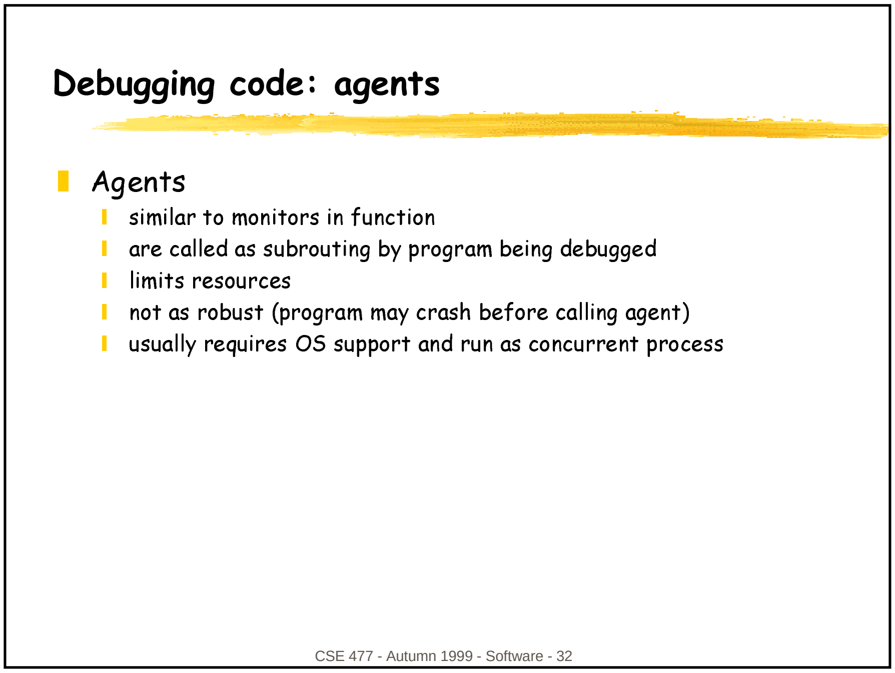
Debugging code: emulation

- Emulator
  - replaces microcontroller in target system being designed
  - pin-compatible (timing also, although not always perfect)
  - provides access to internal memory and registers
    (hard to get to otherwise due to limitations of I/O pins)
  - single-step capability
    (with links to source code)
  - relaxes memory bounds
    (fakes external memory including "infinite" stack)



Debugging code: monitor

- Monitor
  - adds small program to microcontroller code
  - usually in ROM inside microcontroller or on target board
  - provides system initialization
  - runs user program as subroutine
  - can always get control of program (via interrupts)
  - uses system resources (e.g., timer, serial line, LAN)
  - makes it difficult to debug device drivers and real-time code (e.g., OS)
  - provides many of the same functions as emulator at no hardware cost
  - e.g., Angel monitor for StrongARM



Debugging code: agents

- Agents
  - similar to monitors in function
  - are called as subrouting by program being debugged
  - limits resources
  - not as robust (program may crash before calling agent)
  - usually requires OS support and run as concurrent process