CSE 237B Fall 2009
Low Level Programming

Rajesh Gupta
Computer Science and Engineering
University of California, San Diego.
Outline

- **Problem addressed:**
  - how do we extend/modify process and communication models to enable modeling and control of “devices”?
- Review hardware input/output mechanisms
- Identify language requirements
- Define various ways of building device drivers
  - Examples from Modula-1, Ada, Occam and C for device driving.

- **References:**
  - Burns/Wellings: Chapter 15
  - Architecture of Device I/O Drivers, David Kalinsky
  - Abstracting Device Drivers, Steve Finseth
  - O’Nils, Jantsch, “Device Driver and DMA Controller Synthesis from HW/SW Protocol Specifications”
I/O In Embedded Systems

• Generally, all IO is slower than the CPU, CPU/Memory interface
  – Mechanisms needed to control the rate of data transfer
• Generally, all IO is asynchronous to the CPU
  – Mechanisms needed to coordinate data transfer

• Mechanisms built around polling, interrupt, DMA

• IO efficiency measured in terms of transfer rate and latency
  – Transfer rate: bytes per second, max rate gives a measure of the bandwidth capability for a particular IO method. Overheads degrade this.
  – Latency is a measure of the delay from the instant the device is ready until the time the first data byte is transferred.
Component Interfaces & Timing

- Two types of CPU-Component interfacing
  - Memory Interface: Read/Write
  - Peripheral Interface: Memory-Mapped I/O or Separate I/O
Architectural Interfaces to Devices

• Interface using registers
  – separate memory versus I/O accesses using separate assembly instructions (e.g., Intel x86)
  – in memory-mapped I/O, a part of the memory is mapped to device registers (e.g., Moto M68K)
  – **Pros**
    • Rich set of addressing modes for I/O operation (e.g. bit manipulation)
    • No overhead on special instructions
  – **Cons**
    • Processor’s address space is allocated to I/O devices
    • Risk of errors due to spurious accesses (elaborated address decoding)
    • Lack of special purpose I/O signals to control the I/O operation

• Interface is used
  – to control device functionality
  – to control data transfer to/from device

• Control (for data transfer) can be
  – **status** driven or **interrupt** driven.
Programming Status Driven Control

• Also referred to as a “polled waiting loop”
• A program performs explicit tests to determine status of a device
• Three kinds of hardware instructions that support status-driven control:
  – test operations
    • enable the program to determine status of a device
  – control operations
    • direct device to perform non-transfer device dependent actions (e.g., positioning read heads)
  – I/O operations
    • perform the actual transfer of data between device and CPU
• Polling latency determines the maximum transfer rate
  – Limited by available memory bandwidth
  – Latency is unpredictable
Basic I/O Techniques: Programmed, Interrupt, DMA
Interrupt-driven control

• Interrupt-driven control is more common

• Three types:
  
  A. interrupt-driven program-controlled
  • device interrupts, program suspended, handler performs the data transfer

  B. interrupt-driven program-initiated (DMA)
  • though IO is initiated by the program, DMAC controls xfer, once complete, use (A) to transfer control

  C. interrupt-driven channel-program controlled
  • reduces main CPU involvement even further
  • devices and hardware channel, channel program, IO instructions; channel program initiated by application.
  • Channel program (or scripts) normally consist of one or more channel control words that are decoded and executed one at a time by the channel.

• DMA and Channel programs can cause cycle stealing from the processor.
  – This may make it difficult to estimate the WCET of a program.
Elements required to support Interrupt-driven devices

- **Context switching mechanisms**
  - preserve the state of the processor immediately prior to the occurrence of the interrupt
  - place processor in new state to process the interrupt
  - restore suspended process after interrupt processing is complete.

- **State: process and processor**
  - Memory address of the current (or next) instruction
  - Program status information (current priority, memory protection, allowed interrupts etc.)
  - Contents of the programmable registers

- **Support for context switching in hardware ranges from**
  - basic (only PC)
  - partial (PC and PSW, to other status registers)
  - complete (full context is saved).

- **It may be necessary to supplement the HW actions by explicit SW support**
  - E.g., partial context switch may be ok for an interrupt handling model that views the handler as a procedure or subroutine.
  - Or the handler can be a separate process (with its own stack and data areas)
Elements Required (contd.)

• Identification of interrupting-device
  – vectored mechanisms
    • consists of a set of dedicated, contiguous memory locations (interrupt vector); and HW mapping of device addresses onto the interrupt vector
  – status mechanism:
    • each interrupt has an associated status word which specifies the device and reason for interrupt
  – polling mechanism:
    • identification mechanism includes interrogating the status of each device.

• With some processors, interrupt handling associated with a high-level language primitive
  – an interrupt is often viewed as a synchronization message down an associated channel.
  – So, a device is identified when the associated channel becomes active.
Elements required (contd.)

• identification of interrupt
  – once the interrupting device has been identified, the interrupt handling routine must determine *why* it generated the interrupt
  – this information can be supplied by either status information from the device or by having different interrupts from the same device through different vectored locations or channels.
Elements required (continued)

• control of interrupt: enable/disable interrupts by
  – status mechanisms that provide flags to enable/disable interrupts. These flags may be accessible by normal bit-oriented instructions.
  – mask interrupt control mechanisms that associate device interrupts with particular locations in an interrupt mask word. The interrupt mask word may be addressable by normal or special instructions.
  – level interrupt control mechanisms have devices associated with certain levels (that determines which devices may or may not interrupt).

• Static or dynamic priority control
A Simple Example IO System

• Each device has many different types of registers as are necessary for its operation.
• Memory-mapped control/status registers
  – Control and Status registers (CSRs) contain all info on device status
  – Data Buffer Registers (DBRs) for temporarily storing device data
    • 15-8 bits: unused; 7-0: data
    – a device may have more than one CSR and DBR
      • the exact number being dependent on the nature of the device.

<table>
<thead>
<tr>
<th>bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>Errors</td>
</tr>
<tr>
<td>11</td>
<td>Busy</td>
</tr>
<tr>
<td>10-8</td>
<td>Unit select</td>
</tr>
<tr>
<td>7</td>
<td>Done/ready</td>
</tr>
<tr>
<td>6</td>
<td>Interrupt enable</td>
</tr>
<tr>
<td>5-3</td>
<td>reserved</td>
</tr>
<tr>
<td>2-1</td>
<td>Device function</td>
</tr>
<tr>
<td>0</td>
<td>Device enable</td>
</tr>
</tbody>
</table>
Example continued

- On interrupt
  - processor stores PC and PSW on the system stack
  - PSW layout
    
    | Bit | Description                  |
    |-----|------------------------------|
    | 15 - 11 | Mode information          |
    | 10 - 8  | Unused                      |
    | 7 - 5   | Priority                    |
    | 4 - 0   | Condition codes             |

- The condition codes contain information on the result of the last processor operation.

- new PC and PSW are loaded from interrupt vector (two pre-assigned consecutive memory locations)
  - the first word contains the address of the ISR and the second word contains the PSW including its priority.
- A low priority interrupt handler can be interrupted by a higher priority interrupt.
Language Requirements

• Modularity and encapsulation facilities
  – portability of application code requires separation of machine dependent information
    • C: file
    • Ada: package
    • Modula-1: devices encapsulated in special modules

• An abstract model of device handling
  – device = an abstract processor performing some (fixed) task
  – system = set of processes with communication and synchronization.
  – Synchronization provided by interrupt control.
Abstract Models for Device Handling

- Need facilities for addressing, manipulating device registers
- A device register may be represented as
  - a program variable
  - an object
  - or (even) as a communications channel
- Need suitable representation of interrupts
  - procedure call – a remote procedure call from a device process
  - sporadic process invocation
  - asynchronous event
  - shared-memory based condition synchronization
  - message-based synchronization
- All, except procedure, view the handler as executing in the scope of a process, and therefore require a full context switch
  - Optimizations possible by restricting the handlers
Representation of Interrupt - 1

- **procedure call**
  - *interrupt is viewed as a procedure call*
    - an RPC coming from device process
    - any communication & synchronization must be programmed into the interrupt handler (procedure).
  - **non-nested procedures**: only *global data or state local to the handler* is accessible

- **sporadic process invocation**
  - *interrupt = request to execute a process*
  - *handler is a sporadic process*
  - *the process can access both local persistent data as well as global data*
    - if shared variable communication is available.
• Asynchronous event to a process
  – the interrupt handler has access to local state of the process and global data; after handling, resumption or termination of process.

• Shared variable condition synchronization within a shared variable synchronization mechanism
  – For example, as a signal operation on a semaphore; the handler can access both local state of the process and the global state.

• Message based synchronization
  – a contentless message on a channel (Occam). The handler (receiving process) can access the local state of the sending process.
Various Ways of Building Device Drivers

• C and C++
  – use a procedural model with variables as device registers
  – for sequential systems, an asynchronous event model is equivalent to the procedural model as there is only one process to interrupt (therefore, there is no need to identify the process)

• Ada
  – hybrid between procedural and shared variable condition synchronization model
    • ‘protected’ procedure calls represent interrupts, variables used for device registers
    • Each interrupt has a unique ID supported by the system (may be address of the hardware interrupt vector)

• Occam2
  – message-based model
    • devices registers as PORTS
    • interrupts are messages over a channel.
Device Driving in C

- Device registers are addressed by pointer variables which can be assigned to the memory location of the register
- Device registers are manipulated by low-level bitwise logical operators
- Example: unsigned short int *reg; reg = 0AA12;
  - assign n bits starting at position p in register pointed at by reg to x

```c
unsigned int setbits(unsigned int *reg, unsigned int n, 
unsigned int p, unsigned int x)
{
   unsigned int data, mask;
   data = (x & (~(~0 << n))) << (p); /* data to be masked in */
   mask = (~(~0 << n)); /* mask */
   *reg &= ~(mask << (p)); /* clear current bits */
   *reg |= data; /* or in data */
}
```

- interrupt handling
  - assign interrupt handler by placing address of a parameterless procedure in the appropriate interrupt vector location
  - explicit communication, synchronization with the rest of the program.
Various Ways of Building Device Drivers

- **Modula-1**
  - Modula-1 was one of the first languages with facilities for programming device drivers
  - Unit of modularity: module
  - Special type of module: interface module
    - It has properties of a monitor, can be used to control access to shared resources
    - Processes interact via signals (condition variables)
  - Special type of module: device module
    - Special type of interface module, can be used to encapsulate interaction with a device
    - Only from within a device module can one use the facilities for handling interrupts (device process)
    - Each device module has a hardware priority specified in its header
    - A device process holds the monitor lock using the ceiling priority of the device module
Device Drivers

• Ada
  – Three ways tasks can synchronize and communicate with each other
    • Through the rendezvous
    • Using protected units
    • Via shared variables
  – Ada assumes that the device and the program have access to the shared-memory device registers
  – Preferred method of writing device drivers is to encapsulate them into a protected unit.
Device Driving in Ada

• Facilities for addressing and manipulating device registers
  – through REPRESENTATION CLAUSES
  – These define how types of the language are to be mapped onto the underlying hardware.

• A representation clause can be
  – attribute definition clause: size, alignment, storage space for tasks, address
  – enumeration representation clause: internal values for literals
  – record representation clause: offsets and lengths of components

```ada
type Error_T is (Read_Error, Write_Error, Power_Fail, Other);

type Function_T is (Read, Write, Seek);

type Unit_T is new Integer range 0 .. 7;

type Csr_T is record
  Errors  : Error_T;
  Busy    : Boolean;
  Unit    : Unit_T;
  Done    : Boolean;
  Ienable : Boolean;
  Dfun    : Function_T;
  Denable : Boolean;
end record;
```

**Enumeration clause**
• specifies the internal codes for the literals of the enumeration type.
  01 - Read
  10 - Write
  11 - Seek

```ada
for Function_T use (Read=>1,Write=>2,Seek=>3);
```
Representation Clauses

• Record representation clause
  – specifies the storage representation of records; that is, the order, position and size of its components.

```pascal
Word : constant := 2; -- no. of bytes in a word
Bits_In_Word : constant := 16;
for Csr_T use
  record
    Denable at 0 range 0..0;
    Dfun at 0 range 1..2;
    Ienable at 0 range 6..6;
    Done at 0 range 7..7;
    Unit at 0 range 8..10;
    Busy at 0 range 11..11;
    Errors at 0 range 12..15;
  end record;
for Csr_T’Size use Bits_In_Word;
for Csr_T’Alignment use Word;
for Csr_T’Bit_Order use Low_Order_First;
```
Register definition and placement in memory

Tcsr : Csr_T;
for Tcsr’Address use
System.Storage.Elements.To_Address( 8#177566#);
Tmp :Csr_T;

- The hardware register can be manipulated

Tmp := (Denable => True,
Dfun => Read,
Ienable => True,
Done => False,
Unit => 4,
Errors => None );
Tcsr := Tmp;
- to ensure all bits are set at once
- To test for errors

if Tcsr.Error = Read_Error then
  raise Disk_Error;
end if;

In effect, the object Tcsr is a collection of shared variables between the device control task and the device itself. Mutual exclusion between these two processes is achieved by using a protected object.
Ada Interrupt Handling

• The occurrence of an interrupt (HW or SW) consists of
  – interrupt generation
    • underlying hw/sw mechanism that makes an interrupt available to the program
  – interrupt delivery
    • action that invokes a part of the program (interrupt handler)
• In between generation and delivery, the interrupt is blocked.
  – The handler is invoked once for each delivery of an interrupt
  – when blocked: all future occurrences are prevented.
• Reserved interrupts: handled directly by the runtime system
  – e.g., clock interrupt used to implement the delay statement.
• Each non-reserved interrupt has a default handler assigned by the runtime system
  – ceiling priorities are used in interrupt priority.
Handling Interrupts using Protected Procedures

- An interrupt handler in Ada is a parameterless protected procedure
- Each interrupt has a unique discrete id supported by the system
- Implementation of this unique id is hardware dependent
  - for example, the address of hw interrupt vector associated with the interrupt
- Identifying interrupt handling protected procedures is done using one of two pragmas:
  - pragma Attach_Handler (Handler_Name, Expression)
  - pragma Interrupt_Handler (Handler_Name)
Example

• An ADC that has a 16-bit result register at hardware address 8#150000# and a control register at 8#150002#

• 16-bit machine with CSR:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A/D Start</td>
<td>Set to 1 to start a conversion.</td>
</tr>
<tr>
<td>6</td>
<td>Interrupt</td>
<td>Set to 1 to enable interrupts</td>
</tr>
<tr>
<td>7</td>
<td>Done</td>
<td>Set to 1 when conversion is complete.</td>
</tr>
<tr>
<td>8-13</td>
<td>Channel</td>
<td>64 analogue inputs</td>
</tr>
<tr>
<td>15</td>
<td>Error</td>
<td>Set to 1 by the converter if device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>malfunctions.</td>
</tr>
</tbody>
</table>

• Driver as a protected type within a package
  – so that its interrupt processed as a protected procedure call
  – allows more than one ADC handling.

```vhdl
package ADC_Device_Driver is
  Max_Measure : constant := (2**16)-1;
type Channel is range 0..63;
subtype Measurement is Integer range 0..Max_Measure;
procedure Read(Ch: Channel;
               M : out Measurement());
  -- potentially blocking
Conversion_Error : exception;
private
  for Channel'Size use 6;
  -- indicates that six bits only must be used
end ADC_Device_Driver;
```
with Ada.Interrupts.Names; use Ada.Interrupts;
with System; use System;

package body ADC_Device_Driver is

   Bits_In_Word : constant := 16;
   Word : constant := 2; -- bytes in word

   type Flag is (Down, Set);

type Control_Register is
   record
      AD_Start : Flag;
      IEnable : Flag;
      Done : Flag;
      CH : Channel;
      Error : Flag;
   end record;

for Control_Register use
   record
      AD_Start at 0#Word range 0..0;
      IEnable at 0#Word range 6..6;
      Done at 0#Word range 7..7;
      CH at 0#Word range 8..13;
      Error at 0#Word range 15..15;
   end record;

for Control_Register'Size use Bits_In_Word;
   -- the register is 16 bits long
for Control_Register'Alignment use Word;
   -- on a word boundary
for Control_Register'Bit_Order use Low_Order_First;

   type Data_Register is range 0 .. Max_Measure;

for Data_Register'Size use Bits_In_Word;
   -- the register is 16 bits long

Contr_Reg.Addr : constant Address :=
   System.Storage.Elements.To_Address(8#150002#);
Data_Reg.Addr : constant Address :=
   System.Storage.Elements.To_Address(8#150000#);

ADC_Priority : constant Interrupt_Priority := 63;

Control_Reg : aliased Control_Register;
   -- aliased indicates that pointers are used to access it
for Control_Reg'Address use Contr_Reg.Addr;
   -- specifies the address of the control register

Data_Reg : aliased Data_register;
for Data_Reg'Address use Data_Reg.Addr;
   -- specifies the address of the data register
protected type Interrupt_Interface(Int_Id : Interrupt_Id;
   CR : access Control_Register;
   DR : access Data_Register)
is
  entry Read(Chan : Channel; M : out Measurement);
private
  entry Done(Chan : Channel;
   M : out Measurement);
procedure Handler;
pragma Attach_Handler(Handler, Int_Id);
pragma Interrupt_Priority(Adc_Priority);
Interrupt_Occurred : Boolean := False;
Next_Request : Boolean := True;
end Interrupt_Interface;

ADC_Interface : Interrupt_Interface(Names,Adc,
   Control_Reg'Access,
   Data_Reg'Access);
-- this assumes that 'Adc' is registered as an
-- Interrupt_Id in Ada.Interrupts.Names

protected body Interrupt_Interface is
  entry Read(Chan : Channel; M : out Measurement)
   when Next_Request is

  Shadow_Register : Control_Register;

  begin
    Shadow_Register := (Ad.Start => Set, IE =>
      Set, Done => Down, Ch => Chan, Error => Down);
    CR.all := Shadow_Register;
    Interrupt_Occurred := False;
    Next_Request := False;
    requeue DONE;
  end READ;

procedure Handler is
begin
  Interrupt_Occurred := True;
end Handler;

entry Done(Chan : Channel; M : out Measurement)
  when Interrupt_Occurred is

  begin
    Next_Request := True;
    if CR.Done = Set and CR.Error = Down then
      M := Measurement(DR.all);
    else
      raise Conversion_Error;
    end if;
  end Done;
end Interrupt_Interface;

• Inside the Read entry, the control register, CR, is set up with appropriate
  values.

• Once the control register has been assigned, the client task is requeued on
  a private entry to await the interrupt.

• The Next_Request flag is used to ensure only one call to Read is outstanding.

• Once the interrupt has arrived (as a parameterless protected procedure call),
  the barrier on the Done entry is set to true.

• This results in the Done entry being executed (as part of the interrupt
  handler), which ensures that CR.Done has been set and that the error flag has
  not been raised. If this is the case, the out parameter M is constructed, using a
  type conversion, from the value on the buffer register.

procedure Read(Ch : Channel; M : out Measurement) is
begin
  for I in 1 .. 3 loop
    begin
      Adc_Interface.Read(Ch,M);
      return;
      exception
      when Conversion_Error => null;
    end;
  end loop;
  raise Conversion_Error;
end Read;
Types of Device Drivers

• **Basic need**
  – **Ensure mutual exclusivity of device access**
    • only one application task can request an input or output operation on a specific device
    • typically done using (binary) semaphores
  – **Usually it can be done right inside the device driver**
    • The driver executes semaphore’s `get` as soon as it is called by an application task
    • After completion of device operation, the driver `puts` the semaphore
    • Sometimes, access is granted on a basis of a “session”
      – Even though a driver may only do a small subset of the device operation, so it `opens a session...closes a session`
      – A session bundles many IO operations.
  – **Can also be source of unbounded priority inversion**
    • Use mutex, monitors

• **Types**
  – **Synchronous, Asynchronous and Hybrid**
Synchronous versus Asynchronous

• What does the requesting application task do, while a driver is in action?

• Synchronous driver
  – Requesting task waits for driver IO operation to complete
  – That is, the RTOS is blocked, or suspended
    • Other tasks may continue to execute, ISR may continue

• Asynchronous driver
  – Requesting tasks continues while driver IO is happening

• There are two different blocking actions on some drivers
  – Waiting for device operation to complete
  – Waiting for access to a device
Synchronous IO

- Now need two locks (semaphores): mutex, block
- The task treats the entry point area of the driver as a subroutine call
Asynchronous IO

- Requesting tasks processes previous data
  - While HW is working on the new data
- Calling task is not blocked
  - Overlapped IO and execution
- Need a place to hold data, say, a mailbox.

Now, the driver initialization operation needs to create a message queue. When is a message delivered to the queue?
Free-Running Asynchronous Input

• Hardware is doing repeated interrupts **without** an explicit request from application/driver

• The queue may soon fill up with really old data…
One Possibility

- Handle only latest input from hardware
  - Use a semaphore to protect shared data area

- Problem: ISR may not get the semaphore since some RTOS disallow ISRs to do P() on a semaphore
  - Solution: Device driver may disable interrupt
Latest input only
Asynchronous Output

- Deposit data on a queue

```c
• ADevWrite()
ASYNC_OP:
BEGIN
  queue_receive ( STAT, Wait);
  /* Wait for H/W done/status */
  IF( STAT is O.K. )
    Start next Output Operation;
  ELSE ...
END
```

```c
• Device Interrupt Service
DEVICE_ISR:
BEGIN
  Transfer Status from H/W;
  queue_send (STAT);
  /* Send new status */
END
```

[Diagram of asynchronous output process]
Output Spooling

What happens when we run out of data to output? The ISR quits and then there is no way to send data to the output device even when we have it available, since the hardware is no longer interrupting. One way to solve it is to revive the output spooling by splitting ISR into two parts...
Priming Pump?

- "Priming the Pump"

```
request a buffer /* pt_getbuf() */
fill buffer with string for printing
put buffer pointer into a message
put character count into same message
queue this message /* queue_send() */
if semaphore is set /* sm_getnowait() */
then call 'Get More Output' directly
endif
```
Abstracting Drivers

• Problem
  – Each SBC generally comes with its own board support package (BSP)
  – There are over 450 CPU boards with many processor variants
  – If you make a peripheral, you have to customize its drivers for each of these BSPs

• This can be simplified by adding an abstraction layer between the device driver and the BSP
  – Allows design of one set of device drivers for all BSPs
  – No need for recompiling drivers for a given board

Steve Finseth, Embedded Systems Programming, May 2004
Abstraction Layer

- Includes software routines that allow the OS to manipulate hardware
  - BSPs actually handle the access to HW resources such as memory, IO, interrupts and device registers.
- A device driver is written for a specific OS and BSP
- BSP abstraction for the device driver
  - Functions like an API for the driver
- D2B defines routines for
  - Handling interrupts, address translation, memory reads and writes, clocking functions.
  - Pulls out of the device driver all HW dependent routines.
Abstracting Interrupts

• Interrupt code is at D2B with no hardware specific references

```c
STATUS connect_irq( int vector, voice *routine, int parameter ) {
    STATUS return_val = OK;

    #if defined (PCPENTIUM_BSP)
        return_val = pciIntConnect( //describe the way PENTIUM BSP places interrupt vector
    ...
    #elif define (MCP750_BSP)
        return_val = intConnect(...
    #endif

    return (return_val);
}
```

- A “generic” interrupt routine with code snippets that modularize on BSP packages.
- E.g., since there is no common definition for the interrupt vector, each BSP defines different what information re interrupt vector is presented.
Abstracting Address Translations

• Machine addresses are generated either through table lookup for algorithms
• Once again use “generic” routines to convert addresses back and forth
  – As seen from the processor (local memory)
  – As seen from the IO device (bus address)

– Functions:
  • Local2bus_adr()
  • Bus2local_adr()
/*****************************************************************************
**  Prototypes for BSP to device driver API
*****************************************************************************/
STATUS sbs_pci_find_device(int vendor_id, int device_id, int index, int *p_bus_num, int *p_device_num, int *p_func_num, int *p_bus_type);
STATUS sbs_local2bus_adrs(int adrs_space, char *local_adrs, char **p_bus_adrs, int length);
STATUS sbs_bus2local_adrs(int adrs_space, char *bus_adrs, char **p_local_adrs, int length);
STATUS sbs_connect_irq(int int_line, void *routine, int parameter);
STATUS sbs_disconnect_irq(int int_line, void *routine);
STATUS sbs_disconnect_irq2(int int_line, void *routine, int parameter);
STATUS sbs_enable_irq(int int_line);
STATUS sbs_disable_irq(int int_line);
UINT8 sbs_pci_io_read8(char *p_io);
UINT16 sbs_pci_io_read16(char *p_io);
UINT32 sbs_pci_io_read32(char *p_io);
void sbs_pci_io_write8(char *p_io, UINT8 value);
void sbs_pci_io_write16(char *p_io, UINT16 value);
void sbs_pci_io_write32(char *p_io, UINT32 value);
UINT8 sbs_pci_mem_read8(char *p_memory);
UINT16 sbs_pci_mem_read16(char *p_memory);
UINT32 sbs_pci_mem_read32(char *p_memory);
void sbs_pci_mem_read_from_device(char *p_local, char *p_device, int bytes);
void sbs_pci_mem_write_to_device(char *p_device, char *p_local, int bytes);
char * sbs_get_model_name(void);
char * sbs_get_bsp_rev(void);
int sbs_get_sys_clk_rate(void);
Using C for Low Level Programming

Examples and Snippets

References:
Lewis, Fundamentals of Embedded Software
Barr: Programming Embedded Systems in C, C++
Example: Blinking Lights (on 80188EB)

```c
void main (void) {
    while (1) {
        toggleLed(LED_GREEN);
        delay(500); } }

LED status controlled by a bit in the on-chip P2LTCH register located in the IO space at offset 0xFF5E
→ use inline assembly to flip this bit

#define LED_GREEN 0x40 /* bit 6 of this register */

void toggleLed(unsigned char ledMask) {
    asm { mov dx, P2LTCH /* load address of P2LTCH */
         in al, dx /* read contents of register */
         mov ah, ledMask /* move the ledMask into a register */
         xor al, ah /* flip bit */
         out dx, al }; } /* write output */

If P2LTCH was in the memory space it would be easier, e.g.,

volatile unsigned short *pP2LTCH = (unsigned short *) 0x7200005E
void toggleLed (void) { *pP2LTCH ^= LED_GREEN; }
```
Startup Code

• Generally an assembly language file, startup.asm, crt0.s

• Functions
  – Disable all interrupts
  – Copy any initialized data from ROM to RAM
  – Zero the uninitialized data area
  – Allocate space for and initialize the stack
  – Initialize the processor’s stack pointer
  – Create and initialize the heap
  – C++: Execute the constructors and initializers for all global variables
  – Enable interrupts
  – Call main()
    • Recovery code
Compiling, Linking and Loading

- Linker (Gnu ld) plus open library for standard functions
  *(newlib on Cygnus)*
  - Merge object files into a single object file
    - text (code), data (initialized vars) and bss (uninitialized vars) sections
    - Resolve references
  - Produce a *relocatable program*
    - No memory addresses have been assigned to the code and data sections of the program

- Locating program
  - Assign physical memory addresses to code, data sections
  - Produce binary image (sometimes including OS) to be loaded into the target ROM

- Gnu ld can do the job by passing to it memory information as a linker script
Example

- An embedded target with 512 KB each of RAM and ROM

MEMORY {
    ram: ORIGIN = 0x00000, LENGTH = 512K
    rom: ORIGIN = 0x80000, LENGTH = 512K
}

SECTIONS {
    data ram: {
        _DataStart = .;
        *(.data)
        _DataEnd = .;
    } > rom
    bss: {
        _BaseStart = .;
        *(.bss)
        _BaseEnd = .;
    }
    _BottomofHeap = .;
    _TopOfStack = 0x80000;
    text rom: {
        *(.text)
    }
}
Structure of Device Drivers

In general, there are five components:

1. A data structure that overlays the memory-mapped control and status registers of the device
2. A set of variables to track the current state of the hardware and device driver
3. A routine to initialize the hardware to a known state
4. A set of routines that provide an API for the user of the device driver
5. One or more ISRs
C allows mixing with assembly

• Generally not a good idea but sometimes
  – Speed sensitive, predictable code (with specialized ISA)
  – Accessing hardware where the library fails

• C supports this by
  – Inline assembly
  – Use of symbolic registers (even in assembly)
  – In X86, “Protected-mode” compilation requires adhering to some conventions:
    • Functions return all pointers and integer values upto 32 bits in EAX
    • Functions return 64-bit values in EDX:EAX pair
    • EBP is used to access function arguments
    • EBX, ESI, EDI, EBP, DS, ES, SS must be preserved by functions written in assembly language
    • EAX, ECX, EDX, FS, GS can be used as “scratch” registers.