EE458 - Embedded Systems
I/O and Memory Management

• Outline
  – The I/O Subsystem
  – Memory Management

• References
  – RTC: Chapters 12, 13
  – CUG: Chapters 18, 19, 20
I/O and Memory Management

Introduction to I/O

- All embedded systems perform some form of I/O. Touch screens, numeric displays, video terminals, temperature sensors, keyboards, network cards, sound cards, disk drives are all examples of I/O devices.

- Each device is unique. Each device uses particular I/O ports or memory addresses. Data may be written/read either a single word at a time or in blocks.
I/O and Memory Management

The I/O Subsystem

• Most OSes provide an I/O subsystem that shields the application developer from the specific details of communicating with the device.

• A **device driver** must be written for each device by a systems software developer. The **device driver** provides a standard set of functions that are used by the I/O subsystem.
I/O and Memory Management

The I/O Subsystem

The Layered Software Model

- Application Software
- I/O Subsystem
- Device Drivers
- Interrupt Handlers
- I/O Device Hardware
I/O and Memory Management

Basic I/O Concepts

- At the bottom layer of the I/O subsystem is the I/O device hardware. All I/O devices must be initialized by writing to the device control registers.

- Some processors (Intel) support port-mapped I/O in which special processor instructions (IN and OUT) are used to read and write to the ports. The devices occupy a separate address space than memory.
I/O and Memory Management

Basic I/O Concepts

- Other processors (Motorola) use memory-mapped I/O in which the device space is part of the system memory address space. The same instructions that are used to transfer data between the CPU and memory (MOVE) are used to transfer data between the CPU and the I/O devices.

- With direct memory access (DMA) the CPU is not directly used when transferring data between memory and I/O devices.
I/O and Memory Management

Basic I/O Concepts

• Character-mode devices transfer data a single byte at a time. Examples: serial port, parallel port, keyboard. The driver may need to buffer the data if the device is slow.

• Block-mode devices transfer data in blocks (512 B, 1024 B, etc). Examples: hard disks, tape drives. When large amounts of data are written the device driver must break the data into blocks of the required size.
I/O and Memory Management
Basic I/O Concepts

- The I/O subsystem presents a standard interface (API) to the applications programmer for all I/O devices.
- Each device driver must implement the functions in the API.
- There are typically commands to create, open, read, write, close, and destroy a device. An I/O control function (ioctl()) is used to send special commands to a device.
I/O and Memory Management
Basic I/O Concepts

- The create routine creates a virtual instance of an I/O device. The I/O routines do not operate directly on the I/O device, but rather on the device driver.
- After the virtual instance has been created, the device is available for subsequent operations (open, read, write, etc).
- A virtual instance is deleted by a call to the destroy routine.
Each device driver provides the actual implementation of each function in the uniform I/O set. For example, the TTY (serial) device driver may provide a function named tty_open() that is called when the application opens a TTY device.

The parallel port driver would similarly provide a par_open() routine.
I/O and Memory Management
Basic I/O Concepts

• A **driver** table is used to associate each routine in the API with a device specific routine. (Create() to tty_Create() for example. See figure on the following slide.)

• A driver table entry is created when the device driver is loaded.

• The create routine creates an entry in the device table. An entry in the device table contains a pointer to the appropriate routines in the **driver** table
The Driver Table maps API calls to device specific routines.
The Device Table contains a pointer to the appropriate routines in the Driver Table.
The RTEMS I/O manager provides the following directives: `rtems_io_initialize`, `rtems_io_open`, `rtems_io_close`, `rtems_io_read`, `rtems_io_write`, `rtems_io_control`, `rtems_io_register_name`, and `rtems_io_lookup_name`.

Each device driver may contain the following entry points: **Initialization**, **Open**, **Close**, **Read**, **Write**, and **Control**.
I/O and Memory Management
RTEMS I/O

- Device driver entry points are stored in the Device Driver Table. If a driver does not support an entry, then that entry should be NULL in the Driver Table.

- Device drivers can be registered and unregistered. That allows device drivers to be linked into the Driver Table at run time.

- The CONFIGURE_MAXIMUM_DRIVERS parameter defines the Driver Table length.
A virtual instance of a device is identified by two numbers known as the major and minor device numbers. The major number identifies a particular device driver. The minor number is used to differentiate between multiple virtual instances of a device all handled by a single driver.

Device drivers run in the task environment. A driver may invoke other RTEMS directives. If a driver blocks the invoking task will block.
I/O and Memory Management

RTEMS I/O

- We will not be writing RTEMS device drivers in this course, but an understanding of the RTEMS I/O Manager is useful when working with existing device drivers.

- The number and type of device drivers is BSP dependent. A BSP might provide serial (console in RTEMS), clock (PIT), real-time clock, filesystem, and networking drivers.

- See CUG Ch. 20 and the “RTEMS BSP and Device Driver Devel. Guide” (bsp-howto.pdf)
I/O and Memory Management
Memory Management

- Memory not occupied by program code, program data or the stack may be used by the RTOS for dynamic memory allocation. This area of memory is called the heap.

- Information about the heap is stored in a control block. Typically the starting address, the heap size and a memory allocation table are stored in the control block.
The allocation table indicates which areas of the heap are in-use and which are free.

In C the `malloc()` and `free()` routines are used for allocating variable sized memory blocks.

An example allocation table is shown on the following slide. Here the table is only a byte in length. Each bit indicates whether a 32 byte segment of memory has been allocated or is free.
## I/O and Memory Management

### Memory Management

#### Snapshots of a Memory Allocation Table

<table>
<thead>
<tr>
<th>Block</th>
<th>Allocation/Free</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 1 1 1 0 0 0</td>
<td>256 bytes</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1 1 1 1 0 0</td>
<td>A = malloc (120) 1 free block = 128 bytes</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1 1 1 1 1 0</td>
<td>B = malloc (20) 1 free block = 96 bytes</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 1 1 1 1 1</td>
<td>C = malloc (50) 1 free block = 32 bytes</td>
</tr>
<tr>
<td>5</td>
<td>1 1 1 1 0 1 1 1</td>
<td>D = malloc (32) No free blocks left</td>
</tr>
<tr>
<td>6</td>
<td>1 1 1 1 0 1 1 0</td>
<td>free(B) 1 free block = 32 bytes</td>
</tr>
<tr>
<td>7</td>
<td>1 1 1 1 0 0 0 0</td>
<td>free(D) 2 free blocks, 32 bytes each</td>
</tr>
<tr>
<td></td>
<td></td>
<td>free(C) 1 free blocks = 128 bytes</td>
</tr>
</tbody>
</table>
// malloc() and free() example
float *array;
int N = 100;
array = (float *)malloc(N*sizeof(float));
if (array == (float *)NULL)
    fatal_error();
// Use the memory
array[50] = 34.5;
// Release memory when done
free(array);
I/O and Memory Management

RTEMS Memory Managers

- Memory managers provide for allocation of fixed size or variable sized blocks.
- The RTEMS Partition Manager is used to dynamically allocate memory in fixed-size units (called buffers). The Region Manager is used to allocate memory in variable sized units (called segments).
- You can create multiple partitions or regions. Partitions and regions will typically be created out of memory allocated via malloc.
The Region manager can (optionally) force tasks to wait (block) until memory is available.

Tasks will not block when requesting buffers from the Partition manager. (The status return indicates whether or not memory is available.)
I/O and Memory Management

RTEMS Memory Managers

- The Partition Manager is useful when implementing flip buffers.
- The maximum number of partitions and regions is configured via parameter settings.
- Buffers and segments may be shared. Access is usually protected by a mutex.
- See CUG Chapters 17 and 18 for details regarding available RTEMS directives.