CS 5372
Specification and Design of Real-Time Systems

Lecture 2: Platforms & RTOS

1

Lecture 2

- Outline
  - Lab Setup (20 min)
  - Labs work
    - Workbench + vxWorks Documentations (15 min)
    - Project Management (25 min)
    - Host Shell (25 min)
  - Lecture: Platform + RTOS

2

Quiz

Microprocessor Primer (1)

- Microcomputer components:
  - central processing unit: CPU
  - memory: RAM, (P)ROM
  - input/output devices: I/O
  - interrupt controller
  - real-time clock, timers
  - system bus: data, address, control
- CPU components;
  - program counter (PC)
  - arithmetic-logic unit (ALU)
  - internal registers control unit (CU)
  - bus interface unit (BIU)

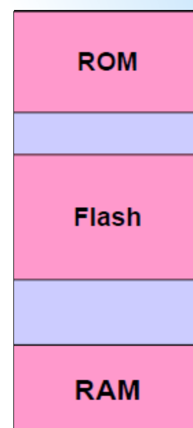
Microprocessor Primer (2)

- 8/16/32/64 bit wide data path: 64KB-4GB memory
- Programs are stored in the memory: downloaded to RAM and/or permanent in (E)(P)ROM, FlashRAM
- Instruction pointed by PC is processed in Fetch-Decode-Execute cycle
- Internal registers used for holding data/address while device registers have addresses mapped to memory
- Compiled programs go through link/locate process to be loaded into RAM relative to the program load point or into predefined memory locations in embedded systems (P)ROM
- Interrupts triggered by external or internal events
- Interrupt processing sequence saves the current context and loads PC with relevant address kept in interrupt pointer table

5

Memory Address Assignment

- Memory chips and peripheral device registers are mapped into the processor address space (e.g. 0x00000 to 0x1FFFF)
- Memory map keeps the address ranges
- Developer needs to know microprocessor start-up characteristics to properly place boot ROM and the interrupt vector area
- Each downloaded program, permanent in (E)PROM/FlashRAM or temporary in RAM, has a predefined load address
- An appropriate symbol table constitutes the record of stored programs and other software objects



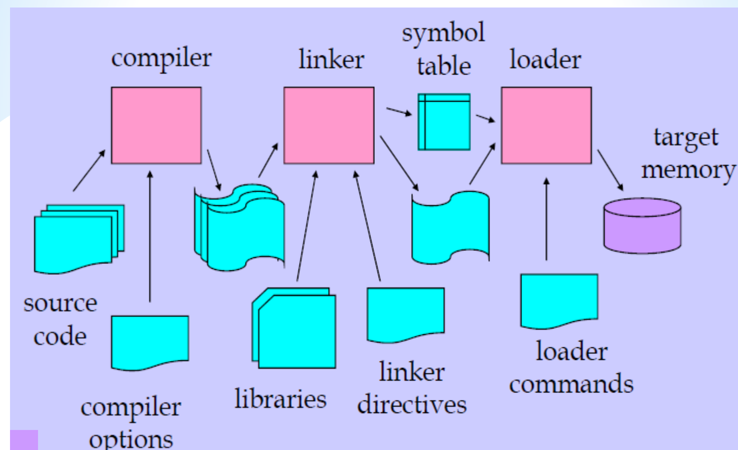
6

Location of Program and Data

- What are the three components of a loadable application program?
  - .text – assigned for the program code
  - .data – assigned for initialized data
  - .bss – assigned for un-initialized (null) data
- Where would you typically store program code (.text)? Why?
- Where would you typically store data (.data & .bss)? Why?
- The location and sizes are defined by hexadecimal address (start and end of the memory area)

7

Development: Object Code Location



8

Development: Loading and Debugging

- Created object files in ELF (executable/linking format)
- include well defined sections including the respective addresses of target memory
- Executable image is loaded into target memory
- Embedded system debuggers often reside on the host with only minimal debug kernel (target agent) running on the target
- Host side has knowledge of source, object, symbols, and cross-references
- Target side works like an interrupt service when receiving a command from communication link
- Any debugging is intrusive and may change the timing characteristics of the application

9

Embedded System Initialization

- Questions:
  - How to load the image onto the target system?
  - Where in the memory to load the image?
  - How to initiate program execution?
  - How the program produces recognizable output?
- Image transfer - an executable image built for target system can be transferred from the host development system onto the target (loading the image)
- Image resides in the EEPROM or flash memory
- Downloading the image onto the target using serial or network connection
- Downloading the image via JTAG or BDM interface
  - JTAG (Joint Test Action Group)
  - BDM (Background Debug Mode)

10

Three System Initialization Phases
(What are they?

- Hardware initialization:
  - Start execution at the reset vector
  - Put processor in a known state
  - Get clock speed
  - Initialize memory
  - Etc.
- RTOS initialization:
  - Initialize RTOS objects and services (tasks, interrupts, semaphores, timers, message queues, etc.)
  - Create stack for RTOS
  - Initialize file system (if applicable)
  - Initialize TCP/IP stack (if applicable)
  - Etc.
- Application software initialization:
  - RTOS transfers the control to the application
  - Application starts its tasks, the kernel scheduler runs, sets up timers, semaphore, etc., to perform the application work

11

Three Image Execution Scenarios
What are they?

- Executing from ROM using RAM for data
  - Limited memory, no download of .text to RAM, but
  - .data must still be loaded in the RAM
  - Boot image contains the .text section (at reset there is where the computer start the execution)
- Executing from RAM after image transfer from ROM
  - Compressed image permanently in ROM is decompressed by the loader and copied to the RAM
- Executing from RAM after image transfer from host
  - The target debug agent transfers an application image from the host into the RAM
  - The target agent is aware of RTOS resources
  - NOTE: *VxWorks system uses debug agent*

12

Simple Solutions for Real-Timeness: 1) Cyclic Executive

- "A big loop" – time-driven sequential execution of tasks assigned to predetermined time slots
- Good predictability and determinism
- Difficult modifications when new task needs to be added
- Response to events may be delayed
- Difficult design for variable task periods and sporadic tasks

13

Example:
- Embedded system designed to monitor a temperature sensor and update an LCD display.
- The LCD may need to be written ten times a second (i.e., every 100 ms).
- If the temperature sensor must be read every 50 ms for other reasons, we might construct a loop of the following appearance:

```
int main(void) {
        // initialization code here
        while (1) {
                currTemp = tempRead();
                lcdWrite(currTemp);

                // waste CPU cycles until 50 ms
                currTemp = tempRead();
                // do other stuff

                // waste CPU cycles until 100 ms
                }
            }
```

14

Simple Solutions for Real-Timeness: 2) Interrupt Driven Executive

- Each task is triggered (activated) by interrupt (HW or SW)
- No synchronization possible as only one task is running to completion
- An interrupt can happen at any time (Asynchronous).
- CPU invokes the Interrupt service routine (ISR).
- Around the code that reads/writes to the shared resources interrupts are disabled in the application.
- Synchronization mechanisms cannot be used in ISR, because ISR should not wait indefinitely.
- When interrupts are disabled, the system's ability to receive stimuli from the outside world is minimal.
- It is important to keep the critical sections of code in which the interrupts are disabled as short as possible.
- A snapshot of the machine – called the context – must be preserved upon switching tasks so that it can be restored upon resuming the interrupted process.

Simple Solutions for Real-Timeness: 3) Background-Foreground Executive

- Main program loop (background)
- Foreground tasks triggered (activated) by interrupts
- The time-critical operations performed in the interrupt routines
- Limited scalability but perfect in dedicated applications
- Good response time but the same problems as with interrupt-driven

Kernel-Based RT Solutions (1)

- Multitasking: scheduling and switching the processor between several tasks
- Separate tasks of a defined priority are created and maintained by the kernel
- The kernel serves as the intermediary between the application and hardware
- The task control (assigning and unassigning the CPU) is done by kernel
- The hardware supports the kernel via clock ticks and external interrupts

17

Kernel-Based RT Solutions (2)

- Arguments FOR:
  - **scalability -** tasks can be added and removed
  - **modularity -** tasks can be configured into different modules
  - **protection -** tasks live in a separate virtual space
- Arguments AGAINST:
  - **size -** separate tasks require more memory
  - **time -** operating system adds overhead scheduling
  - **cooperation -** external communication required

18

Priority-Based RT Kernels

- NON-PREEMPTIVE:
  - Running task (assigned to the CPU) execute to completion
  - Newly activated task is assigned to **ready queue**
  - The queue assignment is according to task priority
  - When executing task completes the task at the front of the ready queue gets executed
- PREEMPTIVE:
  - Executing task may be preempted by activation of a higher priority task
  - It occurs at a preemption point (at clock tick)
  - The preempted task returns to the ready queue according to its priority level
  - Upon completion of the highest priority task the next (highest priority) task is serviced

19

What is an RTOS?

- Operating System (OS) is software that interfaces the user application with the hardware
- History
  - Applications with no OS, machine code, simple systems
  - General Purpose GPOS: allows for abstraction of the specialized hardware from general application
  - UNIX: 60s-70s, developed for multi-user access to expensive, limited availability computing systems
  - PC era: 80s, Microsoft Windows and Apple OS
  - Industrial applications: late 80s, RTOS, VxWorks
- Real-Time Operating System is characterized having unique requirements in five areas:
  - Determinism
  - Responsiveness
  - User control
  - Reliability
  - Fail-safe operation

20

Why an RTOS?

- Asynchronous or synchronous events from multiple sources are implemented
- Outputs to control the environment conforming to **timing** constraints
- Many independent components - **concurrency**
- Components dependency - via **synchronization** and **communication**
- Dependability is critical (**exception handling)**
- Functional similarities, RTOS and GPOS
  - Some level of multitasking
  - SW and HW resource management
  - Provision of underlying OS services to applications
  - Abstracting the HW from SW applications

21

RTOS vs. GPOS

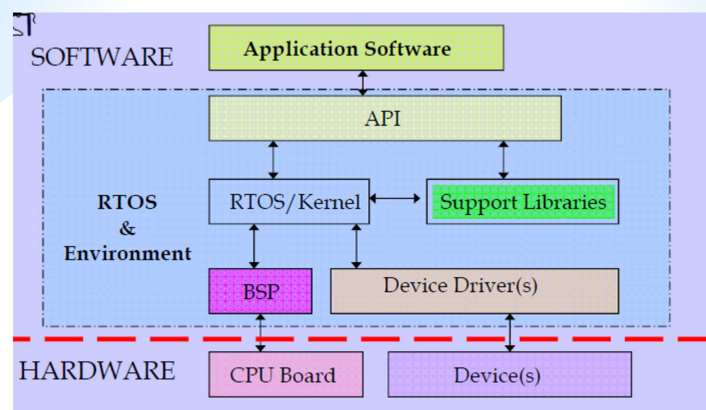
- Functional differences, RTOS and GPOS
  - Better reliability in embedded applications context
  - Ability to scale up or down to meet application needs
  - Faster performance
  - Reduced memory requirement
  - Scheduling policies tailored for RT embedded systems
  - Support for executables to boot and run from ROM or RAM
  - Better portability to different HW platforms
- A real-time operating system is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code

22

RTOS Components In Groups

- Scheduler – set of algorithms that determines which task executes when (round-robin, preemptive scheduling, etc.)
- Objects – kernel constructs for developer (tasks, semaphores, etc.)
- Services – operations that the kernel performs on the objects (timing, resource management, etc)

Place of RTOS Within an Embedded System



RTOS Application Programming Interface (API)

- API is the basic feature of a software system – a boundary between the application (functionality) and the system programming (hardware interactions)
- API defines how application program requests services of system software: via a function call, with a specific name and arguments
- RTOS provides services dealing with managing resources and interactions with the hardware (task control, memory management, synchronization and communication signals)
- The RTOS executes the function, typically in a kernel context returning the results to application
- The API is defined as a list of available calls with their input and output arguments

25

Typical RTOS/Kernel Structure

- **System Tables (Blocks):**
  - **Task-Control-Blocks** (TCB) - hold information necessary to run the tasks
  - **Device-Control-Blocks** (DCB) - hold the information necessary for the operating system to use the I/O devices
  - **Service-Control-Blocks** (SCB) - specify the parameters for a request for the operating system to perform some function
- **Scheduler?**
  - a program module that determines which task will run next
- **Dispatcher?**
  - a program module that prepares and activates scheduled program to run
- There are dedicated **Interrupt Service Routines** (ISR) for each I/O device and each specific event
- RTOS **Services** are implemented as interrupt service routines

26

Typical RTOS/Kernel Structure

- Each task has a Task Control Block (TCB)
- To start a task, the kernel:
  - builds a TCB
  - allocates stack space
  - passes the control to the task entry point
- Kernel places tasks in queues (ready, delayed, pending)
- Whenever a task changes state, the kernel is entered
- On exiting the kernel state, the scheduler is invoked
- To switch execution to a different task, the kernel must manage the context switch (must be very fast for R-T)
  - save context of the running task in TCB
  - restore context of the next task from TCB

27

How is the RTOS/Kernel entered?

- Task makes a **system call**
  - The kernel performs the requested action
- A task attempts to execute an **illegal instruction** or an illegal memory access
  - The kernel may kill the task, allow the task to continue, or call a special handler routine provided by the task
- An **external device** signals the computer for attention, by requesting an interrupt
  - The kernel will call a special handler routine to attend to the external device
- A **timer** (like an alarm clock) has expired, interrupting the CPU
  - The action taken by the kernel depends upon why the timer was set

28

Board Support Package (BSP)

- a dedicated software that allows the RTOS to initialize the board and utilize the hardware
- The typical services of BSP include system identification, initialization of registers, cache, memory, interrupts, timers, etc.

29

Resources Handled by RTOS

- Type: CPU time, main memory, devices
- Operations:
  - The RTOS may reserve access to these devices for use by one task
  - The RTOS may also perform the actual reading and writing of the devices
- Resources are managed on behalf of tasks: task requests the resource
- RTOS grants the request if the requesting task:
  - is allowed the resource (does it have privileges?) AND
  - can be given the resource (is it available?)
- RTOS keeps track of how much of each resource is available and of conditions that requests can be granted

30

Desired RTOS Functionality

- Direct control of input/output devices
- Deterministic scheduling and operation
- Easy access to interrupts
- Time services to supports timers and interrupts
- Synchronization and communication primitives
- Cache handling
- Memory management
- Concurrency/Multitasking
- Good Performance
- Protection

31

RTOS Selection – Questions To Be Asked:

- Is run-time number of task limited?
- What types of inter-task communication are supported?
- Is time-slice adjustable?
- Does RTOS prevent priority inversion?
- Does RTOS support selected synchronization mechanism?
- Does it allow you to select the scheduling algorithm?
- Are task switching times acceptable?
- What tool support does the RTOS offer?
- Is supporting data/documentation available?

32

Typical RTOS

- Run on a variety of target processors
- Supports host-based development environment
- Provides multitasking, reliability, performance, ease of use
- Typical examples:
  - VxWorks – Wind River Systems
  - Integrity – Green Hills Software
  - QNX – QNX Software
  - LynxOS – Lynux Works
  - OSE Enea
  - VRTX – Mentor Graphics
  - Windows CE – Microsoft

33

Summary

- A **traditional** role of operating system is resource management based on the **fairness** principle
- A **real-time** operating system focus on **deterministic response** and data/program **corruption prevention**
- The real-time operating systems must provide solution to **priority**, **timing**, and **task interaction**
- Synchronization, communication, exception handling, and recovery mechanisms take timing into account

34

Host shell Lab

- Virtual shell using any target server connection
- Any number of channels supported
- Runs in host-specific file space
- Four modes:
  - C interpreter: executes C-language expressions
  - Command (cmd): a Unix style command interpreter for debugging and monitoring a system (including RTPs)
  - TCL: access to WTX TCL API and for scripting
  - GDB: used for debugging target using GNU Debugger (GDB) commands
- Any number of host shells can be run simultaneously
- Switching between shells:
  - cmd
  - gdb
  - tcl
  - C (capital c)
- Lab work on C interpreter.

35

Semester Project Teams

- All Dove
- Javier Garza
- Derek Maese

- Ricardo Nunez
- Alejandro Cano
- Edgar????

- Matthew Giandoni
- Danny Lopez
- Rafael Ortega

- Yadira Jacquez
- David Pruitt
- Lauren Waldrop

36

Semester Project

- DigitalHome (DH) Project:
  - www.softwarecasestudy.org
  - SRS 1.3
- Team Assignment for Monday
  - Come up with a project proposal based on the DH with RT aspects
  - Identify required hardware for project completion
    - Links to material
    - Cost (around $400.00 per team)
  - Due by midnight on Sunday