Real-Time Programming

Week 1:
Introduction and Getting Started

Instructors - Tony Montiel & Ken Arnold
rtp@hte.com

These are the course notes to accompany the UCSD Extension class: Real-Time Programming

FIRST send an e-mail listing any e-mail addresses you would like to have class notices sent to:

rtp@hte.com

The class e-mail will consist of updates between meetings, Q&A, important notices, and interaction with the instructor and other students between classes.

web site: http://www.hte.com/uconline/rtp

Instructor contact info:
Tony Montiel (phone 858-513-4941)
Ken Arnold (phone 858-335-9361)

rtp@hte.com

fax 858-530-1458 (@Ken's office)
fax 858-679-7569 (@Ken's home)

Mailing address:
HiTech Equipment Corporation
9672-101 Via Excelencia
San Diego, CA 92126
Welcome!

- Administrative Issues
- Course Overview.
- Instructor and Student Introductions.
- Terms & Definitions.
- Target Design Review.
- Homework #1.
Administrative Issues

- Fill out student forms, PLEASE!
- Class web page:
  http://www.hte.com/uconline/rtp
Grading

- Homework - Due week after assigned!
- Report/Project - Last day of class.
- Mid-term Exam - After 4th class.
  - Gives instructors feedback on progress.
- Final Exam - Electronically, after last class.
Course Objectives

- Microcontroller vs. Desktop PC
- Real-Time Programming
- Hands-On Exposure Required
- Low-level Programming, Interfacing
- Microcontroller Applications
- Polite, Invisible, Reliable Computing

Today the microcontroller has become one of the most powerful tools available to the scientist and engineer. Millions of desktop computers are sold directly to industry and consumers every year, so a great deal of attention is given to them, but the vast majority of new designs are for embedded applications. In fact, microcontrollers have been embedded in so many products that it is easy to overlook the fact that they greatly outnumber personal computers. Billions of microcontrollers are sold every year. For every desktop computer developer there are many more developers using microcontrollers in an embedded application.

Microcontrollers are everywhere around us. Product incorporating these devices have been described as invisible or ubiquitous computers. They are often used as building blocks and incorporated into other integrated circuits. And the number of embedded designs is growing rapidly.

The purpose of this course is to give the student the advanced skills to design real-time software for microcontroller based systems. The emphasis is on the practical aspects of programming and interfacing the processor to real world I/O devices to the outside world.

Since most of the students in this class are already familiar with the development of programs at a high level, the emphasis will be on the low level code which deals directly with the hardware.
Course Format

- **In-class:**
  - Lecture and demonstrations
  - 3 hours * 9 meetings
  - Please Ask Questions!!

- **Outside of Class:**
  - Software Development Kit (SDK),
  - Development Setup:
    - SDK, Prototyping, and Test Equipment
  - Mid-term and Final exams
  - Project
Resources - Hardware

- SDK
- Prototyping Board
- Components
  - Component Kit, or
  - Order from:
    - www.jameco.com
    - www.digikey.com
    - Radio Shack
Resources - Software Tools

- Keil 8051 uVision2 C Compiler/IDE
  - http://www.keil.com

- Assembler

- MicroC/OS-II
  - Need to purchase the Labrosse book.
  - We can send you the 8051 port.
  - Requires full $300 Keil compiler.
General Information

- **Required Text**

- **Additional References**:
Support Web Sites

- http://www.hte.com/uconline
  - .../rtp/51info.htm
  - .../library/8051info/sdk31man.pdf
- http://www.keil.com
- http://www.ucos-ii.com
- http://www.engg.le.ac.uk/books/Pont
Online References

  - The fosterX.pdf files.
    - Scanned images of “Real Time Programming - Neglected topics” by Caxton Foster. Out of print.
  - paranoid.pdf
    - General guidelines for “safe” programming.
- http://www.embedded.com
- http://www.ganssle.com
Instructors

Tony:
- Father of 3 boys.
- ALM Engineering, Inc.
- UCSD Extended Studies Instructor

Ken:
- Father of 5, age 5 to 22, misc. creatures.
- HiTech Equipment Corporation
- UCSD Extended Studies Instructor

Ken Arnold is President of HiTech Equipment Corporation, where he is responsible for the development of embedded computer designs and wireless data communication systems. He is actively involved in several application areas, including industrial electronics, communications, and signal processing. He has designed several highly integrated general and special purpose computers, and has written compilers for two different computer languages. With an extensive background in the design, application, hardware, and software aspects of computers, he has a unique overview of computer technology and its real world applications. Ken holds several patents and currently teaches embedded computing courses at UCSD extension. He is the program coordinator for the Embedded Computer Engineering certificate courses offered by UCSD. Ken is the author of Embedded Computer Hardware Design and is currently working on the companion book, Embedded Computer Software Design. His most important job is being a father to five children, whose ages range from three to twenty-one. Ken also provides consulting services to large multi-national corporations involved in the design and application of a wide range of technologies to real world problems. Prior to founding HTE, he was Engineering Chief of the Computer Technology and Special Purpose Processor Departments at General Dynamics. He is listed in Who's Who in California.

At UCSD Extension, Ken is the embedded certificate Program Coordinator as well as an Instructor. Mr. Arnold has been involved in curriculum development and instructing courses in Engineering and Computer Science at UCSD since 1982. Ken developed and presented the following UCSD courses: Embedding the Internet, Embedded Controller Hardware Design, Microcomputer Operating Systems, Real Time Programming (Embedded Systems), Real Time Embedded Operating Systems, Computer Hardware for Programmers, Signal Processing Hardware, Analog-Digital Conversion. His seminars include: Embedding the Internet, Invisible Computing: The Future of Embedded Systems, Low Cost Wireless Options for Embedded Internet.
Student Introduction

- Your Name and Background

- What Do You Do?
  - (i.e. - EE at XYZ Corp., etc.)

- What Do You Want to Get out of This Class?
Terms Covered in Class

- Real-Time
- Events
- Responses
- Interrupts
- Determinism
- Priority Inversion
- Critical Section
- Atomic Operation
- Semaphore
- Mutex
- RTOS
- ...and many more
What is “Real-Time”?  

- Must respond to an event.
- Only a fixed amount of time for processing.
- Inputs and Outputs are handled at the rate they naturally occur.

To be effective, a real time system must respond to input events, process data, and generate outputs prior to a deadline. The deadline is usually imposed because of external system requirements.

Current PCs are capable of very high processing rates. Yet, a PC running Windows may perform poorly compared to a simpler processor. Why is that? Because the processor spends most of its time doing things which are not part of the intended task. Operating systems, such as Windows are not designed to process tasks in real time, so they often fall behind. This is especially true of those things for which we humans have little ability to tolerate. Delays in spoken communications are intolerable, as anyone who’s used one of the early Internet phones can attest.
Examples

Real Time Systems
- Aircraft flight controls
- Live Action Games
- Simulations
- Cardiac Pacemaker
- Music Synthesizer
- Microwave Oven
- VCR Controller
- Voice/Video

Non-real Time Systems
- IR Remote Control
- Internet
- Accounting Program
- Database
- E-mail programs
- Solitaire Game
- Spreadsheets
- Math Programs

Some examples of real time and non-real time systems will help to illustrate the difference between the two. There are many examples of both types of systems, and here we see a few of them.

Aircraft controls are obviously real time systems, since they must react to the pilot’s inputs in a short period of time to avoid making a really nasty crater in the ground. Games and simulations must operate at the same rate as we would expect in the real world in order to be convincing. When we program our VCR to record a Twilight Zone rerun from midnight to 12:30, we expect it to start and end on time. Assuming, of course, that one is able to program the VCR at all!

There are many non-real time systems as well, such as the processor that sends the IR (infra-red) remote control signals to your TV. If you have to push the button a second time to turn on your TV, there’s no serious negative consequence. In fact, considering what on TV, it might be positive… As long as a spreadsheet program eventually coughs up the right results, we usually wait for them.
Hard Real-Time Example

- Small telemetry computer
- One end of 1-meter rod
- Other end is a nuclear device
- 2.25 msecs to measure and transmit data.
- Things go badly after ~3 msecs.

Quoted from “Patterns...”:

“We build systems that reside in a small telemetry computer, equipped with all kinds of sensors to measure electromagnetic fields and changes in temperature, sound and physical disturbance. We analyze these signals and transmit the results back to a remote computer over a wide-band channel. Our computer is at one end of a one-meter long bar and at the other end is a nuclear device. We drop them together down a big hole in the ground and when the device detonates, our computer collects data on the leading edge of the blast. The first two-and-a-quarter milliseconds after detonation are the most interesting. Of course, long before millisecond three, things have gone down hill badly for our little computer. We think of that as a real-time constraint.”

(de Marco, writing in the forword to Hatley and Pirbhai, 1987).
Soft vs. Hard Real-Time

- **Soft R-T Systems**
  - Missing a deadline results in degraded performance
  - Examples:
    - ATM machines
    - Transaction processing
    - Navigation
    - Inventory control

- **Hard R-T Systems**
  - Missing a deadline may be catastrophic
  - Examples:
    - Life support
    - Engine control
    - Flight controls
    - Broadcast audio/video
    - Robot controls

Like everything else in the software world, real time is not an absolute characteristic of a system. There are “hard” real time systems and “soft” real time systems.

A soft real time system is one which can allow missing a deadline and just degrade in performance rather than failing completely. A hard real time system is one that may suffer catastrophic failure if a deadline is missed.

An example of a soft real time system is an automated teller machine. We expect to get our cash out within a reasonable time after pushing a button on the ATM, but we will wait around until it disgorges our cash.

An automotive engine controller must sequence the fuel injection and ignition timing within a narrow time window to allow proper engine operation. A piece of life support equipment, such as a pacemaker, must act in a short, finite amount of time or it wearer might come up against the final deadline.

An embedded microcontroller driven toaster could be considered soft real time, because it won’t really matter if it doesn’t toast accurately down to the second. On the other hand, we probably wouldn’t want to wait for the toaster to boot up Windows before we could get our toast, either.
Event Driven Systems

- The hardware and software together take action as a direct result of the occurrence of an event.
- When an event occurs:
  - Other processing is suspended
  - The event is processed
  - Other processing is resumed
- Usually these are also “Real-Time” systems

In an event driven system, low priority processing is suspended temporarily to execute another, more important (and usually time critical) task. The purpose of the task is to process the event. This is analogous to the following situation: you’re sitting there reading a good book, and the telephone rings. You respond to the event by getting up and answering the phone, and the event processing task is to convince the mortgage broker that you really don’t want to refinance your home. That task concluded, you go back to your reading.

Many types of embedded systems react to external events, with a limited amount of time available to handle those events. These kinds of applications are "event driven" because no action will take place until an event occurs. In the case of a typical embedded application, event driven programs are used when it is necessary to respond to an external event within a fixed time period. A system which has to respond and process an event in a fixed amount of time is referred to as a "real time, event driven" system.
Polling for the occurrence of an event is a very simple way to make a program respond only when an event occurs. It is particularly appropriate when there is nothing else for the processor to do while waiting for the event.

The program is written with a simple loop that tests for the event, and waits until that event occurs. Such a program will only respond to the event when it is looking for it. It is possible to implement a program which can respond to multiple events, by checking for every possible event in the polling loop. When a particular event occurs, the program calls the appropriate event handling routine.

It is possible to write code for a co-operative multitasking system such as this, but if any task hogs all the CPU time, there is no way for any other tasks to execute. Some operating systems allow certain events, such as a call to an operating system I/O function, to allow other tasks to run. This is approximately how Windows works, and explains why some programs can lock up the PC. It is a multi-threaded operating system, which allows multiple programs to execute by switching them at predetermined points in the program.

Polling is not appropriate for most real time event driven systems, as it results in an indeterminate delay from the time an event occurs until it is processed.
What is an Interrupt?

- Imagine a pushbutton that forces the CPU to call a specific program subroutine.
- When the switch is closed, it effectively CALLs a special subroutine (the Interrupt Service Routine, or ISR).
- This routine can be called independently of the instruction executing when it occurs.
- This is literally what is done for many microcontroller interrupt applications.

A hardware interrupt is like a subroutine call that is initiated by a signal that is external to, and independent of, the running program. A pushbutton can be used to force a call to a specific ISR, regardless of what the CPU is doing at the time of the interrupt.

When a hardware interrupt request is enabled and activate, the CPU saves its current program counter, performs an interrupt cycle in place of the usual program fetch cycle. The interrupt cycle typically consists of the interrupt source identification, and the transfer of the interrupt vector information. The interrupt vector is often a pointer to the place in memory where the address of the interrupt service routine is stored.

The CPU will then fetch that address and perform what amounts to a subroutine call to that address. When the interrupt subroutine has completed processing of the event that caused the interrupt, the processor executes a "return from interrupt" instruction and goes back to the part of the main program that was executing before the interrupt.
Interrupts - Software vs. Hardware

Software Interrupts: subroutine CALLs
- Used to call routines without knowing address
- Example: Operating System Calls
  - Allows access to standard system functions

Hardware Interrupts: a “hardware” CALL
- Used to allow response to external events
- Minimizes wasted CPU time
- Eliminates need to poll for an event

There are two kinds of interrupts: software and hardware. **Software interrupts** are just another kind of subroutine call that can be used to access subroutines with entry points at fixed memory locations. Operating system services are often accessed using software interrupts, which are simply instructions that cause an interrupt subroutine to be called at whatever point in the program they are placed. These interrupts are synchronized with the program in that they always occur at the same place in the program. They are referred to as "synchronous events" because their execution is solely dependent upon the execution of the program instructions. Unless otherwise specified, the word "interrupt" is generally used to imply a hardware interrupt. **Hardware interrupts** are triggered by physical events, such as the closure of a switch, that cause a specific subroutine to be called. They can and do occur at any time in the program, depending on when the event occurs. These are referred to as "asynchronous events" because they may occur during the execution of any part of the program. Interrupts allow the programs to respond to an event when it occurs. In a printing application, the printer may invoke a hardware interrupt to inform the program currently running that it has printed all the data in its buffer and is ready for more.
Software Interrupts

- Specialized Subroutine CALLs
  - Always occur at the same point in a program
  - Synchronized to the program execution
  - Location Independent
  - Uses:
    - Calling a subroutine in ROM
    - Predefined O.S. Functions
    - Handling Exceptions (Overflow, divide by zero)

A software interrupt is a special subroutine call, it is synchronous, meaning that it always occurs at the same time and place in the program that is interrupted. It is frequently used as a quick and simple way to do a subroutine call for accessing programs such as the operating system and I/O programs. The old PC disk operating system used on the PC uses interrupt number 21 (hex) to invoke operating system functions such as reading a disk file, output data to the printer and so on.

For the purposes of this class the word interrupt by itself will be taken to mean hardware interrupt, which is defined next.
Hardware Interrupts

- Hardware Interrupts: a “hardware” CALL
  - Asynchronous to program
    - Can occur at any point in program execution
    - Timing is unrelated to main program execution
  - Forces the CPU to call interrupt subroutine
    - Interrupt Service Routine (ISR)
  - Allows timely response to an event
    - Important events are processed on demand

A hardware interrupt can be thought of as hardware induced subroutine call. When an external event such as the depression of a key occurs, an interrupt subroutine is called to store the key code for later use. This type of event driven subroutine call is asynchronous, meaning that it can occur at any time and place in the program that is interrupted.

Without using interrupts, the program would have to use a loop to poll for an event wasting a lot of CPU time while waiting for the event to occur. This polling process has several disadvantages. It can delay response to the event, since it will only be detected when the program is looking for it. It also wastes CPU time, since the time spent polling for the event is providing no useful purpose except when the event actually occurs.

By responding to the event only when it occurs, the CPU can wait until there is an event before processing it. This allows the event to be processed immediately after it occurs, and avoids wasting CPU cycles testing for something that hasn’t happened yet.
Target Design

P1.0
Jumper or Switch

P1.7
High efficiency LED
Interrupt Turn Signal Input

- The Event
  - Turn signal switch closes

- Interrupt Generated
  - Selects Turn Signal ISR

- Interrupt is Processed
  - CPU suspends main
  - CPU calls ISR
  - ISR stores turn direction
  - ISR returns to main
Turn Signal Interrupt Timing

Switch Closes

Interrupt Controller to CPU: “Interrupt Requested”

Interrupt Latency

CPU: “OK. What interrupt number?”

Interrupt Controller: “Interrupt Request Complete”

CPU: “Interrupt Received”

ISR Begins Execution

Switch Opens

Interrupt ACK

CPU Bus

Interrupt Request

P1.0

Copyright (c) 2002 Ken Arnold, Anthony L. Montiel
Interrupt Latency

- Interrupt Latency is the CPU time it takes to recognize, and start up the ISR
- Non-useful "overhead" delay time
- We want to minimize this delay
- Determined by hardware and software
  - Hardware latency determined by chip design
  - We control the software latency (of our own code)
  - External code (C Library and OS) affects latency
Software Design - Polling

Initialization

Main_Loop:

SW_Input go low?

Yes

No

Switch_Handler:

SW_Input go high?

Yes

No

Toggle LED
Software Design - IRQ

Main Loop:
- LED_Active = SW_Active

Switch_ISR:
- SW_Active = !SW_Input

INT0

Timer2_ISR:
- LED_Active
- Blink LED

What’s missing here?
(Hint: it is missing in the polled version too!)
Interrupt Driven Programs have 3 parts:

1. Initialization
   - Getting everything into a known state
2. Main Program (not event driven)
   - Performs non-time critical processing
3. Interrupt Service Routines (ISRs)
   - Processes the events

Interrupt Driven Program Elements

1. Initialization - performed once
   a. initialize variables, states, flip flops, etc....
   b. initialize interrupt control hardware
   c. initialize interrupt vector

2. Main routine - performed whenever there are no other events occurring not tied to anything occurring, for any non time-critical (non-event driven) things you want done (sometimes referred to as foreground).

3. Interrupt service routine (ISR) - performed once for each interrupt
   a. save processor state
   b. process event causing interrupt
   c. restore processor state
   d. enable interrupts and return
Initialization

- Only happens once (hopefully!)
- Puts all variables, memory, in known state
  - Initialize variables, flags, I/O, IRQs
- Sets up CPU interrupt control system
  - Initialize interrupt priorities, enables
  - Clear pending interrupt requests
- Enables interrupts to occur

Initialization (executed once)

a. disable interrupts (always do this first!)
b. clear buffers/ pointers/ flags
c. store address of ISR(s) in vector table
d. initialize interrupt hardware
e. clear any interrupt requests
f. enable interrupts and enter MAIN routine

All of the hardware and software resources relating to the interrupt system and related data must be initialized here. Note that this is a critical code segment: It may not be interrupted without unpredictable and probably disastrous effects, since the interrupt system is being set up at this phase.

This is usually one of the most difficult and error prone tasks, as it requires the programmer to find and correctly interpret the documentation of the various hardware and interrupt functions.
Main Program

- Processes non-critical functions
  - The things that can be done at any time
- Anything which is not tied to an event
- When there’s nothing better to do:
  - May just wait in a tight loop
  - Execute software diagnostics
  - Execute hardware diagnostics

This is where the boring stuff goes.

The **Main routine** is typically executed many times, when no interrupts are pending. It performs processes that are not time critical or dependent upon external events. Functions such as diagnostic self tests and low priority housekeeping tasks are often performed in this section.

Code in this, non-event driven section may access any resource that is NOT re-entrant. The main routine may be just a loop, waiting for an interrupt to occur.
The **Interrupt service routine (ISR)** is executed once per interrupt. The ISR is really the central part of an event driven system, since this is where events are actually handled.

- a. save processor state: registers, flags, interrupt level, etc.
- b. process the event (what we really wanted to do in the first place)
- c. restore processor state: registers, flags, interrupt level, etc.
- d. enable interrupts (may be located at different points in the ISR)
- e. tell interrupt controller we are finished processing this interrupt
- f. return from interrupt

The ISR must first save the state of the system, so that it does not have any unintended effects on any other programs that may be running (the main program or other ISRs). Any shared resources must be handled carefully. Once the system state is saved, the ISR can proceed to process the event. Finally, the ISR must restore the system to the same state it was in when the interrupt occurred, to allow proper operation after returning to the interrupted program. This is the single most important part of an interrupt driven program, and typically deserves more coding time per line than any other type of program. Like the initialization section, the programmer must have a thorough understanding of the interrupt processing mechanism and any other related hardware. In addition, this code should be as short and simple as possible, to minimize interrupt processing time, latency, and to facilitate debugging.
Summary

- Events can occur at any time
- R-T systems process events as they occur
- Turn signal interrupt input example
- Event driven program structure
  - Initialization
  - Main
  - ISR
Questions

- Classify the following systems, R-T or non-R-T:
  - Pacemaker, Credit card processing machine, printer, calculator, microwave oven, collision avoidance system, heart-lung machine, ATM machine, dialysis machine, payroll processing, music synthesizer

- Classify the R-T systems above as Hard or Soft R-T

- What are the three parts of an interrupt driven program?

- Can software design affect interrupt latency?
Answers

- **Real-Time (Hard/Soft):**
  - Pacemaker(H), microwave oven(H), collision avoidance system(H), heart-lung machine(H), dialysis machine(S), Credit card processing machine(S), ATM machine(S), payroll processing(S), music synthesizer(H)

- **Non-Real-Time:**
  - Credit card processing machine, printer, calculator, ATM machine, payroll processing

- **What are the three parts of an interrupt driven program?** Initialization, Main program, ISR

- **Can software design affect interrupt latency?** Yes.
Homework #1

- Your project proposal is due NEXT WEEK!
- Get TURNSIG working!
  - Download + Extract
  - Make changes.
  - Compile (or assemble)
  - Build hardware
  - Execute code, capture output and e-mail.
- Read sema4s.pdf from FTP site.
- Read Pont pages 2-26, 109-117, review Ch. 3.