UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE 118[218]/L: INTRODUCTION TO MECHATRONICS

LABORATORY ASSIGNMENT NUMBER 0 FOR CMPE 118/218

Due by 6:00pm on Wednesday, January 13, 2009
Pre-Lab due by 12:00 noon on Thursday, January 8, 2009

Purpose: This lab is intended to acquaint you with 'C' programming for embedded systems and soldering.
- Learning the ropes for the CodeWarrior programming environment
- How to write, debug and run a simple C program on an embedded micro-computer
- How to get simple character input from the user in C
- How to order samples from a semiconductor company
- How to solder simple components to printed circuit boards

Minimum Parts Required: A 'CockRoach 3000' and blank PCB with associated components.

References: Course Notes: The Events and Services Framework. State Machines
- http://www.instructables.com/id/How-to-Solder-Videos%3a-Why-is-soldering-difficult-s/
- http://blog.makezine.com/archive/2007/01/soldering_tutor_1.html
- http://www.elexp.com/t_solder.htm

Pre-Lab:

Complete the following exercises AFTER you have read through the lab assignment and BEFORE coming into the CMPE 118 Lab to complete the lab.

☐ 0.1) Brainstorm with your design team to identify the essential 'Events' necessary for your desired behavior, determine how you will test for them in software and how you want to respond. Your Pre-Lab write-up should explain the results of your brainstorming session in the form of a state diagram and pseudo-code for the event checking routines.

For the purposes of the Pre-Lab, your "design team" should consist of you and 1 other person.

0.2) Familiarize yourself with soldering by reviewing the online references above.

In the Pre-Lab: Include your concepts from part 0.1

Part 1 – Exploring the MC9S12C32 and the CW12 Compiler

Reading: The Freescale CodeWarrior Quickstart manual (located in the Freescale folder on the Start Menu)

Assignment: Complete the Following Exercises:

☐ 1.1) Start the CodeWarrior (CW12) IDE (Integrated Development Environment) by clicking on the Windows 'Start Menu', selecting the 'Programs'→'Freescale CodeWarrior'→' CodeWarrior Development Studio for HCS12(X) V4.7' option, then selecting the 'CodeWarrior IDE' shortcut. CW12 is a program that has provisions for creating and editing programs (*.C files), creating and editing project files (*.mcp files), compiling programs, and downloading finished programs to targets (in our case, the "target" is the MC9S12C32 board).

☐ 1.2) First, create a temporary directory to act as your "Home" directory from the root directory (for example C:\JON) Using the Windows Explorer, copy the provided file "C:\CMPE118\SRC\SESTEST.C" into your home directory. Open this file with CW12. You may wish to edit and save changes to this file, so you might want to rename it at this point, also saving this new version into your home directory.

☐ 1.3) Next, you will want to compile and test your program. In order to compile the source file shown in the editor window, you will first need to create a Project file for this program. From the menu bar, select "File → New". Give your project file a name and use your home directory for the location. Pick the "HC(S)12 New Project Wizard" and select "OK". The next screen asks you to pick your micro so select "MC9S12C32". The next screen asks you to choose your languages; we will be using "C". Choose "No" for using the "Processor Expert", "OSEKturbo", "PC-lint", and "Floating Point Format". Choose "Small" for the memory model. In the connections screen choose the "HCS12 Serial Monitor."

1.4) You will need to define all of the *.C source code files needed to build your project. You will need SESTEST.C (or whatever you are calling it now), so click on the "Project→Add" option and select your SESTEST.C file. In addition, you will need to add the files ADS12.C, PWMS12.C, ROACHLIB.C, SES.C, TIMERS12.C, and TERMIO.C located in C:\CMPE118\SRC. To build your project, all you need to do now is click the "Project→Make" option. If there are any errors or problems, CW12 will display them at this time. If there are errors with your code, you will need to correct them all before your code will function (of course), so you should go back to the Editor window with your source code, correct the errors that the Status window revealed, and then try re-building your project. Repeat as required to remove all errors and warnings. To enable debugging information for each source file enable the "Bug" checkbox option next each file listed in the project window.

☐ 1.4) When you are ready for the next step, plug the RS232 (serial) connector into the CockRoach connector (on top of the CockRoach). Make sure you have the battery power connected and switched to the "ON" position (the toggle switch next to the CockRoach wheel).

☐ 1.5) You are now ready to download your new code. To do this, you will need to open uBug12. This is found in the "Start → Programs → Technological Arts → uBug 12" menu. You should now see a terminal window with a command prompt at the bottom. Set SW2 on the MC9S12C32 to the BOOT position and reset the chip. In the uBug12 command prompt enter **CON 1** assuming you are using COM port 1. uBug12 should now display a short message. To erase any existing program use **fbulk**. Now, everything is ready for the new code to be downloaded. You need to tell uBug12 which file to download, so enter **fload ;b** and select your file, which should be named something like MONITOR.ABS.S19 (the *.S19 file is compiled code in the proper format for downloading) and located in the BIN directory of your project. Once your code is finished downloading, the uBug12 will inform you, and you should see a prompt. You are now ready to execute your code.

☐ 1.6) At this point, you are ready to try out your program. **Be sure that the CockRoach is up on the blocks so that when the motors start running it will not drive off of the bench**. When the download was complete, you should have gotten the message LOADED OKAY in the terminal window. To run your program, type in "go" at the prompt and press <ENTER>. This tells the MC9S12C32 to execute your program, which has been loaded into memory. If all has gone well, the CockRoach will begin executing your code.

☐ 1.7) Congratulations! You've just written and downloaded your first embedded program.

☐ 1.8) Next, spend some time exploring the library functions provided for you. Write some simple programs to turn the motors on & off, get a character from the keyboard, print a value to the screen, read the light sensor and test the bumper switches.

In the report: Include the listings from the programs that you wrote to explore the libraries. (Note that we encourage you to print this two to a page and double sided to save paper; you may also use the printer in the CMPE-118 for this).

Part 2 – A Mechanical CockRoach

Reading: Carryer, Kenny, Olhine (CKO) Ch. 5 (Make sure you understand Ch. 1-4)

Background: The behavior of many types of simple creatures can be described in terms of their response to outside stimulus. This will be your opportunity to give a mechanical creature an interesting behavior.

Assignment: This is your mission: you are to write a C program that will run on the MC9S12C32 and cause the CockRoach 3000 to behave like a real cockroach.

The fundamental behaviors are that they:
run around when the lights are on
hide in the dark
won't get stuck in corners or against chair legs, etc.

☐ 2.1) Begin by writing some simple programs to explore the inputs and outputs to/from the 68HC12. Use the library functions provided to control the motors, read the bumper states, and read the light level.

☐ 2.2) Complete the detailed design and implementation of the program that you worked out in Part 0.

☐ 2.3) Write and test your program to imitate a cockroach. Be sure to develop and test incrementally to avoid a huge headache! Use typedef enum to name your states!

☐ 2.4) Get your functional roach checked off by the TA, Tutor, or Professor.

In the report: Include a complete description of the design and implementation of your program. This should include high-level descriptions (like a state diagram) as well as the PDL (Program Design Language) or Pseudo-Code and the final C code that resulted.

Part 3 – A 'Smarter' CockRoach

Reading: Nothing new.

Assignment: Now that you have the basic functions down pat, see if you can make it more interesting to watch by endowing the machine with a more complex behavior. Use your imagination, but as a suggestion I will offer the idea of searching for the dark by following some recognizable pattern.

Part 4 – PCB Assembly and Samples

Reading/Viewing:
- http://www.instructables.com/id/How-to-Solder-Videos%3a-Why-is-soldering-difficult-s/
- http://blog.makezine.com/archive/2007/01/soldering_tutor_1.html
- http://www.elexp.com/t_solder.htm
- http://www.sparkfun.com/commerce/tutorial_info.php?tutorials_id=59
- http://www.sparkfun.com/commerce/tutorial_info.php?tutorials_id=58

Assignment: Later in the class, everyone will need motor driver boards and analog voltage followers to protect the larger I/O boards. This assignment is to assemble and debug one of the supplied PCBs. Boards and parts will be distributed in lab after a brief soldering demo. Another useful resource for this class (and further in your engineering career) is free samples. We'll get you started here on ordering free samples and resupply some parts for the class at the same time.

☐ 4.1) Start assembly only after becoming comfortable soldering and after examining a completed board of the same type. Remember to only solder sockets, not the full ICs.

4.2) If you are feeling extra ambitious, you can do some surface mount solder paste and skillet to get a feel for how to do this. If you plan to do some surface mount fabrication with PCBs in the future,

this is a very good technology to get a handle on. Basically, you can cut your production time for a fairly complex design to well below a week, and including parts which are very challenging to hand solder.

☐ 4.3) Show your completed board to the TA, Tutor, or Professor for check-off.

☐ 4.3) Go to the Texas Instruments website (www.TI.com) and make an account for yourself, order some parts as free samples (for instance op-amps and comparators might be useful later). For a single supply op-amp, navigate to the part page by following this link: http://focus.ti.com/docs/prod/folders/print/tl3414a.html and clicking on the samples button that corresponds to the line TL3414A in the PDIP package (you can get them to send you up to 10). You will need to set up an account (free). Parts should be sent to your own mailing addresses (they will send it FedEx), if you cannot do that, have it sent to: Computer Engineering, 1156 High Street – SOE3, Santa Cruz, CA 95064-1077. Look around TI's website for any parts you think might be useful later on. Motor drivers, filter chips, and rail to rail op-amps are always useful. You will notice that most of the parts they sample are surface mount, and these can be an issue to prototype with. We'll talk about ways of dealing with this in class.

4.4) Navigate to the Allegro site and request samples for the 16-lead SOIC of the 3949 Motor Driver and the A3982 Stepper Driver in 24-lead SOIC: http://www.allegromicro.com/en/. Again, take a look around and see if there are other things they sample that you are interested in having in your parts bins.

4.5) For the report, include a printout of the parts that were sampled, and when they come in, please bring them to the 118 TAs or Professor (only the motor drivers, the rest you keep for your lab kits).

Notes on writing your code

As you may have noticed from studying the example code provided, there are a number of things that you must do in order to have access to the libraries provided for programming the 'CockRoach 3000'. In particular, notice the header files that need to be included:

```
#include <mc9s12c32.h>
#include <roachlib.h>
#include <ses.h>
#include <timers12.h>
#include <stdio.h>
```

These header files are located in `C:\CMPE118\INCLUDE` (except for mc9s12c32.h) and the compiler is aware of them through the `LIBRARYPATH` environmental variable.

`mc9s12c32.h` brings in a set of definitions and prototypes that are used throughout the CMPE118 labs, it should always be included in your programs.

`roachlib.h` is the header file for a simple set of input and output routines provided for use on the CockRoach 3000. This gives you access to functions that control the motors, read the state of the bumpers, and read the state of the light sensor.

`ses.h` is the header file for the Software Events and Services library. This library provides access to a pre-defined and convenient implementation of the Events and Services Framework.

`timers12.h` is the header file for the Timer library. This library provides access to several timers and several useful functions associated with initializing, setting, and checking them.

Use the `typdef enum` construct for code readability and understanding. `Enum` is short for *enumerated constants*, and is used as follows: `typdef enum colors {red, blue, green, brown}; colors` is now a variable type (just like char or int), and has only the declared types (red, blue, green, brown) as valid types. This is especially useful for state machines, where the state variable should have well named enums (reversing, driving_forward, turning_right, stopped, etc.).

Lab #0

Time Summary

Be sure to turn this in with your lab report

This information is being gathered solely to produce statistical information to help improve the lab assignments.

Pre-Lab	Preparing Outside of the lab _____	In the lab working this part_____
Part 1	Preparing Outside of the lab _____	In the lab working this part_____
Part 2	Preparing Outside of the lab _____	In the lab working this part_____
Part 3	Preparing Outside of the lab _____	In the lab working this part_____
Part 4	Preparing Outside of the lab _____	In the lab working this part_____
Report	Preparing the Lab Report _____	

INTRODUCTION to THE COCKROACH 3000

Rev : January 9, 2006

Purpose of Document

This document will serve as an introduction to the functionality of the Cockroach 3000. It includes descriptions of how to drive the Cockroach, use its light sensors to detect changes in ambient light levels and read its bumpers to sense hits.

Cockroach 3000 Control

Driving Forward

Forward motion is implemented by setting both of the Cockroach's motors to the same speed. The right and left motors are controlled by the RightMtrSpeed and LeftMtrSpeed functions, respectively. The functions require integer arguments between 1 and 10 to move the motors forward.

```
Example: RightMtrSpeed(3)
         LeftMtrSpeed(3)
```

Driving Backwards

Reverse motion is also implemented by setting the Cockroach motors to the same speed. However, the arguments into the RightMtrSpeed and LeftMtrSpeed functions are required to be integers between –1 and –10.

```
Example: RightMtrSpeed(-5)
         LeftMtrSpeed(-5)
```

Stopping

The Cockroach is stopped by setting both its motor speeds to 0. This is accomplished by using the RightMtrSpeed and LeftMtrSpeed functions.

Turning

Turning the Cockroach is accomplished by driving the two motors at different speeds. Depending on the desired effect, there are various ways of turning the Cockroach:

- If both motor speeds are positive, the turn will be gradual
- If one motor is set to a positive speed while the other is set to a negative speed, the turn will be sharp
- Identical but opposite speeds will make the Cockroach spin its current position
- Stopping one motor and driving the other will cause the Cockroach to turn on the stopped wheel.

```
Example: RightMtrSpeed(7)
         LeftMtrSpeed(2)
                 represents a gradual turn to the left.
```

Reading Changes in Light Level

The amount of light hitting the Cockroach is obtained by using the function LightLevel. The function returns a 10-bit value corresponding to the amount of light seen by the Cockroach's light sensors. A transition by the Cockroach from light to dark, or vice-versa, is sensed by detecting a change in this measured value.

Often, successive values returned by the LightLevel function will vary by a few bits. This causes a problem in implementations that rely on discrete measurements, as in the case of detecting light-to-dark (or dark-to-light) transitions. In order to avoid repeated sensing of a transition due to fluctuating values returned by the LightLevel function, it is beneficial to add a tolerance band (hysterisis) to the transition condition.

Hysterisis may be implemented by running two different threshold tests that depend on two values, LIGHT_THRESHOLD and DARK_THRESHOLD. LIGHT_THRESHOLD defines the minimum light level required for a valid "light" condition and DARK_THRESHOLD defines the maximum light level for a valid "dark" condition. If LIGHT_THRESHOLD and DARK_THRESHOLD are slightly higher and lower than the nominal transition point, respectively, false transitions will be minimized. In the example below a light level of 50 is the mid-point of the hysterisis band.

Example: An event function that tests if the roach has entered the dark may be implemented like this:

```
#define DARK_THRESHOLD = 47;
#define LIGHT_THRESHOLD = 53;

uchar TestIfDark (EVENT_PARAM)
{
  static uchar LastLight = 0;
  static uchar Threshold = DARK_THRESHOLD;

  char GoneDark = if((LightLevel() < Threshold) &&
                     (LastLight >= Threshold));

  if (GoneDark)
    Threshold = LIGHT_THRESHOLD;
  LastLight = LightLevel();
  return (GoneDark);
}
```

The roach's entry into light would then be detected by testing if LightLevel() is greater than Threshold, which is now equal to LIGHT_THRESHOLD. Once the roach is in the light, Threshold is returned to DARK_THRESHOLD, completing the hysterisis implementation.

Reading Changes in Bumper State

The state of the Cockroach's bumpers is accessed through the functions IsFrontRtBumped(), IsFrontLtBumped(), IsBackRtBumped(), IsBackLtBumped(). These functions return a Boolean value that is true when the bumper is hit and false under normal conditions. These functions use the analog to digital library ADS12 that reads the magnetic field changes for each sensor. These 10-bit magnetic field values can be read using the ReadFrontRt(), ReadFrontLt(), ReadBackRt(), ReadBackLt() functions.

DOCUMENTATION for ROACHLIB module

Rev : January 9, 2006

Purpose of Module

This module provides functions to control the motors, read the light sensor, and read the bumper sensors on the CMPE118 Cockroach 3000.

INTERFACE

Module Functions

RoachInit

```
PROTOTYPE:  void RoachInit(void)
CONTENTS:   This is the initialization routine for the Roach functions.
PARAMETERS:
            none
RETURNS:
            nothing
```

RightMtrSpeed

```
PROTOTYPE:  uchar RighMtrSpeed(char newSpeed)
CONTENTS:   This function is used to set the speed and direction of the right
              motor.  A negative value is reverse and 0 is stop.
PARAMETERS:
            NewSpeed –- a speed value between –10 and 10
RETURNS:
            OK_OPERATION == The registration was successful.
            ERR_BADSPEED == An invalid speed was given.
```

LeftMtrSpeed

```
PROTOTYPE:  uchar LeftMtrSpeed(char newSpeed)
CONTENTS:   This function is used to set the speed and direction of the left
              motor.  A negative value is reverse and 0 is stop.
PARAMETERS:
            NewSpeed –- a speed value between –10 and 10
RETURNS:
            OK_OPERATION == The registration was successful.
            ERR_BADSPEED == An invalid speed was given.
```

LightLevel

```
PROTOTYPE:  uchar LightLevel(void)
CONTENTS:   This will read the light sensor, returns a number proportional to
              the amount of light being received.
PARAMETERS:
            none
RETURNS:
            An 8-bit value indicating the amount of light present.
```

ReadFrontLeft, ReadFrontRight, ReadBackLeft, ReadBackRight

```
PROTOTYPE:  int ReadFrontLeft(void)
CONTENTS:   This will read the 10-bit magnetic field value for a bumper
              sensor.
PARAMETERS:
            none
RETURNS:
            An integer value corresponding to the magnetic field reading for
            the bump sensor.  The value should normally be around 512. If a
            bumper is hit, the corresponding value will swing high (towards
            1024) or low (towards 0) depending on the position of the magnet
            to the sensor.
```

IsFrontLeftBumped, IsFrontRightBumped, IsBackLeftBumped, IsBackRightBumped

```
PROTOTYPE:  int IsFrontLeftBumped(void)
CONTENTS:   This will check a bumper sensor and determine if a bump has
              occurred.
PARAMETERS:
            none
RETURNS:
            A non-zero value if a bump is detected or zero under normal
          conditions.
```

DOCUMENTATION for SES module

Adam Braun, Ed Carryer, Kevin Wooley
Rev : January 9, 2002

Purpose of Module

This module provides a system to automate a software events and services protocol. Up to 128 event and service function pairs can be registered with the system using 2 different scheduling algorithms to handle the events.

INTERFACE

Defined Constants

Standard Parameters –- These definitions are used to share data between an Event checking routine and the corresponding service routine.

EVENT_PARAM –- standard event checker parameter, use this in your function prototype
SERVICE_PARAM –- standard service routine parameter, use this in your function prototype
SET_SHARED_VAR_TO(z) – set the variable shared by the event checker and service routine to the value z.
GET_SHARED_BYTE() – return the low byte value of the shared variable
GET_SHARED_WORD() – return the 2 byte (integer size) value of the shared variable.

Scheduling Algorithms -- These constants are used to specify the scheduling for the handling of events. The priority of events is based on the order in which they are registered.

SES_ROUND_ROBIN -- In this algorithm a full pass through all the pairs is made (in order of priority) and every event that is detected is serviced. When the pass is completed, it is restarted with the highest priority pair if any of the events were detected.
SES_PRIORITY -- In this algorithm the events are serviced in order of priority until an event is detected. After the event is serviced, the servicing restarts with the highest priority pair. This process continues until a full pass is made with no events being serviced.

Service Timing -- These constants are used to specify the timing period for the SES_TimeToService function. They are used to approximately control the time between event services in a foreground loop. This is to allow the program to perform other tasks in the background loop if necessary.

SES_NO_UPDATE -- No service timer. SES_TimeToService is always true.
SES_4MS_UPDATE -- Update every 4 ms.
SES_8MS_UPDATE -- Update every 8 ms.
SES_16MS_UPDATE -- Update every 16 ms.
SES_32MS_UPDATE -- Update every 32 ms.

Data Types

uchar -- typedef to be unsigned char
schar -- typedef to be signed char
uint -- typedef to be unsigned int
sint -- typedef to be signed int

Module Functions

SES_Init

PROTOTYPE : uchar SES_Init(uchar aScheduleType, uchar aTimePeriod)
CONTENTS : This is the initialization routine for the SES functions.
PARAMETERS :
aScheduleType -- The scheduling algorithm type.
aTimePeriod -- The timing for servicing.
RETURNS :
OK_OPERATION == The setup was done successfully.
ERR_BADSCHEDULE == The scheduling value was invalid.
ERR_BADTIMEPERIOD == The time period was invalid.

If any of the ERR_xxx returns occur, then no action is taken in the module.

SES_Register

PROTOTYPE :
uchar SES_Register(char (*aEvent)(void**), void (*aService)(void*))
CONTENTS : This will register an event routine and an associated service routine. The priority of the pairs is descending in the order they were registered.
PARAMETERS :
aEvent -- The event detection routine to register.
aService -- The associated service routine to register.
RETURNS :
OK_OPERATION == The registration was successful.
ERR_NOTINSTALLED == The module was not installed with SES_INIT
ERR_TOOMANYEVENTS == There are already the maximum number of event/service pairs registered.

If any of the ERR_xxx returns occur, then no action is taken in the module.

SES_TimeToService

PROTOTYPE : uchar SES_TimeToService(void)
CONTENTS : This will return a flag indicating whether it is time to service the events or not. It the time schedule is set to SES_NO_UPDATE.
PARAMETERS :
none
RETURNS :
Flag for whether the time has expired or not. (1 == time to service).

SES_HandleEvents

PROTOTYPE : void SES_HandleEvents(void)
CONTENTS : This will run the handle events loop to process the events and service functions using the scheduling algorithm set in the SES_Init function. It will return when a full pass through the event/service pairs finds no events to be serviced.
PARAMETERS :
none
RETURNS :
nothing

SES_End
PROTOTYPE : void SES_End(void)
CONTENTS : This will end the SES system.
PARAMETERS :
none
RETURNS :
nothing

CONSTRAINTS/NOTES

1. Once an event has been registered, it can not be de-registered.

2. SES_Init must be called before the module becomes active. If any other SES functions are called before the module is initialized, they will have no effect.

3. The event/service priorities are based on the order they are registered. Pairs that are registered earlier will have priority over those that are registered later.

4. It is the users responsibility to ensure that the event function returns the correct value. A non-zero return value will be interpreted as indicating that an event has occurred. The void ** parameter (EVENT_PARAM) that is passed to the event function is de-referenced and passed to the service function to enable data transfer from one to the other.

THEORY OF OPERATION

This code operates using an array of function pointer pairs. As the event/service functions are registered, the addresses of the functions are entered into the next available array location.

When the SES_HandleEvents function is called, it starts at the top of the array and executes the event functions in order until one of them returns with a non-zero value. It then executes the associated service function. When this is completed, it will either start at the top of the list or continue with the next event/service pair depending on the scheduling algorithm selected at initialization. This continues until an entire passed is made through the array with no events being serviced.