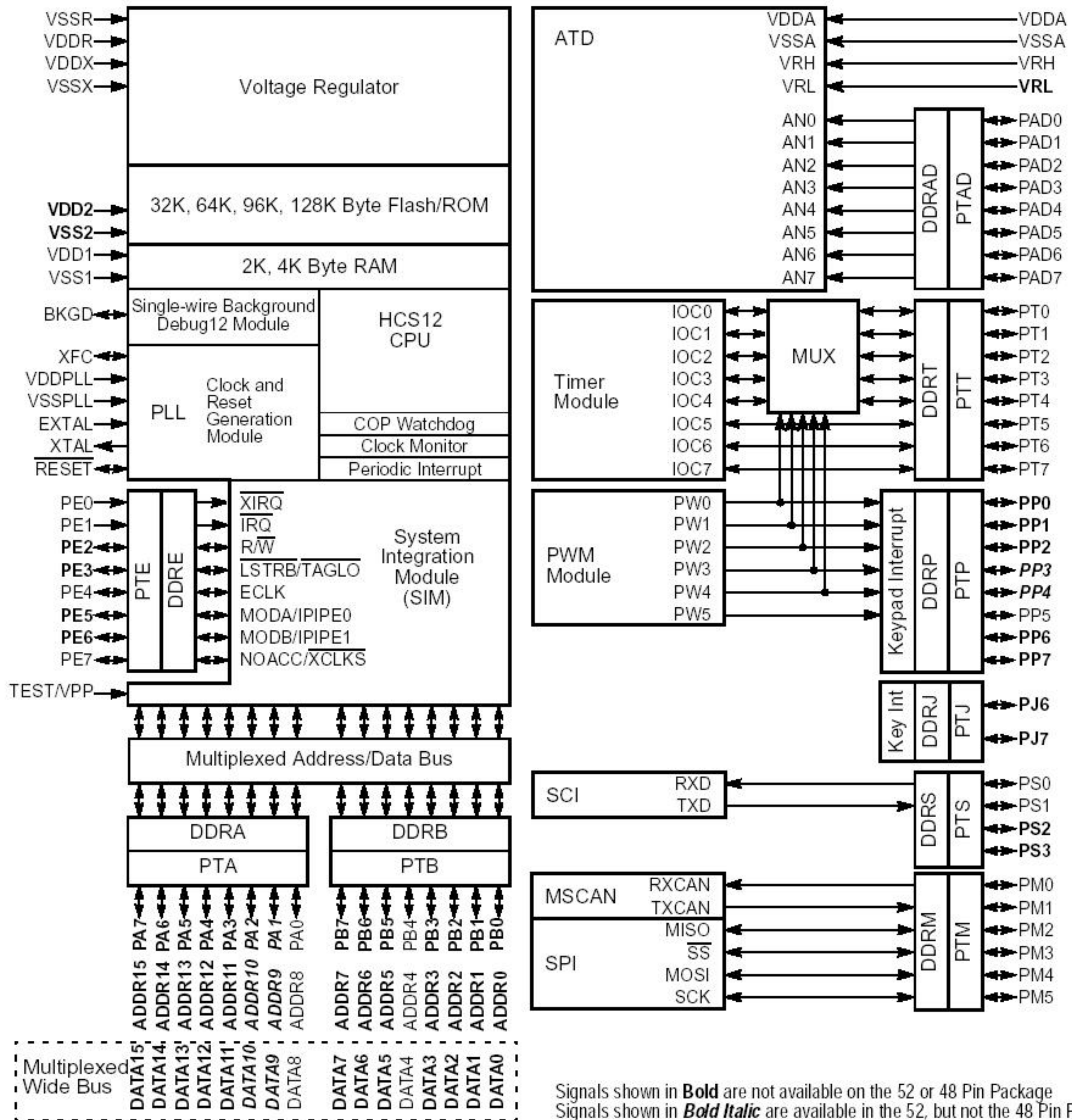
Embedded Microcomputer Systems: Real Time Interfacing
EE345L Laboratory Manual



Spring 2005

Jonathan W. Valvano

Electrical and Computer Engineering

University of Texas at Austin

email: valvano@mail.utexas.edu

web: http://www.ece.utexas.edu/~valvano

how to program: http://www.ece.utexas.edu/~valvano/embed/toc1.htm

This laboratory manual accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Table of Contents

I. Introduction to EE345L Laboratory 2
9S12C32 board information 7
Programming Style Guidelines 13
Important changes to Embedded Microcomputer Systems, 3rd printing 31
Lab 1d. Fixed-point Conversions 35
Lab 2f Performance Debugging 41
Lab 3d Position Data Acquisition System and Interrupting SCI 47
Lab 4f LCD Alarm Clock 51
Lab 5e Stepper Motor Finite State Machine 55
Lab 6f Music generation using a Digital to Analog Converter 59
Lab 7e Preliminary Design and Layout of an Embedded System 63
Lab 8e Interrupting Keyboard Interface and Calculator 65
Lab 9b Final Design and Testing of an Embedded System 71

I. Introduction to EE345L Laboratory

I.1. Grading Policies

Except for Lab 7, groups will consist of exactly two students. The Lab 7 preparation will be done in groups of two, but the final PCB layout and report will be performed in groups of 6. Lab partners have separate attendance[+] and checkout grades[+], but share the preparation[*], report[*], software quality*, and late penalty grades[*]. For each lab assignment, there are a number of preparation tasks that must be completed before the laboratory period. The following activities occur in this order at the beginning of lab, you turn in your preparation, the TA takes attendance, the TA will gives a short lab lecture, the TA will check your preparation, and if possible the TA will return the preparation to you.

<u>Responsibility</u>		<u>grade</u>
Attendance		10[+]
Preparation		20[*]

Preparation includes a printout of your software, which is due at the start of lab. The software should be typed into the computer and compiled with no syntax errors. Preparation does not usually involve running and debugging, but it should not be handwritten. Preparation also includes hardware circuit diagrams, which are also due at the start of lab. Handwritten diagrams are acceptable for the prep, but diagrams in the report must be generated using a CAD package, like **ExpressSCH**. Hardware diagrams include chip numbers, pin numbers, resistor/capacitor types and tolerances, connections to computer. You are responsible for the procurement of all necessary parts before lab starts. As part of the preparation, hardware circuits but not necessarily built or debugged.

Software Quality	(see the section on software style guidelines later on in the lab manual)	20[*]
	Documentation, comments, choice of good variable and function names	
	Proper style, organization, modular structure, ease of understanding	
Report	(10) Final hardware circuit diagrams (must be generated using a CAD program)	25[*]
	(15) Results printout, performance/data graphs (handwritten drawings are OK)	
Checkout, demonstration to TA		
	Performance, correctness of the program function	15[*]
	Interface to the human operator, menus, error messages	
	Oral understanding of engineering tradeoffs	10[+]
Penalty	Late Checkout	-5/day[*]
	Late Report	-5/day[*]
Total		100[+]

Please include the following information at the beginning of each of your software files:

```
1) Students names
2) TA name
3) Date of last change
4) Lab assignment number
```

I.2. SAFETY REGULATIONS:

IN CASE OF EMERGENCY DIAL 911 or 9-911

Since there will be times when students will work other than the regularly scheduled lab sections, it is necessary that certain regulations be observed for the convenience and safety of all. Since the possibility of lethal shock exists in those circuits utilizing low potentials, the following should always be observed:

1. Working alone in a lab room is not permitted.

2. Working after regular hours without written permission is not permitted.

3. Work benches must be clear of all coats, knapsacks and extraneous materials. Coat racks are desired for those desiring this convenience. Otherwise all materials must be stored under the work area or out of the way.

4. Shoes must be worn in the lab at all times. Shoes represent a significant protection against electrical shock.

5. Smoking, food and beverages (e.g., coffee) are not permitted anywhere in the lab area.

IN CASE OF INJURY OR SHOCK:

Turn off power, do not move the injured. Start artificial respiration if breathing has stopped. Have someone else call 911 or 9-911 if CPR is needed.

IN CASE OF FIRE:
Turn off the power, call 911 or 9-911, fight fire with available extinguisher, have someone clear the building.

I.3. LAB PROCEDURES AND POLICIES

FIRST WEEK OF CLASSES: Go to the regularly scheduled lab during the first week of classes. During this time, you will be introduced to the lab equipment. You will also be instructed on lab procedures and grading policy. If you missed your regularly scheduled lab, attend one of the other lab periods. Note that attending a lab session for which you are not registered is not permitted except during the first week of classes.

LAB PARTNERS: Every student is required to have a lab partner. You will perform all labs with a partner. Students choose their own lab partners during the first week.

LAB EQUIPMENT USAGE: Lab hours are posted in the laboratory. There are no sign-up sheets, but cooperation is expected. If you start debugging on a station, you may stay as long as you like, with three exceptions:
You must leave when the second floor labs are closed for the day;
You must leave during the first half-hour of the other regularly scheduled lab periods;
You may not leave the station unattended for more than 15 minutes.

If you would like to use a station that has been left unattended for more than 15 minutes:
1) Carefully disconnect the hardware and eject any floppy disks;
2) Do not save any software files;
3) Return all materials (hardware, disks, paper) to the front desk;
4) Leave a note on the station with your name and time;
5) Write a note to the TA describing exact times listing what you turned in.

LAB LECTURE: The purpose of the lab lecture is to provide necessary information to complete the lab. The scope of the lectures will be material relevant to the lab. The lecture will be conducted during the first 15 minutes of each lab session.

LAB PREPARATION: Lab preparation must be performed prior to the regularly scheduled lab period. All software must be written, edited and designed before coming to the lab. Hardware must be designed down to the pin numbers. Label all resistance and capacitance values and types. For example, 1k? 5% carbon, or 0.01µF 5% ceramic. In this way, the lab period may be spent in debugging your system with the TA's help. The preparation is due at the start of the lab. Preparation includes gathering all the physical components required to perform the lab.

LAB REPORT: Each lab report is bound separately. There is a **Deliverables** section that details the specific components required for that lab report. Lab report typically include the following items:

A) Objectives (1/2 page maximum)

B) Hardware Design. Detailed hardware designs with pin numbers.
- Generated using a CAD program like **ExpressSCH**
- Include all external devices used (chips, R's C's values and types)
- Show connections to the 6812 board.

C) Software Design (no software printout in the report)
- Draw figures illustrating the major data structures used,
- A call-graph illustrating the modularity of the software components
- Draw data-flow graph showing how data is processed

D) Measurement Data
- Whenever appropriate, use the printer to get a hardcopy listing of your results. Include graphs and figures as specified in the assignment.

E) Analysis and Discussion (1 page maximum)

CHECKOUT: A rough draft of your hardware diagrams and hard copy printout of your source code listings must be given to the TA before you demonstrate. If your experiment works, you will be assigned a good score on the performance part. The TA will ask the partners oral questions that test your "understanding" of the computer engineering concepts of the lab. The partners will answer separate questions and receive separate "understanding" grades. You must get your rough draft software listings signed and dated by a TA to prove that the lab was completed in a satisfactory manner. Late checkouts will result in lost points. **Your software files will be copied onto a class web server during checkout. We will submit the files into an automated system that searches for plagiarism.**

I.4. Backing up software onto your floppy

One of the things you need to do often is copying files between the floppy and the hard drive. A batch file is a convenient way to do this. Create a text file on your floppy called **LOD.BAT** with the similar contents:

```
copy A:\*.c D:\TEMP
copy A:\*.h D:\TEMP
copy A:\*.bat D:\TEMP
```

When you execute this file (from DOS or Windows File Manager) it will copy your 6812 development files from your floppy disk to the system hard drive. If you are working with other types of files you can add them to the list.

Create a second file called **SAV.BAT** with the following contents:

```
copy D:\TEMP\*.c A:
copy D:\TEMP\*.h A:
copy D:\TEMP\*.bat A:
```

When you execute **SAV.BAT**, it will copy files from the hard drive back to your floppy disk. During software development you should rotate at least 3 floppy disks, so that you can return to previous states when you get a computer crash, when one file becomes corrupted, or when you make a software modification you wish to undo. Notice how the three floppy disks are used in a rotation. In each case, you load the files from the newest floppy, edit the files on the hard drive, then save the files onto the oldest floppy disk. When you have a software system that runs enough to allow a partial credit demonstration to the TA, put that disk aside and enter new floppy disk into the rotation. The following table shows the contents of the disks after each software development session.

time	session	disk A	disk B	disk C
1	create new programs, save on A	session 1	blank	blank
2	load from A, edit, save on B	session 1	session 2	blank
3	load from B, edit, save on C	session 1	session 2	session 3
4	load from C, edit, save on A	session 4	session 2	session 3

I.5. Variables In C

Consider the following simple C program. Assume this code is located in **`file.c`**.

```
short A1; int A2;
short B1; short static B2;
short C1; short volatile C2;
short D1=5; short const D2=5;
short F1(short n){ return n+1;}
short static f2(short n){ return n+1;}
short FreezingPoint1=32; short FreezingPoint2=0x20; // degrees F
void program(void){
  short e1; short static e2;
}
```

a) *What is the difference between **`A1`** and **`A2`**?* **`short`** is always 16 bits, while **`int`** is compiler-dependent. On Metrowerks CodeWarrior, they are both 16 bits.

b) *What is the difference between **`B1`** and **`B2`**?* **`B1`** is public, and can be accessed from anywhere in the software system. In particular, any file can define **`B1`** as

```
extern short B1;
```

and access this variable. The linker will resolve the address uncertainty. **`B2`** is private, and can be only be accessed from software in this particular file.

c) *What is the difference between **`C1`** and **`C2`**?* **`C2`** will not be optimized by the compiler (**`C1`** can be optimized). In particular, it will not keep a copy of **`C2`** in a register, rather it will reload a new value each time it is needed. It assumes **`C2`** can be changed by operations other than direct software action. Two good applications of **`volatile`** are I/O ports and global variables shared by two or more threads. Assume **`C1`** is incremented by a background interrupt and **`wait`** is called from the foreground. Because **`C1`** is not **`volatile`** then the compiler could take this

```
void wait(void){
  while (C1<100){};
}
```

and create the following "optimized" assembly (Metrowerks does not optimize this way, but some compiler might)

```
wait::  ldd  C1      ; get a copy of C1
loop:   cpd  #100    ; assume RegD has count, check for greater than 100
        blo  loop
        rts
```

d) *What is the difference between **`D1`** and **`D2`**?* Formally, **`const`** means can't be changed. On an embedded system, it means **`D2`** is stored in ROM, and **`D1`** is stored in RAM. There will be two copies of **`D1`**.

e) *What is the difference between **`e1`** and **`e2`**?* When used inside a function **`static`** has a different meaning than when used outside a function. In this situation, it means **`e2`** is statically allocated in permanent RAM and the values persist from one function call to the next. **`e1`** is dynamically allocated on the stack, used inside the function, and deallocated at the end of the function. The values of **`e1`** do not persist from one function call to the next.

f) *What is the difference between **`F1`** and **`f2`**?* In this context, **`static`** is used in a similar manner to the way static is used for a variable outside a function. **`F1`** is public, and can be called from anywhere in the software system. In particular, any file can define **`F1`** as

```
extern short F1(short n);
```

and call this function. The linker will resolve the address uncertainty. **`f2`** is private, and can be only be called from software in this particular file.

g) *What is the difference between **`FreezingPoint1`** and **`FreezingPoint2`**?* The only difference is style. 32 is much better style (easy to read) than 0x20 in this context.

I.6. Web sites

see 6812 products **http://www.technologicalarts.com**
electronic parts **http://www.bgmicro.com**
electronic parts **http://www.mouser.com/**
mechanical parts **http://www.allcorp.com/**
Metrowerks **http://www.metrowerks.com/MW/download/default.asp**
Writing in C for 6812 **http://www.ece.utexas.edu/~valvano/embed/toc1.htm**
Freescale **http://www.freescale.com**
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC9S12C32&nodeId=01624684492994
links to companies **http://www.ece.utexas.edu/~valvano/book.htm**

I.7. Legal Stuff

The opinions expressed in these notes do not necessarily reflect the opinions of the University, its management or its big time financial donors. Also, there shall be no bologna, Bevis, mustard, chewing the cables, free lunch, sob stories, running & screaming, whining, hitting, spitting, kicking, biting, or tag backs. Quit it or we're telling. (Enjoy the course.)

9S12C32 board information

For more information on using the board to develop assembly programs see
http://www.ece.utexas.edu/~valvano/EE319K/CW12asm.pdf

For more information on using the board to develop C programs see
http://www.ece.utexas.edu/~valvano/EE345M/Howtobuild_S12C32.pdf

To see/download many example C programs see
http://www.ece.utexas.edu/~valvano/metrowerks/

To download data sheets for the 9S12C32 system
http://www.ece.utexas.edu/~valvano/metrowerks/TechArts.zip

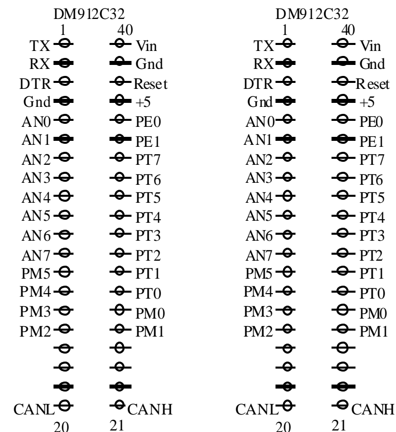
Department policies concerning the 9S12C32 kit
Each student will be given kit, which must last for the EE319K, EE345L EE345M sequence
You should test the board within 7 days (see link below or ask your TA how to test it)
If it broken at the time of this initial test,
show it to your TA who will verify that it is indeed broken,
then bring it back to the department for an exchange
If the board stops working after this initial test, you are responsible for purchasing a replacement

Buying another 9S12C32 board
The specific part to buy is ORDER CODE: NC12C32SP-SB $49.95 plus shipping. To order, click on the University of Texas at Austin link at http://www.technologicalarts.com/

How to test the 9S12C32 board.
Check out from the second floor, the orange 9S12C32 board tester.
0) Download and unzip the Tester.zip programs from this website
http://www.ece.utexas.edu/~valvano/metrowerks/
1) plug the 9S12C32 docking module into testing protoboard/ZIF socket
2) connect RS232 cable to docking module and a PC COM port
3) place the 9S12C32 in BOOT mode
4) apply power to the docking module
5) hit reset on the docking module
6) run Metrowerks and download this program (Project->Debug)
7) quit Metrowerks and start a terminal program e.g., `HyperTerminal`
set COM port to match the cable, and baud rate 115200 bits/sec
8 bit data, no parity, no hardware flow control
8) place the 9S12C32 in RUN mode
9) hit reset on the docking module

The interrupt vectors of the 9S12C32 that we will use are

```
0xFFD6    interrupt 20  SCI
0xFFDE    interrupt 16  timer overflow
0xFFE0    interrupt 15  timer channel 7
0xFFE2    interrupt 14  timer channel 6
0xFFE4    interrupt 13  timer channel 5
0xFFE6    interrupt 12  timer channel 4
0xFFE8    interrupt 11  timer channel 3
0xFFEA    interrupt 10  timer channel 2
0xFFEC    interrupt 9   timer channel 1
0xFFEE    interrupt 8   timer channel 0
0xFFF0    interrupt 7   real time interrupt
0xFFF6    interrupt 4   SWI software int
0xFFF8    interrupt 3   trap software int
0xFFFE    interrupt 0   reset
```



Valvano's general guidelines

1) Test the CPU/docking module combination within 7 days of receiving the kit.
2) Touch a grounded object before handling CMOS electronics. Try not to touch any exposed wires.
3) Cut out a piece of paper trimming it as close to the writing as possible, and place it between the docking module and the protoboard. The pins of the docking module straddle the gap running down the center of the protoboard. To push the docking module into the protoboard, push straight down.
4) Never remove the CPU module from the docking module. THE PINS ON THE CPU MODULE ARE VERY FRAGILE. On the other hand, the male pins on the docking module have been very robust as long as you limit the twisting forces. To remove the docking module from the protoboard pull straight up (or at least pull up a little at a time on each end.)
5) Use and store the system with the docking module plugged into a protoboard (this will reduce the chances of contacting the metal pins tied directly to the 6812 with either your fingers or stray electrical pulses).
6) Do not use the 9S12C32 with any external power sources, other than the supplied wall-wart. In particular, avoid connecting signals to the 6812 that are not within the 0 to +5V range.
7) Do not connect any wires to the pins labeled Vin, DTR, TX, or RX. These pins contain voltages outside the safe 0 to +5V range. Also do not connect to the Reset pin.
8) Label all your pieces (CPU module, docking module, cable, wall wart, and protoboard) with your name.

Installing Metrowerks

1) http://www.metrowerks.com/MW/download/default.asp
2) select "CodeWarrior Development Studio for HC12 Microcontrollers"
3) fill in name information
   email must be correct
   decide whether or not you want email from Metrowerks
4) Wait for email, should contain these links
To download your software, please use the following link:
ftp://ftp.metrowerks.com/pub/embedded/HC12/HC(S)12_v3_0.exe
To use your software you must have a license key, please use the following link to download one:
ftp://ftp.metrowerks.com/pub/embedded/HC12/CW12R30_SElicense.zip
5) Download HC(S)12 v3.0, install
6) Download license, unzip
7) Copy license.bat overwriting existing limited version license
   This license is available to all educational users
8) Download starter projects from
http://www.ece.utexas.edu/~valvano/metrowerks/

If you have comments or suggestions, email me at valvano@mail.utexas.edu
Jonathan Valvano

How to develop programs Metrowerks/Tech Arts 9S12C32 board
Jonathan Valvano, valvano@mail.utexas.edu, January 7, 2004

First, you need to install Metrowerks CodeWarrior for HC(S)12. You can go to the Metrowerks web site **http://www.metrowerks.com/MW/download/default.asp** get the CodeWarrior for the 6812 (you can download it or request a CD in the mail). Follow the Metrowerks instructions about downloading, installing and registering the application. Second, you put the 12K learning edition "`license.dat`" file in your Metrowerks folder. The first few lines look like

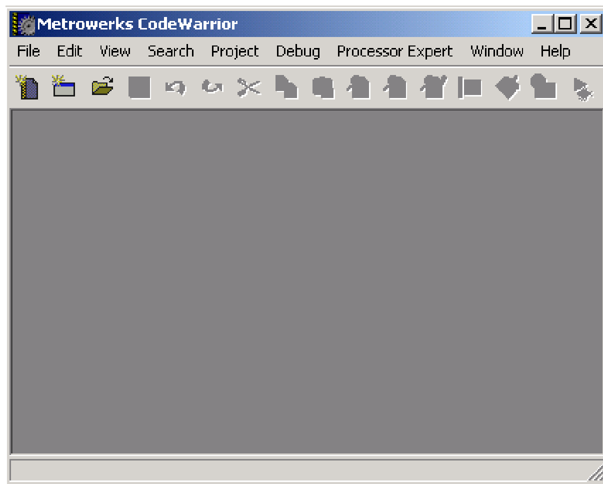
```
FEATURE Win32_CWIDE_Limited metrowks 5.5 permanent uncounted 2589EF7E8174 HOSTID=ANY
#######################################################
# ( 436): HC12 Special Edition for V3.x
# IDE: learning edition (max 32 files, no subprojects).
# Build/Debug: Unlimited Assembly/Hex/S19. C code up to 12K. ELF/Dwarf object file format.
```

A) To open an existing Metrowerks project

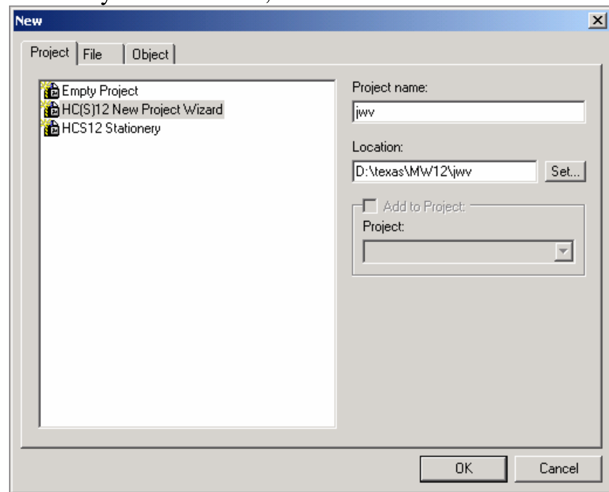
1) Start Metrowerks CW12 3.0
2) Execute File->Open, navigate to an existing *.mcp file, and click "OK"

B) How to configure create a new Metrowerks project

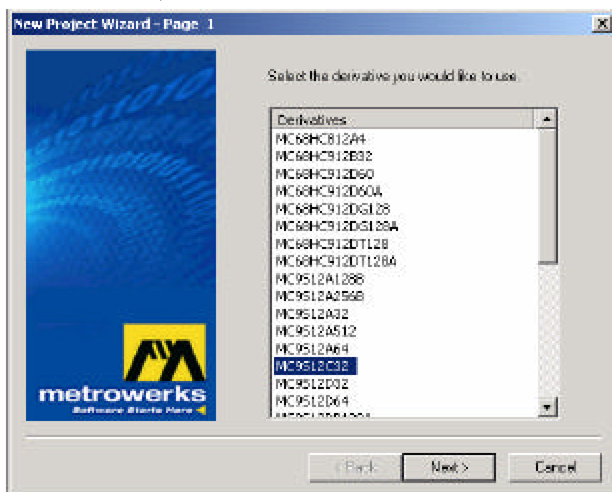
1) Start Metrowerks CW12 3.0



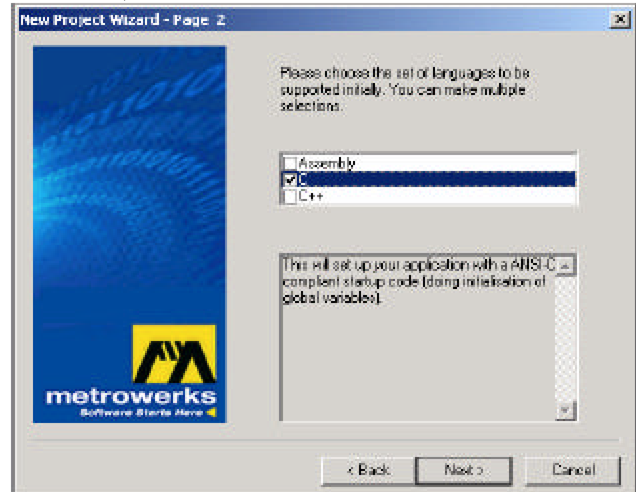
2) Execute File->New, click "Project" Tab
select "HC(S)12 New Project Wizard"
specify the "Project name"
verify its "Location", click "OK"



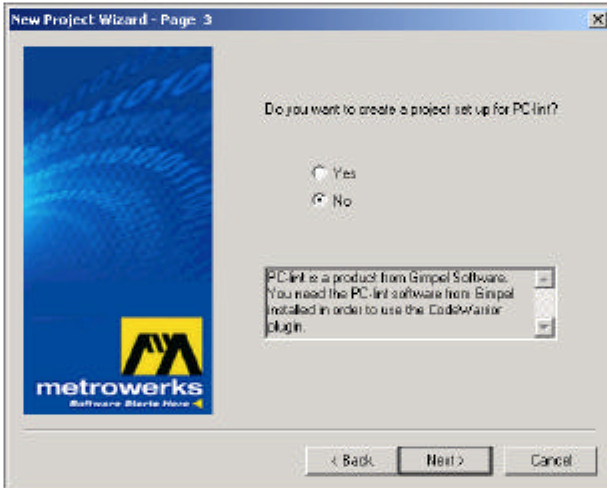
3) Select the derivative you want to use
"MC9S12C32", click "next"



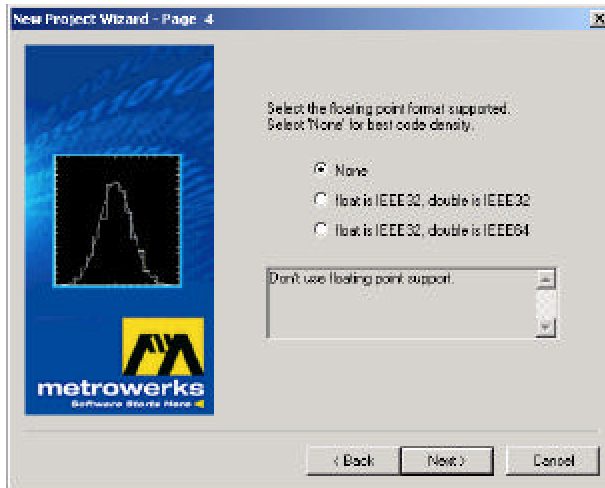
4) Choose the set of languages supported
select "C", click "next"



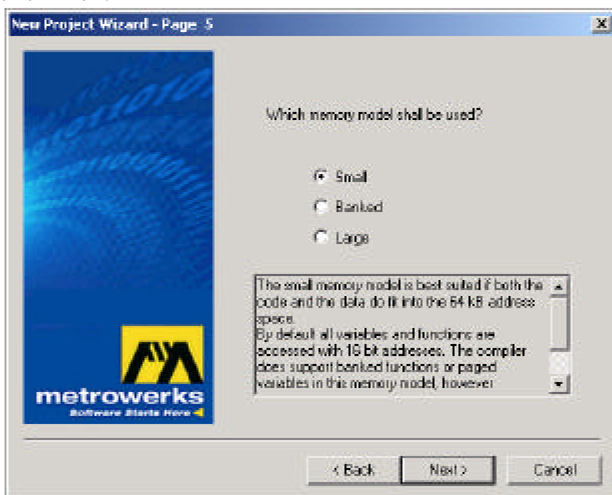
5) Do you want to create a setup for PC-lint
select "no", click "next"



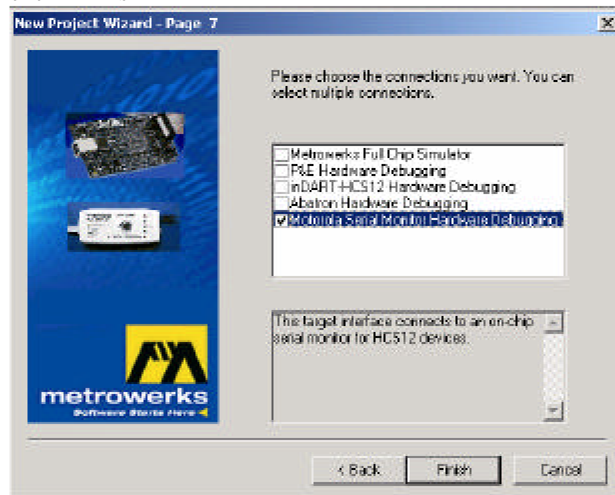
6) Select the floating point format supported
select "None", click "next"



7) Which memory model should be used?
select "Small"
click "next"



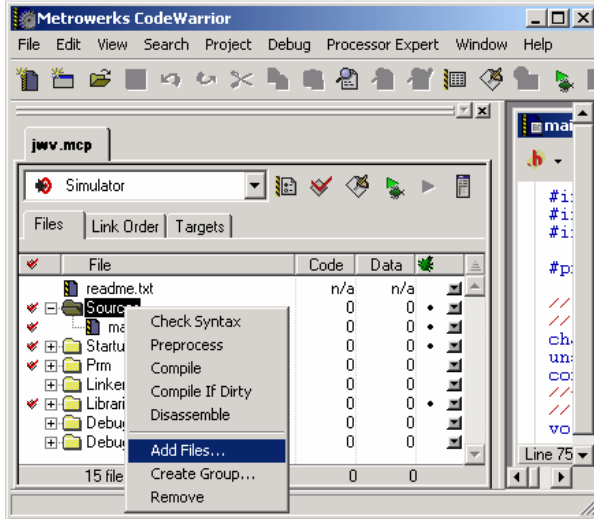
8) Please choose the connections you want
select "Serial Monitor Hardware Debugging"
click "Finish"

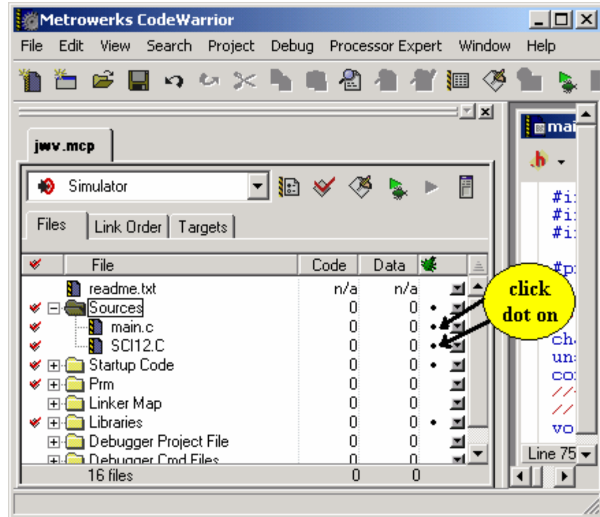


9) Create or copy program files `*.c` and `*.h`
place them into the "`Sources`" directory of your project

10) Add the necessary C files to project

click on Sources in the "`mcp`" window
right click and execute "`Add Files...`"
“click dot on” in the field associated all C source files under the "bug" icon





13) Change compiler/linker options

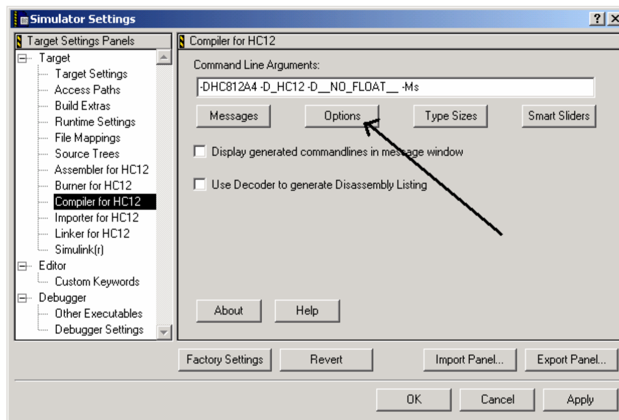
click the right-most toolbar ICON called "`Simulator settings`"

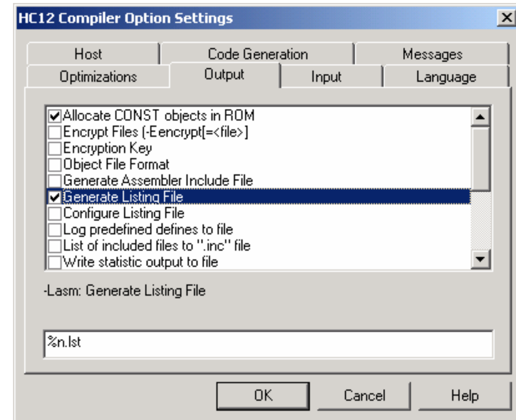
click "`Compiler for HC12`" choice

click "`Options`"

click "`Output`" tab

select "`Allocate CONST objects in ROM`" and "`Generate Listing File`"



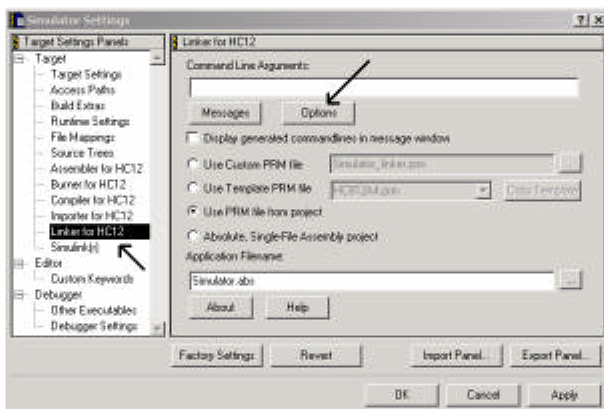


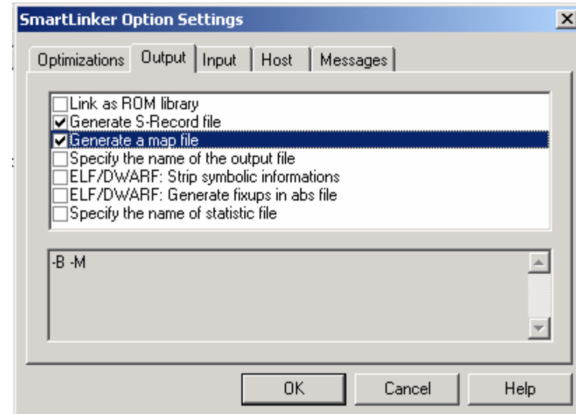
click "`Linker for HC12`" choice

click "`Options`"

click "`Output`" tab

select "`Generate S-Record`" and "`Generate a map file`"

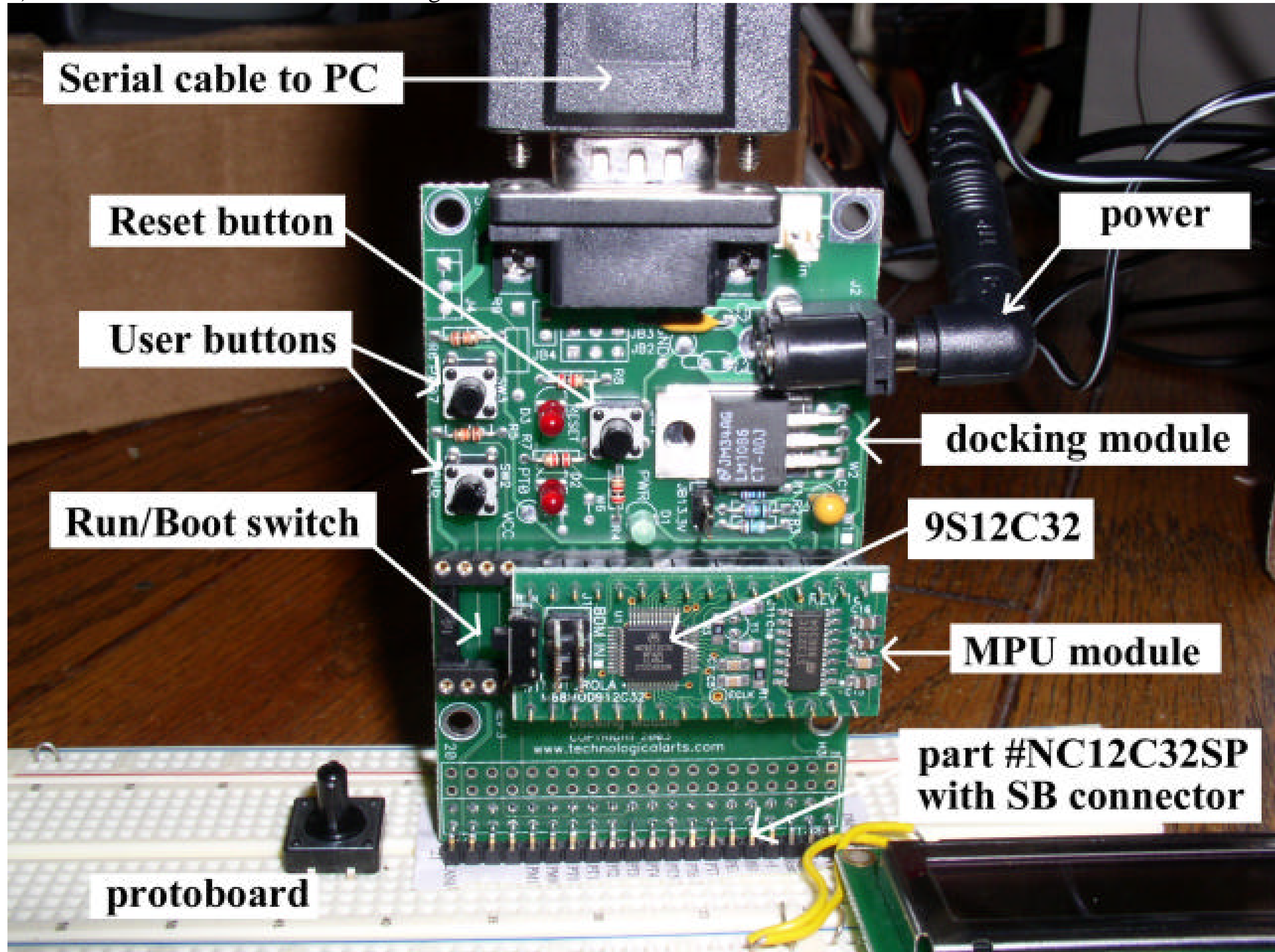




C) How to run Metrowerks on the Real 9S12C32 board

Do this once

1) Connect PC-COM1 to the 9S12C32 docking station,
2) Place the Run/Boot switch on the 9S12C32 board in Boot mode
3) Connect power to 9S12C32 docking station.
4) Touch the reset switch on the docking station



For each edit/compile/run cycle for software that does not use SCI

1) In Metrowerks, perform editing to source code
2) In Metrowerks, compile/Link/Load
   Execute **`Project->Debug`**
3) Click the green arrow in the debugger to start. Runs at 24 MHz.

For each edit/compile/run cycle for software that does use SCI

1) set the Run/Boot switch to Boot mode, push the reset button on the 9S12C32 docking station
2) execute Project->Debug (compiles and downloads code to 9S12C32)
3) quit MW debugger once programming complete. Quitting the debugger will release the COM port.
4) start a terminal program (like HyperTerminal)
   specify proper COM port, 19200 bits/sec, no flow control
5) set the Run/Boot switch to Run mode and push the reset button on the 9S12C32 docking station. Runs at 4 MHz.
6) when done, quit terminal program. Quitting the terminal program will release the COM port.

Programming Style Guidelines

1.2. Attitude

Good engineers employ well-defined design processes when developing complex systems. When we work within a structured framework, it is easier to prove our system works (verification) and to modify our system in the future (maintenance.) As our software systems become more complex, it becomes increasingly important to employ well-defined software design processes. Throughout this book, a very detailed set of software development rules will be presented. This book focuses on real-time embedded systems written in assembly language, but most of the comments should apply to other situations as well. At first, it may seem radical to force such a rigid structure to software. We might wonder if creativity will be sacrificed in the process. True creativity is more about good solutions to important problems and not about being sloppy and inconsistent. Because software maintenance is a critical task, the time spent organizing, documenting, and testing during the initial development stages will reap huge dividends throughout the life of the software project.

***Observation**: The easiest way to debug is to write software without any bugs.*

We define *clients* as programmers who will use our software. A client develops software that will call our functions. We define *coworkers* as programmers who will debug and upgrade our software. A coworker, possibly ourselves, develops, tests, and modifies our software.

Writing quality software has a lot to do with attitude. We should be embarrassed to ask our coworkers to make changes to our poorly written software. Since so much software development effort involves maintenance, we should create software modules that are easy to change. In other words, we should expect each piece of our code will be read by another engineer in the future, whose job it will be to make changes to our code. We might be tempted to quit a software project once the system is running, but this short time we might save by not organizing, documenting, and testing will be lost many times over in the future when it is time to update the code.

As project managers, we must reward good behavior and punish bad behavior. A company, in an effort to improve the quality of their software products, implemented the following policies.

The employees in the customer relations department receive a bonus for every software bug that they can identify. These bugs are reported to the software developers, who in turn receive a bonus for every bug they fix.

***Checkpoint 1.2:** Why did the above policy fail horribly?*

We should demand of ourselves that we deliver bug-free software to our clients. Again, we should be embarrassed when our clients report bugs in our code. We should be mortified when other programmers find bugs in our code. There are a few steps we can take to facilitate this important aspect of software design.

Test it now. When we find a bug, fix it immediately. The longer we put off fixing a mistake the more complicated the system becomes, making it harder to find. Remember that bugs do not go away on their own, but we can make the system so complex that the bugs will manifest themselves in a mysterious and obscure fashion. For the same reason, we should completely test each module individually, before combining them into a larger system. We should not add new features before we are convinced the existing system is bug-free. In this way, we start with a working system, add features, then debug this system until it is working again. This incremental approach makes it easier to track progress. It allows us to undo bad decisions, because we can always revert back to a previous working system. Adding new features before the old ones are debugged is very risky. With this sloppy approach, we could easily reach the project deadline with 100% of the features implemented, but have a system that doesn't run. In addition, once a bug is introduced, the longer we wait to remove it, the harder it will be to correct. This is particularly true when the bugs interact with each other. Conversely, with the incremental approach, when the project schedule slips, we can deliver a working system at the deadline that supports some of the features.

***Maintenance Tip:** Go from working system to working system.*

Plan for testing. How to test each module should be considered at the start of a project. In particular, testing should be included as part of the software design. Our testing and the client's usage go hand in hand. In particular, how we test the software module will help the client understand the context and limitations of how our software is to be used. On the other hand, a clear understanding of how the client wishes to use our software is critical for both the software design and its testing.

Maintenance Tip: *It is better to have some parts of the system that run with 100% reliability than to have the entire system with bugs.*

Get help. Use whatever features are available for organization and debugging. Pay attention to warnings, because they often point to misunderstandings about data or functions. Misunderstanding of assumptions that can cause bugs when the software is upgraded, or reused in a different context than originally conceived. Remember that computer time is a lot cheaper than programmer time.

Maintenance Tip: *It is better to have a software system that runs slow than one that does run at all.*

In the early days of microcomputer systems, software size could be measured in 100's of lines of source code or 1000's of bytes of object code. These early systems, due to their small size, were inherently simple. The explosion of hardware technology (both in speed and size) has lead to a similar increase in the size of software systems. The only hope for success in a large software system will be to break it into simple modules. In most cases, the complexity of the problem itself can not be avoided. E.g., there is just no simple way to get to the moon. Nevertheless, a complex system can be created out of simple components. A real creative effort is required to orchestrate simple building blocks into larger modules, which themselves are grouped. Use our creativity to break a complex problem into simple components, rather than developing complex solutions to simple problems.

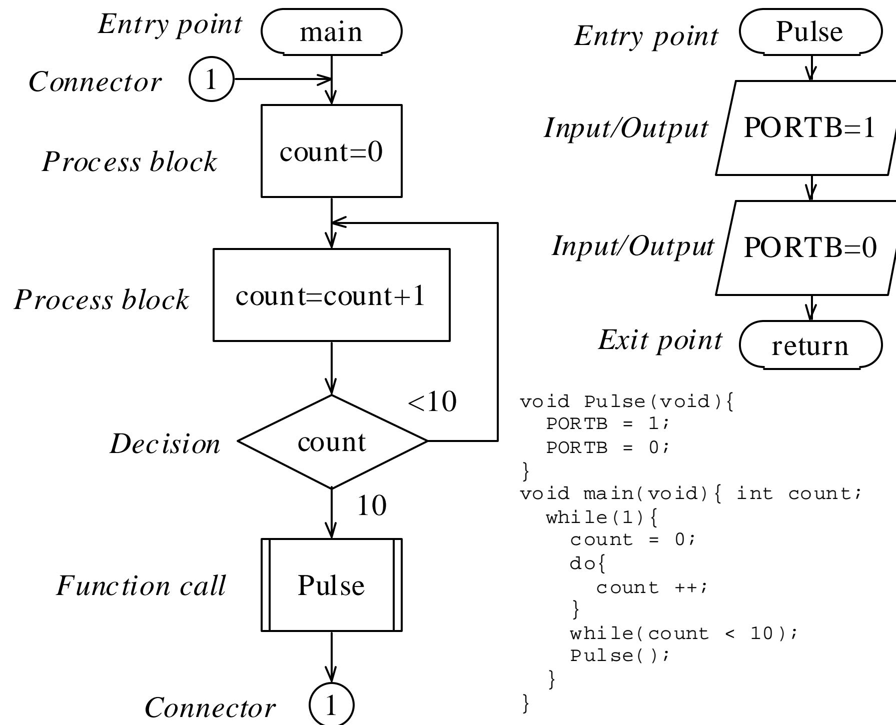
Observation*: There are two ways of constructing a software design: one way is to make it so simple that there are* obviously *no deficiencies and the other way is make it so complicated that there are no* obvious *deficiencies. C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb. 1981.*

1.4. Flowcharts and structured programming

Next, we introduce the flowchart syntax that will be used throughout the book. Programs themselves are written in a linear or one-dimensional fashion. Writing programs this way is a natural process, because the computer itself usually executes the program in a top-to-bottom sequential fashion. This one-dimensional format is fine for simple programs, but conditional branching and function calls may create complex behaviors that are not easily observed in a linear fashion. Flowcharts are one way to describe software in a two-dimensional format, specifically providing convenient mechanisms to visualize conditional branching and function calls. You will not have to draw a flowchart for every program you write. However, flowcharts are very useful in the initial design stage of a software system to define complex algorithms. Flowcharts can also be used in the final documentation stage of a project, once the system is operational, in order to assist in its use or modification.

Observation*:* ***TExaS*** *is one of the few software development systems that allow you to add flowcharts directly into your software as part of its documentation.*

Figure 1.2 illustrates the flowchart syntax, showing both the flowcharts and corresponding C program. The oval shapes define entry and exit points. The main *entry point* is the starting point of the software. Each function, or subroutine, also has an entry point. The *exit point* returns the flow of control back to the place from which the function was called. When the software runs continuously, as is typically the case in an embedded system, there will be no main exit point. We use rectangles to specify *process* blocks. In a high-level flowchart, a process block might involve many operations, but in a low-level flowchart, the exact operation is defined in the rectangle. The parallelogram will be used to define an *input/output* operation. Some flowchart artists use rectangles for both processes and input/output. Since input/output operations are an important part of embedded systems, we will use the parallelogram format, which will make it easier to identify input/output in our flowcharts. The diamond-shaped objects define a branch point or *decision* block. The rectangle with double lines on the side specify a call to a *predefined function*. In this book, functions, subroutines and procedures are terms that all refer to a well-defined section of code that performs a specific operation. Functions usually return a result parameter, while procedures usually do not. Functions and procedures are terms used when describing a high-level language, while subroutines often used when describing assembly language. When a function (or subroutine or procedure) is called, the software execution path jumps to the function, the specific operation is performed, and the execution path returns to the point immediately after the function call. Circles are used as *connectors*.




```
void Pulse(void){
  PORTB = 1;
  PORTB = 0;
}
void main(void){ int count;
  while(1){
    count = 0;
    do{
      count ++;
    }
    while(count < 10);
    Pulse();
  }
}
```

Figure 1.2. Example flowchart showing some common flowchart symbols.

> ***Common error.** In general, it is bad programming style to develop software that requires a lot of connectors when drawing its flowchart.*

> ***Checkpoint 1.7:** Using a flowchart describe the control algorithm that a toaster must use to cook toast. Assume the inputs are toast temperature in F, and desired temperature in F. The output is heat (on/off).*

There are an almost infinite number of operations one can perform on a computer, and the key to developing great products is to select the correct ones. Just like hiking through the woods, we need to develop guidelines (like maps and trails) to keep us from getting lost. One of the fundamental issues when developing software, especially in assembly language, is to maintain a consistent structure. One such framework is called **structured programming**. Most high-level languages (with the exception of `goto`) force the programmer to write structured programs. Structured programs are built from three basic building blocks: the **sequence**, the **conditional**, and the **while-loop**. At the lowest level, the process block contains simple and well-defined commands, like the process blocks shown in Figure 1.2. I/O functions are also low-level building blocks. Structured programming involves combining existing blocks into more complex structures, as shown in Figure 1.3.

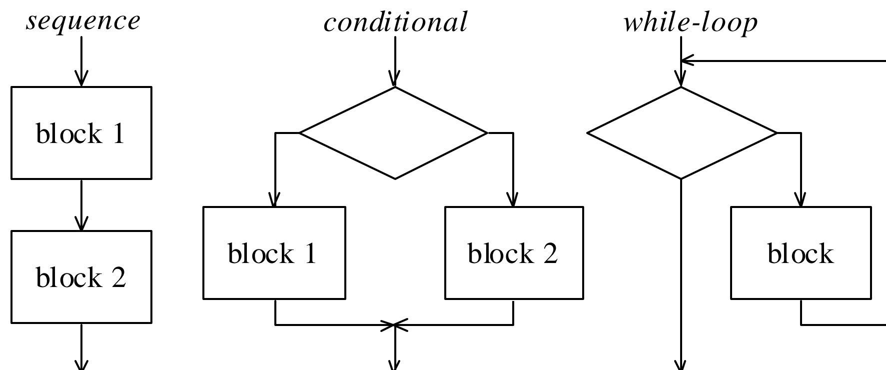


Figure 1.3. Flowchart showing the basic building blocks of structured programming.

1.5. Product development cycle

In this section, we will introduce the product development process in general. The basic approach is introduced here, and the details of these concepts will be presented throughout the remaining chapters of the book. As we learn software/hardware development tools and techniques, we can place them into the framework presented in this section. As illustrated in Figure 1.4, the development of a product follows an analysis -design-implementation-testing cycle. For complex systems with long life-spans, we tranverse multiple times around the development cycle. For simple systems, a one-time pass may suffice.

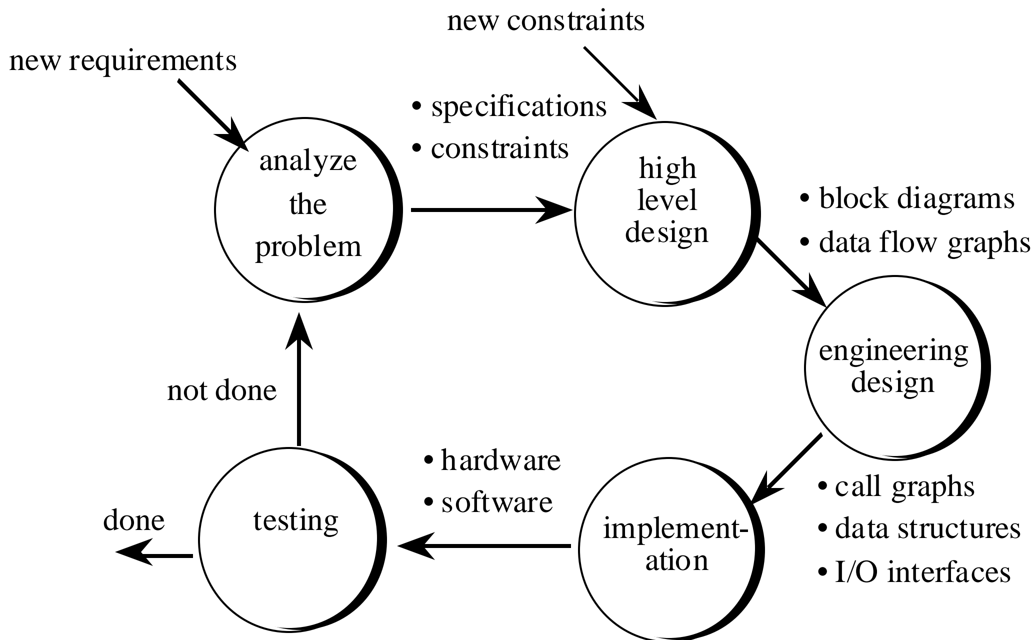


Figure 1.4. Software development cycle.

During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information. A *requirement* is a specific parameter that the system must satisfy. We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed *specifications.* In general, specifications are detailed parameters describing how the system should work. For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device. For example, suppose we wish to build a thermometer. During the analysis phase, we would determine obvious specifications such as range, resolution, accuracy, and speed. There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of calibration, display readability, and reliability. A *constraint* is a limitation, within which the system must operate. The system may be constrained to such factors as compatibility with other products, use of specific electronic and mechanical parts as other devices, interfaces with other instruments and test equipment, and development schedule.

Checkpoint 1.8: *What's the difference between a requirement and a specification?*

During the **high-level design** phase, we build a conceptual model of the hardware/software system. It is in this model that we exploit as much abstraction as appropriate. The project is broken in modules or subcomponents. Modular design will be presented in Chapter 9. During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide if the project has a high enough potential for profit. A *data flow graph* is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components and the ovals are software modules. We use data flow graphs in the high-level design, because they describe the overall operation of the system while hinding the details of how it works. A data flow graph for a simple thermometer is shown in Figure 1.5. The sensor converts temperature in an electrical resistance. The amplifier converts resistance into the 0 to +5V voltage range required by the ADC. The ADC converts analog voltage into a digital sample. The `ADC routines`, using the ADC and timer hardware, collect samples and calculate voltages. The `calculation` software uses a table data structure to convert voltage to temperature. Voltage and temperature data are represented as fixed-point

numbers within the computer. The temperature data is passed to the `LCD routines` creating ASCII strings, which will be sent to the *liquid crystal display* (LCD) module. The user will be able to select the Fahrenheit or Centigrade scale using a switch.

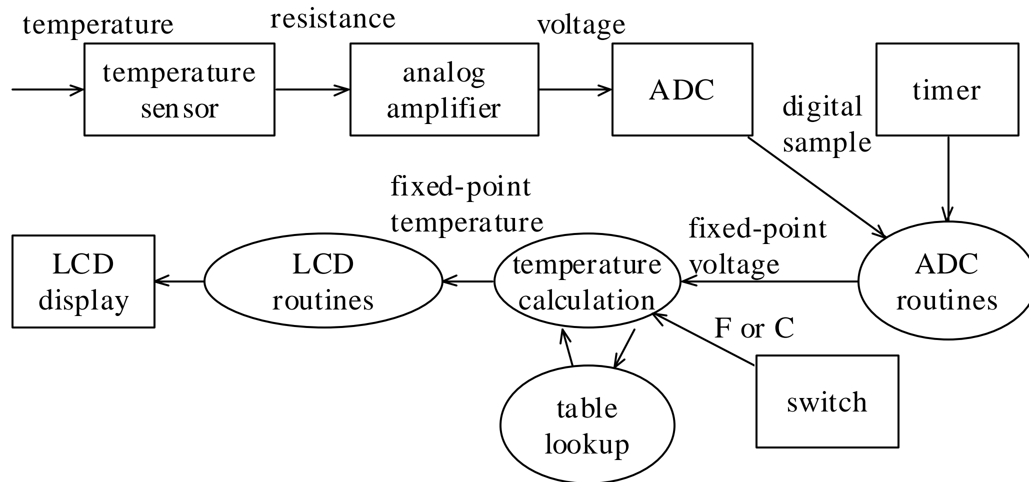


Figure 1.5. A data flow graph showing how the temperature signal passes through a simple thermometer.

The next phase is **engineering design**. We begin by constructing a preliminary design. This system includes the overall top down hierarchical structure, the basic I/O signals, shared data structures and overall software scheme. At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design. Next, we finish the top down hierarchical structure, and built mock-ups of the mechanical parts (connectors, chassis, cables etc.) and user software interface. Sophisticated 3-D CAD systems can create realistic images of our system. Detailed hardware designs must include mechanical drawings. It is a good idea to have a second source, which is an alternative supplier that can sell our parts if the first source can't deliver on time. *Call-graphs* are a graphical way to define how the software/hardware modules interconnect. *Data structures*, which will be presented in Chapter 10, include both the organization of information and mechanisms to access the data. A call-graph for a simple thermometer is shown in Figure 1.6. Again, rectangles represent hardware components and ovals show software modules. An arrow points from the calling routine to the module it calls. The I/O ports are organized into groups and placed at the bottom of the graph. A high-level call-graph, like the one shown in Figure 1.6, shows only the high-level hardware/software modules. A detailed call-graph would include each software function and I/O port. Normally, hardware is passive and the software initiates hardware/software communication, but as we will learn in Chapter 11, it is possible for the hardware to interrupt the software and cause certain software modules to be run. In this system, the timer hardware will cause the `ADC` software to collect a sample. The `main program` gets the next sample from the `ADC` software, converts it to temperature, and displays the result by calling the `LCD` interface software.

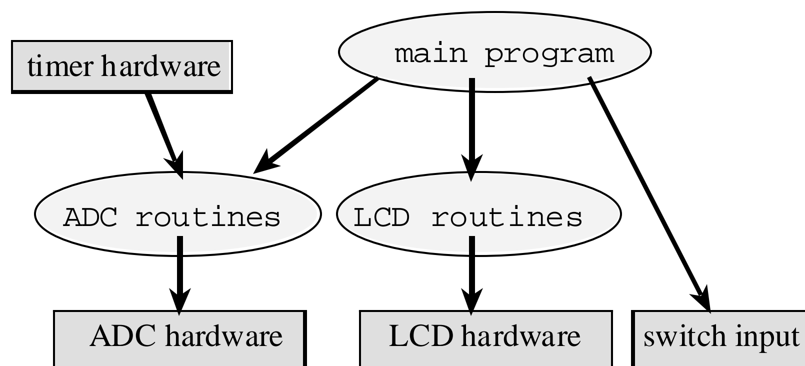


Figure 1.6. A call flow graph for a simple thermometer.

***Observation**: If module A calls module B, and B returns data, then a data flow graph will show an arrow from B to A, but a call-graph will show an arrow from A to B.*

The next phase is **implementation**. An advantage of a top-down design is that implementation of subcomponents can occur concurrently. During the initial iterations of the development cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator versus constructing a physical device out of actual components. Rapid prototyping is important in the early stages of product development. This allows for more loops around the analysis -design-implementation-testing cycle, which in turn leads to a more sophisticated product.

Recent software and hardware technological developments have made significant impacts on the software development for embedded microcomputers. The simplest approach is to use a cross-assembler or cross-compiler to convert source code into the machine code for the target system. The machine code can then be loaded into the target machine. Debugging embedded systems with this simple approach is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even although the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evalutate the performance of our system. First, we debug the system and validate basic functions. Next, we use careful measurements to optimize performance such as static efficiency (memory requirements), dynamic efficiency (execution speed), accuracy (difference between truth and measured), and stability (consistent operation.)

Maintenance is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints. To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the development cycle.

1.6. Quality Programming

Embedded system development is similar to other engineering tasks. We can choose to follow well-defined procedures during the development and evaluation phases, or we can meander in a haphazard way and produce code that is hard to test and harder to change. The ultimate goal of the system is to satisfy the stated objectives such as accuracy, stability, and input/output relationships. Nevertheless it is appropriate to separately evaluate the individual components of the system. Therefore in this section, we will evaluate the quality of our software. There are two categories of performance criteria with which we evaluate the "goodness" of our software. Quantitative criteria include dynamic efficiency (speed of execution), static efficiency (memory requirements), and accuracy of the results. Qualitative criteria center around ease of software maintenance. Another qualitative way to evaluate software is ease of understanding. If your software is easy to understand then it will be:

easy to debug (fix mistakes)
easy to verify (prove correctness)
easy to maintain (add features)

***Common error:** Programmers who sacrifice clarity in favor of execution speed often develop software that runs fast, but does work and can't be changed.*

Golden Rule of Software Development
Write software for others as you wish they would write for you.

1.6.1. Quantitative Performance Measurements

In order to evaluate our software quality, we need performance measures. The simplest approaches to this issue are quantitative measurements. *Dynamic efficiency* is a measure of how fast the program executes. It is measured in seconds or CPU cycles. *Static efficiency* is the number of memory bytes required. Since most embedded computer systems have both

RAM and ROM, we specify memory requirement in global variables, stack space, fixed constants and program object code. The global variables plus maximum stack size must be less than the available RAM. Similarly, the fixed constants plus program size must be less than the ROM size. We can judge our software system according to whether or not it satisfies given constraints, like software development costs, memory available, and time-table.

1.6.2. Qualitative Performance Measurements

Qualitative performance measurements include those parameters to which we can not assign a direct numerical value. Often in life the most important questions are the easiest to ask, but the hardest to answer. Such is the case with software quality. So therefore we ask the following qualitative questions. Can we prove our software works? Is our software easy to understand? Is our software easy to change? Since there is no single approach to writing the best software, we can only hope to present some techniques that you may wish to integrate into your-own software style. In fact, this book devotes considerable effort to the important issue of developing quality software. In particular, we will study self-documented code, abstraction, modularity, and layered software. These issues indeed play a profound effect on the bottom-line financial success of our projects. Although quite real, because there is often not a immediate and direct relationship between a software's quality and profit, we may be mistakenly tempted to dismiss their importance.

To get a benchmark on how good a programmer you are, take the following two challenges. In the first test, find a major piece of software that you have written over 12 months ago, then see if you can still understand it enough to make minor changes in its behavior. The second test is to exchange with a peer a major piece of software that you have both recently written (but not written together), then in the same manner, see if you can make minor changes to each other's software.

Observation: *You can tell if you are a good programmer if 1) you can understand your own code 12 months later, and 2) others can make changes to your code.*

9.3. Naming convention

Choosing names for variables and functions involves creative thought, and it intimately connected to how we feel about ourselves as programmers. Of the policies presented in this section, naming conventions may be the hardest habit for us to break. The difficulty is that there are many conventions that satisfy the "easy to understand" objective. Good names reduce the need for documentation. Poor names promote confusion, ambiguity, and mistakes. Poor names can occur because code has been copied from a different situation and inserted into our system without proper integration (i.e., changing the names to be consistent with the new situation.) They can also occur in the cluttered mind of a second-rate programmer, who hurries to deliver software before it is finished.

Names should have meaning. If we observe a name out of the context of the program in which it exists, the meaning of the object should be obvious. The object `TxFifo` is clearly the transmit first in first out circular queue. The function `LCD_outString` will output a string to the LCD display.

Avoid ambiguities. Don't use variable names in our system that are vague or have more than one meaning. For example, it is vague to use `temp`, because there are many possibilities for temporary data, in fact, it might even mean temperature. Don't use two names that look similar, but have different meanings.

Give hints about the type. We can further clarify the meaning of a variable by including phrases in the variable name that specify its type. For example, `dataPt timePt putPt` are pointers. Similarly, `voltageBuf timeBuf pressureBuf` are data buffers. Other good phrases include `Flag Mode U L Index Cnt`, which refer to boolean flag, system state, unsigned 16-bit, signed 32-bit, index into an array, and a counter respectively.

Use the same name to refer to the same type of object. For example, everywhere we need a local variable to store an ASCII character we could use the name `letter`. Another common example is to use the names `i j k` for indices into arrays. The names `V1 R1` might refer to a voltage and a resistance. The exact correspondence is not part of the policies presented in this section, just the fact that a correspondence should exist. Once another programmer learns which names we use for which types of object, understanding our code becomes easier.

Use a prefix to identify public objects. An underline character will separate the module name from the function name. As an exception to this rule, we can use the underline to delimit words in all upper-case name (e.g., `#define MIN_PRESSURE 10`). Functions that can be accessed outside the scope of a module will begin with a prefix specifying the module to which it belongs. It is poor style to create public variables, but if they need to exist, they too would begin with the module prefix. The prefix matches the file name containing the object. For example, if we see a function call, `LCD_OutString("Hello world");` we know the public function belongs to the LCD module, where the policies are defined in `LCD.h` and the implementation in `LCD.c`. Notice the similarity between this syntax (e.g., `LCD_init()`) and the corresponding syntax we would use if programming the module as a class in C++ (e.g., `LCD.init()`). Using this convention, we can easily distinguish public and private objects. If the variable is public, because the name has an

underline, then the first letter of the name after the underline should be capitalized (e.g., `my_Count` is a public variable belonging the the module "`my`" and defined in the header file `my.h`.)

Use upper and lower case to specify the scope of an object. We will define I/O ports and constants using no lower-case letters, like typing with caps-lock on. In other words, names without lower-case letters refer to objects with fixed values. `TRUE FALSE` and `NULL` are good examples of fixed-valued objects. As mentioned earlier, constant names formed from multiple words will use an underline character to delimit the individual words. E.g., `MAX_VOLTAGE UPPER_BOUND FIFO_SIZE`. Global objects will begin with a capital letter, but include some lower-case letters. Local variables will begin with a lower-case letter, and may or may not include upper case letters. Since all functions are global, we can start function names with either an upper-case or lower-case letter. Using this convention, we can distinguish constants, globals and locals.

An object's properties (public/private, local/global, constant/variable) are always perfectly clear at the place where the object is defined. The importance of the naming policy is to extend that clarity also to the places where the object is used.

Use capitalization to delimit words. Names that contain multiple words should be defined using a capital letter to signify the first letter of the word. Recall that the case of the first letter specifies whether is the local or global. Some programmers use the underline as a word-delimiter, but except for constants, we will reserve underline to separate the module name from the variable name. Table 9.1 overviews the naming convention presented in this section.

Checkpoint 9.5: *How can tell if a function is private or pubic?*

Checkpoint 9.6: *How can tell if a variable is local or global?*

type	examples
constants	`CR SAFE_TO_RUN PORTA STACK_SIZE START_OF_RAM`
local variables	`maxTemperature lastCharTyped errorCnt`
private global variable	`MaxTemperature LastCharTyped ErrorCnt`
public global variable	`DAC_MaxTemperature Key_LastCharTyped Network_ErrorCnt file_OpenFlag`
private function	`ClearTime wrapPointer InChar`
public function	`Timer_ClearTime RxFifo_Put Key_InChar`

Table 9.1. Examples of names.

Observation: *Software can be made easier to understand by reworking the approach in order to reduce the number of conditional branches.*

```
short my_Data1; // in global ram, public
static short Data2; // in global ram, private
const short DATA3=5; // in EEPROM, if in C file, public if in h file
```

9.5. C language Style Guidelines

9.5.1. Code File Structure, the *.c file

One of the recurring themes of this software style section is consistency. Maintaining a consistent style will help us locate and understand the different components of our software, as well as prevent us from forgetting to include a component or worse including it twice. The following regions should occur in this order in every code file (e.g., `file.c`).

Opening comments. The first line of every file should contain the file name. This is because some printers do not automatically print the name of the file. Remember that these opening comments will be duplicated in the corresponding header file (e.g., `file.h`) and are intended to be read by the client, the one who will use these programs. If major portions of this software are copied from copyrighted sources, then we must satisfy the copyright requirements of those sources. The rest of the opening comments should include

- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module, and
- any copyright information.

Including .h files. Next, we will place the `#include` statements that add the necessary header files. Adding other code files, if necessary, will occur at the end of the file, but here at the top of the file we include just the header files.

Normally the order doesn't matter, so we will list the include files in a hierarchical fashion starting with the lowest level and ending at the highest high. If the order of these statements is important, then write a comment describing both what the proper order is and why the order is important. Putting them together at the top will help us draw a call-graph, which will show us how our modules are connected. In particular, if we consider each code file to be a separate module, then the list of `#include` statements specifies which other modules can be called from this module. Of course one header file is allowed to include other header files. In general, we should avoid having one header file include other header files. This restriction makes the organizational structure of the software system easier to observe. Be careful to include only those files that are absolutely necessary. Adding unnecessary include statements will make our system seem more complex than it actually is.

`extern` references. After including the header files, we can declare any external variables or functions. External references will be resolved by the linker, when various modules are linked together to create a single executable application. Placing them together at the top of the file will help us see how this software system fits together (i.e., is linked to) other software systems.

`#define` statements. After external references, we should place the `#define` macros and `#define` constants. Since these definitions are located in the code file (e.g., `file.c`), they will be private. This means they are available within this file only. If the client does not need to use or change the macro or constant, then it should be made private by placing it here in the code file. Conversely, if we wish to create a public constant or macro, then we place it in the header file for this module.

`struct union enum` statements. After the define statements, we should create the necessary data structures using `struct union` and `enum`. Again, since these definitions are located in the code file (e.g., `file.c`), they will be private.

Global variables and constants. After the structure definitions, we should include the global variables and constants. If we specify the global as `static` then it will be private, and can only be accessed by programs in this file. If we do not specify the global as `static` then it will be public, and can only be accessed any program (that program defines it as `extern` and the linker will resolve the reference). We put all the globals together before any function definitions to symbolize the fact that any function in this file has access to these globals. If we have a permanent variable that is only access by one function, then it should be defined as a static local. The **scope** of a variable includes all the software in the system that can access it. In general, we wish to minimize the scope of our data.

```
short publicGlobal; // accessible by any function via extern declaration
static short privateGlobal; // accessible in this file only
void function(void){
static short veryPrivateGlobal; // accessible by this function only
}
```

Maintain order in our system by restricting direct access to our data.

Prototypes of private functions. After the globals, we should add any necessary prototypes. Just like global variables, we can restrict access to private functions by defining them as `static`. Prototypes for the public functions will be included in the corresponding header file. In general, we will arrange the code implementations in a top-down fashion. Although not necessary, we will include the parameter names with the prototypes. Descriptive parameter names will help document the usage of the function. For example, which of the following prototypes is easier to understand?

```
static void plot(short, short);
static void plot(short time, short pressure);
```

Implementations of the functions. The heart of the implementation file will be, of course, the implementations. Again, private functions should be defined as static. The functions should be sequenced in a logical manner. The most typical sequence is top-down, meaning we begin with the highest level and finish with the lowest level. Another appropriate sequence mirrors the manner in which the functions will be used. For example, start with the initialization functions, followed by the operations, and end with the shutdown functions. For example:

open
input
output
close

Including .c files. At the end of the file, we will place the `#include` statements that add the necessary code files. If our compiler supports projects, then it is a good idea to take advantage of this feature. The project simplifies the management of large software systems by providing organizational structure to the software system. If we use projects, then including code files will be unnecessary, and hence should be avoided. If our compiler does not support projects, or if we are writing software for multiple compilers, then including code files allows a large software project to be constructed simply

by compiling the file with the `main()` program in it. Including header files at the top of the file allows this module to accessing public variables and functions of the other module. On the other hand, including code files at the end of the file prevents this module from accessing private variables and functions of the other module.

If our compiler supports **assert()** functions, use them liberally. In particular, place them at the beginning of functions to test the validity of the input parameters. Place them after calculations to test the validity of the results. Place them inside loops to verify indices and pointers are valid. There is a secondary benefit to using **assert()**. The **assert()** statements themselves provide documentation of the assumptions made by the programmer.

9.5.2. Header File Structure, the *.h file

Once again, maintaining a consistent style facilitates understanding and helps to avoid errors of omission. Definitions made in the header file will be public, i.e., accessible by all modules. As stated earlier, it is better to make global variables private rather than placing them in the header file. Similarly, we should avoid placing actual code in a header file.

There are two types of header files. The first type of header file has no corresponding code file. In other words, there is a `file.h`, but no `file.c`. In this type of header, we can list global constants and helper macros. Examples of global constants are I/O port addresses (e.g., `HC12.h`) and calibration coefficients. Debugging macros could be grouped together and placed in a `debug.h` file. We will not consider software in these types of header files as belonging to a particular module.

The second type of header file does have a corresponding code file. The two files, e.g., `file.h`, and `file.c`, form a software module. In this type of header, we define the prototypes for the public functions of the module. The `file.h` contains the policies (behavior or what it does) and the `file.c` file contains the mechanisms (functions or how it works.) The following regions should occur in this order in every header file (e.g., `file.h`).

Opening comments. The first line of every file should contain the file name. This is because some printers do not automatically print the name of the file. Remember that these opening comments should be duplicated in the corresponding header file (e.g., `file.c`) and are intended to be read by the client, the one who will use these programs. We should repeat copyright information as appropriate. The rest of the opening comments should include

- the overall purpose of the software module,
- the names of the programmers,
- the creation (optional) and last update dates,
- the hardware/software configuration required to use the module, and
- any copyright information.

Including .h files. Nested includes in the header file should be avoided. As stated earlier, nested includes obscure the manner in which the modules are interconnected. On the other, an implementation file can include other header and implementation files.

`#define` statements. Public constants and macros are next. Special care is required to determine if a definition should be made private or public. One approach to this question is to begin with everything defined as private, and then shift definitions into the public category only when deemed necessary for the client to access in order to use the module. If the parameter relates to what the module does or how to use the module, then it should probably be public. On the other hand, if it relates to how it works or how it is implemented, it should probably be private.

`struct union enum` statements. The definitions of public structures allow the client software to create data structures specific for this module.

Global variables and constants. If at all possible, public global variables should be avoided. Public constants follow the same rules as public definitions. If the client must have access to a constant to use the module, then it could be placed in the header file.

Prototypes of public functions. The prototypes for the public functions are last. Just like the implementation file, we will arrange the code implementations in a top-down fashion. Comments should be directed to the client, and these comments should clarify what the function does and how the function can be used.

9.5.3. Formatting

The rules set out in this subsection are not necessary for the program to compile or to run. Rather the intent of the rules are to make the software easier to understand, easier to debug, and easier to change. Just like beginning an exercise program, these rules may be hard to follow at first, but the discipline will pay dividends in the future.

Make the software easy to read. I strongly object to hardcopy printouts of computer programs during the development phase of a project. At this time, there are frequent updates made by multiple members of the software development team. Because a hardcopy printout will be quickly obsolete, we should develop and debug software by observing it on the computer screen. In order to eliminate horizontal scrolling, no line of code should be more than 80 characters wide. If we do make hard copy printouts of the software at the end of a project, this rule will result in a printout that is easy to read.

Indentation should be set at 2 spaces. When transporting code from one computer to another, the TAB settings may be different. So, what looks good on one computer may look ugly on another. For this reason, we should avoid TABs and use just spaces. Local variable definitions can go on the same line as the function definition, or in the first column on the next line.

Be consistent about where we put spaces. Similar to English punctuation, there should be no space before a comma or a semicolon, but there should be at least one space or a carriage return after a comma or a semicolon. There should be no space before or after open or close parentheses. Assignment and comparison operations should have a single space before and after the operation. One exception to the single space rule is if there are multiple assignment statements, we can line up the operators and values. For example

```
  data     = 1;
  pressure = 100;
  voltage  = 5;
```

Be consistent about where we put braces {}. Misplaced braces cause both syntax and semantic errors, so it is critical to maintain a consistent style. Place the opening brace at the end of the line that opens the scope of the multi-step statement. The only code that can go on the same line after an opening brace is a simple local variable declaration or a comment. Placing the open brace near the end of the line provides a visual clue that a new code block has started. Place the closing brace on a separate line to give a vertical separation showing the end of the multi-step statement. The horizontal placement of the close brace gives a visual clue that the following code is in a different block. For example

```
void main(void){ int i, j, k;
  j = 1;
  if(sub0(j)){
    for(i = 0; i < 6; i++){
      sub1(i);
    }
    k = sub2(i, j);
  }
  else{
    k = sub3();
  }
}
```

Use braces after all if else for do while case and switch commands, even if the block is a single command. This forces we to consider the scope of the block making it easier to read and easier to change. For example, assume we start the following code.

```
  if(flag)
    n = 0;
```

Now, we add a second statement that we want to execute also if the flag is true. The following error might occur if we just add the new statement.

```
  if(flag)
    n = 0;
    c = 0;
```

If all of our blocks are enclosed with braces, we would have started with the following.

```
  if(flag){
    n = 0;
  }
```

Now, when we add a second statement, we get the correct software.

```
  if(flag){
    n = 0;
    c = 0;
  }
```

9.5.4. Code Structure

Make the presentation easy to read. We define presentation as the look and feel of our software as displayed on the screen. If at all possible, the size of our functions should be small enough so the majority of the code fits on a single computer screen. We must consider the presentation as a two-dimensional object. Consequently, we can reduce the 2-D

area of our functions by encapsulating components and defining them as private functions, or by combining multiple statements on a single line. In the horizontal dimension, we are allowed to group multiple statements on a single line only if the collection makes sense. We should list multiple statements on a single line, if we can draw a circle around the statements and assign a simple collective explanation to the code.

> ***Observation:*** *Most professional programmers do not create hard copy printouts of the software. Rather, software is viewed on the computer screen.*

Another consideration related to listing multiple statements on the same line is debugging. The compiler often places debugging information on each line of code. For example, the ICC11 and ICC12 compilers place a label specifying the starting address of the assembly code that implements the line. Breakpoints in some systems can only be placed at the beginning of a line.

Consider the following three presentations. Since the compiler generates exactly the same code in each case, the computer execution will be identical. Therefore, we will focus on the differences in style. The first example has a horrific style.

```
void testFilter(short start, short stop, short step){ short x,y;
  initFilter(); SCI_OutString("x(n) y(n)"); SCI_OutChar(CR);
  for(x=start;x<=stop; x=x+step){ y=filter(x); SCI_OutUDec(x);
  SCI_OutChar(SP); SCI_OutUDec(y); SCI_OutChar(CR);} }
```

The second example places each statement on a separate line. Although written in an adequate style, it is unnecessarily vertical.

```
void testFilter(short start, short stop, short step){
short x;
short y;
  initFilter();
  SCI_OutString("x(n) y(n)");
  SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x);
    SCI_OutChar(SP);
    SCI_OutUDec(y);
    SCI_OutChar(CR);
  }
}
```

The last example groups the two variable definitions together because the collection can be considered as a single object. The variables are related to each other. Obviously, `x` and `y` are the same type (16-bit signed), but in a physical sense, they would have the same units. For example, if `x` represents a signal in mV, then `y` is also a signal in mV. Similarly, the SCI output sequences cause simple well-defined operations.

```
void testFilter(short start, short stop, short step){ short x, y;
  initFilter();
  SCI_OutString("x(n) y(n)"); SCI_OutChar(CR);
  for(x = start; x <= stop; x = x+step){
    y = filter(x);
    SCI_OutUDec(x); SCI_OutChar(SP); SCI_OutUDec(y); SCI_OutChar(CR);
  }
}
```

The "make the presentation easy to read" guideline sometimes comes in conflict with the "be consistent where we place braces" guideline. For example, the following example is obviously easy to read, but violates the placement of brace rule.

```
  for(i = 0; i < 6; i++) dataBuf[i] = 0;
```

When in doubt, we will always be consistent where we place the braces. The correct style is also easy to read.

```
  for(i = 0; i < 6; i++){
    dataBuf[i] = 0;
  }
```

Employ modular programming techniques. Complex functions should be broken into simple components, so that the details of the lower-level operations are hidden from the overall algorithms at the higher levels. An interesting question arises:

"Should a subfunction be defined if it will only be called from a single place?"

The answer to this question, in fact the answer to all questions about software quality, is yes if it makes the software easier to understand, easier to debug, and easier to change.

Minimize scope. In general, we hide the implementation of our software from its usage. The scope of a variable should be consistent with how the variable is used. In a military sense, we ask the question, *"Which software has the need to know?"* Global variables should be used only when the lifetime of the data is permanent, or when data needs to be passed from one thread to another. Otherwise, we should use local variables. When one module calls another, we should pass data using the normal parameter-passing mechanisms. As mentioned earlier, we consider I/O ports in a manner similar to global variables. There is no syntactic mechanism to prevent a module from accessing an I/O port, since the ports are at fixed and known absolute addresses. The Intel Pentium does have a complex hardware system to prevent unauthorized software from accessing I/O ports, but the details are beyond the scope of this book. So for our embedded system, we must rely on the **does-access** rather than the **can-access** method. In other words, we must have the discipline to restrict I/O port access only in the module that is designed to access it. For similar reasons, we should consider each interrupt vector address separately, grouping it with the corresponding I/O module. In particular, rather than having one long list of interrupt vectors for the entire system, each interrupt vector should be separately defined along with the software that supports the other I/O hardware of the module. For example, the serial port interrupt vector should be specified in the same file as the serial port interrupt handler.

Use types. Using a `typedef` will clarify the format of a variable. It is another example of the separation of mechanism and policy. New data types and structures will begin with an upper case letter. The `typedef` allows use to hide the representation of the object and use an abstract concept instead. For example

```
typedef short Temperature;
void main(void){ Temperature lowT, highT;
}
```

This allows us to change the representation of temperature without having to find all the temperature variables in our software. Not every data type requires a `typedef`. We will use types for those objects of fundamental importance to our software, and for those objects for which a change in implementation is anticipated. As always, the goal is to clarify. If it doesn't make it easier to understand, easier to debug, or easier to change, don't do it.

Prototype all functions. Public functions obviously require a prototype in the header file. In the implementation file, we will organize the software in a top-down hierarchical fashion. Since the highest level functions go first, prototypes for the lower-level private functions will be required. Grouping the low-level prototypes at the top provides a summary overview of the software in this module. Include both the type and name of the input parameters. Specify the function as `void` even if it has no parameters. These prototypes are easy to understand:

```
void start(unsigned short seriod, void(*functionPt)(void));
short divide(short dividend, short divisor);
void SCI_Init(void);
```

These prototypes are harder to understand:

```
start(unsigned short, (*)());
short divide(short, short);
SCI_Init();
```

Declare function return types explicitly. In general, we can remove ambiguities by clarifying exactly what we want. Unless the number of parameters is large, we will place the return type, the function name, and the input parameters on a single line. If there is still room within the 80-character line limit, we can add some local variable declarations to this line. The following are good examples of the first line of several functions.

```
void main(void){ int i;
void SCI_OutUDec(unsigned short number){
unsigned short SCI_InUHex(void){
int RxFifo_Put(char data){
```

Declare data and parameters as `const` whenever possible. Declaring an object as const has two advantages. The compiler can produce more efficient code when dealing with parameters that don't change. The second advantage is to catch software bugs, i.e., situations where the program incorrectly attempts to modify data that it should not modify.

goto statements are not allowed. Debugging is hard enough without adding the complexity generated when using `goto`. A corollary to this rule is when developing assembly language software, we should restrict the branching operations to the simple structures allowed in C.

++ and -- should not appear in complex statements. These operations should only appear as commands by themselves. Again, the compiler will generate the same code, so the issue is readability. The statement

```
  *(--pt) = buffer[n++];
```

should have been written as

```
  --pt;
  *(pt) = buffer[n];
  n++;
```

If it makes sense to group, then put them on the same line. The following code is allowed

```
  buffer[n] = 0; n++;
```

Be a parenthesis zealot. When mixing arithmetic, logical, and conditional operations, explicitly specify the order of operations. Do not rely on the order of precedence. As always, the major issue is clarity. Even if the following code were actually to perform the intended operation (which in fact it does not),

```
  if( x + 1 & 0x0F == y | 0x04)
```

the programmer assigned to modify it in the future will have a better chance if we had written

```
  if( ((x + 1) & 0x0F) == (y | 0x04))
```

Use `enum` instead of `#define` or `const`. The use of `enum` allows for consistency checking during compilation, and provides for easy to read software. A good optimizing compiler will create the exact object code for the following four examples. So once again, we focus on style. In the first example, we needed comments to explain the operations.

```
int Mode;  // 0 means error
void function1(void){
  Mode = 1; // no error
}

void function2(void){
  if(Mode == 0){ // error?
    SCI_OutString("error");
  }
}
```

In the second example, no comments are needed.

```
#define NOERROR 1
#define ERROR 0
int Mode;
void function1(void){
  Mode = NOERROR;
}
void function2(void){
  if(Mode == ERROR){
    SCI_OutString("error");
  }
}
```

In the third example, the compiler performs a type-match, making sure `mode`, `NOERROR`, and `ERROR` are the same type.

```
const int NOERROR = 1;
const int ERROR = 0;
int Mode;
```

```
void function1(void){
  Mode = NOERROR;
}
void function2(void){
  if(Mode == ERROR){
    SCI_OutString("error");
  }
}
```

Enumeration provides a check of both type and value. We can explicitly set the values of the enumerated types if needed.

```
enum Mode_state{ ERROR, NOERROR};
enum Mode_state mode;
void function1(void){
  mode = NOERROR;
}
void function2(void){
  if(mode == ERROR){
    SCI_outString("error");
  }
}
```

Don't use bit-shift for arithmetic operations. Microcomputer architectures and compilers used to be so limited that it made sense to perform multiply/divide by 2 using a shift operation. For example, when multiplying a number by 4, we might be tempted to write `data<<2`. This is wrong; if the operation is multiply, we should write `data*4`. Compiler optimization has developed to the point where the compiler can choose to implement `data*4` as either a shift or multiply depending on the instruction set of the computer. When we use `data*4`, we have code that is easier to understand than `data<<2`.

Reference:

Mike Dahlin, ed., "LESS Software Engineering", coding standard for the LESS group.

9.7. Comments

Discussion about comments was left for last, because they are the least important aspect involved in writing quality software. It is much better to write well-organized software with simple interfaces having operations so easy to understand that comments are not necessary.

The beginning of every file should include the file name, purpose, hardware connections, programmer, date, and copyright. E.g.,

```
// filename  adtest.c
// Test of 6812 8-bit ADC
// 1 Hz sampling and output to the serial port
// Last modified 1/7/05 by Jonathan W. Valvano
// Copyright 2005 by Jonathan W. Valvano, valvano@mail.utexas.edu
//    You may use, edit, run or distribute this file
//    as long as the above copyright notice remains
```

The beginning of every function should include a line delimiting the start of the function, purpose, input parameters, output parameters, and special conditions that apply. The comments at the beginning of the function explain the policies (e.g., how to use the function.) These comments, which are similar to the comments for the prototypes in the header file, are intended to be read by the client. E.g.,

```
//----------------------SCI_InUDec-------------------------------
// InUDec accepts ASCII input in unsigned decimal format
//     and converts to a 16 bit unsigned number
//     valid range is 0 to 65535
// Input: none
// Output: 16-bit unsigned number
// If you enter a number above 65535, it will truncate without an error
// Backspace will remove last digit typed
```

Comments can be added to a variable or constant definition to clarify the usage. In particular, comments can specify the units of the variable or constant. For complicated situations, we can use addition lines and include examples. E.g.,

```
short V1;             // voltage at node 1 in mV, range -5000 mV to +5000 mV
unsigned short Fs;    // sampling rate in Hz
int FoundFlag;        // 0 if keyword not yet found, 1 if found
unsigned short Mode;  // determines system action, as one of the following three
cases
#define IDLE 0
#define COLLECT 1
#define TRANSMIT 2
```

Comments can be used to describe complex algorithms. These types of comments are intended to be read by our coworkers. The purpose of these comments is to assist in changing the code in the future, or applying this code into a similar but slightly different application. Comments that restate the function provide no additional information, and actually make the code harder to read. Examples of bad comments include:

```
  time++;    // add one to time
  mode = 0;  // set mode to zero
```

Good comments explain why the operation is performed, and what it means:

```
  time++;    // maintain elapsed time in msec
  mode = 0;  // switch to idle mode because no more data is available
```

We can add spaces so the comment fields line up. As stated earlier, we avoid tabs because they often do not translate from one system to another. In this way, the software is on the left and the comments can be read on the right.

I taught a large programming class one semester, and being an arrogant and lazy fellow, I thought I could write a grading program that accepts the students' programming assignments and automatically generates and records their grades. (The second step would have been to write a self-study book, then I could teach the masses without ever having to show up for work.) My grading program worked OK for the functional aspects of the students' software. My program generated inputs, called the students' program and compared the results with expected behavior. Where I utterly failed was in my attempts to automatically grade their software on style. I used the following three part "quality" statistic. First, I measured execution speed the student's software, s_i. Smaller times represent improved dynamic efficiency. Next, I measured the number of bytes in the object code, b_i. Again, a smaller number represents better static efficiency. Third, I used the number of ASCII characters in the source code, c_i, as a quantitative measure of documentation. For this parameter, bigger is better. In a typical statistical fashion, I used the average and standard deviation to calculate

$$\text{quality} = \frac{\bar{s}-s_i}{\sigma_s} + \frac{\bar{b}-b_i}{\sigma_b} + \frac{c_i-\bar{c}}{\sigma_c}$$

Half way through the semester, I happened to look at some assignments and was horrified to find the all-time worst software ever written from both a style and content basis. To improve speed and reduce size, the students cut so many corners that their code didn't really work anymore, it just appeared to work to my grading program. Then they took the ugly mess and filled it with nonsense comments, giving it the appearance of extensive documentation. To my students in that class that semester, I sincerely apologize. We should write comments for coworkers who must change our software, or clients who will use our software.

Appendix 9. Solutions Manual

A9.1. Checkpoint Solutions

Checkpoint 1.2: It failed because employees were rewarded for poor behavior. It is much better to punish poor behavior and reward good behavior.

Checkpoint 1.7:

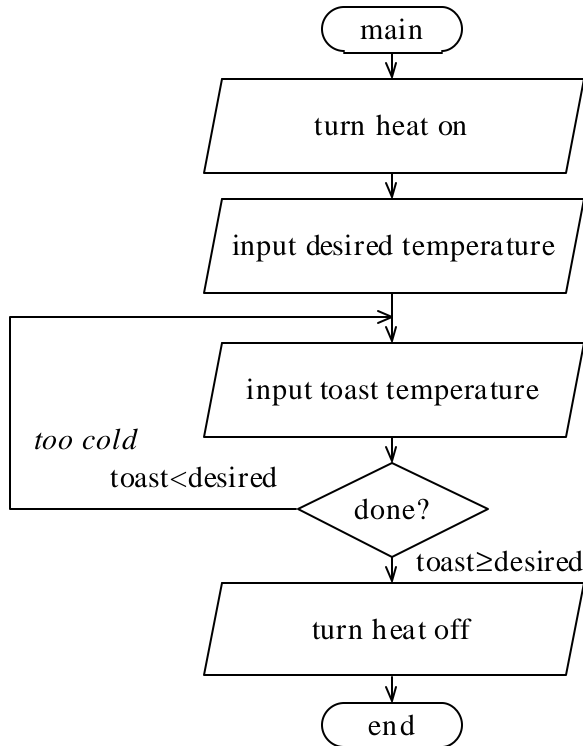


Figure A9.1. Flowchart showing a toaster algorithm.

Checkpoint 1.8: Both terms refer to parameters of a system, but the differences lie in the level of detail used to describe the parameter. A requirement is usually defined in general terms, whereas a specification entails detailed engineering rigor. A requirement often refers to an objective of the system, while a specification describes how well the actual device works.
Checkpoint 9.5. Public functions have an underline. E.g., `SCI_OutString`.
Checkpoint 9.6. Local variables begin with a lower case letter E.g., `myKey`. Global variables begin with an upper case letter E.g., `TheKey`.

Important changes to Embedded Microcomputer Systems, 3rd printing

Page 118, last line change
```
port<unsigned char> InPort(0x0003,0x0007);  // bidirectional port
```
to
```
port<unsigned char> InPort(0x1003,0x1007);  // bidirectional port
```
Page 199, last line change
```
(Resulttx)>>1
```
to
```
(Result+x)>>1
```
Page 208, program 4.16 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 209, program 4.17 change
```
cpx PutPt  Empty if initially the same
```
to
```
cpy PutPt  Empty if initially the same
```
Page 209, program 4.17 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 211, program 4.20 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 212, program 4.21 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 214, program 4.24 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 215, program 4.25 change
```
tpa        restore CCR to previous value
tba
```
to
```
tap        restore CCR to previous value
tba
```
Page 215, program 4.25 change
```
stab 0,x      Return by reference
ldab GetI
```
to (line up op code, operand, comments with other lines)
```
stab 0,x      Return by reference
dec  Size     one less element in FIFO
```

```
          ldab GetI
```

Page 279, change the sequence of Program 5.13 so it reads as follows

```
* To block a thread on semaphore S, execute SWI
SWIhan ldx  RunPt   running process "to be blocked"
       sts  SP,x    save Stack Pointer in its TCB
* Unlink "to be blocked" thread from RunPt list
       ldy  Next,x  find previous thread (Figure 5.10)
       sty  RunPt   next one to run
look   cpx  Next,y  search to find previous
       beq  found
       ldy  Next,y
       bra  look
found  ldd  RunPt  one after blocked (Figure 5.11)
       std  Next,y  link previous to next to run
* Put "to be blocked" thread on block list
       ldy  BlockPt (Figure 5.12)
       sty  Next,x   link "to be blocked"
       stx  BlockPt
* Launch next thread
       ldx  RunPt
       lds  SP,x    set SP for this new thread
       ldd  TCNT    Next thread gets a full 10ms time slice
       addd #20000  interrupt after 10 ms
       std  TOC5
       ldaa #$08    ($20 on the 6812)
       staa TFLG1   clear OC5F
       rti
```

Page 307, Program 6.10, 6811 C code, change ")" to "}"

```
    First = TIC1; Count=0; Mode=1;
    if(((TIC1&0x8000)==0)
       &&(TFLG2&0x80)) Count--;)
```

to

```
    First = TIC1; Count=0; Mode=1;
    if(((TIC1&0x8000)==0)
       &&(TFLG2&0x80)) Count--;}
```

Page 382, Change **SCxDR** to **SCxDRL** (in RDRF description)

page 418 Change PUPEJ to PULEJ three places, two places in the laast paragraph, once in program 8.1

Page 434, change

```
  PUPEJ = 0;      // regular input
```

to

```
  PULEJ = 0;      // regular input
```

Page 437, Program 8.11, MC68HC812A4 version, remove spaces after the two comma, change

```
Ritual: clr  DDRJ ;PJ3-PJ0 inputs
        movb #$0F, PUPSJ
        movb #$0F, PULEJ
        rts       ;PJ7-PJ0 oc outputs
```

to

```
Ritual: clr  DDRJ ;PJ3-PJ0 inputs
        movb #$0F,PUPSJ
        movb #$0F,PULEJ
        rts       ;PJ7-PJ0 oc outputs
```

Page 450, Program 8.15, MC68HC812A4 version, change

```
   TC5=TOC5+10000; // every 5 ms
```

to

```
   TC5=TC5+10000;  // every 5 ms
```

Page 468, line 2, change ***made*** to ***make***
Page 506, Table 9.5, change

```
PF4     CSD     0xxxxxxxxxxxxxxx   $0000    $7FFF    32K (CSDFH=0)
```

to

```
PF4     CSD     0xxxxxxxxxxxxxxx   $0000    $7FFF    32K (CSDFH=1)
```

Page 525, Change in the last line from
SMODN, MODB, MODA register.
to
SMODN, MODB, MODA bits in the MODE register .
Page 569, Figure 9.82, wrong figure. Should be

Page 569, Change
"The write timing when controlled by **C1** is shown on the left in Figure 9.82; the write timing when controlled by **C2** is shown on the right in Figure 9.82."
to
"The write function occurs on either the fall of **C1** or the rise of **C2**, whichever occurs first. Let the setup time be $\mathbf{t_{su}}$ and assume the hold time is zero."
Page 635, 7 lines from the bottom change
last time of interface
to
last type of interface
Page 636, 13 lines from the bottom change
output capture interrupt
to
output compare interrupt
Page 809, Program 15.2 (6811 version)
change

```
for(i=5;i>0;i++)
```

to

```
for(i=5;i>0;i--)
```

Page 814, Program 15.8,
change

```
unsigned char Median(unsigned char u1,unsigned char u2,unsigned char u3){
unsigned char result;
  if(u1>u2)
    if(u2>u3)   result=u2;     // u1>u2,u2>u3       u1>u2>u3
      else
        if(u1>u3) result=u3;   // u1>u2,u3>u2,u1>u3 u1>u3>u2
        else      result=u1;   // u1>u2,u3>u2,u3>u1 u3>u1>u2
  else
    if(u3>u2)   result=u2;     // u2>u1,u3>u2       u3>u2>u1
      else
        if(u1>u3) result=u3;   // u2>u1,u2>u3,u1>u3 u2>u1>u3
        else      result=u1;   // u2>u1,u2>u3,u3>u1 u2>u3>u1
  return(result):}
```

to

```
unsigned char median(unsigned char u1,unsigned char u2,unsigned char u3){
unsigned char result;
  if(u1>u2)
    if(u2>u3)   result=u2;     // u1>u2,u2>u3       u1>u2>u3
      else
```

```
      if(u1>u3) result=u3;   // u1>u2,u3>u2,u1>u3 u1>u3>u2
      else      result=u1;   // u1>u2,u3>u2,u3>u1 u3>u1>u2
  else
    if(u3>u2)   result=u2;     // u2>u1,u3>u2       u3>u2>u1
      else
      if(u1>u3) result=u1;   // u2>u1,u2>u3,u1>u3 u2>u1>u3
      else      result=u3;   // u2>u1,u2>u3,u3>u1 u2>u3>u1
  return(result):}
```

Back cover, change **74505** to **74S05**

Lab 1d. Fixed-point Conversions

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- To introduce the lab equipment,
- To familiarize yourself with Metrowerks CodeWarrior for the 6812,
- To develop a set of useful fixed-point conversion routines.

Review
- "How to program…" section located at the beginning of this laboratory manual,
- Read "Developing C Programs using ICC11/ICC12/Hiware" on the TExaS CD or at http://www.ece.utexas.edu/~valvano/embed/toc1.htm
- Valvano Sections 1.1, 1.5, 1.7, 2.1, 2.2, 2.3, 2.5, and 2.7.2 from the book Embedded Microcomputer Systems: Real Time Interfacing,
- Valvano Excerpts from Introduction to Embedded Microcomputer Systems: Motorola 6811 and 6812 Simulation located in the lab manual.

Starter files
- none

Background

The objectives of this lab are to introduce the 9S12C32 programming environment and to develop a set of useful fixed-point routines that will be used in the subsequent labs. A **software module** is a set of related functions that implement a complete task. In particular, you will create `Fixed.H` and `Fixed.C` files implement the fixed-point conversion module. An important factor in modular design is to separate the policies of the interface (how to use the software is defined in the `Fixed.H` file) from the mechanisms (how the programs are implemented, which is defined in the `Fixed.C` file.) You will develop a third file, `main.c` containing the main program, which will be used to test the fixed-point routines. You should place the prototypes for the public functions in the `Fixed.H` file. The implementations of all functions and any required private global variables should be included in the `Fixed.C` file. The two files `Fixed.H` and `Fixed.C` will be used in subsequent labs, whereas software in the `main.c` file will only be used in this lab to verify the software is operational. There are two long term usages of the main program. Sometimes we deliver the main program to our customer as an example of how our module can be used, illustrating the depth and complexity of our module. Secondly, the main program provides legal proof that we actually tested the software. A judge can subpoena files relating to testing to determine liability in a case where someone is hurt using a product containing our software.

Because the 6812 has no hardware floating point instructions, we will use fixed-point numbers when we wish to express values in our software that have noninteger values. A **fixed-point number** contains two parts. The first part is a **variable integer**, called **I**. This integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer. The **precision** of a number is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On the 6812, we typically use 8 bits or 16 bits. Extended precision can be implemented, but the execution speed will be slower because the calculations will have to be performed using software algorithms rather than with hardware instructions. This integer part is saved in memory and is manipulated by software. These manipulations include but are not limited to add, subtract, multiply, divide, convert to BCD, convert from BCD. The second part of a fixed-point number is a **fixed constant**, called **Δ**. This value is fixed, and can not be changed. The fixed constant is not stored in memory. Usually we specify the value of this fixed constant using software comments to explain our fixed-point algorithm. The value of the fixed-point number is defined as the product of the two parts:

fixed-point number ≡ **I•Δ**

The **resolution** of a number is the smallest difference in value that can be represented. In the case of fixed-point numbers, the resolution is equal to the fixed constant (Δ). Sometimes we express the resolution of the number as its units. For example, a decimal fixed-point number with a resolution of 0.001 volts is really the same thing as an integer with units of mV. When interacting with humans, it is convenient to use **decimal fixed-point**. With decimal fixed-point the fixed constant is a power of 10.

decimal fixed-point number = **I** • $10^{\mathbf{m}}$ for some fixed integer **m**

Again, the integer **m** is fixed and is not stored in memory. Decimal fixed-point will be easy to convert to ASCII for display, while **binary fixed-point** will be easier to use when performing mathematical calculations. With binary fixed-point the fixed constant is a power of 2.

binary fixed-point number = $\mathbf{I} \cdot 2^{\mathbf{n}}$ for some fixed integer **n**

In the next example, we will develop the equations that 6812 software could need to implement a digital scale. Assume the range of the position measurement system is 0 to 3 cm, and the system uses the 9S12C32's ADC to perform the measurement. The 10-bit ADC analog input range is 0 to +5 V, and the ADC digital output varies 0 to 1023. Let **x** be the distance to be measured in cm, $\mathbf{V_{in}}$ be the analog voltage and **N** be the 10-bit digital ADC output, then the equations that relate the variables are

V_{in} = 5 * N/1023 = 0.0048876* N and x = 3cm * V_{in}/5V = 0.6 (cm/V) * V_{in}

thus

x = 3 * N/1023 = 0.00293255* N where x is in cm

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.003 cm. In other words, the distance must increase or decrease by 0.003 cm for the digital output of the ADC to change by at least one number. It would be inappropriate to save the distance as an integer, because the only integers in this range are 0, 1, 2 and 3. Since the 6812 does not support floating point, the distance data will be saved in fixed-point format. Decimal fixed-point is chosen because the distance data for this distance-meter will be displayed for a human to read. A fixed-point resolution of Δ=0.001 cm could be chosen, because it matches the resolution determined by the hardware. Table 1.1 shows the performance of the system with the resolution of Δ=0.001 cm. The table shows us that we need to store the variable part of the fixed-point number in a signed or an unsigned 16-bit variable.

x distance in cm	$\mathbf{V_{in}}$ (V) Analog input	**N** ADC output	**I** (Δ= 0.001cm) variable part	**Approximation** **`(44*N+7)/15`**
0	0.000	0	0	0
0.003	0.005	1	3	3
0.600	1.000	205	600	601
1.500	2.500	512	1500	1502
3.000	5.000	1023	3000	3001

Table 1.1. Performance data of a microcomputer-based distance measurement.

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is **overflow**, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug

```
I = (3000*N)/1023;
```

because when **`N`** is greater than 21, **`3000*N`** exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an approximate calculation can be performed without overflow

```
I = (44*N)/15;
```

You can add one-half of the divisor to the dividend to implement rounding. In this case,

```
I = (44*N+7)/15;
```

The addition of "7" has the effect of rounding to the closest integer. The value 7 is selected because it is about one half of the divisor. For example, when N=4, the calculation (44*4)/15=11, whereas the "(44*4+7)/15" calculation yields the better answer of 12.

No overflow occurs with this equation using unsigned 16-bit math, because the maximum value of **`44*N`** is 45012. If you can not rework the problem to eliminate overflow, the best solution is to use promotion. Promotion is the process of performing the operation in a higher precision. For example, in C we cast the input as **`unsigned long`**, and cast the result as **`unsigned short`**

```
I = (unsigned short)((3000*(unsigned long)N)/1023);
```

Again, you can add one-half of the divisor to the dividend to implement rounding. In this case,

```
I = (unsigned short)((3000*(unsigned long)N+512)/1023);
```

The above equation will run slowly on a 6812 because there are no instructions to implement 32-bit by 32-bit arithmetic. When speed is important we can implement the calculation in assembly

```
ldd  N
ldx  #3000
emul      32-bit Y:D is 3000*N
ldx  #1023
ediv      16-bit Y is (3000*N)/1023
sty  I
```

The other error is called **drop out**. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result looses its ability to represent all of the values. It is very important to divide last when performing multiple integer calculations. If you divided first, e.g.,

```
I=44*(N/15);
```

then the values of I would be only 0, 44, 88, … or 2992.

The display algorithm for unsigned decimal fixed point with 0.001 resolution is simple:

1) display (`I/1000`) as a single digit value
2) display a decimal point
3) display (`I%1000`) as a three digital value
4) display the units "`cm`"

When adding or subtracting two fixed-point numbers with the same **D**, we simply add or subtract their integer parts. First, let x ,y, z be three fixed-point numbers with the same **D**, having integer parts I, J, K respectively. To perform z=x+y, we simply calculate K=I+J. Similarly, to perform z=x-y, we simply calculate K=I-J. When adding or subtracting fixed-point numbers with different fixed parts, then we must first convert two the inputs to the format of the result before adding or subtracting. This is where binary fixed-point is more convenient, because the conversion process involves shifting rather than multiplication/division.

In this next example, let x,y,z be three binary fixed-point numbers with the different **D**s. In particular, we define x to be $\mathbf{I}\bullet 2^{-5}$, y to be $\mathbf{J}\bullet 2^{-2}$, and z to be $\mathbf{K}\bullet 2^{-3}$. To convert x, to the format of z, we divide I by 4 (right shift twice). To convert y, to the format of z, we multiply J by 2 (left shift once). To perform z=x+y, we calculate

K=(I>>2)+(J<<1)

For the general case, we define x to be $\mathbf{I}\bullet 2^{n}$, y to be $\mathbf{J}\bullet 2^{m}$, and z to be $\mathbf{K}\bullet 2^{p}$. To perform any general operation, we derive the fixed-point calculation by starting with desired result. For addition, we have z=x+y. Next, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}+\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}\bullet 2^{n-p}+\mathbf{J}\bullet 2^{m-p}$$

For multiplication, we have z=x•y. Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}\bullet\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}\bullet\mathbf{J}\bullet 2^{n+m-p}$$

For division, we have z=x/y. Again, we substitute the definitions of each fixed-point parameter

$$\mathbf{K}\bullet 2^{p}=\mathbf{I}\bullet 2^{n}/\mathbf{J}\bullet 2^{m}$$

Lastly, we solve for the integer part of the result

$$\mathbf{K}=\mathbf{I}/\mathbf{J}\bullet 2^{n-m-p}$$

Again, it is very important to carefully consider the order of operations when performing multiple integer calculations. We must worry about both overflow and drop out. In particular, in the division example, if (n-m-p) is positive then the left shift ($\bullet 2^{n-m-p}$) should be performed before the divide (/**J**). On the other hand, if (n-m-p) is negative then the divide (/**J**) should be performed before the right shift ($\bullet 2^{n-m-p}$).

We can use these fixed-point algorithms to perform complex operations using the integer functions of our 6812. For example, consider the following digital filter calculation.

```
y = x -0.0532672*x1 + x2 + 0.0506038*y1-0.9025*y2;
```

In this case, the variables `y`, `y1`, `y2`, `x`, `x1`, and `x2` are all integers, but the constants will be expressed in binary fixed-point format. The value -0.0532672 will be approximated by $-14 \cdot 2^{-8}$. The value 0.0506038 will be approximated by $13 \cdot 2^{-8}$. Lastly, the value -0.9025 will be approximated by $-231 \cdot 2^{-8}$. The fixed-point implementation of this digital filter is

```
y = x + x2 + (-14*x1+13*y1-231*y2)/256;
```

Preparation (do this before your lab period)

There is no hardware for this lab. Please create a project that contains your three files:

`fixed.h`	header file for the fixed-point conversion module
`fixed.c`	implementation file for the fixed-point conversion module
`main.c`	main program that tests the fixed-point conversion module

A "syntax-error-free" hardcopy listing for procedure Part (2) of the software is required as preparation. The TA will check this off at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines. The comments included in `fixed.h` are intended for the client (programmers that will use your functions.) The comments included in `fixed.c` are intended for the coworkers (programmers that will debug/modify your functions.) Your programs should not perform any SCI input or output, rather you should learn how to use the interactive features of the Metrowerks debugger.

The format is signed 16-bit with a fixed-point constant of 0.001. The full-scale range is from -32.767 to 32.767. The fixed-point value -32.768 (integer -32768) signifies an error has occurred. The `Fixed_Str2Fix` function should convert an ASCII string into the signed 16-bit integer part of the fixed-point number. The specifications of `Fixed_Str2Fix` are illustrated in Program 1.1. `Fixed_Str2Fix` should round to the closest fixed-point result (e.g., 1.5995 rounds to 1.600 and 1.6004 also rounds to 1.600). You may limit the system to 16-bit signed math. In particular, some numbers like 1.2345678 might be considered illegal because they cause overflow of intermediate results. In the comments of the software, please discuss why you chose your particular implementation method over the other available choices. You are free to modify the prototypes in any way you feel is appropriate. You must check for illegal inputs, returning -32768, which is defined as an illegal number. The main program then can decide what to do on an illegal input.

Your second function converts the signed 16-bit integer part of a fixed-point number into an ASCII string. This function also performs no SCI input/output. The input/output specifications for `Fixed_Fix2Str` are also illustrated in the Program 1.1.

You are free to develop a testing file in whatever style you wish. This main program has three important roles. First, you will use it to test all the features of your program. Second, a judge in a lawsuit can subpoena this file. In a legal sense, this file documents to the world the extent to which you verified the correctness of your program. When one of your programs fails in the marketplace, and you get sued for damages, your degree of liability depends on whether you took all the usual and necessary steps to test your software, and the error was unfortunate but unforeseeable, or whether you rushed the product to market without the appropriate amount of testing and the error was a foreseeable consequence of your greed and incompetence. Third, if you were to sell your software package (`fixed.c` and `fixed.h`), your customer can use this file to understand how to use your package, its range of functions, and its limitations.

Procedure (do this during your lab period)

Part (1) Experiment with the different features of Metrowerks and its debugger. Familiarize yourself with the various options and features available in the editor/assembler/terminal. Edit, compile, download, and run your project working through all aspects of software development. In particular, learn how to:

- create hard copy listings of your source code;
- open and read the assembly listing files;
- save your source code on floppy disk;
- compile, download, and execute a program.

Part (2) Debug your software module (`fixed.h fixed.c main.c`).

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design (no software printout in the report)
D) Measurement Data (none for this lab)
E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)

You should be able to demonstrate correct operation of each routine:

- show the TA you know how to observe global variables and I/O ports using the debugger;
- demonstrate to the TA you know how to observe assembly language code;
- verify proper input/outputs of the I/O functions;
- verify the proper handling of illegal formats;
- demonstrate your software does not crash.

Your software files will be copied onto the TA's zip drive during checkout.

Hints

1) Create a new project first, then make small changes and test. Make backups of the previous versions, so that when you add something that doesn't work, you can go back to a previous working version and try a new approach. Please add documentation that makes it easier to change and use in the future. Your job is to organize these routines to facilitate subsequent laboratories.

2) You must always look at the assembly language created by the compiler to verify the appropriate function. We are using a new compiler, and bugs have been found in earlier versions. I.e., sometimes the compiler will not create the proper executable code. If you think you've found a bug, email the source and assembly listing to the TA explaining where the bug is.

3) You may find it useful read to the calculator and position measurement labs to get a feel for the context of the fixed-point routines you are developing. In particular, you should be able to use these functions in these two labs without additional modification.

```
const struct TestCase{        // used to test Fixed_Str2
  unsigned char InBuffer[10];   // Input String
  unsigned short CorrectAnswer; // proper result
  unsigned char OutBuffer[10];  // Output String
};
typedef const struct TestCase TestCaseType;
TestCaseType Tests[33]={
{  "0",            0,  "   0.000" }, // 0.000
{  "+1",        1000,  "   1.000" }, // 1.000
{  ".02",         20,  "   0.020" }, // 0.020
{  ".1",         100,  "   0.100" }, // 0.100
{  "1.",        1000,  "   1.000" }, // 1.000
{  "+12.34",   12340,  "  12.340" }, // 12.340
{  "5.05",      5050,  "   5.050" }, // 5.050
{  "-10.720", -10720,  " -10.720" }, // -10.720
{  "0.00023",      0,  "   0.000" }, // 0.000
{  "14.5995",  14600,  "  14.600" }, // 14.600
{  "14.6004",  14600,  "  14.600" }, // 14.600
{  "19.9994",  19999,  "  19.999" }, // 19.999
{  "19.9995",  20000,  "  20.000" }, // 20.000
{  "-21",     -21000,  " -21.000" }, // -21.000
{  "32.767",   32767,  "  32.767" }, // 32.767
{  "-32.767", -32767,  " -32.767" }, // -32.767
{  "33",      -32768,  "  **.***" }, // illegal, too big
{  "-33",     -32768,  "  **.***" }, // illegal, too small
{  "-32.769", -32768,  "  **.***" }, // illegal, too small
{  "32.769",  -32768,  "  **.***" }, // illegal, too big
{  "32.77",   -32768,  "  **.***" }, // illegal, too big
{  "32.8",    -32768,  "  **.***" }, // illegal, too big
{  "3*2",     -32768,  "  **.***" }, // illegal, illegal character
{  "-3A.769", -32768,  "  **.***" }, // illegal, illegal character
{  "-32.7b9", -32768,  "  **.***" }, // illegal, illegal character
{  "3-2.767", -32768,  "  **.***" }, // illegal, illegal order
{  "3.2.767", -32768,  "  **.***" }, // illegal, two decimal points
{  ".",       -32768,  "  **.***" }, // illegal, no numbers
{  "-",       -32768,  "  **.***" }, // illegal, no numbers
```

```
{  "+",        -32768,  "  **.***" }, // illegal, no numbers
{  "",         -32768,  "  **.***" }, // illegal, no numbers
{  "-+1",      -32768,  "  **.***" }, // illegal, too many signs
{  "+-1",      -32768,  "  **.***" }, // illegal, too many signs
};
unsigned short Result;       // fixed-point resolution 0.001
unsigned short I;
unsigned short Errors;
unsigned char Buffer[10];
void main(void){ // possible main program that tests your functions
  Errors = 0;
  EnableInterrupts;
  for(I=0; I<33; I++){
    Result = Fixed_Str2Fix(Tests[I].InBuffer); // convert string to fixed point
    if(Result != Tests[I].CorrectAnswer){
      Errors++;
    }
    Fixed_Fix2Str(Tests[I].CorrectAnswer,Buffer);
    if(strcmp(Buffer, Tests[I].OutBuffer)){
      Errors++;
    }
  }
  for(;;) {} /* wait forever */
}
```

Program 1.1. One approach to software testing

Lab 2f Performance Debugging

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- to develop software debugging techniques,
  - Performance debugging (dynamic or real time)
  - Profiling (detection and visualization of program activity)
- to pass data using a FIFO queue,
- to learn how to use the logic analyzer.

Review
- Valvano Section 2.11 on debugging,
- Port T and TCNT of the 9S12C32 Technical Data Manual,
- RTI interrupts on the 9S12C32 in the Technical Data Manual,
- Logic analyzer instructions.

Starter files
- `Lab2f` project

Background

Every programmer is faced with the need to debug and verify the correctness of his or her software. In this lab, we will study hardware-level techniques like the oscilloscope; and software-level tools like simulators, monitors, and profilers. **Nonintrusiveness** is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. **Intrusiveness** is used as a measure of the degree of perturbation caused in program performance by the debugging instrument itself. For example, a `SCI_OutUDec` statement added to your source code is very intrusive because it significantly affects the real-time interaction of the hardware and software. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. In a real microcomputer system, breakpoints and single-stepping are also intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. On the other hand, dumps, dumps with filter and monitors (e.g., output strategic information on LEDs) are much less intrusive. A logic analyzer that passively monitors the activity of the software by observing the memory bus cycles is completely nonintrusive. An in-circuit emulator is also nonintrusive because the software input/output relationships will be the same with and without the debugging tool. Similarly, breakpoints and single-stepping on a simulator like **TExaS** are nonintrusive, because the simulated hardware and the software are affected together.

Often on an embedded system like the 9S12C32 with limited debugging facilities, initially we define strategic variables as global (e.g., `BackData ForeData`), when proper software principles dictate they should be local. We define them as global to simplify the debugging procedures. Once the system is debugged, we can redefine them as local. On the other hand, some variables must be defined global (e.g., `NumLost`), because we want them to be permanent.

Preparation (do this before lab starts)

1. Copy the `Lab2f` project from class web site. Compile this system and observe the assembly listing file (e.g., `main.lst`). Print out that portion of the assembly listing that implements RTI interrupt service routine. Circle on the listing, those assembly instructions that access the local variable `i`. Also look at the associated map file. Print out that portion of the map file showing where is the variables `BackData ForeData NumLost` are stored in memory. We will be using the listing and map files to debug our C programs.
2. Connect PT1 and PT0 to the dual channel scope, run the program and describe its behavior.

   the falling edge of PT1 means start of foreground waiting

the rising edge of PT1 means start of foreground processing
the rising edge of PT0 means start of interrupt
the falling edge of PT0 means end of interrupt

3. In this part, we will study the execution speed of the routine **`LLFifo_Get`** the hard way. Open the assembly listing files showing **`LLFifo_Get`** (**`LLFIFO.lst`**) and **`Heap_Release`** (**`HEAP.lst`**). This time edit the assembly listing files leaving just the assembly code that implements **`LLFifo_Get`** and **`Heap_Release`**. Include also the assembly code from the main program that calls **`LLFifo_Get`**. Print out this assembly listing of these two functions. Please don't print the entire listing files, just the parts that implement these two functions. Look up the cycle counts for each instruction in the two functions, and record them on the printout. Estimate the total time in µs required to call and execute the **`LLFifo_Get`** function. In boot mode, the 9S12C32 runs at 24MHz. In run mode, the 9S12C32 executes at 4 MHz. Later, we will learn how to initialize the phase-lock-loop (PLL) so that we can run the 9S12C32 at 24MHz in run mode. When you will get to a conditional branch, you need to make assumptions about which way execution will go (i.e., assume the fifo is not empty).

Procedure (do this during lab)

A. Instrumentation measuring with an independent counter, `TCNT`

In the preparation, you estimated the execution speed of the **`LLFifo_Get`** routine by counting instruction cycles. This is a tedious, but accurate technique on a computer like the 6812 (when running in single chip mode). It is accurate because each instruction (e.g., **`LDY 2,X`**) always executes in exactly the same amount of time. Cycle counting can't be used in situations where the execution speed depends on external device timing (e.g., think about how long it would take to execute **`SCI_InChar`**.) If the 6812 were to be running in expanded mode, the time for each instruction would depend also on whether it is accessing internal or external memory. On more complex computers, there are many unpredictable factors that can affect the time it takes to execute single instructions, many of which can't be predicted *a priori*. Some of these factors include an instruction cache, out of order instruction execution, branch prediction, data cache, virtual memory, dynamic RAM refresh, DMA accesses, and coprocessor operation. For systems with these types of activities, it is not possible to predict execution speed simply by counting cycles using the processor data sheet. Luckily, most computers have a timer that operates independently from these activities. In the 6812, there is a 16-bit counter, called **`TCNT`**, which is incremented every E clock. The 9S12C32 has a prescaler that can be placed between the E clock and the **`TCNT`** counter. Leave the prescaler at its default value of divide by 1 (**`TCNT`** incremented each bus cycle). It automatically rolls over when it gets to $FFFF. If we are sure the execution speed of our function is less than (65535 counts), we can use this timer to directly measure execution speed with only a modest amount of intrusiveness. Let **`First`** and **`Delay`** be unsigned 16-bit global integers. The following code will set the variable **`Delay`** to the execution speed of **`LLFifo_Get`**.

```
LLFifo_Init();     // Initialize linked list
LLFifo_Put(1);     // make sure there is something in the fifo
First = TCNT;
LLFifo_Get(&ForeData);
Delay = TCNT-First-8;
```

The constant "8" is selected to account for the time overhead in the measurement itself. In particular, run the following,

```
First = TCNT;
Delay = TCNT-First-8;
```

and adjust the "8" so that the result calculated in **`Delay`** is zero. Use this method to verify the general expression you developed as part of the preparation.

`// with debugging print`	`// with debugging dump`
`int LLFifo_Get(unsigned short *datapt){`	`unsigned short ptBuf[10];`

```
NodePtr pt;  int success;
  success = 0;
  if(GetPt){
    *datapt = GetPt->value;
    pt = GetPt;
    GetPt = GetPt->NextPt;
    if(GetPt==NULL
      PutPt = NULL;
    }
    SCI_OutUHex(pt);
    SCI_OutChar(SP);
    SCI_OutUDec(pt->value);
    SCI_OutChar(CR);
    Heap_Release((unsigned short*)pt);
    success = 1;
  }
  return(success);
}
```

```
unsigned short dataBuf[10];
unsigned short Debug_n=0;
int LLFifo_Get(unsigned short *datapt){
NodePtr pt;  int success;
  success = 0;
  if(GetPt){
    *datapt = GetPt->value;
    pt = GetPt;
    GetPt = GetPt->NextPt;
    if(GetPt==NULL
      PutPt = NULL;
    }
    if(Debug_n<10){
      ptBuf[Debug_n] = pt;
      dataBuf[Debug_n++] = pt->value;
    }
    Heap_Release((unsigned short*)pt);
    success = 1;
  }
  return(success);
}
```

Collect execution times for the function 1) as is, 2) with debugging print statements, and 3) with debugging dump statements. For the dump case, you are measuring the time to store into the array and not the time to print the array on the screen. The slow-down introduced by the debugging procedures defines its level of intrusiveness.

B. Instrumentation Output Port.

Another method to measure real time execution involves an output port and an oscilloscope. Connect PORTT bit 0 to an oscilloscope. You will create a debugging instrument that sets PORTT bit 0 to one just before calling **`LLFifo_Get`**. Then, you will set the output back to zero right after. You will set the port's direction register to 1, making it an output. If you were to put the instruments inside **`LLFifo_Get`**, then you would be measuring the speed of the calculations and neglecting the time it takes to pass parameters and perform the subroutine call. On the other hand, in a complex system this method allows you to visualize each call to **`LLFifo_Get`**, regardless from where it was called. For this lab, stabilize the input, and repeat the operation in a loop, so that the scope can be triggered. The time measured in this way includes the overhead of passing parameters. E.g.,

```
// 9S12C32
void main(void){
unsigned short data;
  DDRT |= 0x01;
  LLFifo_Init();  // initialize
  for(;;){
    LLFifo_Put(1);
    PTT |= 0x01;
    LLFifo_Get(&data);
    PTT &= ~0x01;
  }
}
```

Compare the results of this measurement to the **`TCNT`** method. Discuss the advantages and disadvantages between the **`TCNT`** and scope techniques. Determine the measurement error of the scope technique using the following code. The time the signal is high represents the error introduced by the measurement itself.

```
// 9S12C32
```

```
void main(void){
  DDRT |= 0x01;
  for(;;){
    PTT |= 0x01;
    PTT &= ~0x01;
  }
}
```

C. Profiling using a software dump to study execution pattern

The objective of this part is to develop software and hardware techniques to visualize program execution in real time. This system has two threads: the foreground thread (**main** program) and a background thread (RTI interrupt handler). The background thread, invoked by the RTI clock hardware, executes periodically. The foreground thread runs in the remaining intervals. The background thread calls **LLFifo_Put** and the foreground thread calls **LLFifo_Get**. In this way data is passed between the threads. In this lab, we will study the execution pattern of this two-threaded system that uses the linked-list FIFO. E.g.,

```
unsigned short timeBuf[100];
unsigned short placeBuf[100];
unsigned short Debug_n=0;
void Debug_Profile(unsigned short thePlace){
  if(Debug_n>99) return;
  timeBuf[Debug_n] = TCNT;        // record current time
  placeBuf[Debug_n] = thePlace;   // record place from which it is
called
  Debug_n++;
}
```

Calls to **Debug_Profile** have been placed at strategic places in the software system. The **thePlace** parameter specifies where the software is executing. The **timeBuf** records when the debugging profile was called. Notice that the debugging instrument saves in an array (like a dump). The debugger for 9C12C32 allow you to observe memory while the program is executing. In particular use the debugger to collect the profile data. Except for a modest increase in the execution time, your instrument should not modify the operation. Printout the C source code of the instrumented system. Run the instrumented system and make a hardcopy printout the results of the debugging instruments. On the source code listings draw "tail to head" arrows (e.g.,→) illustrating the execution pattern as one piece of data is passed through the system. Draw a data-flow graph of this system. If the data you collect is confusing, repeat the experiment with more (or less) instruments.

D. Thread Profile using hardware.

When the execution pattern is very complex, you could use a hardware technique to visualize which program is currently running. In this section, choose the RTI interrupt service routine and at least three other regular functions to profile. You will associate one output pin with each function you are profiling. You will connect all the output pins to the logic analyzer to visualize in real time the function that is currently running. For each regular routine, set its output bit high when you start execution and clear it low when the function completes. E.g., assume PT3 is associated with **Heap_Allocate**

```
// 9S12C32
```

```
unsigned short *Heap_Allocate(void){
unsigned short *pt;
  PTT |= 0x08;
  pt = FreePt;
  if(pt!=null){
    FreePt = (unsigned short*)*pt;
  }
  PTT &= ~0x08;
  return(pt);
}
```

For RTI interrupt service routine, save the previous value, set its output bit high when you start execution and restore the previous value when the function completes. E.g., assume PT0 is associated with **`RTIHan`**

```
// 9S12C32
interrupt 7 void RTIHan(void){
unsigned short i; char previous;
  previous = PTT;
  PTT = 0x01;
  RTIFLG = 0x80;
  for(i=0; i<=BackData; i++){
    if(LLFifo_Put(i)==0){
      NumLost++;
    }
  }
  BackData++;
  if(BackData==20){
    BackData = 0;   // 0 to 19
  }
  PTT = previous;
}
```

Compile, download, and run this system observing on the oscilloscope the behavior of the two output pins. Explain what is happening. If the data you collect is confusing, change which functions you are profiling and repeat the profiling.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)

B) Hardware Design (none for this lab)

C) Software Design (no software printout in the report)

none

D) Measurement Data

Prep part 3) Show the cycle counting of execution speed

Part A) Show the three results of the execution times

Part B) Show the execution time measured with the oscilloscope

Part C) The software profile data, draw arrows on the listings, and data-flow graph

Part D) The hardware profile data

Part E) Analysis and Discussion (1 page maximum)

Checkout

You should be able to demonstrate:

Part B. Instrumentation output port.

Part C. Profiling using a software dump.

Part D. Profiling using an output port.

Your software files will be copied onto the TA's zip drive during checkout.

Part C) The programs that implement software profiling

Part D) The programs that implement hardware profiling

Lab 3d Position Data Acquisition System and Interrupting SCI

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals • Study ADC conversion,
• Develop an interrupt-driven SCI device driver,
• Develop a real-time position measurement system using a slide potentiometer.

Review • Operation of the 6812 ADC system in the Technical Data on 9S12C32 manual,
• Valvano Section 11.10 on the 6812 ADC, Section 12.1 on DAS parameters,
Subsection 12.2.4 on position sensors.

Starter files • **`RTI SCIA`** and **`ADC`** projects

Background

You will design a **position meter** with a range of about 3 cm. A linear slide potentiometer converts position into resistance ($0<R<R_{max}$). You will use an electrical circuit to convert resistance into voltage (V). Since the potentiometer has three leads, one possible solution is shown in Figure 3.1. The 6812 ADC will convert voltage into a 10-bit digital number. Your software will calculate position from the ADC sample as a decimal fixed-point number. The position measurements will be displayed on the PC via the SCI-COM- HyperTerminal interface. You will use interrupting I/O for the SCI interface.

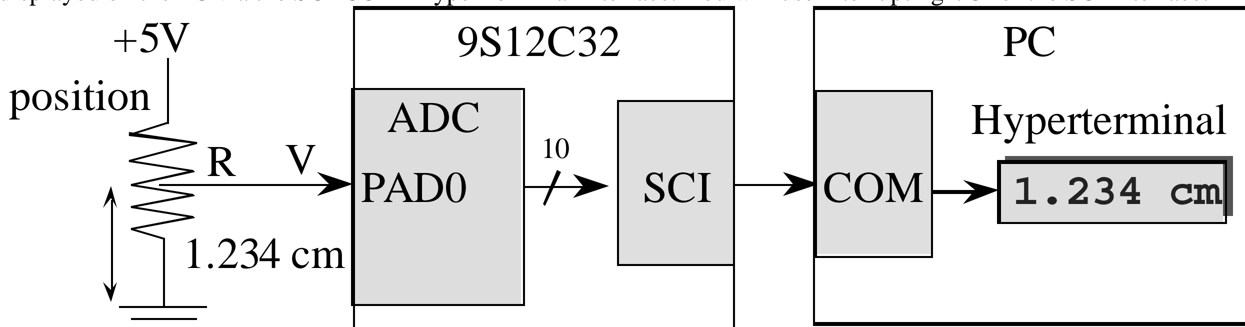


Figure 3.1. Block diagram of the position meter system.

You should make the position resolution and accuracy as good as possible. The **position resolution** is the smallest change in position that your system can reliably detect. In other words, if the resolution were 0.01 cm and the position were to change from 1.00 to 1.01 cm, then your device would be able to recognize the change. Resolution will depend on the amount of electrical noise, the number of ADC bits, and the resolution of the output display software. **Accuracy** is defined as the absolute difference between the true position and the value measured by your device. Accuracy is dependent on the same parameters as resolution, but in addition it is also dependent on the stability of the transducer and the quality of the calibration procedure.

Since the transducer is not linear, you could use a piece-wise linear interpolation to convert the ADC sample to position (Δ of 0.001 cm.) The 6812 assembly language **`etbl`** instruction is an efficient mechanism to perform the interpolation. The **`etbl.RTF`** assembly program included with TExaS is an example of a piece-wise linear interpolation using the **`etbl`** instruction. There are two small tables `Xtable` and `Ytable`. The `Xtable` contains the ADC results and the `Ytable` contains the corresponding positions. The ADC sample is passed into the lookup function. This function first searches the `Xtable` for two adjacent of points that surround the current ADC sample. Next, the function uses the **`etbl`** instruction to perform a linear interpolation to find the position that corresponds to the ADC sample. You are free to implement the conversion in any acceptable manner, with the exception that you are not allowed to use a simple linear equation. You may use C, assembly, or a mixture of C/assembly.

The 10-bit ADC converters on the 9S12C32 are successive approximation devices with a short conversion time. You need to enable the ADC in **`ATDCTL2`**. You can define the number of ADC conversions (1 to 8) in a sequence using **`ATDCTL3`**. You specify the ADC clock in **`ATDCTL4`**, which will determine the time to perform an ADC conversion. Writing to the ADC Control register (**`ADCTL5`**) begins a conversion. The ADC chip clocks itself. After the first sample is complete, **`CCF0`** is set and the result can be read out of the first result register, **`ATDDR0`**. After the entire sequence has been converted, the **`SCF`** bit is set.

In this lab, you will be measuring the position of the armature on the slide potentiometer. This signal has very few frequency components (0 to 1 Hz.) According to the Nyquist Theorem, we need a sampling rate greater than 2 Hz. You may choose any sampling rate in the range of 2 to 50 Hz. You will sample the ADC at that rate and calculate position using decimal fixed-point with Δ of 0.001 cm. You should display the results in the HyperTerminal window on the PC, including

units. No floating point is allowed. RTI interrupts will be used to sample the ADC in a background thread. This high priority interrupt will establish the sampling rate.

Nyquist Theorem: If f_{max} is the largest frequency component of the analog signal, then you must sample more than twice f_{max} in order to faithfully represent the signal in the digital samples. For example, if the analog signal is A + B sin(2πft + ϕ) and the sampling rate is greater than 2f, you will be able to determine A, B, f, and ϕ from the digital samples.

Valvano Postulate: If f_{max} is the largest frequency component of the analog signal, then you must sample more than ten times f_{max} in order for the reconstructed digital samples to look like the original signal when plotted on a voltage versus time graph.

Preparation (do this before your lab period)

1. Review the technical information on the ADC system of 6812. What initiates the conversion process? What are two ways of knowing that the conversion process has been completed?

2. Firmly attach the frame (the fixed part) of the potentiometer to a solid object. Place a metric ruler on the frame but near the armature (the movable part) of the sensor. You could Xerox a metric ruler and tape the paper onto the frame. Attach or draw a hair-line to the armature, which will define the position measurement. Solder three solid wires to the slide potentiometer. If you do not know how to solder, ask your TA for a lesson.

3. Design and build the electrical circuit that interfaces the transducer to the ADC. If you use external devices be careful not to produce voltages above +5 or below 0 volts. Typically, all that is required here is soldering three wires to the potentiometer, and connecting the three wires to +5V, ADC input, and ground. Make sure the +5V and ground connections go to the two potentiometer leads that are the fixed resistance. The center variable lead is connected to the ADC input, as shown in Figure 3.1.

4. Load and run the **`ADC`** project. You can visualize the **`Data`** result using the debugger, without having to connect the LCD display that this project normally uses. You will use it to perform a calibration to determine the ADC sample for 6 positions. You will determine the true position using the metric ruler and hairline you built in step 2. You can measure the analog voltage using a DVM. The **`ADC`** project will set the **`Data`** variable to the results of the analog to digital conversion. Make a three-column (position, voltage, and ADC sample) six-row table for this data. As an option, you could also disconnect the potentiometer from the circuit and measure the resistance using an ohmmeter for each position as well. Plot the position as a function of ADC sample. Does the response have a 1-1 correspondence? Is the response linear? These six measurements could become the data for the `Xtable Ytable` method described above.

5. Load and run the **`SCIA`** project. Learn how to use HyperTerminal together with the 9S12C32 board. This system uses interrupt synchronization for both input and output. Rename the project **`SCIB`** and modify the driver software (**`SCIB.c`** and **`SCIB.h`**) and the main program so they implement just SCI output. I.e., remove all the SCI input routines. You should also remove the **`RxFifo`** module.

6. Combine the **`RTI ADC`** and **`SCIB`** projects to create the real-time DAS software required for this lab. Within the RTI interrupt handler, you will start the ADC, wait for the ADC sample, convert the sample to fixed-point position, convert the fixed-point to an ASCII string (Lab 1), and output the string using **`SCI_OutString`**. Include units in the output. You will have to modify **`SCI_OutChar`** so it does not crash when the **`TxFifo`** is full. In addition, you will also have to slow down the ADC sampling rate and/or increase the SCI baud rate so the **`TxFifo`** never becomes full. In particular, review the procedure sections. A "syntax-error-free" hardcopy listing for the software is required as preparation. The TA will check off your listing at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines.

Procedure (do this during your lab period)

1. *System checkout*: Run your hardware/software system to verify operation.

2. *Real-time*: One of the critical factors when designing a real-time system is the maximum time the software runs with interrupts disabled. Use one of the techniques you learned in Lab 2 to measure the worst case (longest) execution time of the RTI handler.

3. *Accuracy determination*: Collect 10 data points equally spaced throughout the range. Again the true position will be determined using the metric ruler and hairline. The measured position is defined as the value on the PC HyperTerminal window. Calculate average accuracy, maximum error, standard error, average accuracy of reading, average accuracy of full scale, maximum accuracy of reading, and maximum accuracy of full scale.

4. *Reproducibility determination*: Place the slide potentiometer at the center position and use your system to measure position on three separate days. Calculate the drift as the maximum difference between these three measurements

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)

B) Hardware Design

Detailed circuit diagram all external circuits interfaced to the microcomputer

C) Software Design (no software printout in the report)

Draw a data flow graph of the software system

Draw a call graph of the software system

D) Measurement Data

Give the software execution times of the RTI interrupt handler (procedure 2)

Give the accuracy measurements including details of how the data was obtained (procedure 3)

Give the reproducibility measurements including details of how the data was obtained (procedure 4)

E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)

You should be able to demonstrate the proper operation of position measurement instrument. Connect a DVM to the analog signal and be prepared to discuss with the TA the various aspects of the mechanical, electrical and software design.

Your software files will be copied onto the TA's zip drive during checkout.

Lab 4f LCD Alarm Clock

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Optimize an existing hardware/software interface between a LCD display and a microcomputer,
- Design a hardware/software interface between three switches and a microcomputer,
- Design a hardware/software driver for generating single tones on a speaker,
- Implement a digital alarm clock.

Review
- Valvano Chapter 3 on Basic Handshake Mechanisms,
- Valvano Section 8.3 on LCD fundamentals,
- Valvano Section 2.7 on device drivers.
- `hd44780.pdf` and `LCD.pdf` data sheets

Starter files
- `RTI` and `LCD` projects

Background

Microprocessor controlled LCD displays are widely used having replaced most of their LED counterparts, because of their low power and flexible display graphics. This experiment will illustrate how a handshaked parallel port of the microcomputer will be used to output to the LCD display. The hardware for the display uses an industry standard HD44780 controller. The low-level software initializes and outputs to the HD44780 controller. Because the 9S12C32 has so few I/O pins, you will implement the 4-bit data interface.

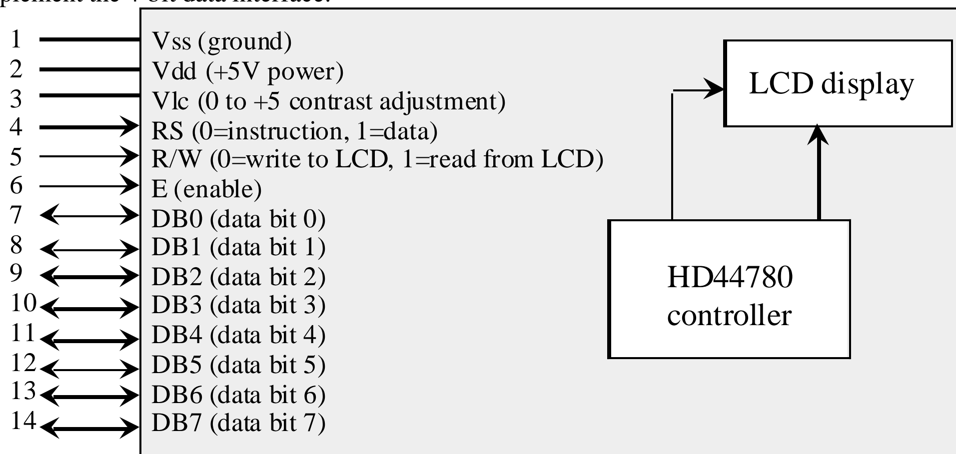


Figure 4.1. 1 by 16 LCD display.

There are four types of access cycles to the HD44780 depending on RS and R/W

RS	R/W	Cycle
0	0	Write to Instruction Register
0	1	Read Busy Flag (bit 7)
1	0	Write data from µP to the HD44780
1	1	Read data from HD44780 to the µP

Two types of synchronization can be used with the LCD, blind cycle and gadfly. Most operations require 40 µs to complete while some require 1.64 ms. The example implementation shown in the **`LCD`** project uses **`TCNT`** to create the blind cycle wait. YOU ARE REQUIRED TO DEVELOP SOFTWARE THAT READS THE BUSY FLAG. A gadfly interface provides feedback to detect a faulty interface, but has the problem of creating a software crash if the LCD never finishes. You will implement an excellent solution utilizing both gadfly and blind cycle, so that the software can return with an error code if a display operation does not finish on time (due to a broken wire or damaged display.) Your low-level functions must perform both gadfly (wait for the busy flag to be clear) and blind cycle. For each operation you will return with the error code equal to success if the busy becomes clear, but return with a failure if the busy does not become clear after waiting the appropriate amount of time.

The paragraph discusses issues involved in developing software that will be sold. Because the software for embedded systems is much smaller than software for a general-purpose computer, it is customary to provide sell copyrighted source files (e.g., **`LCD.H`** and **`LCD.C`**) that the user can compile with their application. In a large software system, one typically sells the header file together with precompiled object code. In our embedded system, the compiler will

perform the linking. You are encouraged to modify/extend this example, and define/develop/test your own format. Normally, we group the device driver software into four categories. Interrupt service routines would be classified as protected support software, not directly callable by the user.

1. Data structures: global, protected (accessed only by the device driver, not the user.)
`OpenFlag` boolean that is true if the display port is open
 initially false, set to true by `LCD_Open`
 static storage (or dynamically created at bootstrap time, i.e., when loaded into memory)

2. Initialization routines (used/called by the user)
`LCD_Open` Initialization of display port
 Sets `OpenFlag` to true
 Initialize hardware, other data structures
 Returns an error code if unsuccessful
  hardware non-existent, already open, out of memory, hardware failure, illegal parameter
 Input Parameters(`display`, `cursor`, `move`, `size`)
 Output Parameter(none)
 Typical calling sequence

```
        if(!LCD_Open(LCDINC+LCDNOSHIFT,LCDNOCURSOR+LCDNOBLINK,
                     LCDNOSCROLL+LCDLEFT,LCD2LINE+LCD7DOT)) error();
```

3. Regular I/O calls (called by user to perform I/O)
`LCD_Clear` clear the LCD display
 Returns an error code if unsuccessful. E.g., device not open, hardware failure (happens when a wire is loose)
 Input Parameter(none)
 Output Parameter(error code)
 Typical calling sequence

```
        if(LCD_Clear()) error();
```

`LCD_OutChar` Output an ASCII character to the LCD port
 Returns an error code if unsuccessful. E.g., device not open, hardware failure (happens when a wire is loose)
 Input Parameter(ASCII character)
 Output Parameter(error code)
 Typical calling sequence

```
        if(LCD_OutChar(letter)) error();
```

`LCD_OutString` Output a NULL-terminated ASCII string to the LCD port
 Returns an error code if unsuccessful. E.g., device not open, hardware failure (happens when a wire is loose)
 Input Parameter(pointer to ASCII string)
 Output Parameter(error code)
 Typical calling sequence

```
        if(LCD_OutString("Hello world")) error();
```

4. Support software (protected/ `static`, not directly accessible by the user).
`outCsr` sends one command code to the LCD control/status
 Input Parameter(command is 8-bit function to execute)
 Output Parameter(none}
`wait` fixed time delay
 Input Parameter(time in microseconds to wait)
 Output Parameter(none}

You will write the first main program in order to test and evaluate the device driver software and hardware interface. For a device this simple you should be able to test all modes and all characters. Purposefully cause errors and see if the software gives the appropriate response. For example, the LCD project contains a main program that is used to test the features of the `LCD.H`, `LCD.C` device driver.

The second main program will implement a simple digital alarm clock. You can connect individual push-button switches (see Figure 4.2) to input pins of the 6812, which the user can use to set the current time and the alarm time. The LCD display will be used to display the current time. You are free to implement whatever features you wish, but there must be a way to set the time. The system maintains three global variables `hour`, `minute`, `second`. No SCI input is allowed,

and the time parameters must be maintained using the RTI interrupts. In particular, the RTI interrupt service routine should increment **`second`** once a second, increment **`minute`** once a minute, and increment **`hour`** once an hour. The foreground (main) will output to the LCD display, and interact will the operator via the switch inputs. If the correct sequence of switches is pushed, the main program can initialize the values of **`hour`**, **`minute`**, **`second`**.

Figure 4.2. S.P.S.T. momentary, normally open. 0.48" square x 0.18" body. Plunger stands 0.3" above body. Four PC leads on 0.2" x 0.5" centers. CAT# MPB-127, http://www.allelectronics.com.

You can interface a speaker using a NPN transistor like PN2222 to an output port. The resistor controls the loudness of the sound. Please make it quiet, try 1 kΩ. The maximum I_{CE} of the transistor must be larger than 5V/32Ω. The 1N914 diode is used as a snubber to remove back EMF when the transistor switches off. If you toggle the output pin in the background ISR, then sound will be generated. See Figure 4.3.

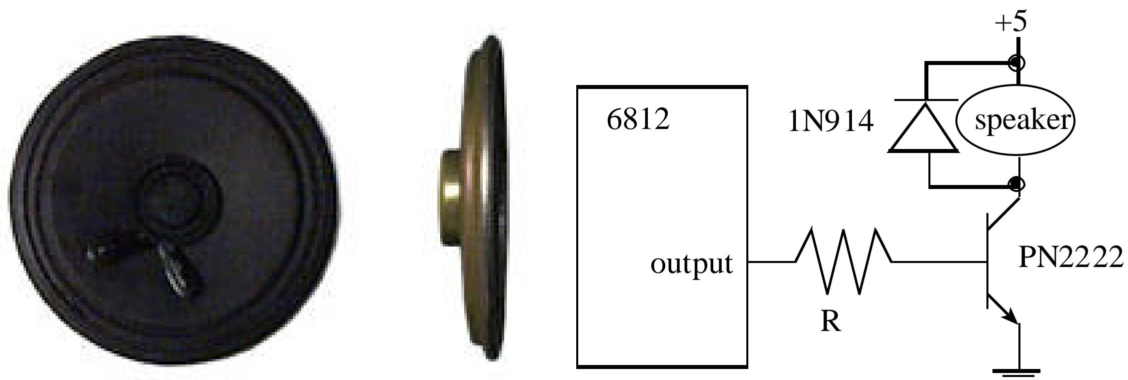


Figure 4.3. 2.25" 32 OHM SPEAKER. 0.38" thick. CAT# SK-232, http://www.allelectronics.com.

Preparation

Show the required hardware connections. Label all hardware chips, pin numbers, and resistor values. Modify the existing low-level LCD device driver to implement the combined blind-cycle/gadfly synchronization. You must have a separate **`LCD.H`** and **`LCD.C`** files to simplify the reuse of these routines. Write one main program that tests all features of the driver.

Write a second main program that implements the digital alarm clock. The interrupt service routine used to maintain time must run as fast as possible. This means you must perform all LCD I/O from the main program. You must be careful not to let the LCD show an intermediate time of 1:00:00 as the time rolls over from 1:59:59 to 2:00:00. You must also be careful not to disable interrupts too long (more than one RTI interrupt period), because a time error will result if any RTI interrupts are skipped. You will have to do some 32-bit math to maintain the exact time. For example, if the RTI interrupts at 30.517Hz, then interrupts are requested at exactly 32768 µs. One method is to add 32768 to a 32-bit **`counter`**. When the **`counter`** exceeds 1 million, increment **`second`** and subtract 1 million from the **`counter`**.

Procedure

You should look at the +5 V voltage versus time signal on a scope when power is first turned on to determine if the LCD "power on reset" circuit will be properly activated. The LCD data sheet specifies it needs from 0.1 ms to 10 ms rise time from 0.2 V to 4.5 V to generate the power on reset. Connect the LCD to your microcomputer. Use the scope to verify the sharpness of the digital inputs/outputs. If needed, adjust the contrast potentiometer for the best looking display. Test the device driver software and two main programs in small pieces.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
LCD interface, showing all external components
speaker interface, showing all external components
C) Software Design (no software printout in the report)
A call-graph illustrating the modularity of the hardware/software components of the alarm clock

D) Measurement Data

Plot the LCD supply voltage versus time as the system is powered up

Plot the speaker voltage versus time during an alarm sound

E) Analysis and Discussion (1 page maximum)

Checkout

Using the first main program, you should be able to demonstrate all the "cool" features of your LCD display system. Next, download the digital alarm program. Demonstrate that your digital alarm clock is stand-alone by turning the power off then on. The digital alarm clock should run (the time will naturally have to be reprogrammed) without downloading the software each time.

Your software files will be copied onto the TA's zip drive during checkout.

Hints

1) Make sure the 14 wires are securely attached to your board.

2) One way to test for the first call to **`LCD_Open`** is to test the direction register. After reset, the direction registers are usually zero, after a call to **`LCD_Open`**, some direction register bits will be one.

3) Observe the starter files in the **`LCD`** project. These C language routines only output to **`PTM`** and **`PAD0`**. Notice that they do not perform any input (either status or data), therefore they leave **`DDRM=0x3F`**, **`DDRAD|=0x01`**. Because you have to include inputs, then you must toggle **`DDRM`**, so that **`PTM`** bits 3-0 are output for writes and an input for reads.

4) Although many LCD displays use the same HD44780 controller, the displays come in various sizes ranging from 1 row by 16 columns up to 4 rows by 40 columns.

Lab 5e Stepper Motor Finite State Machine

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- To interface a stepper motor,
- To implement background processing with periodic interrupts,
- To develop a dynamic linked command structure.

Review
- Valvano Section 1.6 about open collector logic,
- Valvano Section 1.7 about initializing and accessing I/O ports,
- Valvano Section 2.4 about abstraction, linked lists and FSM's,
- Valvano Chapter 8, about stepper motor interfacing,
- Valvano Section 13.2.3, about an open loop stepper motor control system.

Starter files
- `MOORE OC` and `OC3` projects

Background

Stepper motors are popular with digital control systems because they can be used in an open loop manner with predictable responses. Such applications include positioning heads in disk drives, adjusting fuel mixtures in automobiles, and controlling robot arms in robotics. In this lab, you will control a stepper motor using a finite state machine. The finite state machine must be implemented as a linked data structure. Two switches will allow the operator to control the motor. A background periodic interrupt (either `OC` or `RTI`) thread will perform inputs from the switches and outputs to the stepper motor coils. The worm gear on the motor, as shown in Figure 5.1, produces a linear motion as the stepper motor turns. It takes about 200 full-steps to move the worm gear from one end to the other. If you step the motor too far in either direction, the worm gear will disengage and you will have to manually re-engage the worm gear. Once your software is debugged, the system should move up and down along these 200 steps without disengaging regardless of how the TA pushes the input buttons. Two switches determine the motor operation. If both buttons are released, then the motor should stop. If switch 1 is pressed the motor should spin slowly in one direction. If switch 2 is pressed the motor should spin slowly in the other direction. The motor should stop when it reaches the end of the worm gear even if the TA continues to press the switch. If both switches are pressed, the motor should continuously spin as fast as possible back and forth across the full range of the worm gear. If the system is performing one operation, and another command is issued, then the first operation is terminated and the second command is performed.

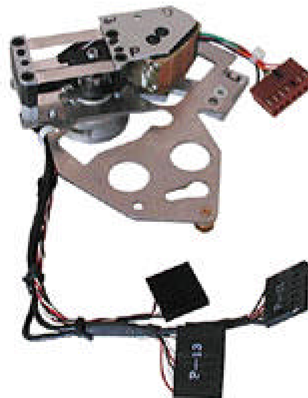


Figure 5.1. Stepper motor with worm gear, CAT# EX-82, http://www.allelectronics.com.

Preparation (do this before your lab period)

With an ohmmeter, measure the resistance of one coil. Apply the power across the coil and simultaneously measure using both a voltmeter and a current-meter the voltage and current required to activate one coil of your stepper motor. Do this before your lab period, without any connections to the 6812 board. Design the hardware interface between the 6812, and prepare a circuit diagram labeling all resistors and diodes. The pin out for the stepper is shown in Figure 5.2. Include pin numbers and resistor/capacitor types and tolerances. Be sure the interface circuit (e.g., 7406 or 2N2222) you select can sink enough current to activate the coil. In particular, verify the driver can supply (I_{OL}) the required current for the stepper motor. Make sure you have all the parts you need before lab starts. This interface will require four 1N914 snubber diodes. You can use bigger diodes like the 1N4001-1N4005 series, but if do not properly connect the diodes, the back EMF will destroy your board.

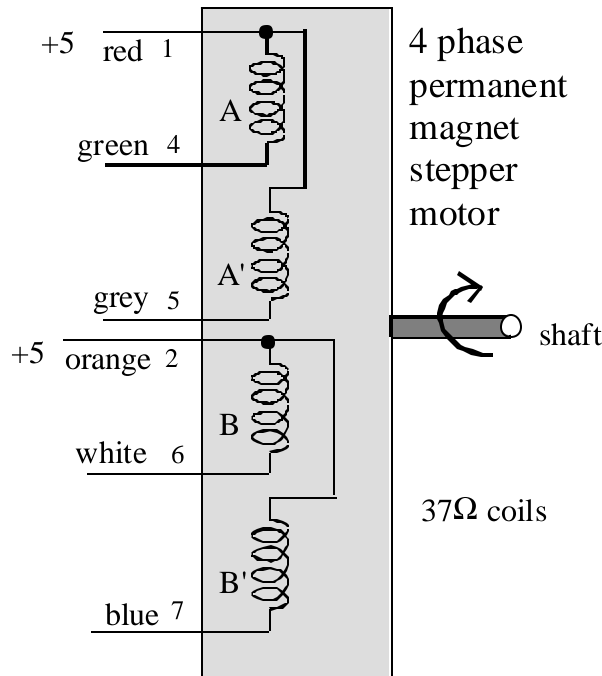


Figure 5.2. The EX-82 stepper motor has unipolar configuration with four+5V 37-W coil.

A "syntax-error-free" hardcopy listing for the software is required as preparation. This will be checked off by the TA at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines. The periodic interrupt-driven background thread will execute the finite state machine, while the foreground thread initializes the system then does nothing.

Procedure (do this during your lab period)

Make sure your TA checks your hardware diagram before connecting it to the 6812. We do not have extra boards, so if you fry your board, you may not be able to finish. First build the digital interface on a separate protoboard from the 6812, as shown in Figure 5.3. Use switches as inputs for the stepper interface. You should be able to generate the 5,6,10,9 sequence with your fingers. Using a scope, look at the voltages across the coils to verify the diodes are properly eliminating the back EMF. The very fast turn-off times of the digital transistors can easily produce 100 to 200 volts of back EMF, so please test the hardware before connecting it to the 6812.

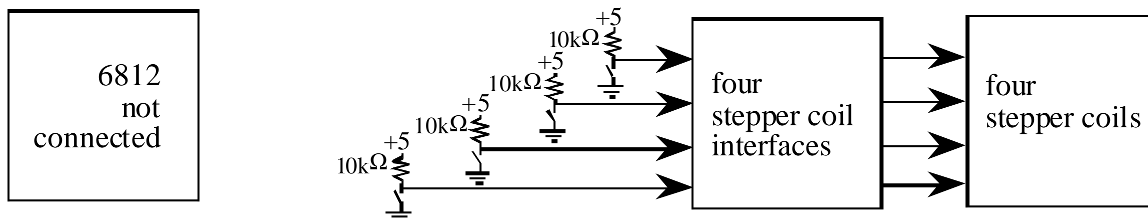


Figure 5.3. Hardware test procedure.

The switches will eventually be replaced by 6812 outputs, as shown in Figure 5.4. Once you are sure the hardware is operating properly, run a simple constant velocity software function to verify the hardware/software interface.

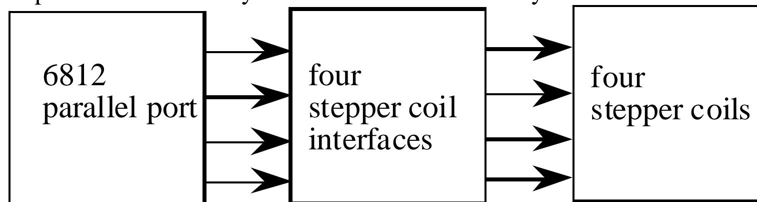


Figure 5.4. Hardware block diagram.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
    stepper motor interface, showing all external components
    switch interfaces, showing all external components
C) Software Design (no software printout in the report)
    Draw figures illustrating the major data structures used, e.g., the finite state machine
    A call-graph illustrating the modularity of the software components
D) Measurement Data
    Prep) Give the voltage, current, and resistance measurements
    Determine the range of speed possible for the motor
E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)

You should be able to demonstrate correct operation of the stepper motor system. Be prepared to describe how your stepper interface works. Explain how your system handles the switch bounce. Demonstrate debugging features that allow you to visualize the software behavior.
Your software files will be copied onto the TA's zip drive during checkout.

Hints

1. Be sure the interface driver (e.g., the 7406 or 2N2222) has an I_{OL} large enough to deliver the needed coil current. You may use the 9S12C32 +5V regulated supply to run this stepper motor because it requires less than 500 mA total current. Although many steppers will operate at voltages less than the rated voltage, the torque is much better when using the proper voltage. Remember to actually measure the coil current rather than dividing voltage by resistance.
2. Be sure to put an appropriate delay between each step to prevent the motor from burning up.
3. To prevent the power supply from overheating, make sure the supply can deliver the required current.
4. Debug the lab in small steps.
5. The **TExaS** simulator does include a stepper motor, so you can develop and test the software in parallel with the hardware development. In particular, the `OC` project can be run on the **TExaS** simulator.

Lab 6f Music generation using a Digital to Analog Converter

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- DAC conversion,
- SPI interface,
- Design a data structure to represent music,
- Develop a system to play sounds.

Review
- Data sheets on 74HC595,
- Valvano Chapter 7 on SPI interfacing,
- Valvano Chapter 8 on NPN transistors,
- Valvano Chapter 11 on DAC converters.

Starter files
- `OC3` project

Background

Most digital music devices rely on high-speed DAC converters to create the analog waveforms required to produce high-quality sound. In this lab you will create a very simple sound generation system that illustrates this application of the DAC. Your goal is to play your favorite song. For the first step, you will interface a 74HC595 serial in/parallel out shift register to the SPI port. Please refer to the 74HC595 data sheets for the synchronous serial protocol. The second step you need to perform is to create a DAC from the 8-bit digital output of the 74HC595. You are free to design your DAC with a precision anywhere from 5 to 8 bits. You will convert the binary bits (digital) to an analog output current using a simple resistor network. The third step is to convert the DAC analog output to speaker current using a current-amplifying audio amplifier. It doesn't matter what range the DAC is, as long as there is an approximately linear relationship between the digital data and the speaker current. To do this you will have to run the amplifier in its linear range. The performance score of this lab is not based on loudness, but sound quality. The quality of the music will depend on both hardware and software factors. The precision of the DAC, the linearity of the audio amp, the frequency response of the audio amp and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the complexity of the stored music data. It is important to add a 0.1µF bypass capacitor on the power connection of the 74HC595 to prevent output glitches during serial input transmissions. Consider using the LM386 when designing the audio amp.

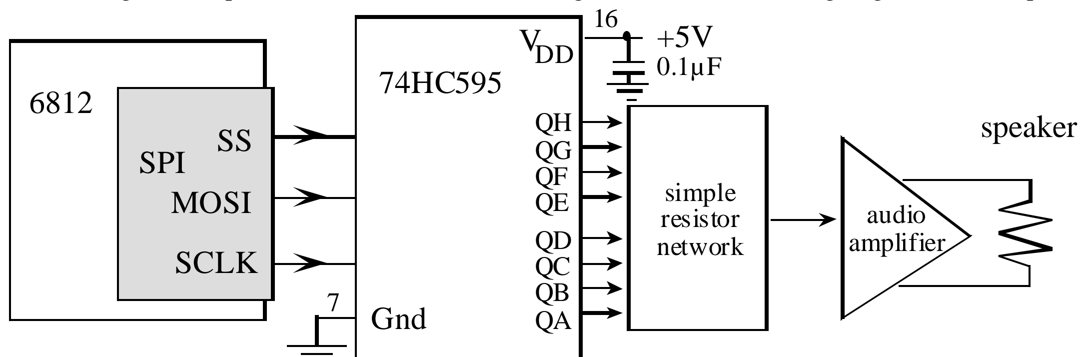


Figure 6.1. DAC allows the software to create music.

The fourth step is to design a low-level device driver for the DAC. A single 8-bit SPI frame is all that is required to set the DAC output. The fifth step is to design a data structure to store the sound waveform. You are free to design your own format, as long as it uses a formal data structure (i.e., **`struct`**). Compressed data occupies less storage, but requires runtime calculation. On the other hand, a complete list of points will be simpler to process, but requires more storage than is available on the 6812. The sixth step is to organize the music software into a device driver. Although you will be playing only one song, the song data itself will be stored in the main program, and the device driver will perform all the I/O and interrupts to make it happen. You will need public functions **`Rewind`**, **`Play`** and **`Stop`**, which perform operations like a cassette tape player. The **`Play`** function has an input parameter that defines the song to play. A background thread implemented with output compare will fetch data out of your music structure and send them to the DAC. The last step is to write a main program that inputs from three binary switches and performs the three public functions.

If you output a sequence of numbers to the DAC that form a sine wave, then you will hear a continuous tone on the speaker, as shown in Figure 6.2. The **loudness** of the tone is determined by the amplitude of the wave. The **pitch** is defined as the frequency of the wave. Table 6.1 contains frequency values for the notes in one octave.

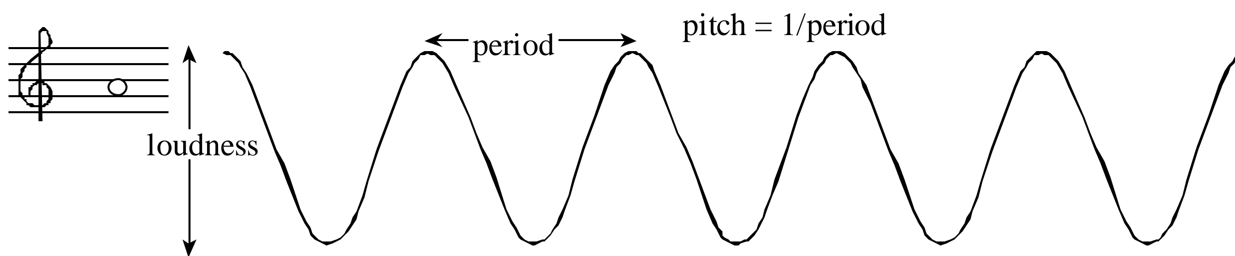


Figure 6.2. A sine wave generates a pure tone.

Note	frequency
C	523 Hz
B	494 Hz
B^b	466 Hz
A	*440 Hz*
A^b	415 Hz
G	392 Hz
G^b	370 Hz
F	349 Hz
E	330 Hz
E^b	311 Hz
D	294 Hz
D^b	277 Hz
C	262 Hz

Table 6.1. Fundamental frequencies of standard musical notes. The frequency for 'A' is exact.

The frequency of each note can be calculated by multiplying the previous frequency by $\sqrt[12]{2}$. You can use this method to determine the frequencies of additional notes above and below the ones in Table 6.1. There are twelve notes in an octave, therefore moving up one octave doubles the frequency. Figure 6.3 illustrates the concept of **instrument**. You can define the type of sound by the shape of the voltage versus time waveform. Brass instruments have a very large first harmonic frequency.

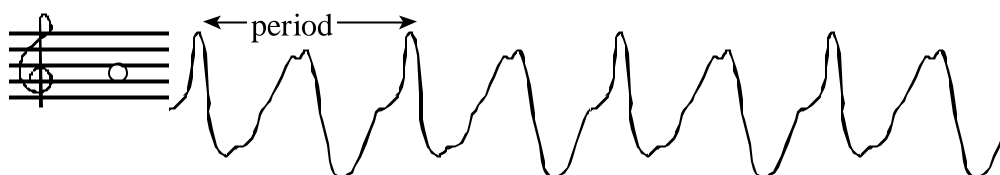


Figure 6.3. A waveform shape that generates a trumpet sound.

The **tempo** of the music defines the speed of the song. In 2/4 3/4 or 4/4 mu sic, a **beat** is defined as a quarter note. A moderate tempo is 120 beats/min, which means a quarter note has a duration of ½ second. A sequence of notes can be separated by pauses (silences) so that each note is heard separately. The **envelope** of the note defines the amplitude versus time. A very simple envelope is illustrated in Figure 6.4. The 9S12C32 has plenty of processing power to create these types of waves.

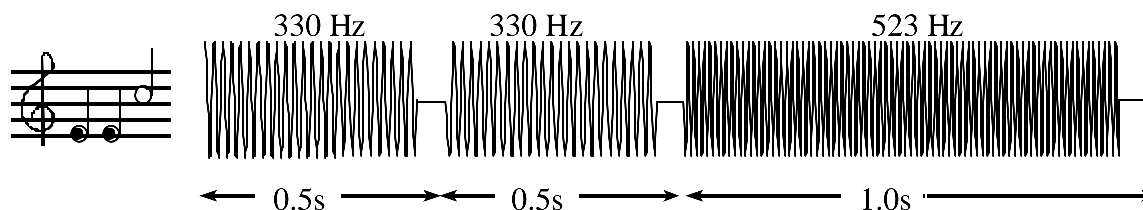


Figure 6.4. You can control the amplitude, frequency and duration of each note (not drawn to scale).

The smooth-shaped envelope, as illustrated in Figure 6.5, causes a less staccato and more melodic sound. This type of sound generation may be difficult to produce in real-time on the 9S12C32.

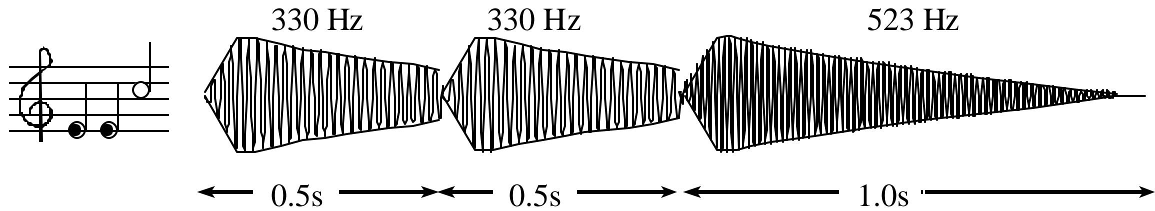


Figure 6.5. The amplitude of a plucked string drops exponentially in time.

A chord is created by playing multiple notes simultaneously. When two piano keys are struck simultaneously both notes are created, and the sounds are mixed arithmetically. You can create the same effect by adding two waves together in software, before sending the wave to the DAC. Figure 6.6 plots the mathematical addition of a 262 Hz (low C) and a 392 Hz sine wave (G), creating a simple chord.

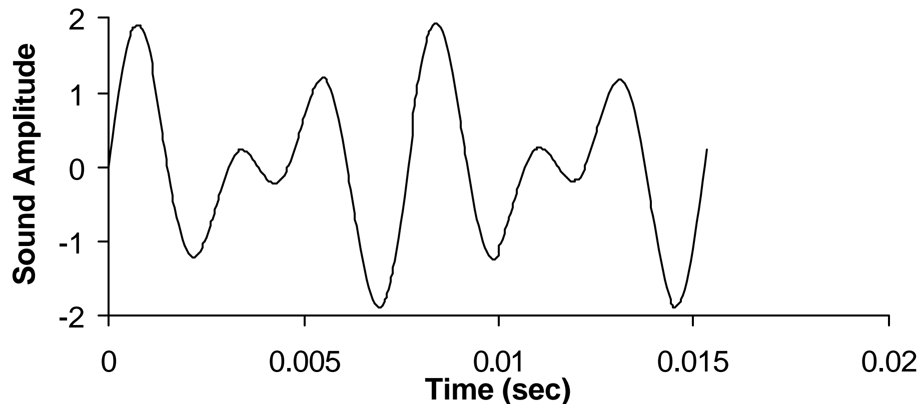


Figure 6.6. A simple chord mixing the notes C and G.

Preparation (do this before your lab period)

1. Draw the circuit required to interface the 74HC595 to the 6812 SPI port. Include signal names and pin numbers. The bypass capacitor on the +5 V supply of the 74HC595 can be any value from 0.1 µF to 0.22 µF.

2. Design the DAC converter using a simple resistor-divider technique. Use resistors in a 1/2/4/8/16/32 resistance ratio. Select values in the 5 kΩ to 200 kΩ range. For example, you could use 5 kΩ, 10 kΩ, 20 kΩ, and 40 kΩ. Notice that you could create double/half resistance values by placing identical resistors in series/parallel. Design the audio amplifier which runs on the +5V power from the board. Using Ohm's law and the properties of the audio amplifier, make a table of the speaker voltage and current as a function of digital value.

3. Write a low-level device driver for the SPI interface. Include two functions that implement the SPI/DAC interface. The function **`DAC_Init()`** initializes the SPI protocol, and the function **`DAC_Out()`** sends a new data value to the DAC. Create separate **`DAC.h`** and **`DAC.c`** files.

4. Write a couple of simple main programs that test the SPI/DAC interface. You will use this software to test the SPI interface, and the DAC hardware. This main program can be used for static testing.

```
void main(void){unsigned char number;
  SCI_Init(19200);   // initialize SCI interface
  DAC_Init();        // initialize SPI/DAC interface
  while(1){
    number = SCI_InUHex(); // read from PC keyboard
    number = number&0x1F;  // 5-bit only
    DAC_Out(number);       // output to SPI
  }
}
```

This main program can be used for dynamic testing. It creates a triangle waveform.

```
void main(void){unsigned char n;
  DAC_Init();    // initialize SPI/DAC interface
  while(1){
```

```
    for(n=0; n<32 ; n++){ // up 0 to 31
      DAC_Out(n);         // output to SPI
    }
    for(n=30; n>0 ; n--){ // down 30 to 1
      DAC_Out(n);         // output to SPI
    }
  }
}
```

5. Design and write the music device driver software. Create separate **Music.h** and **Music.c** files. Place the data structure format definition in the header file. Add minimally intrusive debugging instruments to allow you to visualize when interrupts are being processed.

6. Write a main program to run the entire system.

A "syntax-error-free" hardcopy listing for the software is required as preparation. The TA will check off your listing at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines.

Procedure (do this during your lab period)

1. Use the simple main programs to debug the SPI/DAC interface. Experimentally measure the speaker voltage/current versus digital value. Compare the measured data from the predicted data calculated as part of the preparation. Adjust resistance and capacitor values to get an approximately linear relationship between the digital output and the speaker current.
2. Using debugging instruments, measure the maximum time required to execute the periodic interrupt service routine. Adjust the interrupt rate to guarantee no data are lost.
3. Debug the music system.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
  Detailed circuit diagram of all hardware attached to the 6812 (preparation 1 and 2)
C) Software Design (no software printout in the report)
  Draw pictures of the data structures used to store the sound data
  Draw a data flow graph illustrating out information in the data structure is converted to music
D) Measurement Data
  Show the theoretical response of speaker current versus digital value (preparation 2)
  Show the experimental response of speaker current versus digital value (procedure 1)
E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)

You should be able to demonstrate the three functions **Rewind**, **Play** and **Stop**. You should be prepared to discuss alternative approaches and be able to justify your solution.

Your software files will be copied onto the TA's zip drive during checkout.

Lab 7e Preliminary Design and Layout of an Embedded System

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals • To interface a memory to a microcomputer,
• To study issues of power, clock, reset, and programming for a embedded system,
• To layout a PCB board.

Review • Valvano Chapter 9 on interfacing to your microcontroller,
• Data sheets for your microcontroller,
• Data sheets for your memory.

Starter files `Lab7eprep.sch Lab7eprep.pcb`

Background

You will use the CAD program **ExpressPCB** to layout a complete embedded system including reset, clock, power, memory and I/O. The software to activate the interface, hardware construction, and device testing will be performed in a later lab. The design of the system must satisfy certain requirements (things it must do) within a set of constraints (limitations dictated by the realities of the project). The embedded system must do something useful. There are three options listed below, but you have flexibility to define exactly what it is to do.

Option 1. You can create a general-purpose microcomputer development board. You can bring out the unused I/O pins to a set of 0.3in wide 0.1in male pins similar to the SB connector option on the Technological Arts boards. The final checkout for this option (Lab 9) will be to redo one of the previous labs (e.g., stepper motor, alarm clock, or music player) on this platform.

Option 2. You can make a hand-held game of pong. You will interface 8 LEDs to the microcontroller using a shift register. The LEDs will be positioned in a straight line. At any given time exactly one of the LEDs is on, signifying the position of the ball. The ball is always moving to the left or to the right. There are two switches on each end of the LED line for the paddles. When the ball reaches the end, the player must hit the switch to volley the ball back in the other direction. Score can be displayed using additional LEDs.

Option 3. You can make a voltage recorder. You will interface two switches and a DAC to the microcontroller. When you press the record switch, the analog voltage is sampled by the ADC and restored in memory. When you press the other switch, the previously recorded waveform is output using the DAC. The status of the system can be displayed using LEDs.

Requirements

- A microcontroller must run in expanded mode (with an external address and data bus),
- Memory must be interfaced to the address/data bus of the microcontroller,
- PCB layout of the system must include microcontroller, reset, clock, memory and I/O devices,
- There must be at least one input switch and one LED output,
- The final system (Lab 9) will be an actual device with chips soldered onto the PCB,
- The system should performs something useful similar to the above options.

Constraints

- The layout will be performed using **ExpressPCB**,
- Each **ExpressPCB** must done using the $51 Miniboard service,
- Although 3 groups will combine to produce one shared PCB layout (one PCB file),
  there will be separate Lab 7 preparations,
  there will be separate Lab 7 reports,
  there will be separate Lab 9 software and reports,
- There will be a specific list of parts that we will be willing to give you for building this system,
- You must purchase any additional parts that you require.

Preparation (do this before your lab period)

Begin by drawing a circuit diagram of the entire embedded system using any available CAD software. One possibility is to use the **ExpressSCH** CAD drawing program (see starter file **`Lab7eprep.sch`**). Label all pin numbers on every connector and chip. Search for details on how to generate an appropriate reset signal in the data sheets of the microcontroller. You can also search for existing circuit designs based on the same microcontroller, which are commercially available (e.g., www.technologicalarts.com or http://www.axman.com/). Next, layout the circuit using the **ExpressPCB** program (see starter file **`Lab7eprep.pcb`**). Follow the layout advice for the crystal in the data sheets of the

microcontroller. Please read the instructions included in **ExpressPCB** help menu. View the "Tips for Making PC Boards" page at **http://www.expresspcb.com**.

Add labels for your initials, the date, and the purpose of the board,
Place all components on the top side,
If possible align all chips in the same direction,
Configure the board so that all soldering occurs on the bottom side,
Add labeling to the top side to assist in construction and debugging,
Add test points at strategic points to assist in debugging (e.g., power, **/CS1**, **/WE**, and ground),
Each IC should have a bypass capacitor, placed as close to the chip as possible,
All components need labels (e.g., U1 R1 C1 J1), shown both on the board and the circuit diagram,
Avoid 90-degree turns, convert them to two 45 degree turns,
After the circuit is done, cover the unused area on the bottom side with a ground plane.

Every lab group of two students will produce an independent circuit diagram and layout as part of the preparation. You must select the MiniBoard service, which creates three identical 3.8 by 2.5 inch boards for $51. Add features such as test points and labels that will make it easy to test the hardware. Please print out this initial layout and turn it in along with the regular circuit diagram.

Write a one-page proposal to the TA describing exactly what your final system in Lab 9 will do. In other words, start with one of the above three *options* and add specific details on how it will work.

Procedure (do this during your lab period)

The lab groups will be combined so that three groups form a layout team. Discuss with your TA and the other layout team members about the proposed operation of the final Lab 9 system. The layout team will meet and integrate the three individual layouts into a single final layout. It is possible for one layout to solve similar, but not identical problems. Again using the simple layout program provided by **ExpressPCB**, create the final layout for the layout team. Please print out this final layout and turn it in to the TA for approval and manufacture. **ExpressPCB** will make three identical PC boards, one for each lab group.

You should collect all the devices that will be required to finish the Lab 9 construction.

You should print both sides of the PCB layout (without the mask). Create a mirror image of the bottom layer and glue the two pieces of paper to a thin piece of cardboard. Drill holes in the paper/cardboard "PCB". Create a complete 3-D mockup of the system placing the actual components on this paper/cardboard "PCB".

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
Regular circuit diagram (must be prepared on the computer using any CAD program)
PCB layout and three printouts (top, bottom and combined)
C) Software Design
Write a product description defining exactly what this embedded does (different for each group).
D) Measurement Data (none)
E) Analysis and Discussion (none)

Checkout (show this to the TA)

Show the 3-D mockup of the system to your TA.

Hints:

1) RAM memory will function properly with any connection between the RAM and computer address pins, as long as each of the computer address pins is connected to a separate RAM address pin. This is not true for ROM interfaces, which will require exact A0 to A0, etc.
2) RAM memory will function properly with any connection between the RAM and computer data pins, as long as each of the computer data pins is connected to a separate RAM data pin. This is not true for ROM interfaces, which will require exact D0 to D0, etc.
3) The Miniboard service does not produce the silk layer, but you may still wish to use it to help document the design.
4) **/CSD** and **CSD*** are two equivalent ways to specify the signal is negative logic

Lab 8e Interrupting Keyboard Interface and Calculator

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Redesign the hardware interface between a keyboard and a microcomputer using interrupts,
- Study the concept of critical sections and nonintrusive debugging,
- Develop routines for fixed-point arithmetic.

Review
- Valvano Chapter 4 on basic interrupt mechanisms and reentrant programming,
- Valvano Chapter 6 on input capture interrupts,
- Valvano Section 8.1 on keyboard scanning and debouncing,
- The chapter on output compare in the Motorola Reference Manual.

Starter files
- `OC3` and `IC` projects, `RXFIFO.H`, and `RXFIFO.C`

Background

The interface to the keyboard will be performed using input capture interrupts. Microprocessor controlled keyboards are widely used, having replaced most of their mechanical counterparts. This experiment will illustrate how a parallel port of the microcomputer will be used to control a keyboard matrix. The hardware for the keyboard is shown in Figure 9.1. Your computer will drive the rows (output 0 or HiZ) and read the columns. The low level software that inputs, scans, debounces, and saves key's in a FIFO runs in the background using interrupts. To scan the keyboard, the software drives the first row low (output 0), while the other rows are off (output HiZ). The software then reads the columns, and any keys are pressed in that row will be identifies as zeros in the column position. If no keys are pressed in that row, then all column inputs will be high. In a similar manner the software checks the other three rows. To recognize that a key has been pressed (or released), your software will drive all four rows low (output 0), and detect a rise or fall on any of the column signals using input capture. **Your system will not need to handle two-key rollover.** For example, when some people type "1,2,3", they push "1", push "2", release "1", push "3", release "2", then release "3". In this lay, when we type "1,2,3", we push "1", release "1", push "2", release "2", push "3", then release "3".

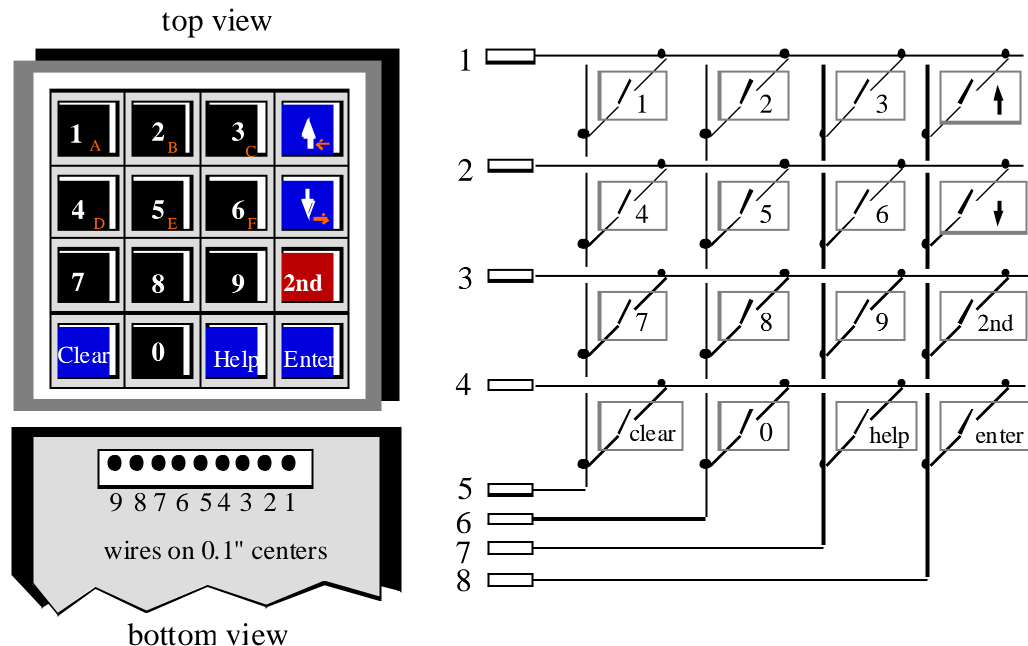


Figure 8.1. 0-9 keyboard, with up arrow, down arrow, 2nd, CLEAR, HELP, and ENTER. Pin 9 is not connected.

Low-level ***device drivers*** normally exist in the BIOS ROM and have direct access to the hardware. They provide the interface between the hardware and the rest of the software. Good low-level device drivers allow:
- new hardware to be installed;
- new algorithms to be implemented
  synchronization with gadfly, interrupts, or DMA
  error detection and recovery methods
  enhancements like automatic data compression
- higher level features to be built on top of the low level

OS features like blocking semaphores
user features like function keys
and still maintain the same software interface. In larger systems like the Workstation and IBM-PC, the low level I/O software is compiled and burned in ROM separate from the code that will call it, it makes sense to implement the device drivers as software TRAP's (SWI's) and specify the calling sequence in assembly language. In embedded systems like we use, it is OK to provide **KEY.H** and **KEY.C** source code files that the user can compile with their application. ***Linking*** is the process of resolving addresses to code and programs that have been complied separately. In this way, the routines can be called from any program without requiring complicated linking. In other words, when the device driver is implemented with a TRAP, the linking is simple. In our embedded system, the compiler will perform the linking.

In this keyboard lab, you will design the keyboard interface using interrupt synchronization. You will use both input capture and output compare interrupts to read and debounce the switch. There are two advantages of interrupts in an application like this. Placing the key input into a background thread, frees the main program to execute other tasks while the software is waiting for the operator to type something (unfortunately this system doesn't have anything else to do). The second advantage of interrupts is the ability to create accurate time delays even with a complex software environment. In particular, the output compare interrupt can be used to accurately wait for the bouncing to stop. A prototype keyboard device driver follows. As always, you are encouraged to modify this example, and define/develop/test your own format. This time we have all four categories of the device driver software.

1. Data structures: global, protected (accessed only by the device driver, not the user)

OpenFlag boolean that is true if the keyboard port is open
initially false, set to true by **Key_Open**, set to false by **Key_Close**
static storage

Fifo FIFO queue, with **Clr, Put, Get** functions
static storage initialized by **Key_Open**
linkage between Keyboard interrupt and **Key_InChar**

2. Initialization routines (called by user)

Key_Open Initialization of keyboard port
Sets **OpenFlag** to true
Initialized hardware, size of FIFO queues
Returns an error code if unsuccessful
hardware non-existent, already open, out of memory, hardware failure, illegal parameter
Input Parameters(none)
Output Parameter(error code)
Typical calling sequence

```
if(!Key_Open()) error();
```

Key_Close Release of keyboard port
Sets **OpenFlag** to false
Returns an error code if not previously open
Output Parameter(error code)
Typical calling sequence

```
if(!Key_Close()) error();
```

3. Regular I/O calls (called by user to perform I/O)

Key_InChar Input an ASCII character from the keyboard port
Tries to **Get** a byte from the Fifo
Returns data if successful
Returns an error code if unsuccessful
device not open, Fifo empty, hardware failure (probably not applicable here)
Output Parameter(data, error code)
Typical calling sequence (you are free to change it so `Key_InChar` waits for next input)

```
while(!Key_InChar(&data)) process();
```

Key_Status Returns the status of the keyboard port (checks FIFO to see if data is waiting)
Returns a true if a call to **Key_InChar** would return with a key
Returns a false if a call to **Key_InChar** would not return right away, but rather it would wait
Returns a true if device not open, hardware failure (probably not applicable here)
Typical calling sequence

```
if(Key_Status()) Key_InChar(&data);
```

4. Support software (protected, not directly accessible by the user).

There are five interrupt service handlers. A separate input capture interrupt is attached to each column

`ICHan0, ICHan1, ICHan2, ICHan3`

Occurs when a key is touched or released
This handler disarms all input captures, and arms an OC handler to occur 20 ms from now

`OCHan`

Occurs 20 ms after a key is touched or released
Scans the matrix, if exactly one key, it puts ASCII code into the Fifo
This handler disarms itself and arms all input captures

Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a **`printf`** statement added to your source code and single-stepping are very intrusive because they significantly affect the real time interaction of the hardware and software. When a program interacts with real time events, the performance is significantly altered. On the other hand, dumps, dumps with filter and monitors (e.g., output strategic information on LED's) are much less intrusive. A logic analyzer that passively monitors the address and data by is completely non-intrusive. An in-circuit emulator is also nonintrusive because the software input/output relationships will be the same with and without the debugging tool.

A program segment is *reentrant* if it can be concurrently executed by two (or more) threads. This issue is very important when using interrupt programming. To implement reentrant software, place local variables on the stack, and avoid storing into global memory variables. Use registers, or the stack for parameter passing (normal C call/return method). Typically each thread will have its own set of registers and stack. A nonreentrant subroutine will have a section of code called a *vulnerable window* or *critical section*. An error occurs if

1) one thread calls the nonreentrant subroutine
2) is executing in the "vulnerable" window when interrupted by a second thread
3) the second thread calls the same subroutine or a related subroutine. There are a couple of scenarios

A) 2nd thread is allowed to complete the execution of the subroutine
control is returned to the first thread
the first thread finishes the subroutine.

B) 2nd thread executes part of it, is interrupted and then re-entered by a 3rd thread
3rd thread finishes
control is returned to the 2nd process and it finishes
control is returned to the 1st process and it finishes

C) 2nd thread executes part of it, is interrupted and the 1st thread continues
1st thread finishes
control is returned to the 2nd thread and it finishes

A vulnerable window may also exist when two different subroutines access the same memory-resident data structure. Consider the situation where two concurrent threads are communicating with a FirstInFirstOut (FIFO) queue. What would happen if the PUTFIFO subroutine executed in between any two assembly instructions of the GETFIFO routine (or vice versa.)

An *atomic operation* is one that once started is guaranteed to finish. In most computers, once an instruction has begun, the instruction must be finished before the computer can process an interrupt. Therefore, the following read-modify-write sequence is atomic because it can not be reentered.

```
inc counter        where counter is a global variable
```

On the other hand, this read-modify-write sequence is not atomic because it can start, then be interrupted.

```
ldaa counter       where counter is a global variable
inca
staa counter
```

In general, nonreentrant code can be grouped into three categories all involving *nonatomic writes to global variables*. The first group is the *read-modify-write* sequence.

1) a read of global variable produces a copy of the data
2) the copy is modified
3) a write stores the modification back into the global variable

Example: **`Money +=100;`** which may be implemented in assembly as

```
ldd  Money  where Money is a global variable
addd #$100
std  Money  Money=Money+$100
```

In the second group is the *write followed by read*, where the global variable is used for temporary storage:

1) a write to the global variable is used to save the only copy important data
2) a read from the global variable expects the original data to still be there

Example:

```
short thePort;
void function(void){
  thePort = PTT; // save in global
// a bunch of stuff that may modify PTT, but not thePort
  PTT = thePort;} // restore original value
```

In the third group, we have a *non-atomic multi-step write* to a global variable:

1) a write part of the new value to a global variable
2) a write the rest of the new value to a global variable

Example:

```
short position[2];  // (x,y) location
void function(void){
  position[0] = PTT;  // x position
  position[1] = PTM;  // y position
}
```

Reentrant programming is very important when writing software in the context of multiple threads (interrupts). Obviously, we minimize the use of global variables. But when global variables are necessary must be able to recognize potential sources of bugs due to nonreentrant code. We must study the assembly language output produced by the compiler. For example, we can't determine whether the following read-modify-write operation is reentrant or not without knowing if it is atomic:

```
  time++;
```

The following read-modify-write operation is reentrant when using Metrowerks, because it is atomic:

```
  PTT = PTT | 0x01;  // set PT0
```

Preparation

Show the required hardware connections. Label all hardware chips, pin numbers, and resistor values. You will need 10 kΩ pull-up resistors on the column inputs. You should look at the voltage versus time signals on a scope to determine if hardware drivers are required, and to check if your particular keyboard has switch bounce. *Please check for valid (0 to +5V) digital signals on your external hardware before connecting them to your computer.*

The first main program you write will be used to test the keypad device driver. You are allowed to add lots of SCI output to assist in testing and debugging the keyboard interface. You could write this main program so that it inputs from your keyboard and outputs to the LCD display.

In the second main program you will design a four-function 16-bit signed fixed-point calculator. All numbers will be stored in signed 16-bit fixed-point format with a constant of 0.001. The full-scale range is from -32.767 to +32.767. You should be able to use the fixed-point routines developed in a previous lab, by converting to keyboard input and LCD output. The matrix keyboard will include the numbers '0'-'9', and the letters '+', '-', '*', '/', '=' and '.'. The HD44780 LCD display will show both a 16-bit global accumulator, and a 16-bit temporary register. You are free to design the calculator functionality in any way you wish, but you must be able to: 1) clear the accumulator and temporary; 2) type numbers in using the matrix keyboard; 3) add, subtract, multiply, and divide; 4) display the results on the HD44780 LCD display. No SCI input/output is allowed in the calculator program.

Procedure

Configure the keyboard and connect it to the system. Once again, test the device driver software in small pieces. You can use output ports and a scope to visualize when interrupts are occurring, when data is put into the Fifo, and when data is get from the Fifo. Collect some latency data (time from key touch to Fifo put) measurements and discuss them in your report. The exact time the key is touch will be recorded in the timer latch by the input capture hardware.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
    keyboard interface, showing all external components
C) Software Design (no software printout in the report)
    Explain how your software removes switch bounce

A call-graph illustrating the modularity of the software components of the calculator system

D) Measurement Data

Keyboard latency data

E) Analysis and Discussion (1 page maximum)

Checkout

You should be able to demonstrate the calculator functions. You should show the TA your method(s) to nonintrusively visualize the background thread interrupting the critical section of the foreground thread. Prove to your TA that your Fifo implementation has no critical sections (proof could be theoretical or experimental.)

Your software files will be copied onto the TA's zip drive during checkout.

Hints

1) Try using the debugging techniques developed in earlier labs.

2) Look at how the `RxFifo` is used to pass data from the SCI input interrupt to the `SCI_InChar` function in the file `SCIa` project.

3) The time executing in an interrupt service routine must be small and bounded. It is not appropriate to wait 20 ms inside an ISR.

Lab 9b Final Design and Testing of an Embedded System

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals • Build and test an embedded system,
• Obverse the read/write cycles of a computer/memory interface.

Review • Valvano Chapter 9 on interfacing to your microcontroller,
• Data sheets for your microcontroller,
• Data sheets for your memory.

Starter files • **RAMTEST** project

Background

This experiment includes the sequence of steps required to test an embedded system. Verification will involve many interrelated steps. The first step is to verify power and ground signals are properly applied to all the chips. The second step is to test the ability to download and run a program. It is important that this program be so simple that if the system doesn't work, then there is a hardware fault and not a software bug. One method to diagnose hardware failures is to do a side-by-side comparison with an operational system. Next, you run a program to test the microcomputer-memory interface. Finally, you will develop and test the final application as approved by your TA in Lab 7.

Preparation (do this before your lab period)

1: Recheck the PCB layout on the actual board looking for dropout (missing traces) and inadvertent shorts. Follow the +5V and ground signals around the board verifying with an ohmmeter than there are no shorts to inappropriate places. Assemble the components required to complete the interface, and solder them to the PC board.

2: Verify that Read Data Available overlaps Read Data Required and that Write Data Available overlaps Write Data Required. Draw the combined READ timing diagram and draw the combined WRITE timing diagram, as described in Chapter 9 of the book.

3: Before plugging the microcomputer and memory chips into their sockets, check the power supply circuit. Look on the oscilloscope to verify the absence of noise in the power signal. Using a DVM make sure the voltage level is within tolerance of the microcomputer and memory.

4: Next, write a very simple program that interacts with the switch input and LED output on your system.

5: Write the main application that implements the final objective of the embedded system. If you want change what your system does please get approval from your TA.

Procedure (do this during your lab period)

1: First test your system with a very simple main program (preparation step 4). This will verify you can download and run software on your system.
2: We have written a memory test program for our system (**RAMTEST**). Download this program, make any modifications and run it to verify that your RAM system is operating correctly.
3: Experimentally measure the actual memory interface timing diagrams. For example if location **0x8000** is external RAM on your computer. Run programs like this (add code to initialize the interface if needed)

```
#define memory *(unsigned char *)(0x8000)
void ReadTest(void){
  while(1){
    memory = 0xC3;     // write data to external RAM
  }
}
void WriteTest(void){  unsigned char data;    // internal RAM
  while(1){
    data = memory;     // read from external RAM
  }
}
```

While running these simple programs over and over, the memory interface signals can be observed on the oscilloscope using the dual channel capability of the scope. A pulse should be received from the **CS1** line during accesses to the memory. Observe **CS1** with the following signals in a pair-wise fashion:

• **CS1** and **E**

- `CS1` and `A14`
- `CS1` and `R/W`
- `CS1` and `D0`

Trigger on the falling edge of `CS1`. Using the dual channel capability, verify the proper relation between `E`, `A14`, `R/W` and `CS1`. In this way you should be able to verify the timing diagram created as part of the lab preparation. Draw the read and write timing diagrams as actually measured. Include both data available and data required. Which one did you measure and which one did you derive? Verify that all the microcomputer and RAM timing restrictions are met. Perform this measurement for both the read and write cycles.

4: Finally debug your embedded system application.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design
    Detailed circuit diagram of the system (from Lab 7)
C) Software Design (no software printout in the report)
    Briefly explain how your software works (1/2 page maximum)
D) Measurement Data
    Draw the theoretical read and write timing diagrams (preparation 2)
    Draw the measured read and write timing diagrams (procedure 1)
E) Analysis and Discussion (1 page maximum)

At the end of the first week Checkout (show this to the TA)

You should demonstrate your very simple main program and our memory test program (procedures 1 and 2) to the TA. At this point you and TA should agree about exactly what your final system is supposed to do.

Final Checkout (show this to the TA)

You should demonstrate the operation of the embedded system.
Your software files will be copied onto the TA's zip drive during checkout.

Hints:

1. If you need help soldering, please ask the TA or one of the 2nd floor lab staff. There will be a +5point bonus if you show your soldering job to the 2nd floor lab staff, and they think your soldering work is high quality.
2. The read access time (delay from address available to the start of read data available) can be determined by the number after the dash on the chip. For example, –7 means 70 ns, –8 means 80 ns, –1 means 100 ns, and –12 means 120 ns. The read access time is also the delay from the fall of CS1 to the start of read data available.
3. Don't try to create stuck high, stuck low, stuck together faults because it may damage the microcomputer or the memory.