Microcontroller Hardware and Software: A Laboratory Manual

Stephen J. Hegner
Department of Computer Science and Electrical Engineering
Votey Building
University of Vermont
Burlington, VT 05405
USA

hegner@emba.uvm.edu

Contents

Preface

Philosophy of this Manual

It is hardly an understatement to say that computers have become central to the practice of modern engineering. On the one hand, desktop computers and workstations have become an essential tool of the modern engineer. On the other hand, as microprocessors become ever more powerful, the use of embedded computers has become widespread as well, and in many sub-disciplines knowledge of embedded systems has become essential.

In the modern engineering curricula of today, virtually all students take at least one or two courses in programming and software development, often side-by-side with computer science and computer engineering majors. These courses typically focus upon the design of correct, readable, and maintainable software, and employ a development environment for a high-level language, such as *C* or *Java*, as the programming vehicle. Sadly, more often than not, the skills acquired in these courses are set aside during the study of embedded microcomputer systems. Typically, modern textbooks for such courses have little or nothing to say about software development. They begin by presenting the instruction set of the processor, and then go on to illustrate various small examples of how one does this or that. It is as though the existing body of knowledge on software development is not relevant to such systems.

It is my firm belief that the modern principles of software design are just as important in the design of small software and firmware programs for embedded systems as they are for the development of larger programs on desktop workstations. Of course, there are some differences. While *C* is making inroads into the domain of embedded systems, most programs are still written in a simple assembly language for the processor. In addition, more often than not, memory is at a premium. However, the need to write compact programs in assembly language is no excuse for discarding a discipline of software development. Particularly, with the availability of cross-platform development environments and simulators, readable and maintainable software for embedded microprocessor-based controllers may be developed on a workstation, using principles not unlike those for larger software systems.

The experiments in this manual were developed for a course which introduces the engineering student to embedded microprocessor-based controllers, while at the same time keeping a clear focus upon quality software development. It describes nine experiments, divided into two groups. The first group consists of four experiments which are

based entirely upon a PC-based development system and simulator. In these experiments, the students are introduced to techniques for programming the microcontroller in assembly language, without the need to be concerned about configuring and interfacing to hardware sensors and devices. Thus, they can focus entirely upon software development, and can even do much of the work on home systems, away from the engineering laboratory. Clear and well documented code, with judicious use of subroutines, is emphasized throughout. The intent is that the students should get into the habit of writing quality code from the very beginning. Since communicating with sensors and devices is the ultimate goal, considerable emphasis is placed upon manipulating ports. The second group, consisting of five experiments, the students take the lessons learned in the first part and apply it to sensing, display, and control of simple although real devices. In each of the experiments of this group, the students are encouraged first to develop prototype solutions on the PC-based development station, and only after that software is working satisfactorily, to move to the actual hardware and do the final design. In this way, the use of the tools introduced in the first group of experiments is reinforced as a means to develop actual microcontroller software, and the principles of quality software design learned in the first group are reinforced as well. Throughout, many skeletons of program organization, as well as suggestions for how to code specific routines, are provided. I feel that it is extremely important to give beginning students adequate guidance on good design strategies.

The equipment used in these experiments includes the P&E Microcomputer Systems MC68HC11 software development suite for PC's, the Motorola MC68HC11 development board, the Knight Electronics Mini-Lab 200 development station, as well as numerous displays, motors, and the like. It represents the available equipment in the microcontroller laboratory at the University of Vermont at the time that this course was developed, rather than a measured decision to select the best resources for such a course. Nonetheless, the principles behind most of these experiments should prove useful with other hardware configurations.

Acknowledgments

Ronald Williams served as a constant source of knowledge about all aspects of this course, as well as a sounding board for proposed experiments. His help in development of these experiments was invaluable.

Part I

Exercises Using the Simulator

Laboratory Exercise 1

Becoming Acquainted with the Software Development Environment

1.1 Purpose

The purpose of this experiment is to introduce you to some of the features of the software development environment which will be used in the course. While the illustrative assembly-language programs used do not compute anything useful, they do illustrate techniques for using the software effectively. Working through this exercise carefully will pay off in benefits which will make later laboratory exercises, in which you will need to focus on problem solving rather than on learning how to use the software tools, much easier.

1.2 Procedure

1.2.1 The Configuration of the Laboratory

The computers in the course laboratory are connected to the EMBA-CF network. In addition to the local floppy drive (`A:`) and the local hard drive (`C:`), a number of network drives are available. The most important is the `M:` drive, which is a private drive associated with your account; it is not shared with other users. You should use the `M:` drive to store programs and other files associated with your work in this course. The particular `M:` drive which is found depends upon the account used. Thus, if you have both a course account (*e.g.*, `ee101xx`) and a personal account (*e.g.*, `jsmith`) on the EMBA-CF system, then logging in as `ee101xx` will yield a different `M:` drive than will logging in as `jsmith`. The file space which is found on the `M:` drive is exactly the same as that found in the home directory of the UNIX account of the same user-id. While the `M:` drive is reasonably secure and reliable, it is always a good idea to save

copies of programs and the like to a floppy disk.

Other network drives (such as `Q:`), are used to store software and files which are shared by all of the PC's in a given laboratory. In general, these drives are read-only; it is not possible for ordinary users to alter files on them.

The local hard drive of the PC (the `C:` drive) is not protected from writing by ordinary users. (Such protection is impossible under Windows 95.) Nonetheless, it is very poor user etiquette to modify installed software, or to clutter directories with your own work. Except for use of temporary directories (*e.g.*, `C:\temp` or `C:\tmp`), never use the `C:` directory to save files. If you inadvertently destroy a system or software file on the `C:` drive, let the laboratory instructor know, so that it may be restored as soon as possible.

The CD-ROM drives on the laboratory machines are disabled. They will not be needed for the course.

1.2.2 Setting Up Your `M:` Drive

The first step in your laboratory work in EE101 is to create a directory on your `M:` drive for the storage of your programs. Start by opening up an MS-DOS window by clicking on the appropriate prompt. Then, type M:, followed by a return. This will place you on the `M:` drive. Create a directory called `Programs` by typing the command `mkdir Programs`. Use this directory to store the programs which you write for the laboratories. After every laboratory meeting, back up the `.ASM` files to a floppy disk as well.

1.2.3 Overall Picture of the Software

There are three software components which will be used in EE101, all products of P&E Microcomputer Systems, Inc. All run in the MS-DOS and Windows environments on a PC.

`IASM11` The *Integrated Assembler* `IASM` is a program which contains an editor for writing assembly-language programs, as well as a cross-assembler which converts 68HC11 assembly-language programs into files which may be downloaded to and executed on a 68HC11 system.

`SIM11A` The *68HC11 Simulator* `SIM11A` is a program which emulates the behavior of a 68HC11 microprocessor. This makes it possible to run and debug 68HC11 programs entirely on a PC, without the need for an actual 68HC11 processor.

`EVB11` The *Evaluation Board Monitor* is a program which enables the PC to communicate with a Motorola 68HC11 evaluation board (EVB). It allows you to run a program on the EVB, while observing values stored in memory locations and registers.

The components `IASM11` and `SIM11A` will be used for both laboratory and homework assignments, and so are part of the student software package which you are expected to purchase. The program `EVB11` is useful only in conjunction with a Motorola EVB, and will be used only in laboratory experiments.

The P&E Microcomputer Systems software which is installed on the laboratory machines is licensed solely for use in these laboratories. The license agreement does not permit students to copy this software for free in lieu of purchasing a student-licensed copy.

1.2.4 Introduction to `IASM11`

The first major step in this laboratory exercise is to become familiar with the environment of the `IASM11` integrated assembler package. It may be started in one of three ways: by double-clicking on an icon on the desktop, by selecting the appropriate item from the `Programs` menu, or by starting an MS-DOS shell and invoking the program manually. Your laboratory instructor will tell you which option is most appropriate for your laboratory.

1. Start up `IASM11` in one of these ways. If a small window comes up which requests arguments, just click on the `OK` button and continue. Otherwise, you will be greeted with a screen which announces the identity and version of the software, and tells you to press any key. Press a key, and the top-level environment screen will appear.

2. The `IASM11` screen should have appeared as a window on the larger screen, but it may have appeared as a full screen. In either case, it is easy to switch from one mode to the other. Hold down either of the two `Alt` keys on the keyboard, and strike the `Enter` key. The screen will switch to its opposite mode. This is a useful keystroke to know about, since the full-screen version can be easier on the eyes, while the window version is useful when other things (such as a `SIM11A` screen) must be viewed simultaneously.

   In the window mode, it is easy to change the size of the font used, and hence the size of the window. Just select the downward-pointing arrow at the upper left, next to the small white window which indicates font size. A different size may be selected from a wide list. Selections may also be made by clicking on the `A` button at the top of the window, and working through the resulting menu. Try some of these steps — you will want to use the most comfortable font and size layout while you work.

   (Note: The commands discussed in this item apply to any MS-DOS window, and are in no way specific to `IASM11`.)

3. The `IASM11` window which you are looking at is an editor window. Try typing anything, and note that the characters appear on the screen.

4. Before proceeding any further, it is best to familiarize yourself with an annoying bug in `IASM11`. This bug will not cause you any problems at all if you are aware of it and plan accordingly, but if you are careless it can cause you to go through some complex steps to avoid losing an entire editing session. Notice that on the bottom of the window there is an item which reads `F2-Save`. So, let us try to save the file. Press the `F2` key. A prompt for a path will appear at the top of the screen. Now, deliberately type a path which includes a nonexistent directory; for example `M:\porgrams\foo.asm`. A message will appear which states:

   ```
   Path not found - Press <Esc> to continue
   ```

   Dutifully press the `Esc` key; it will tell you something like

   ```
   Command aborted - Press <Esc> to continue
   ```

   Hit `Esc` again. All seems well. However, you still want to save the file; so, press `F2` again. See what happens? There is a way around this bug, but it involves creating a new buffer, copying the old buffer contents to the new, and saving the new buffer. We will explain how to do this later. For now, we will just exit `IASM11` and restart it, since the buffer contains nothing useful anyway. Look at the bottom of the screen, and observe that `F5-Exit` appears. Hit the `F5` and the window will disappear. Now, start up `IASM11` again. Before doing anything else, press `F2` and respond by typing the path of a potentially legal file (*e.g.*, `M:\programs\foo.asm`). Now, every time that you hit `F2` it will save to this file. **Get into the habit of providing `IASM11` with a path as soon as you start it up. If you make a mistake at that early point, you will not lose any work.**

5. At the top of the screen, there is a status line which may read something like

   ```
   foo.asm  Line 1  Col 11  Byte 10  Insert   Indent        Save
   ```

   It is providing information on the status of the buffer editor, including the name of the file, and the line, column, and byte position of the cursor. Try typing some lines and moving the cursor around, and notice how these values change. The word `Insert` means that the editor is in *insert* mode — room will be made for new characters which are typed. To switch to *overwrite* mode, press the `Insert` key, and note that the word on the status line has changed to `Over`. Press that key again to return to insert mode.

   The word `Indent` means that *autoindent* mode is enabled. This means that a new line will be indented as much as the current line. Type some lines and see what

it does. To toggle this mode, type `Ctrl-Q` followed by `I`. (Here `Ctrl-Q` means "control-Q;" hold down a `Ctrl` key and press the `Q` key.) Notice that the word `Indent` appears and disappears, as per the mode.

Next, type `Ctrl-Q` followed by `T`, and note that the keyword `Tabs` appears (or disappears, if it was previously present). This toggles the so-called *smart tabs* mode, with the *absence* of the keyword indicating smart tabs. Toggle this mode so that the word `Tabs` does not appear on the status line; *i.e.*, put the window in smart-tabs mode. Type a line of text with some spaces in it. Now go to the next line, and hit the `Tab` key a few times. Each time that you strike the key, note that the cursor jumps to a position just following a string of blanks of the previous line. In other words, it tabs according to the previous line. Toggle the mode again by typing `Ctrl-Q` and then `T`, and try this again. Note this time that the `Tab` key tabs a fixed amount. Unfortunately, the fixed tabbing amount may be adjusted only by initialization with the `IASMINST` program, but the laboratory machines should have this parameter set to tab to columns 1, 5, 9, 13, *etc.*

Finally, the keyword `Save` on the command line indicates that the buffer has been modified since the last save. Strike the `F2` key and note that this word disappears. Type a character and note that it reappears.

7. If you want to know more about the editor, you can press the `F1` key to bring up the help menu, and then use the arrow keys to highlight the topic which you want further information on. (Hit `Enter` after selecting the topic.) To exit from levels of the help system, use the `Esc` key. Generally speaking, the editor follows old "*Wordstar*" keystroke conventions, although you won't need to know these unless you are trying to do advanced things such as move a block of text. Try navigating to a few topics in the help system, so you know where to look to find out how to do advanced editing operations.

6. Now let us see how this utility may be used to edit and to assemble 68HC11 assembly-language programs. To start, we will make it easy. Rather than asking you to write a program, you will merely be asked to type one in. So, type in the program shown in Figure 1, and save it in a file named `M:\programs\lab1.asm`. Copy it verbatim, except put your own name and the current date at the top in place of the instructor's name and the given date. **But first...** If you have been following along, you will realize that the current buffer is already being saved in another file. So, the logical step is to open a new buffer. Hold down an `Alt` key, and note that the menu on the bottom of the screen changes to an "`Alt`" menu. While holding down the `Alt` key, press the `F3` key to create a new buffer. The screen will split in two, with the new buffer in the lower half. Note that either `Alt-F1` or `Alt-F2` will switch buffers. Make sure that the cursor is in the *old* buffer, and use `Alt-X` to close the other buffer. You will be asked whether or not you wish to save the old buffer. Since it contains only garbage, answer no. Now

proceed to type in the program, keeping in mind the editor commands for tabbing, *etc.*, which you have learned. It is a good idea to hit the `F2` key frequently to save the file. You never know when something is going to crash. A motto of all good programmers is **save often.**

7. Having entered the program, it is now time to assemble it. This is very easy in the integrated environment. Just hit the `F4` key. If you typed everything correctly, it should respond with a message indicating success, and request that you "strike any key." If you made an error, the offending line will be highlighted, together with an error message. Since you are just copying a working program, it should be an easy matter to find the error and correct it.

8. Before moving on, let us make sure that the assembler is generating the files that we want. Hit the `F10` key, and then select `Assemble` by highlighting that word on the bottom of the screen and pressing return. (Use the arrow keys to navigate.) The configuration which is displayed should be the following.

```
Assemble   F4
Object     .s19
Listing    on
Debug map  on
Cycle cntr on
Macros     view
Includes   view
```

   If any values are incorrect, highlight the incorrect entry and use the `Enter` key to toggle its values to the correct ones. If a laboratory machine is configured incorrectly, let the laboratory instructor know as well.

1.2.5 Introduction to `SIM11A`

Now it is time to run the program using `SIM11A`. Leave the `IASM11` program running. You will need it later.

9. Start up the `SIM11A` program. You should see a startup screen giving the version number. Strike any key, and a menu should come up. Select first `HC11A1`, and then `Extended`. If such a menu does not appear, it means that an initialization file already exists. In the laboratory, you cannot depend upon that initialization file being correct. So, quit `SIM11A` by typing `quit` at the prompt in the `Debug` window, open an MS-DOS shell, go to the `pemicro` directory, delete the file `SIM11A.CFG`, exit the shell, and then start `SIM11A` anew. Ask your laboratory instructor if you need help.

10. The first step is to load the assembled program. The cursor should be at the "`>`" prompt in the `Debug` window at the bottom of the screen. If not, type `F10` to put

```
;; EE101 Laboratory 1 Example Program.
;; Stephen J. Hegner
;; January 22, 1997

        org     $c000       ;; Begin code area.
code:
        ldaa    #n0         ;; Immediate mode load of A.
        ldaa    $10         ;; Direct mode load of A.
        ldab    n1          ;; Extended mode load of B.
        ldd     n3          ;; Extended mode load of D.
        std     n5          ;; Extended mode store of D.
        staa    $10         ;; Direct mode store of A.
        ldx     n1          ;; Extended mode load of X.
        std     n3          ;; Overwrite n3.
        ldd     #$6869      ;; Immediate mode load of D.
        ldy     #'hi'       ;; Immediate mode load of Y.
        lds     #stack      ;; Load stack location into stack register.
        psha                ;; Push A (byte) onto the stack.
        pshb                ;; Push B (byte) onto the stack.
        pshy                ;; Push Y (word) onto the stack.
        pula                ;; Pop stack (byte) to A.
        pulx                ;; Pop stack (word) to X.
        pulb                ;; Pop stack (byte) to B.
        wai                 ;; Wait for an interrupt.
;;
        org     $c100       ;; Begin data area.
data:
n0:     equ     $55         ;; The symbol n0 represents the constant 55h.
n1:     db      $9A         ;; Define a byte with value 9Ah.
n2:     db      $1F         ;; Define a byte with value 1Fh.
n3:     dw      $1234       ;; Define a word with value 1234h.
n4:     db      n0          ;; Define a byte with the value of n0.
n5:     ds      $4          ;; Reserve four bytes with
                            ;; no initial value given.
str1:   db      'This is a string.'
                            ;; Define a string of length 17.

stack:  equ     $C2FF       ;; Assume that the stack goes from
                            ;; C2FF to C200.

        org     $0000       ;; Fake on-chip memory.
onchip:
        ds      $10
        db      $33         ;; Fake a direct memory location.
;;
end                         ;; End of the program.
```

Figure 1.1: A simple 68HC11 program.

it there. To load the program, type `load m:\programs\lab1.s19`. Adjust this if you named or located the files differently. Make sure that you load the `.s19` file, and not the `.asm` file. The `.s19` file is an encoded object file containing executable 68HC11 code. The `.asm` file is the source-code file. The response should look like

```
Loading ...S19 File Loaded.
MAP file loaded.
```

If not, go back to Step 8 and make sure that you have the `Object` and `Debug map` fields set properly.

11. At this point, it is prudent to warn you of a bug in `SIM11A`. If you type `load` with no argument at the prompt, a pop-up window will appear which will let you navigate to a file. This interface, unfortunately, seems buggy, and we have seen `SIM11A` crash upon its use. So, the best advice is to type the whole argument to the `load` command right on the command line.

12. The program is now loaded in the simulator. To run it, we have to tell the simulator where to start executing. Type the command `pc c000` at the prompt to tell it to set its program counter to point to `C000h`. Upon doing this, code for the program should appear in the `Code` window at the upper right-hand corner of the screen, with the first instruction (at `C000h`) highlighted.

13. To run the program, type the command `run` at the command prompt in the `Debug` window. The instruction `wai` should be highlighted. The program has zipped through the initial part, and is now looping forever waiting for an interrupt. Unfortunately, this simulation is far too rapid for a human to follow. To see what is happening, we need to slow it down.

14. Press the return key to stop the program. The message `Command terminated by operator` should appear in the `Debug` window, and a prompt should follow. At the prompt, type `pc c000` again to restore the program counter to the beginning of the program.[1] The program text should reappear in the code window. Now let us step through the program one instruction at a time. At the `Debug` prompt, type the command `step`.[2] You may repeat the last command entered (in this case `step`) by pressing the `F9` key. Try it, working through the entire program. Without paying attention to details, note that the values of the registers, as shown in the `CPU` window in the upper left-hand corner, change at each step.

[1] Alternately, you may use the mouse to left-click repeatedly on the upward-pointing arrow "↑" at the right side of the `Debug` window until you reach the desired command; this action toggle through the previous commands. The mouse commands are totally undocumented, but some seem to work quite well.

[2] Again, there is a mouse alternative. Left-click on the green `Step` at the bottom of the `Code` window.

15. The simulator also allows us to look at the values of memory locations as the program progresses. Notice that the main data area of the program begins at location `C100h`. To display the contents of memory beginning at this location, type the command `md c100` at the `Debug` prompt. Notice that the contents of memory beginning at location `C100h` appear in the `Memory` window at the middle right of the screen, just below the `Code` window. The display is in two formats; to the left is a byte-by-byte hexadecimal display, and to the right is a character-code display, with nonprinting characters represented by a dot.

16. One further way to view the values of memory locations is to use variables. At the debug prompt, type the following commands

```
symbol loc_ten $10
var $10
```

    In the `Variables` window in the middle-left of the screen, an entry for the variable `loc_ten` should appear, with its values displayed in both hexadecimal and binary. These displayed values will change dynamically as the program alters them.

17. At this point, you should step through the program, one instruction at a time, and watch the changes in the values of the registers and memory locations, making sure that you understand what is going on. When you get to the part of the program which manipulates the stack, alter the memory window display so that the stack locations are displayed. Remember that the stack grows *downward*, so if the top of the stack is at location `C2FFh`, then let the memory window start displaying at, say, `C2E0h`.

18. The simulator allows you to change the values of memory locations. Reload the program by issuing the load command as described in Step 10. Then set the `Memory` window so it displays locations `C100h` through `C11Fh`. Now type `mm c100`. The program will respond with a prompt of the form `C100 = 9A >`. At this prompt, type `B2` followed by `Enter`. Notice that the value of location `C100`, as displayed in the `Memory` window, has changed to `B2h`. Another prompt will be displayed, requesting a new value for location `C101h`. Type `ee`, and note that the value of location `C101h` has changed to `EEh`. To exit this memory modification routine, type a period at the prompt; the single "`>`" prompt will reappear. Finally, type `mm loc_ten 55`, and note that the value of `loc_ten`, as displayed in the `Variables` window, has changed.

19. The simulator is indifferent as to whether you change the values of data items or code items. Type the command `pc c000`, so that the code is again displayed in the `Code` window. Also issue the command `md c000` so that the code values are displayed in the memory window. Type the command `mm c001 29` at the `Debug` prompt, and notice that the highlighted instruction in the `Code` window

has changed from `LDAA #55` to `LDAA $29`, and that the value of memory location `c001` has changed in the `Memory` window as well.

20. Instructions may even be changed in this fashion. Type the command `mm c000 c6` at the debug prompt, and observe the results, particularly in the `Code` window. Think about what has happened; you will be asked a similar question in a homework assignment.

21. Now issue the command `mm c000 f6`. A number of changes have occurred in the `Code` window as a result. Again, think about what has happened; you will be asked a similar question in a homework assignment.

22. Finally, go back to the `IASM11` window. Change the `org $c000` directive identifying the code area to read `org $c100`, and change the `org $c100` directive identifying the data area to read `org $c000`. In other words, switch the locations of the data and code areas. Save and reassemble this new file. Now switch back to the simulator `SIM11A`, and load the new `.s19` file. Since all memory references are via labels, and not absolute numbers, the program should function exactly as before. Type `pc c000`. The code does not appear in the `Code` window. Do you see what is wrong? The code now begins at location `C100h`, so type `pc c100`. Now run the program, and observe that it still works.

23 . You are done with this exercise. To exit `IASM11`, press the `F5` key, and to exit `SIM11A`, type `quit` at the `Debug` prompt. Remember to save your source files (`.asm`) to a floppy disk, and to log off of the laboratory machine.

1.3 Submission Requirements

For this assignment, all that you need submit, in addition to the cover sheet, is a printout of the `.asm` and `.lst` files for the program which you copied. Turn in these printouts for the versions which you modified for Step 22 above.

Laboratory Exercise 2

Basic Controller Design

2.1 Purpose

This exercise has several purposes.

- To illustrate techniques of program design.
- To provide hands-on practice in the coding of conditionals and loops in assembly language.
- To introduce some of the basic ideas behind the use of a microprocessor as a device to measure and control.

2.2 Procedure

2.2.1 The Simulated Situation

In this experiment, you will simulate the operation of an elementary measurement and control situation, which consists of a microcontroller, with three items connected to it.

1. A thermocouple which enables the microcontroller to observe the ambient temperature of a system.
2. A switch which enables the microcontroller to turn a cooling fan on and off.
3. A system which sometimes requires cooling from the fan to avoid overheating.

The system has both a lowest and a highest temperature at which it will operate properly; call these $T_{\mathbf{max}}$ and $T_{\mathbf{min}}$, respectively. If the temperature drops below $T_{\mathbf{min}}$ or rises above $T_{\mathbf{max}}$, it may malfunction. There is furthermore a third temperature $T_{\mathbf{panic}}$, above which continued operation of the system may result not only malfunction but in

serious damage. The mission of the microcontroller is to monitor the temperature of the system, and to perform the following sub-tasks.

st-1. If the temperature rises above a given point `T_fan_on`, the fan is turned on. Typically `T_fan_on` is a little lower than $T_{\mathbf{max}}$, but much higher than $T_{\mathbf{min}}$.

st-2. If the temperature drops below a given point `T_fan_off`, the fan is turned off. Typically `T_fan_off` is a little higher than $T_{\mathbf{min}}$, but much lower than $T_{\mathbf{max}}$.

st-3. If the temperature rises above `T_panic`, the system is shut down, and requires operator intervention to restart. Typically, `T_panic` is slightly lower than $T_{\mathbf{panic}}$.

2.2.2 Modelling with the Simulator

It is not possible to connect real thermocouples, motors, and systems to the simulator. Instead, in this experiment, the following simple modes of emulation will be used.

em-1. The fan and system statuses will be modelled by simple status bits, with one bit for each. If the corresponding bit is nonzero, the device is considered to be on; if it is a zero, the device is considered to be off. The program may thus "turn these devices on and off" by altering these bits.

em-2. The thermocouple connection will be modelled by a "simulated input port." This port will consist of two sub-ports: a *status port* which indicates whether or not there is new data available, and a *data port* which contains the actual data (*i.e.*, the temperature) from the thermocouple.

To test the program, you may halt the simulator manually. Insert new values on the two component ports of the simulated input port, and then restart the simulator. In this way, you will be able to observe the behavior of your program in response to inputs.

2.2.3 Designing the Program

In this exercise, we will provide you with many of the components of this program, while asking you to complete some components yourself. You should follow the procedure identified here, step by step.

Assigning meaningful names: Assembly language is a relatively cryptic and user-unfriendly medium for programming. It is therefore unwise to make it even more cryptic by using numbers directly to represent conditions. Rather, it is a good idea to assign these conditions names. Figure 1 displays an appropriate set of declarations for this program.

```
;;-----------------------------------------------------------------------
;; Data constants:
Fan_on       equ  1     ;; The fan is on.
Fan_off      equ  0     ;; The fan is off.
Status_OK    equ  1     ;; The status of whatever is OK.
Status_NOK   equ  0     ;; The status of whatever is not OK.
New_data     equ  1     ;; There is new data at the input port.
No_new_data  equ  0     ;; There is no new data at the input port.
;; Constants for decision-making points.
T_fan_off    equ  !15   ;; Fan shutoff temperature; degrees C.
T_fan_on     equ  !20   ;; Fan startup temperature; degrees C.
T_panic      equ  !30   ;; System shutdown temperature; degrees C.
;;-----------------------------------------------------------------------
```

Figure 2.1: Name declarations for the program.

Begin by opening up an `IASM11` window, and entering these lines to a file, which you might call `lab2.asm`. Remember to use the `m:` drive to save your work. Notice that these are just `equ` declarations; they do not generate any code; they merely associate names with numerical quantities.

These lines must be placed at the beginning of your program, before the associated names are used.

Declaring the data items: The required data declarations are shown in Figure 2. Notice that three one-byte registers are provided. `Fan_st` and `Sys_status` each

```
;;-----------------------------------------------------------------------
;;Data area.
        org  $c100
;; -----------------
;; Registers:
T_last: ds  1       ;; Last recorded temperature.
Fan_status:
        ds  1       ;; Fan status flag.
Sys_status:
        ds  1       ;; System status flag.
;; -----------------
;; The simulated port:
Port_T_status:
        ds  1       ;; Simulated status port for the temperature.
Port_T_data:
        ds  1       ;; Simulated data port for the temperature.
;;-------------------------------------------------------------------
```

Figure 2.2: Data declarations for the program.

take on binary quantities (their values may be either `0` or `1`, which are repre-

sented symbolically by `Fan_off` and `Fan_on`, respectively, for `Fan_status`, and by `Status_OK` and `Status_NOK`, respectively, for `Sys_status`. The register `T_last` records the last temperature observed, and contains an unsigned integer.

The high-level algorithm: Before developing the code for this problem, it is prudent to sketch out the high level algorithm. For this problem, that task has been completed for you, and is shown in Figure 3. Notice that the entire algorithm is

```
Initialize appropriate items;
While (true) do
  While (Port_T_status = No_new_data) do
    wait;
  end while;
  T_last := Port_T_data;
  Port_T_status := No_new_data;
  if (T_last >= T_panic)
    then
      goto Panic;
  elseif (T_last >= T_fan_on)
    then
      Fan_status := 1;
  elseif (T_last <= T_fan_off)
    then
      Fan_status := 0;
  end if;
end while;
Panic: Shut down the system and wait for operator intervention .
```

Figure 2.3: High-level algorithm for the program.

surrounded by a perpetual while loop. The program repeats its task of measuring the temperature and effecting the proper control endlessly. The only exception is the case in which the panic temperature is reached, in which case the system is shutdown, and the controller ceases operation.

Before you begin low-level coding, make sure that you understand the high-level algorithm. If you have questions, ask the instructor.

Initialization: Because this is not a "real" system, but rather a simulation, we need to initialize the values of certain parameters which would be captured from the environment in a real system. This code to accomplish this is shown in Figure 4. Note the use of meaningful symbolic constants, rather than numbers. Make sure you understand why we use values such as `#Fan_off`, rather than `Fan_off`. Think about what would happen if the `#`'s were omitted. Insert the code of Figure 4 into your program.

Polling: The component of the high-level algorithm which reads

```
        org $c000
;; Install some initial values.
        ldaa #Fan_off
        staa Fan_st        ;; The fan is off.
        ldaa #Status_OK
        staa Sys_status    ;; The current system status is OK.
        ldaa #No_new_data
        staa Port_T_status ;; No new temperature data is available.
```

Figure 2.4: Initialization of data values for the program.

```
While (Port_T_status = No_new_data) do
  wait;
end while;
```

may be realized in two distinct ways; via *interrupts*, and via *polling*. Interrupt handling will be considered later in the course; for now, polling will be used. The idea behind polling is extremely simple. The program just runs in a short loop in which the status port is continually queried for a change in value. The actual code is shown in Figure 5. Notice that to exit the two-instruction loop consisting of

```
poll:   cmpa Port_T_status  ;; New temperature data available?
        beq  poll;          ;; If not, loop and wait.
```

Figure 2.5: Polling for a change in the temperature port status.

the first two lines, the value at location `Port_T_status` must be changed by some external means. We will see how to accomplish that feat shortly. For now, insert this code into your program and make sure that you understand how it works.

The rest of the code: The rest of the code consists of some assignment statements, an if-then-elseif-then-else statement, and a surrounding outer while loop. It is your task to complete the code, using the techniques which were developed in class. At this point, clarity is far more important than cleverness. Do not worry about using a few extra bytes of storage space, or a few extra machine cycles. Focus upon writing something which is as easy as possible to understand. To understand how to test this code, refer to the next two paragraphs.

Placing key variables in the view window: It is extremely useful to be able to follow the values of key data items as the program executes. Enter the five variables identified in Figure 2 into the variable tracking window of the simulator. For example, to enter `T_last`, type the command `var T_last` at the "`>`" prompt in the `Debug` window. An appropriate entry should appear in the `Variables` window.

Later in the course, we will see how to load such variable definitions from an external data file, so they need not be retyped manually every time that the simulator is re-started.

Entering data to the port: To test this program, it would seem necessary to be able to enter data to the `Port_T_status` and `Port_T_data` memory locations, while the programming is running. While this is not quite possible, something almost as good is possible. Namely, the program can be suspended, memory locations altered, and the program resumed. To begin execution of a program in normal mode, just type the command "`g`" at the prompt in the `Debug` window. To suspend execution of the program, just hit the return key. The message `Command terminated by operator` should appear. Typing `g` again should restart things. Try this a few times with at least a program fragment which includes your polling loop, which should loop endlessly waiting for a change in the value of `Port_T_status`.

To alter the port values, suspend the running program. Then type the command `mm Port_T_status`. a prompt of the form `C103 = 00 >` should appear. At this point, you may enter a new value and hit return. You will then be prompted for a value for the next location, and so on. To terminate the process, type a period at the prompt. Here is a sample dialog.

```
> mm Port_T_status
C103 = 00 > New_data
C104 = 1E > !27
C105 = XX > .
>g
```

In each case, text to the right of a `>` was typed by the user. We have changed the status bit to `1`, and provided a temperature of 27 degrees Celsius.

You may use this technique even if you step through your program. In that case however, use the command `st` instead of the command `g`. Also, in that case, since the stepper suspends the program after each step, there is no need to hit return to suspend the program. With these tools in hand, you should be able to debug your program without undue difficulty. Make sure that you test all of the "boundary" cases, because this is what the laboratory instructor will do.

2.3 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of both a program source listing (`.asm` file) and an assembler listing (`.lst` file). Your program code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

Demonstration: The laboratory instructor will evaluate the operation of your program by running it. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 3

Subroutines and Ports

3.1 Purpose

This exercise has two main purposes.

- To provide hands-on practice in the coding of subroutines in assembly language.
- To further some of the basic ideas behind the use of a microprocessor as a device to measure and control, by introducing the use of a "real" parallel I/O port.

3.2 Procedure

3.2.1 The Simulated Situation

In this experiment, you will enhance the simulation of the operation of an elementary measurement and control situation introduced in Laboratory Exercise 2, which consisted of a simulated thermocouple, fan, and system. Refer to the handout for Laboratory Experiment 2 for details of this system..

The enhancement will proceed in two steps. In the first step, you will modify your program of Laboratory Exercise 2 in such a way that subroutines are used to realize several of the key operations. In the second step, you will replace the simulated temperature port with use of the parallel I/O port C.

3.2.2 Using Subroutines

To begin, we will modify the program of Laboratory Exercise 2 to include several subroutines. Since it is always a good idea to save working copies of your previous programs, **begin by making a copy of your program from Laboratory Exercise 2, and perform all work for this experiment on such a copy.** Next, since this new program will use subroutines, it is essential that space be allocated for the stack,

and that the stack pointer be initialized. So, in the declarations of your program, insert the line

```
stack   equ   $DFFF
```

and in the initialization lines, a corresponding command of the form

```
lds    #stack
```

Now let us tackle the task of replacing in-line code with subroutine calls. Figure 1 shows the high-level algorithm for the modified main program. Compare this algorithm

```
Initialize appropriate items;
While (true) do
  Call Get_new_temperature;
  T_last := Subroutine_return_value;
  if (T_last >= T_panic)
    then
      goto Panic;
  elseif (T_last >= T_fan_on)
    then
      Call Turn_on_fan;
  elseif (T_last <= T_fan_off)
    then
      Call Turn_off_fan;
  end if;
end while;
Panic: Shut down the system and wait for operator intervention .
```

Figure 3.1: High-level algorithm with subroutines for the program.

to that shown in Figure 3 of Laboratory Exercise 2. Notice that the only difference is that in-line statements have been replaced with subroutine calls. You will need to write three subroutines to accomplish this: `Get_new_temperature`, `Turn_on_fan`, and `Turn_off_fan`. It is best to write these one at a time. So, to begin, leave the original code to turn the fan on and off intact, and concentrate on writing the subroutine `Get_new_temperature`. The line `Call Get_new_temperature` is realized via a simple `bsr` command; there is nothing more to it. Furthermore, we will adopt the convention that the subroutine returns the temperature in `ACCA` (the `A` register). Thus, the line `T_last := Subroutine_return_value` will be realized by a `staa` instruction.

Now let us tackle design of the subroutine itself. The high-level algorithm is essentially the same as the corresponding in-line code from Figure 3 of Laboratory Exercise 2, and is shown in Figure 2. Only the last two lines are new. The first of these two new lines, `Subroutine_return_value := Port_T_data`, places the value of the temperature which the subroutine obtains into the appropriate return point, which we have already identified as `ACCA`. This line will therefore be realized by a `ldaa` instruction. The

```
While (true) do
  While (Port_T_status = No_new_data) do
    wait;
  end while;
Subroutine_return_value := Port_T_data;
Return to caller;
```

Figure 3.2: High-level algorithm for subroutine to read the temperature.

second line of the subroutine, `Return to caller`, is realized via a simple `rts` instruction. Write this new subroutine, and place it after the `wai` instruction of your current program. Get your modified program to work with only this one added subroutine before proceeding.

Once you have this single subroutine working, add the other two. These latter routines are even easier to write, since they do not involve any parameter passing.

Place documentation in your program as illustrated in the class handout for realizing delay loops via subroutines. This includes the following:

1. Separate each component; *i.e.*, each subroutine, the main program, the data area, and the declarations, with a line of commented asterisks.
2. Put a comment header on your program, identifying the author, date, course, and purpose of the program.
3. For each subroutine, insert a comment header identifying the purpose of the subroutine, as well as any conventions it uses for parameter passing.

Do all of this before proceeding!! Part of your grade will be based upon whether or not you documented and formatted your program properly.

3.2.3 Relocating the Program

In the next step of this experiment, the simulated temperature port will be replaced by use of a real port, the parallel I/O port C. Unfortunately, due to a "feature" in the `SIM11A` emulation of the 68HC11 in enhanced mode, it does not provide an accurate model of the operation of this port, even though the hardware which will be used later in the course (the Motorola EVB) does provide such a port.[1] To recover use of Port C, the simulator must be run in `Single Chip` mode, which is chosen (in lieu of `Expanded`)

[1]Technically, when a 68HC11 is run in enhanced mode, Port C becomes unavailable. However, it is almost always the case that when the chip is run in this mode, it is in association with a companion chip, the 68HC24. This companion chip provides Port C in a manner transparent to the programmer. Unfortunately, the designers of `SIM11A` decided not to provide any emulation of the 68HC24, and so this mode provides no Port C.

when the simulator is started. However, when using `Single Chip` mode, the ability to address memory at `C000` to `DFFF` hexadecimal is lost. So, we must resort to using the 256-byte on-chip memory at addresses `0000` through `00FF`.[2]

Fortunately, relocating a small program to the on-chip memory is very easy. Only two types of modifications need to be made. You should now make these changes **to a copy** of your program. Save the original!!

1. Change all `org` statements to place all code and data between addresses `0000` and `00FF`. Almost surely, you will have only two such statements, one for your code area, and one for your data area. Change the one for the code area to `org $0000`, and the one for the data area to `org $00C0`. That will allow 192 bytes for the program, and 64 bytes for the data area and the stack. This should be more than adequate for this simple little program.

2. Change the parameter for the `lds` instruction to reflect the fact that the stack begins at `00FF`, and not `DFFF`. If you have followed the above guidelines, this will amount to changing the single line `stack equ $DFFF` to `stack equ $00FF`.

Make these modifications to a copy of your program. Then, exit all instances of `SIM11A`, and start a new one, this time selecting `Single Chip` for the mode. Load your program and try running it. (Remember that you must now initialize the `PC` to `0000`, and not `C000`.) It should work exactly as before. If not, seek help from the laboratory instructor.

3.2.4 Using a General-Purpose I/O Port

To realize any sort of control or instrumentation application, the microprocessor must communicate with the outside world. It does so via *ports*. The 68HC11 has five ports, named `Port A`, `Port B`, `Port C`, `Port D`, and `Port E`. Each of these ports has certain special characteristics, and some may be used in a variety of ways. Port C is one of the most versatile. It contains eight parallel data line, and may be used for either input or output. In fact, it may be switched between input and output as the program progresses. In this experiment, it will be used only as an input port, as a simulated connection to a temperature-sensing device.

While this laboratory handout should be self-contained as regards the procedure to be followed, you may wish to refer to the text or to manuals for further information. Sections 5.2 through 5.4 of the text by Kheir contains additional information on this subject. Section 4 of the *Motorola MC68HC11A8 Technical Data* manual contains additional information on parallel I/O, and Section 4.3 on simple strobed I/O containing specific information relevant to this exercise.

[2] On the evaluation board, the on-chip memory at `0000` through `00FF` is used by the tiny on-board monitor, but in the simulator, we may use it as we please.

The mode of Port C which will be used is termed *Simple Strobed I/O*. In this mode, communication is one-way. the input device can signal the the microprocessor that new input has been made available, but the processor cannot reply to the peripheral. Later in the course, we will look at "handshaking" protocols, in which communication is two-way. Nontheless, it is far from trivial to set up simple strobed I/O. There are several registers which are involved in the operation, which we need to discuss in turn. **Note that all I/O-related registers on the 68HC11 are located at memory addresses starting with 1000h.** Look at the chart on pages 3-2 and 3-3 of the *Motorola MC68HC11A8 Technical Data* manual and follow along as you familiarize yourself with these items.

`PORTC` The register `PORTC` is located at address `1003h`. It is called the *Port C data register*. When an input (or output) operation occurs, this is where the data are placed.

`PORTCL` The register `PORTCL` is located at address `1005h`. It is called the *Port C latch register*. When a certain operation, called a *latch operation*, is performed, the contents of `PORTC` are transferred to `PORTCL`. These contents remain until a new latch operation is performed. A latch is performed when the instrument wishes to signal the microprocessor that there is valid new data available at the port. This latching operation also sets a flag in register `PIOC`, thus informing the processor that valid new data is available.

`PIOC` The register `PIOC` is located at address `1002h`. It is called the *Parallel I/O Control Register*. Seven of its bits encodes critical information on how the port behaves, and one bit (`STAF`) serves as a flag which indicates that latching has occurred. The functions of these bits are discussed in Section 4.5 of the *Motorola MC68HC11A8 Technical Data* manual.

`DDRC` The register `DDRC` is located at address `1007h`. It is called the *Data Direction Register for Port C*. The purpose of this register is to define whether a particular bit of Port C is an input bit or an output bit. If a particular bit in `DDRC` is a `0`, the corresponding bit in `PORTC` is an input bit; if it is a `1`, the corresponding bit is an output bit. Since we will use Port C only for input in this exercise, this value of this register will be set to `00h`.

Now let us begin the process of modifying the program to make use of this port as the input port for temperature data. As the first step, add symbol definitions for these ports to your program. These definitions should appear as shown in Figure 3. Add these definitions at the beginning of the file, before the code or data sections.

There are three more definitions which are necessary, and which should be added immediately following the four shown in Figure 3. These new definitions are shown in Figure 4, and should be added now. They will be discussed as they are used.

```
PIOC        equ  $1002 ;; Parallel I/O control register.
PORTC       equ  $1003 ;; Port C data register.
PORTCL      equ  $1005 ;; Port C latch register.
DDRC        equ  $1007 ;; Data direction register C.
```

Figure 3.3: Declarations for Port C.

```
IO_PATTERN   equ  $00   ;; All lines are inputs.
PIOC_config  equ  $02   ;; PIOC configuration.
STAF_bit     equ  $80   ;; Mask for STAF bit of PIOC register.
```

Figure 3.4: Data masks and patterns for Port C.

At this point, it is necessary to explain in much more detail how acquisition of data from Port C is effected. Shown in Figure 5 below is a high-level description of the initialization process. The first line, which stores the I/O pattern in `DDRC`, just

```
Store the I/O pattern (IO_PATTERN) in DDRC;
Configure PIOC (store PIOC_config in PIOC);
```

Figure 3.5: High-level initialization for data read from Port C.

informs the port that all lines are to be used for input (since `IO_PATTERN` is `00h`.) The second line, which configures PIOC, is far more complex. Welcome to one of the most tedious aspects of programming microcontrollers! The information on how to configure PIOC may be found on pages 4-4 and 4-5 of the *Motorola MC68HC11A8 Technical Data* manual. To get it correct, you need to read about what each bit does, and then configure it correctly. Although `PIOC_config` gives the correct configuration for this exercise, let us go through them, one by one, in order to understand why. Compare the comments below to the descriptions in the manual, and try to understand why `02h` (or `00000010` binary) is an appropriate pattern.

STAF This bit cannot be set by a store command. It is a flag which is set when a latch has occurred. The value used in `PIOC_config` does not matter.

STAI This bit is set to `1` when using interrupt-based processing, and to `0` when using polling. Since we are polling, it is set to `0` in this program.

CWOM This bit deals with the implementation technology, and is of no importance in the simulation. We set it to `0`.

HNDS This bit defines the handshake mode. We are using *simple strobe mode*, so it is set to `0`.

OIN This bit is irrelevant in simple strobe mode.

PLS This bit is irrelevant in simple strobe mode.

EGA Latching occurs when the `STRA` pin of the chip is goes through a transition. This bit determines whether the latching transition is low-to-high, or high-to-low. For this exercise, we want low-to-high latching, so it is set to `1`.

INVB This bit is irrelevant for Port C. It pertains to the behavior of Port B.

Having coded these initialization steps, it is now time to turn to the task of understanding how to read data from the port. The steps of the associated subroutine are shown in Figure 6 below.

```
While STAF bit of PIOC is 0 do
    wait;
end while;
read PIOC;
read PORTCL;
return;
```

Figure 3.6: High-level algorithm for subroutine to read data from Port C.

Let us analyze this procedure line-by-line. The while loop is the equivalent of the wait for `Port_T_status` of the previous program. In this case, new data is signalled by the `STAF` bit of `PIOC` taking on the value `1`. The next two lines have special significance. **It is crucial to understand that the `STAF` bit cannot be cleared just by writing a `0` to its position in `PIOC`.** Its value will remain unchanged if this operation is attempted. To clear `STAF`, two operations must be performed in sequence: a read of `PIOC`, followed by a read of `PORTCL`. These "reads" are typically loads into a register. How do we know that this is the way to clear `STAF`? Read the penultimate sentence of the second paragraph under item 4.3.1 on page 4-2 of the *Motorola MC68HC11A8 Technical Data* manual.

Armed with this information, it is a relatively straightforward matter to convert the statements of Figure 6 to assembly language. The two read statements are just loads, and if the last is `ldaa`, it puts the return value for the subroutine into the register `ACCA`, which follows the specification of the first program. The only point for which you may require a little help is the test of the `STAF` bit. There are a number of "trick" ways to do this, but we will suggest a straightforward one here. Since we haven't covered logical operations yet, the code will be provided for you. Figure 7 provides a straightforward realization of the while-wait loop. Once bit operations are covered in the course, you

```
        ldaa #STAF_bit
poll:   bita PIOC
        beq  poll
```

Figure 3.7: Realization of the while-wait loop of Figure 6.

should go back and try to understand why this works.

The final task is to interact with this program by simulating input to Port C. Fortunately, there are several convenient commands which may be entered at the debug prompt of the simulator.

`inputc` The `inputc` command, followed by a value, will place the appropriate value into the register `PORTC`.

`stra` The `stra` command, with an argument, sets the value of the `stra` line.

Figure 8 shows an annotated example of use of the program. Note carefully how the debug commands `stra` and `inputc` are used to enter data values.

	Comments:
`>load m:lab3.s19`	
`Loading .,.S19 File Loaded.`	
`MAP file loaded.`	
`>pc 0000`	Program starts at `0000` now!
`>g`	Loop for a while.
`...`	
`Command terminated by operator.`	Interrupt execution.
`>inputc !22`	Simulate a value at Port C.
`>stra 0`	Drive `stra` low.
`>stra 1`	Drive `stra` high to latch.
`>g`	Compute again.
`...`	Program should record temperature.
`Command terminated by operator.`	Interrupt execution.
`>inputc !15`	Simulate a new value at Port C.
`>g`	Run for a while. Note that program does not
`...`	have new temperature.
`Command terminated by operator.`	Interrupt execution.
`>stra 0`	Drive `stra` low.
`>stra 1`	Drive `stra` high to latch.
`>g`	Run program.
`...`	Note that program has new temperature.

Figure 3.8: A session with `SIM11A` illustrating simulated data input.

As a final step, once your program is working, you should delete extraneous data items which are no longer valid in this version, such as `Port_T_data` and `Port_T_status`.

Clean up your program and make sure that it is clear and well documented.

3.3 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for each of the two programs. developed. The first program is the one described in 2.2, and the second program is the one described in 2.4. Your programs code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 4

Multiple Ports and Handshaking Protocols

4.1 Purpose

This exercise has several purposes.

- To study techniques using one port for input and a second port for output.
- To study the full input handshaking communications protocol for Port C.
- To learn how to manage 32-bit quantities on the 68HC11.
- To prepare the sequence of programs of Laboratory Exercises 2 through 4 for use with the Motorola EVB.

4.2 Procedure

4.2.1 The Simulated Situation

In this experiment, you will further enhance the simulation of the operation of an elementary measurement and control situation introduced in Laboratory Exercises 2 and 3, which consisted of a simulated thermocouple, fan, and system. Refer to the handout for Laboratory Exercises 2 and 3 for details of this system.

The enhancement will proceed in three steps. In the first step, you will modify your program of Laboratory Exercise 3 so that the temperature which is read from Port C is written to Port B. In the second step, you will reconfigure Port C to use full input handshaking. In the final step, you will install a long delay loop to space temperature readings.

4.2.2 Using Port B

In Laboratory Exercise 3, the temperature which was read from Port C was simply stored in a memory location of the processor system. In preparation for the next experiment, in which you will read a data value from Port C and then display it, you will reconfigure your program so that the value read from Port C is written to another port, Port B.

Port B is an eight-bit output-only port. Use is very simple. Basically, all that you do is to write a byte to the memory address of the port, and the value appears on its output lines. To perform the modification for this experiment, take the following steps.[1]

B1. From Table 3.1 of the *Technical Data* manual, or from Section 5.3 of the text by Kheir, determine the memory address associated with Port B, and place an appropriate definition in your program. The definition should have the form `PORTB equ $????` in which `????` is replaced with the port address.

B2. Change the code of your program so that, instead of saving the last recorded temperature to a memory location, it is written to Port B.

B3. Experiment with your program, making sure that the desired value may be read from Port C and written to Port B. Set up and use variables defined within the simulator. Do not use the `portb` debug command, which places a value at the output port. Your program, and not a debug command, should write the value to Port B.

Note that since this exercise continues to use ports, you must use `Single Chip` mode and locate your code, data, and stack within the range `00h` through `FFh`.

4.2.3 Implementing Full Input Handshaking

There are two different input protocols which may be used with Port C. In Laboratory Exercise 3, *simple strobed input* was used. In that mode, the existence of new data at the port is signalled by setting the `STAF` flag, which is cleared by reading it and then the port. However, there is no way to signal the device which is writing to Port C that the data have been read, and that it is safe to write new data. It is therefore possible that unread data will be overwritten before the program has been able to read the previous data, and data may be lost. In some applications, this kind of error may be unacceptable, so it is necessary to have some way of signalling the external device that it is safe to write new data to the port. A handshaking protocol provides such a means of two-way communication. The *full input handshaking protocol* behaves similarly to

[1] As with all programming experiments, you should make these modifications to a copy of your program of Laboratory Exercise 3, saving the original.

the simple-strobed input protocol, except that it uses one extra line of the processor, the `STRB` line, to communicate with the external device. Whenever the external device writes and latches data to the port by writing to `PORTC` and then asserting (*i.e.*, setting to a value denoting `true`) the `STRA` line, the `STRB` line is *de-asserted* (*i.e.*, set to a value denoting `false`). When the program has read the data which has been latched in `PORTCL`, the `STRB` line is asserted, thus signalling the external device that it is safe to write and latch new data to the port.

The task for this part of the experiment is to modify your program to use the full input handshaking protocol for Port C. To help familiarize you with some of the nuances of configuring a processor in this fashion, we will not give you the exact settings of `PIOC` to realize this protocol. Rather, we will tell you what the settings should do; you will then need to determine the exact value to program into `PIOC`. Information on configuring `PIOC` is contained on pages 4-4 and 4-5 of the *Technical Data* manual. Here is the list of conditions which you must configure.

- Polled input (as opposed to interrupt-driven input), will still be used, as was the case in Laboratory Exercise 3.
- Full input handshaking mode will be used.
- Handshaking should be *interlocked* as opposed to pulsed. (In *interlocked* mode, once the `STRB` line is asserted, it can only be de-asserted by a read of `PORTCL`. In *pulsed* mode, it will be de-asserted after two clock pulses.)
- The active level for `STRB` should be logic one. (This means that to *assert* `STRB` means to set it to a logic one value, and to *de-assert* it means to set it to a logic zero value.)
- The active edge for `STRA` will continue to be the rising edge, as it was in Laboratory Exercise 3.
- The *Port C Wire-OR Mode* value does not matter.

Study the information in Section 4.5 of the *Technical Data* manual, and identify the correct configuration byte for `PIOC`. If you have any doubts, check with the laboratory instructor. Once you have identified the correct configuration, place the appropriate statement in your program. Make sure that you document, in your program, the action of this configuration byte.

It is now time to test the new configuration. In addition to those which you used in Laboratory Exercise 3, there is one new `Debug` command which is essential. The command `strb` behaves analogously to `stra`, but for the `STRB` line. If you run it with an argument (`0` or `1`), it sets the `STRB` line to that value. If you run it without an argument, it displays the value of the `STRB` line. For the most part, in this experiment, you will only want to read the `STRB` line. Other commands will set its value implicitly.

The test procedure is a bit awkward, since there is (apparently) no way to maintain a running display of the value of the `STRB` line. Rather, you must explicitly issue the `strb` command at the `Debug` prompt each time that you want to examine its value. Run through several simulated inputs, in a fashion similar to that illustrated in Figure 8 of Laboratory Exercise 3. However, this time, run the `strb` debug command after each command. In addition, instead of using the `g` command, use single stepping (`st`). Make careful note of exactly when `STRB` goes from `0` to `1`, and when it goes from `1` to `0`. Compare this behavior to that described in 4.4.1 of the *Technical Data* manual. **In trying to make sense of this behavior, remember that the `STRB` line is available to an external device, while the internal registers and memory locations of the processor are not.**

There are two possible ways in which a system might be designed to use the handshake signal from `STRB`.

1. When the `STRB` line is de-asserted, the processor does not recognize writes to `PORTC`. The value stored at `PORTC` is not altered by such writes. Only when `STRB` is asserted will such writes have any affect.
2. Writes to `PORTC` will always place the new value in that register. A "well-behaved" input device must check the value of `STRB` before attempting a write, and perform an actual write only if `STRB` is asserted.

These are not configurable options for the 68HC11/68HC24, but rather inherent properties of the processor architecture. Let us design a simple experiment to determine which one is correct. Run the following experiment.

S1. Observe the value of `STRB`.

S2. Write a value to `PORTC` using the `inputc` command of the `Debug` window.

S3. Latch the value by using the `stra 0` and then the `stra 1` command.

S4. Observe the value of `STRB`.

S5. Write a new, different value to `PORTC` using the `inputc` command of the `Debug` window.

S6. Try to latch the value by using the `stra 0` and then the `stra 1` command. Observe whether it has indeed been latched to `PORTCL`.

S7. Observe the value of `STRB`.

Based upon this experiment, determine which of the above possibilities is the correct one for the simulator.

In 4.4.1 of the *Technical Data* manual, the following is stated.

> Deassertion of the STRB line automatically inhibits the external device from strobing new data into port C.

Do the experimental observations agree with the statement in the manual? Sometimes, natural language is not precise enough to convey detailed technical information. In the next laboratory experiment, we will determine whether or not the actual processor pair 68HC11/68HC24 on a Motorola EVB behave in the same way as the simulator.

4.2.4 Installing a Long Delay Loop

In Laboratory Exercise 5, the program which you have just developed will be tested on the Motorola EVB. To test this program in "real time," it is necessary to wait for a noticeable period of time between reads, so that the effect of an input to Port C while the `STRB` line is de-asserted can be observed. Since these inputs will be applied manually by toggling switches on a circuit development board, delays on the order of five to fifteen seconds or more are necessary. The largest unsigned number which can be stored in a sixteen-bit register is $2^{16} - 1 = 65535$. A reasonable delay loop will have a cycle time on the order of five to ten microseconds, so the longest delay which one can expect with a single loop is a fraction of a second. To obtain longer delays, we need to be able to work with larger numbers. Although the 68HC11 does not have any registers larger than 16-bit, there is nothing to prevent us from combining two 16-bit registers to form one 32-bit register. Let us combine `IY` and `IX` in just this fashion, as shown in Figure 1 below. For example, if `IY` contains `03D5h`, and `IX` contains `F4CAh`,

Figure 4.1: Viewing `IY:IX` as a 32-bit register.

then the associated 32-bit quantity is `03D5F4CAh`.

The idea of the delay program is to decrement this "register" until its value becomes zero. Shown in Figure 2 is the full algorithm. While this algorithm will work, it may be simplified greatly. We use the fact that if we decrement a 16-bit register whose contents is `0000h`, the new contents is `FFFFh`. The much simpler realization is shown in Figure 3 below.

Your first task of this section is to realize the algorithm of Figure 3 as a subroutine `Dec_xy` in 68HC11 assembly language. Once that has been done, write a subroutine `Delay` which delays for time proportional to the value in `IY:IX`. In writing the subroutine `Delay`, you may follow the pattern of the programs which were handed out in class, calling the subroutine `Dec_xy` instead of just executing `dex`. Bear in mind that you must now check both `IY` and `IX` for the value zero before exiting.

As a final step, install these subroutines into your program. Configure it so that once a data value has been read and latched, the program enters the delay loop, and

```
If IX > 0
  Then
    IX := IX -1;
Else if IY > 0
  Then
    IY := IY -1;
    IX := FFFFh;
  Else
    IX := FFFFh;
    IY := FFFFh;
End if;
```

Figure 4.2: Decrementing `IY:IX` as a 32-bit register.

```
If IX = 0
  Then
    IY := IY - 1;
End if;
IX := IX -1;
```

Figure 4.3: Simplified algorithm for decrementing `IY:IX`.

does not resume polling for new data at Port C until the delay is complete. Note that in the simulator, even a delay with `IY:IX` = `00010000h` will be a very long one, so you may need to be somewhat inventive in testing these routines.

4.3 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for the final program embodying all three parts identified in 2.2 through 2.4 above. Your programs code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which details how full input handshaking mode works *on the simulator*. Make sure that you answer the questions posed at the end of Section 2.3.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Part II

Exercises Involving the Evaluation Board and Prototyping Unit

Laboratory Exercise 5

Use of the Evaluation Board and the Prototyping Unit

5.1 Purpose

This exercise has several purposes.

- To gain familiarity with the Motorola M68HC11EVB evaluation board and the associated P&E Microcomputer Systems `EVB11` software.
- To gain familiarity with the Knight Electronics Mini-Lab 200 lab station.
- To take an application which has been developed using a simulator and install and test it on a prototype system.

5.2 Equipment

The following equipment will be used in this experiment. Before proceeding, make sure that your group has all of these items available.

1. One PC running the P&E software `IASM11`, `SIM11A`, `SIM11`, and `EVB11`.
2. One Motorola M68HC11EVB evaluation board, with power supply.
3. One Motorola *M68HC11EVB Evaluation Board User's Manual*.
4. One serial port cable, for connecting the evaluation board to the PC.
5. One Knight Electronics Mini-Lab 200 analog-digital lab station.
6. At least 25 wires for configuring the lab station and for connecting the development board to the lab station.

5.3 Procedure

5.3.1 The Simulated Situation

In this experiment, you will take the software which you wrote for Laboratory Exercise 4 and install and test it on a true prototype system, consisting of a Motorola M68HC11EVB evaluation board, coupled to a Knight Electronics development station. The evaluation board is the "computer," while the development station will be used to supply the electronic signals which simulate the input of a temperature, as well as the action of observing and setting the strobe lines `STRA` and `STRB`. In addition, LED's on the lab station will be used to observe the temperature which the program has recorded.

There are two distinct steps to be followed in this experiment. First, you will connect the evaluation board to the PC, and install and run your program on the Motorola evaluation board. Second, you will configure the lab station and interconnect it to the evaluation board. The lab station will provide the input to Port C, and the pulse to `STRA`, and will display, on its LED's, the values on the lines of Port B, as well as the value of `STRB`.

5.3.2 Modifying the Program

Recall that, due to limitations on the ability of `SIM11A` to simulate the behavior of ports on the port expansion chip, it was necessary to develop the program for Laboratory Exercise 4 to use the on-chip memory at addresses `00h-FFh`. On the evaluation board, this on-chip memory is used by the monitor program, which is a tiny operating system which supports communication between the board and the PC. Therefore, to run your program on the evaluation board, it is necessary to reconfigure it to use the external memory at addresses `C000h-DFFFh`. This should be extremely easy; just change the `org` commands identifying the location of the code and data portions of your program, and change the initialization of the stack. Suggested addresses are `C000` for the beginning of the code area, `D000` for the beginning data area, and `DFFFh` for the initial top of the stack.

Once you have reconfigured the program, you should test it to the extent possible on `SIM11A`. You will not be able to use the ports. but you will be able to see whether or not it crashes. You may also wish to test it on `SIM11`, the older 68HC11 simulator, which is also installed on the laboratory machines, and which simulates both external memory and ports at the same time. If you have trouble understanding the operation of `SIM11`, ask the laboratory instructor for assistance.

If you are working in a group, the program of each member of the group should be modified and tested in this fashion.

5.3.3 The Motorola EVB and the Program `EVB11`

The Motorola evaluation board (EVB) is a small computer system which is built around the MC68HC11 processor and the MC68HC24 port replacement unit. It is designed specifically for the development of prototype systems. The configuration includes a tiny monitor program on an EPROM chip, 8Kb. of memory at addresses `C000-DFFFh`, and two RS-232 serial ports. Through one of these serial ports, the unit is connect to the PC. The program `EVB11` is run on the PC, and enables communication between the development board and the PC. The environment provided by `EVB11` looks roughly like that provided by `SIM11`. The difference, of course, is that the programs run on the processor on the evaluation board, rather than emulated by a program on the PC,

The units which are used in the laboratory are mounted on a plexiglass platform, and are connected to a companion power supply. Examine the unit at your station, identifying the following components.

User's manual A copy of the *M68HC11EVB Evaluation Board User's Manual* should be available for each group. These manuals provide detailed information on the configuration and use of the evaluation boards. Take a few minutes to leaf through and familiarize yourself with this book.

To aid in identifying the following three components, refer to Figure 2.1 of the *M68HC11EVB Evaluation Board User's Manual*.

RS-232C port connectors There are two 25-pin female connectors on the EVB, which provide RS-232C compatible serial ports. `P2`, which is the connector furthest from the power supply, is the *terminal port connector*, and is the one which is to be connected to the serial port on the PC to provide communication. The other, `P3`, is the *host port connector*, which provides an independent connection for another device.

60-pin connector On the opposite end of the board from the serial-port connectors is `P1`, the 60-pin *MCU I/O port connector.* In 2.4.4 of the *M68HC11EVB Evaluation Board User's Manual*, you will find a diagram identifying the pin-out of this connector. There should also be a piece of paper pressed over the pins on the board which indicates the pin-out. For our immediate purposes, this connector provides access to Port B and Port C, as well as to the `STRA` and `STRB` lines. The connection wires for prototyping will have a connector on one end which will slide over a pin on this port.

Reset switch On the EVB, just above the power connector and next to `P2`, is the reset switch. Refer to Figure 2.1 of the *M68HC11EVB Evaluation Board User's Manual* for its location. You will use this switch often in this and future experiments. When this switch is depressed, the EVB "resets," meaning that the

monitor program attempts to locate the PC-based program, sends a prompt, and then waits for further input.

The following basic yet important switch lies on the power supply.

Power supply switch The power supply is a separate module, mounted on the plexiglass board. Make sure that you locate the power on-off switch. Note that the EVB does not have a separate power switch. The EVB cannot be "shut off," except by shutting off power to its power connector.

At this point, you should connect the EVB to the PC with a serial cable. You may use either of the two serial ports on the PC. Follow these steps.

3.3:1 Connect the PC to the EVB with a serial cable. Make sure that you connect to port `P2` on the EVB, which is the port furthest from the power supply. The other end of the cable may be connected to either of the two serial ports of the PC.

3.3:2 Make sure that the power switch of the EVB power supply is off, and then plug in the supply.

3.3:3 Turn on the power supply for the EVB.

3.3:4 Start the `EVB11` program on the PC by double-clicking on its icon. A screen with the message "Type any key to continue." should appear. So, type any key. If the `EVB11` display does not appear, try striking the `1` and `2` keys alternately, to see if it responds to an alternate port. If this does not work, press and release the reset button of the EVB and repeat the process of trying key `1` and then `2`. If you still have no success, check your wiring and/or get help from the laboratory instructor.

3.3:5 The `EVB11` window should display a "`>`" prompt in the `Debug` window. Strike the `Enter` key; a new `>`, or perhaps the word "`What?`" followed by a prompt, should appear. This indicates that the PC is communicating with the `EVB`.

3.3:6 It is now time to load the program. To do so, type the command `loadall`,[1] followed by the `Enter` key. At the prompt, type the full path to the `.S19` file of your program. At the resulting prompt, type the path to the `.S19` file of your program. (The maximum length of the path is somewhat limited, so you may need to relocate the file to, for example, the `c:\temp` directory to keep it short enough.)

Upon loading the file, a timeout error message may appear several times in the debug window. If this happens, press and release the reset switch on the EVB,

[1] You should use the `loadall` command rather than just the `load` command, since the latter will not load your `.map` file. Without the contents of the `.map` file, `EVB11` will not have access to the values of symbols which are defined in the program, and the resulting display will be more cryptic.

and then hit the `Enter` key. You should get the debug prompt `>` back, perhaps with a message about loading the `.map` file first. If not, ask for help.

3.3:7 Issue the command `pc C000` to position the program counter at the beginning of the program.

3.3:8 The program may be stepped, in the same way as the `SIM11A` program. Try the `st` command a few times. Note that it takes a substantial amount of time, even for one step. If the go command (`g`) is issued, the program will just run. It cannot be interrupted from the keyboard, To reset the system, press and release the reset button on the EVB, and then hit the `Enter` key. A prompt should reappear in the `Debug` window.

3.3:9 The `EVB11` program has quite a few options. The `F10` key will bring up a help screen which summarizes them. You may also wish to look at the on-line documentation. Generally, things are similar to, but not identical with, the `SIM11` and `SIM11A` programs. Try a few commands.

To be able to supply inputs to Port C, to be able to strobe `STRA`, and to be able to read Port B, it is necessary to connect the EVB to the mini-lab. We next turn to this topic.

5.3.4 The Knight Electronics Mini-Lab 200

The *Knight Electronics Mini-Lab 200* is a small development station. Within the context of this course, it has three principal purposes:

1. It provides both digital and analog signals for connection to the ports and lines of the EVB.
2. It provides indicators for reading logic levels of signals on the ports and lines of the EVB.
3. It provides a small "breadboarding" area for wiring up simple circuits, which are then connected to the EVB.

For this experiment, the following components of the mini-lab are critical. Make sure that you can identify each.

Power switch The power switch for the mini-lab is a toggle switch located near the hinged top, above the LED's. It is the third switch from the right, just to the left of the pulser switches. Before proceeding, make sure that you know the location of this switch.

Eight level-switchable output lines, with toggle switches At the top of the unit, near the hinged cover, is a row of eight toggle switches, labelled `7` through `0`.

Below each switch is a white connector with four small wire holes. The level of the signal for each line may be set to `HIGH` (approximately 5 volts) or to `LOW` (zero volts) by positioning the toggle switch appropriately. In this experiment, these lines will be used to supply the input signal to Port C.

Eight input lines with LED display To the right of the output lines, below the power switch, is a row of eight LED's, mounted within a white connector with two small wire holes. The LED will light up when the voltage level at the connector is `HIGH` (approximately 5 volts). In this experiment, these lines will be used to display the output of Port B, as well as the value on `STRB`.

Two pulser lines To the right of the LED display are two pulser lines, each with two four-wire connectors, one labelled `1` and the other labelled `0`. The number indicates the "normal" level at which the connector is held, with `1` representing `HIGH` and `0` representing `LOW`. These lines behave exactly as do the level-switchable output lines, save that the toggle switches will stay in only one position; they must be held in the other, and will spring back when released. They are thus ideal for generating a pulse.

Ground connection There are white ground connectors labelled `GND`. **Whenever configuring a circuit, unless stated specifically to the contrary. it is always necessary to connect a ground connector on the mini-lab to the ground connector on the EVB (pin 1) of `P1`.**

Prototyping area The two large white plastic strips in the center of the mini-lab are two prototyping boards. They are attached to the station with Velcro (try removing and then replacing them!), but there is no electrical connection. When using the prototyping area, you must make all connections yourself. This prototyping area will not be used in this experiment.

In this section, only one simple experiment will be performed, to make sure that you understand the operation of the mini-lab.

3.4:1 Make sure that the mini-lab is switched off.

3.4:2 Run a wire from one of the eight level-switchable output lines to one of the eight input lines with LED display.

3.4:3 Switch on the mini-lab. Toggle the switch on the selected output line, and observe the behavior of the LED. This should confirm your intuition about how these input and output lines work.

3.4:4 Shut off the mini-lab, and remove the wire which was connected in step 2.

3.4:5 Connect one wire from `1` line of the leftmost pulser to one of the input lines, and connect another wire from the `0` line of the same pulser to different input line.

3.4:6 Switch on the mini-lab, and observe the level values as the pulser switch is toggled. Make sure that you understand what the level labels (`0` and `1`) on the pulser mean.

3.4:7 Shut off the mini-lab, and remove the two wires.

5.3.5 Interconnecting the Two Units

Having familiarized yourself somewhat with the components, it is now time to interconnect the two units, and to run the software. First, follow these steps to connect the EVB to the mini-lab.

3.5-A1 Make sure that the mini-lab power switch is in the `OFF` position.

3.5:A2 Connect the `GND` pin on connector `P1` of the EVB to a `GND` connector on the mini-lab. This step is extremely important; it establishes a reference point between voltage levels on the EVB and voltage levels on the mini-lab. Without it, your circuit will not work properly.

3.5:A3 Connect the eight lines of Port C on the `EVB` to the output lines of the mini-lab. More specifically, for $0 \leq i \leq 7$, connect the pin on connector `P1` labelled `PCi` to the output line labelled `i` on the mini-lab, (The output lines on the mini-lab are those with the toggle switches.)

3.5:A4 Connect lines `0` through `6` of Port B to input lines `0` through `6` of the mini-lab. More specifically, for $0 \leq i \leq 6$, connect the pin on `P1` labelled `PBi` to the input line labelled `i` on the mini-lab. (The input lines on the mini-lab are those with the LED's.)

3.5:A5 Connect the `STRB` pin of `P1` to input line `7` of the mini-lab.

3.5:A6 Connect the `STRA` pin of `P1` to a `0` connector of one of the pulsers on the mini-lab.

Next, a few minor software modifications are necessary. Modify your program as follows. If you are working in a group, the program of each group member should be so altered.

3.5:B1 Modify your program so that the panic temperature is `FFh`. This ensures that the program will not simulate a panic shut-down as various temperature inputs are tested.

3.5:B2 The delay between readings needs to be set to a "reasonable" value. Try `00100000h` as an initial value for the register pair `IY:IX`.

3.5:B3 Reassemble the program, and make sure that it at least runs on the simulator (*i.e.*, does not crash).

Finally, it is time to run the program on the EVB. Here are the steps to follow.

3.5:C1 Turn on the EVB. Reset the unit and and load the program. Set the `PC` to `C000h`.

3.5:C2 Turn on the mini-lab.

3.5:C3 Select a typical temperature (*e.g.*, $20°C$), and "program" it in binary on the toggle switches of the mini-lab. Remember that only the seven least-significant bits will be displayed on Port B.

3.5:C4 Run the program by typing `g` at the debug prompt. The Port C value should **not** appear at Port B, as displayed on the LED's. Now strobe `STRA` by momentarily toggling the appropriate switch, and watch the input value appear on the LED's. LED `7`, which displays the value of `STRB`, should also be low (not lit) at this point. If the LED's do not display these values, you have some "debugging" work to do.

3.5:C6 After a period of time (roughly 10 to 20 seconds), LED `7` should light up again, indicating that Port C is again ready for input. If this time is too long or too short, adjust the delay value in the program accordingly.

3.5:C5 Repeat step 4 several times, until you are satisfied that your program is working properly.

3.5:C6 During the time that `STRA` is low, apply several distinct inputs, by varying the values of the toggle switches on Port C, and then pulsing `STRA`. Determine which of the input.map values, if any, appears at Port B once `STRB` is re-asserted.

3.5:C7 Configure `EVB11` so that the value of `PIOC` (and in particular the `STAF` bit) is displayed in both the `MEMORY` window and in the `MEM2` window. (See the help documentation for `EVB11`, if necessary.) Start stepping through the program. At some point, pulse the `STRA` line. Keep a close eye on the value of the `STAF` bit, as it is displayed in both locations. What conclusions can you draw about the display of this value? If the value is not an accurate display of the true value, give a possible reason.

Repeat the above steps for the program of any partners in the group in which you are working.

5.4 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for the final program embodying all three parts identified in 2.2 through 2.4 above. Your programs code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which includes general observations and conclusions, as well as discussions of the questions raised in 3.5:C6 and 3.5:C7.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 6

Basic Time-Multiplexed Display Control

6.1 Purpose and General Plan

The principal purpose of this exercise is the following:

- To gain familiarity with the use of seven-segment LED displays in a microcontroller application.

Specifically, in this laboratory exercise, you will interface a two-digit seven-segment display unit to Port B of the microcontroller, and use this unit to display the temperature which is read from Port C.

6.2 Equipment

The following equipment will be used in this experiment. Before proceeding, make sure that your group has all of these items available.

1. One PC running the P&E software `IASM11`, `SIM11A`, `SIM11`, and `EVB11`.
2. One Motorola M68HC11EVB evaluation board, with power supply.
3. One Motorola *M68HC11EVB Evaluation Board User's Manual*.
4. One serial port cable, for connecting the evaluation board to the PC.
5. One Knight Electronics Mini-Lab 200 analog-digital lab station.
6. One two-digit seven-segment display unit.
7. At least 25 wires for interconnection purposes.

6.3 Pre-Lab Preparation

It will be to your distinct advantage to prepare in advance for this laboratory. In addition to reading through this handout, you should do the following.

1. Read Section 6.5, and pages 227-232 of Section 8.1 in the text by Kheir.
2. Implement the software described in Section 5.1 of this document, using either `SIM11A` or `SIM11` as a development platform,

6.4 Background — LED-Based Display Units

A *light-emitting diode* (*LED* hereafter) is a diode which emits light when forward biased with a sufficiently high voltage. In all likelihood, the panel indicator lights on your PC (*e.g.*, *Power*, *Turbo*, *etc.*,) use LED's. They look like tiny light bulbs. Many seven-segment displays also use LED's, with each segment containing one LED.

LED-based displays should not be confused with LCD-based displays (*LCD* is an acronym for *Liquid Crystal Display*), as their electrical properties are quite different. Unlike their LED-based counterparts, LCD displays do not emit light, and use much less power, which makes them ideal choices for digital wristwatches and the like. LED-based displays are popular in applications in systems which are not battery-powered and in which high-visibility is important. Examples include displays on appliances, and on clock radios. In this laboratory exercise, we will use an LED-based display.

The seven-segment displays which will be used in this exercise actually contain eight LED's, with the eighth used as a decimal point. A visual representation of such a display is shown in Figure 1. In this representation, each of the seven segments

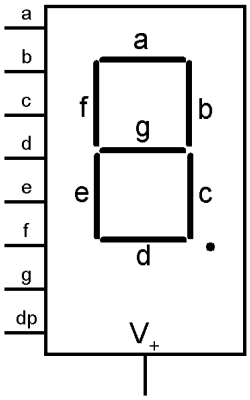


Figure 6.1: Visual representation of a seven-segment LED display.

is labelled with a letter in the range `a-g`. These letters have nothing to do with the hexadecimal representation of values; it is strictly a coincidence that the letters `a-f` are used both in hexadecimal representations and as segment labels. This labelling convention is fairly standard, and will be used throughout this exercise. The decimal point is identified by `dp`.

Typically, one just uses the diode symbol to denote an LED. To depict display consisting of many diodes (such as a seven-segment display), a group of diodes is used, as illustrated in Figure 2. This configuration is termed *common-anode* because the

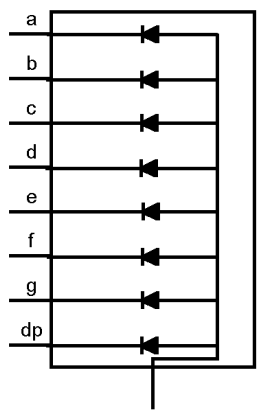


Figure 6.2: Common-anode configuration of a seven-segment LED display.

anodes of the diodes are tied together. A positive voltage is applied to this common point, and to light a given segment, the corresponding line is tied to logical zero (*i.e.*, to ground). A *common cathode* configuration is also possible, in which case the common cathodes are tied to ground, and a positive voltage must be applied to the lines of segments which are to be illuminated. (See Figure 6.10 in the text of Kheir.) The LED displays used in this exercise are in the common-anode configuration.

It is not feasible to connect the LED display unit directly to the output leads of the EVB; rather, some buffering electronics is necessary, for at least two reasons.

- The leads on the ports of the 68HC11 and 68HC24 may not be able to supply enough power to drive an LED.
- Since only one port (Port B) will be used for output, some means of switching use of that port between the two display units must be employed.

The circuit which will be used (and which is already wired on the boards containing the display units), is shown in Figure 3. The eight lines labelled $\overline{a}$ through $\overline{g}$, and $\overline{dp}$, are used to drive the appropriate segments of the LED's. The unit in the upper left, containing the eight triangle-shaped units, is a buffer unit. Logically, it provides

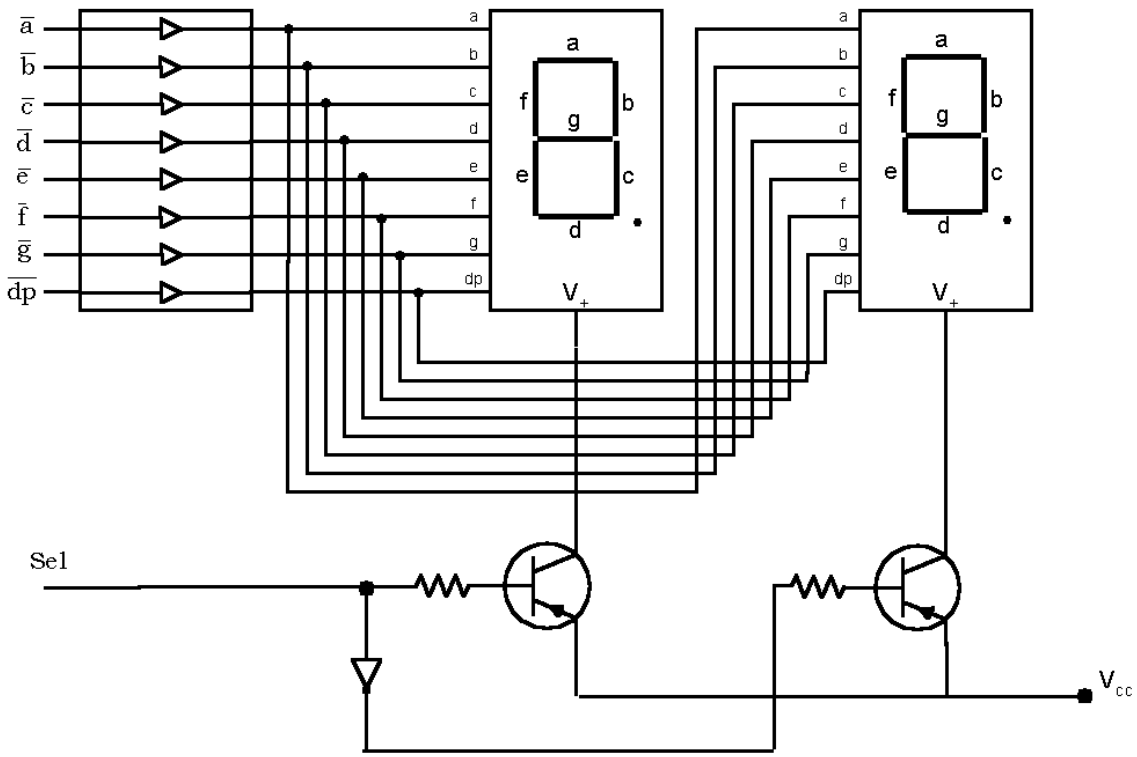


Figure 6.3: Laboratory unit configuration.

a direct connection from each input line to the corresponding output line. However, it also provides current amplification and isolation. The output can deliver a much larger current than is required to drive the inputs. Without this unit, the lines of Port B would not be able to deliver enough current to illuminate the segments.

The line labelled `Sel` is used to select which of the two display units is selected. When `Sel` is held low, the base of the left transistor is grounded, and is is driven into saturation, which places V_{cc} at the V_+ line of the leftmost display.[1] The `Sel` line is connected to the second transistor through an *inverter* (or *NOT-gate*). It converts logical one to logical zero, and conversely. Thus, if `Sel` is held low, a logical-high voltage (*i.e.*, +5 V.) is applied to the base of the rightmost transistor. Since the base-emitter voltage of this transistor is thus zero, it is in cutoff, and V_+ of the rightmost LED unit is near 0. This unit is thus disabled. If `Sel` is held high, the situation reverses, with the left transistor in cutoff, and the right transistor in saturation.

To drive both displays, time multiplexing is used. We alternate between selection

[1] Note that these are *npn* transistors, and not the *pnp* variety which were the focus in EE100. Thus, all voltage and current conventions are reversed.

of the leftmost and the rightmost display unit. If this alternation is performed rapidly enough, it will appear to the human eye that both units are always illuminated. This issue will be discussed further in Section 5.

6.5 Procedure

The procedure is broken into two parts, preliminary programming and hardware wiring. If at all possible, you should complete the preliminary programming portion before coming to the laboratory meeting.

6.5.1 Preliminary Programming

The programming portion of this exercise consists of performing several rather straightforward modifications to your program of Laboratory Exercise 5. You should perform this portion of the experiment individually, on your own program, and test it using either `SIM11A` or else `SIM11`. Remember that if you use `SIM11A`, you will have to load your program in the on-chip memory between addresses `00h` and `FFh`, inclusive. Then, when you transfer your program to the hardware and use `EVB11`, you will need to relocate it to use memory locations between `C000h` and `DFFFh`.

You and your partner must perform this task individually, on your own programs. Because this program will write to the LED display unit, and not to the LED's on the mini-lab, there are two additions which must be made to your program.

1. Conversion of the hexadecimal representation of the temperature to a pair of seven-segment encodings.
2. Writing of the representations to the display in multiplexed fashion, with a controllable delay between the writes for the leftmost digit and for the rightmost digit.

A high-level description of the core of the program is shown in Figure 4. As in the program for Laboratory Exercise 5, you should "disable" the panic stop routine by setting the panic stop temperature to `FFh`.

Let us now go through the new items of this procedure, one by one. The first line, specifying a read from Port C, is performed as in the previous program. No modification is necessary.

The next line stipulates that the temperature be converted to two decimal digits. The internal representation of the temperature is as a single unsigned number, represented in binary. The display unit consists of two LED units, each able to display one digit. It is the encoding for each digit, and not the binary representation of the entire number, which must be sent to the display unit. In this program, it will be assumed that the temperature value is between 0 and 99 inclusive, so that it may be

```
While true do
   Read temperature from Port C;
   Compute representation of temperature as two decimal digits;
   Write the leftmost digit to the display unit;
   Wait specified inter-write delay time;
   Write the rightmost digit to the display unit;
   Wait the specified inter-read delay time;
End while;
```

Figure 6.4: High-level algorithm for the program.

displayed fully using just two digits. It is remarkably easy to obtain these two digits. Just take the internal representation, and divide by 10. The quotient will be the leftmost digit, and the remainder will be the rightmost digit. The `IDIV` instruction does exactly what we want. Put the temperature in register `ACCB` and clear register `ACCA` (effectively placing the temperature, as a 16-bit quantity, in register `ACCD`). Next, place the decimal value 10 in register `IX`, and perform the `IDIV` instruction. The quotient (leftmost digit) will be in `IX`, and the remainder (rightmost digit) in `ACCD`. A template for a subroutine `Conv_temp` which realizes this process is shown in Figure 5. It is your

```
;;*********************************************************************
;;*********************************************************************
;; Subroutine which converts the temperature to decimal.
;; Input: temperature is in ACCA.
;; Output: leftmost digit is in IX.
;;         rightmost digit is in ACCD.
Conv_temp:
        <subroutine-body>
        rts
;;*********************************************************************
;;*********************************************************************
```

Figure 6.5: Template for the subroutine `Conv_temp`.

task to fill in `<subroutine-body>` with the appropriate code. Before proceeding with further programming, write and test this routine, and install it into your existing code.

We now turn to the task of converting a digit to the corresponding LED segment pattern. A table is used which contains the seven-segment pattern for each digit. For example, to display the digit `0`, all segments except `g` must be illuminated. This pattern is represented as a single seven-bit number `11111110`, with `a` corresponding to the leftmost bit, and `g` to the rightmost. However, since the configuration for the hardware is common-anode, a logical zero is used to illuminate the segment, and a

logical one otherwise. Thus, the actual pattern for display of a 0 is `0000001`. This may be represented in hexadecimal as `01`. The reprsentations for the digits `1` through `9` are obtained similarly, and are summarized in the table shown in Figure 6.

Decimal Digit	Bit a	b	c	d	e	f	g	Hex Repr.
0	0	0	0	0	0	0	1	01
1	1	0	0	1	1	1	1	4F
2	0	0	1	0	0	1	0	12
3	0	0	0	0	1	1	0	06
4	1	0	0	1	1	0	0	4C
5	0	1	0	0	1	0	0	24
6	0	1	0	0	0	0	0	20
7	0	0	0	1	1	1	1	0F
8	0	0	0	0	0	0	0	00
9	0	0	0	1	1	0	0	0C

Figure 6.6: Conversion table for a common-anode seven-segment LED display.

In the implementation, the lines $\overline{a}$ through $\overline{g}$ will be connected to lines `PB6` through `PB0`, respectively, of the EVB; that is, to the seven "lowest" lines of Port B, Line `PB7` will be connected to the line `Sel`. It is convenient to set up a table of values in the program; the table would appear as shown in Figure 7 below.

```
Seg_table:
      db $01,$4F,$12,$06,$4C,$24,$20,$0F,$00,$0C
```

Figure 6.7: Data table for driving a common-anode LED display unit.

To write a digit to the leftmost LED display, simply index into this table to obtain the correct pattern, and write it to Port B. Assuming that the digit is stored in register `IX`, the subroutine shown in Figure 8 will do the trick. It is assumed that register `ACCB` contains the `Sel` bit; its value is either `00h` (leftmost LED unit) or else `80h` (rightmost LED unit). Adding this value to the pattern from `Seg_table` yields the value to be written to Port B. Include this subroutine in your program.

Between writing the leftmost digit and the rightmost, your program should wait a specified amount of time, by using a delay loop. You should use the same delay program which you developed for Laboratory Exercise 5. Design your program so that this parameter is easily configurable, by altering one or two named memory locations.

```
;;*********************************************************************
;;*********************************************************************
;; Subroutine which writes one digit to the LED display.
;; Digit is in register X.
;; Mask is in register B.
Out_digit:
        ldaa seg_table,X   ;; Segment pattern for required digit.
        aba                ;; Set leftmost bit to identify display.
        staa PORTB
        rts
;;*********************************************************************
;;*********************************************************************
```

Figure 6.8: Writing a digit to an LED.

In conducting the hardware portion of this exercise, you will need to vary this parameter as part of an experiment.

Note that after you have computed the two digits, the leftmost will be exactly where you want it — in register `IX`. The rightmost digit will be in register `ACCD`, and surely you will need to use that register for other purposes. Thus, you will need to save this digit elsewhere. Of course, you may save it in (two) memory locations. However, there is a slicker way to save it. Just push `ACCB` and `ACCA` onto the stack. Then, when you need this digit, execute a `PULX` to place it into register `IX`.

Note also that your program will have *two* delays loops, one between writing the two digits to Port B, and one between reads of the temperature from Port C. The reason for the latter "inter-read" delay is to stabilize the display. If a new temperature is written out every millisecond, for example, the display might be unreadable if the temperature were to hover between two values, such as 19° and 20°. **As a starting point, set both delays to 0.**

6.5.2 Hardware Wiring

Once the program is working on the simulator, it is time to configure it for the hardware, and to install it. Begin by changing the `org` and stack location commands so that the program uses the on-board memory between `C000h` and `DFFFh`. Next, it is time to wire the hardware. Use the same connections as in Laboratory Exercise 5, except for Port B. Instead of connecting Port B to the mini-lab, connect it to the LED display unit. Shown in Figure 9 is the pin-out of the connector on the LED board. Note that the decimal point line is not available. Make the following connections.

c:1. `GND` to a `GND` on the mini-lab.

c:2. V_{cc} to a +5 V connection on the mini-lab.

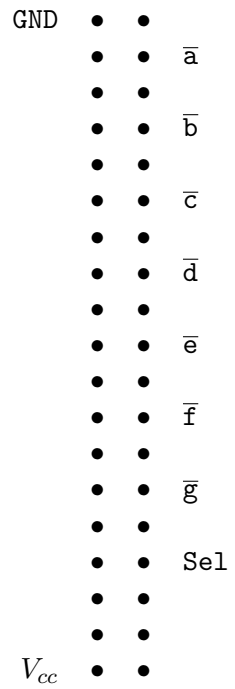


Figure 6.9: Pin-out of the LED board connector.

c:3. `Sel` to `PB7` on the EVB.

c:4. $\overline{a}$ to `PB6` on the EVB;

c:5. $\overline{b}$ to `PB6` on the EVB;

⋮ ⋮

c:10. $\overline{g}$ to `PB0` on the EVB.

Once you have made all connections, it is time to turn on the EVB and the mini-lab and test out the configuration. Test the system exactly as you did in Laboratory Exercise 5. Once you are satisfied that the LED unit is displaying the correct temperature, it is time to conduct an experiment.

Experiment: Vary the inter-write delay time, in order to determine the longest possible time which may be used without producing a noticeable flicker of the display. The time should be a rather small fraction of a second, so you will probably keep register `IX` at `0`, and just vary the value of `IY`. Based upon the values of `IX` and `IY`, as well as an analysis of the approximate time that one loop of your delay routine takes, compute the length of the delay.

In performing this experiment, it is not necessary to modify the source code and re-assemble each time. You may modify the value of register `IY` on the fly. At the debug prompt, just type its name and the new value. For example, to set the value of `IY` to `1000h`, type `IY 1000` at the "`>`" prompt.

6.6 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for the final program embodying all concepts identified in 5.1 above. Your programs code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which includes the observations of the delay-time experiment, as well as general observations and conclusions.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 7

Interrupts and Timer-Based Control

7.1 Purpose and General Plan

The principal purposes of this exercise are the following:

- To gain familiarity with the handling of maskable interrupts.
- To gain familiarity with the use of the programmable timer of the microprocessor.

Specifically, in this laboratory exercise, you will make two modifications to your program of Laboratory Exercise 6.

1. Detection of new data at Port C will be handled via interrupt, rather than via polling.
2. A delay loop will be implemented using the programmable timer. This will permit other computations to proceed while the delay elapses.

7.2 Equipment

The equipment to be used in this experiment is the same as that used in Laboratory Exercise 6. Before proceeding, make sure that your group has all of these items available.

1. One PC running the P&E software `IASM11`, `SIM11A`, `SIM11`, and `EVB11`.
2. One Motorola M68HC11EVB evaluation board, with power supply.
3. One Motorola *M68HC11EVB Evaluation Board User's Manual*.

4. One serial port cable, for connecting the evaluation board to the PC.
5. One Knight Electronics Mini-Lab 200 analog-digital lab station.
6. One two-digit seven-segment display unit.
7. At least 25 wires for interconnection purposes.

7.3 Pre-Lab Preparation

It will be to your distinct advantage to prepare in advance for this laboratory. In addition to reading through this handout, you should do the following reading assignments.

Interrupts: Read Section 5.5 of the text by Kheir, and Section 9.2 of the HC11 Technical Data Manual.

Programmable Timer: Read Section 5.8 of the text by Kheir, and Section 8,1 of the HC11 Technical Data Manual.

In addition, you should attempt the following.

Software: Implement the software described in Section 5.1 and Section 7.1 of this document, using either `SIM11A` or `SIM11` as a development platform,

7.4 Background — Interrupts

Some basic information on interrupts is summarized in this section. Consult the text and reference manuals for more details.

Maskable and non-maskable interrupts: Interrupts come in two basic flavors. A *non-maskable interrupt* is one which cannot be shut off regardless of any programming action. Examples include reset of the processor, and hardware failure indicators. Maskable interrupts are those which can be shut off, either locally or globally. The column labelled *CC Register Mask* of Table 9-2 of the Technical Data Manual indicates which is which; those with *None* as the value are non-maskable.

Global mask: All maskable interrupts may be disabled at once by setting the global mask. On the 68HC11, this mask is the `I` bit of the register `CCR`. When this bit is set, all maskable interrupts are disabled. The instruction `CLI` and `STI` clear and set this bit, respectively. When the processor is reset, the `I` bit is set, so that all maskable interrupts are disabled until the program enables them by clearing that bit.

Local mask: Associated with each maskable interrupt source is a bit called the *local mask*. This bit is in effect a switch, which may be used to turn off and on recognition of interrupts for the given source. For example, bit 6 of register `PIOC` is the local mask for interrupts to Port C. If this bit is `1`, latching new data to Port C

will generate an interrupt. If it is `0`, no interrupt will occur for this source regardless of any latches to that port. Note that if the global mask is set, all maskable interrupts are disabled regardless of the setting of the local flag. Also note that the setting for global masking is `1` of the `I` bit of `CCR`, while most local masks disable interrupts with a `0`. As a rule, upon resetting of the processor, local masks are set to disable the corresponding interrupt. There is no hard and fast rule regarding the location or exact use of local flags; for each interrupt source, the corresponding flag must be considered on an individual basis.

Local flag: Often, a maskable interrupt also has a *local flag* associated with it. When an interrupt occurs, this flag is set. Depending upon the particular source, it may or may not be the case that the program must clear this flag to avoid another interrupt. For Port C, the `STAF` flag of register `PIOC` is the local flag. As we know from previous laboratory exercises, this flag is set when new data are latched to Port C, regardless of whether or not interrupts are enabled for that source.

Pending interrupts: Even if an interrupt source is masked, an event which would generate an interrupt still generates a *pending interrupt*. When interrupts for that source are enabled, an interrupt is generated.

Priority: Different interrupt sources have different priorities. When a given interrupt is being serviced, and an interrupt of a higher priority source occurs, processing of the lower priority interrupt is suspended, and processing of the higher priority one begins. When the higher priority one has completed, processing of the lower priority interrupt resumes. See Section 9.2.4 of the Technical Data manual for more information, including how to alter the default priority.

Interrupt handlers and interrupt vectors: When an interrupt occurs, normal execution is suspended, and a special routine, called the *interrupt handler*, begins execution. The address at which the interrupt handler begins is stored in a special location, called the *interrupt vector*. Different interrupt sources typically have different interrupt vectors, although sometimes two sources will share the same vector. (For example, the external $\overline{\texttt{IRQ}}$ line and Port C share the same interrupt vector.) For the 68HC11, the interrupt vectors lie in the address range `FFC0h-FFFFh`. See Table 5.4 in Kheir, or Table 9-2 in the Technical Data manual, for information on the addresses of individual interrupt vectors.

Interrupt vectors and the EVB: On the evaluation board, it is not possible to write to the interrupt vector locations in the range `FFC0h-FFFFh`. Rather, these locations contain hardwired addresses in a *jump table*, located at addresses in the range `00C4-00FF`. Each jump-table entry is three bytes long, and must contain a `jmp` instruction to the interrupt handler. For example, the interrupt vector for $\overline{\texttt{IRQ}}$, which is `FFF2h-FFF3h`, contains the address `00EEh`, and is associated with the three-byte range `00EEh-00F0h`. The user is allowed to write to these three-byte blocks, and the strategy is to store a jump instruction to the interrupt handler there. See Figure 8 for a specific example connected with this exercise. For more complete information, consult Section 3.3, and

in particular the *Interrupt Vector Jump Table* which is either Table 3-1 or Table 3-2 of the *M68HC11EVB Evaluation Board User's Manual*, depending upon the version of the manual which you have.

Interrupts and the stack: When an interrupt occurs, the contents of registers `PC`, `ACCD`, `IX`, `IY`, and `CCR` are pushed onto the stack. When a return from interrupt is executed, these values are restored to the associated registers. Thus, any values assigned to these registers within an interrupt handler will be overwritten when the handler exits. For specific information on the order of insertion onto the stack, consult Figure 5-11 of Kheir, or Figure 9-4 of the Technical Data manual. It is important to note that nine bytes are pushed at each interrupt. In a program in which interrupts may be nested, make sure that enough stack space is reserved.

7.5 Procedure – Part 1: Basic Interrupt Handling

The procedure is broken into two parts, preliminary programming and hardware wiring. If at all possible, you should complete the preliminary programming portion before coming to the laboratory meeting.

7.5.1 Preliminary Programming

In this part of the programming task, you are to modify your program of Laboratory Exercise 6 so that it uses interrupts (rather than polling) to determine whether new data has been latched to Port C. A high-level description of the software is given in Figure 1. The top part, up to the `End while;`, constitutes a description of the main

```
Enable interrupts for Port C;
While true do
   Compute and display temperature;
End while;

When interrupt for Port C
   Suspend execution of compute and display temperature;
   Read new temperature from Port C;
   Resume execution of compute and display temperature;
End when;
```

Figure 7.1: High-level representation for the program.

program. Note that the main loop does not do anything to get the temperature from Port C; it just computes and displays the temperature endlessly! When an interrupt at Port C occurs, the actions shown in the `When` block are executed.

The representation is no longer one of a simple sequential program, to be executed from top to bottom. Such a representation is not appropriate for the description of interrupt handlers. The `When` block is an auxiliary routine, which is invoked when an interrupt occurs.

Now let us turn to the issue of programming this algorithm in 68HC11 assembly language. Figure 2 shows the code necessary to enable interrupts. Notice that two steps

```
;;*********************************************************************
PIOC         equ  $1002 ;; Parallel I/O control register.
PIOC_config  equ  $53   ;; STAF responds on low-to-high; interrupt driven.
;;*********************************************************************
;;     ...
            cli                    ;; Enable global interrupts.
;;     ...
            ldaa   #PIOC_config
            staa   PIOC            ;; Enable interrupts for parallel I/O.
;;*********************************************************************
```

Figure 7.2: Enabling interrupts.

must be taken. First, global (maskable) interrupts must be enabled by clearing the `I` flag of register `CCR`. Second, interrupts for the parallel I/O registers must be specifically enabled, by setting bit `6` (`STAI`) of `PIOC`. Read the information on configuration of `PIOC` on page 4-4 of the Technical Data Manual to verify and understand why the configuration given for is appropriate.

The body of the while loop, which computes and displays the temperature, is almost the same as that used in Laboratory Exercise 6. The only difference is that the part which polls the `STAF` bit of register `PIOC` is removed, since this function is assumed by the interrupt handling mechanism. The high-level description is shown in Figure 3; compare it to that of Figure 4 of Laboratory Exercise 6.

```
        Compute representation of temperature as two decimal digits;
        Write the leftmost digit to the display unit;
        Wait specified inter-write delay time;
        Write the rightmost digit to the display unit;
        Wait the specified inter-read delay time;
```

Figure 7.3: Summary of compute and display temperature.

The code for a bare-bones interrupt service routine is shown in Figure 4. This routine does only two things. First, it branches to a subroutine which reads the temperature. The same subroutine which worked for Laboratory Exercise 6 should work

```
;;*********************************************************************
;; Service the STAF interrupt.
Svc_STAF_int:
        bsr   Read_Temp
        rti
;;*********************************************************************
```

Figure 7.4: Bare-bones interrupt service routine.

here. Then, it executes the instruction `rti`, a *return from interrupt*. The `rti` instruction bears the same relationship to an interrupt handler that `rts` instruction does to a subroutine. When an `rti` instruction is executed, the register values (which were pushed onto the stack when the interrupt occurred) are restored, and execution resumes, in the program, at the point it was at when the interrupt occurred.

In practice, a "safeguard" check is usually placed in interrupt handling routines, as shown in Figure 5. This check ensures that the appropriate interrupt has really

```
;;*********************************************************************
PIOC         equ  $1002 ;; Parallel I/O control register.
STAF_bit     equ  $80   ;; Interrupt flag bit in PIOC register.
;;*********************************************************************
;;*********************************************************************
;; Service the STAF interrupt.
Svc_STAF_int:
        ldx   #PIOC
        brclr 0,X,STAF_bit,Rti_int
        bsr   Read_Temp
Rti_int:
        rti
;;*********************************************************************
```

Figure 7.5: Interrupt service routine.

occurred before processing it, since glitches can occasionally cause incorrect interrupt calls. In the code shown, the `brclr` instruction checks to see whether a `STAF` interrupt has really occurred. If so, it proceeds normally. If not, it simply returns. In a real system, a more elaborate error-handling routine might be invoked in the case of an erroneous interrupt. You should incorporate the routine of Figure 5 into your program.

The format of the `brclr` instruction, and a number of its relatives, is is shown in Figure 6. Since these instructions are extremely useful in testing and setting bits (*e.g.*, flags), they are widely used in interrupt handling routines. The `bset` and `bclr` instructions are used to *alter* specific bits within a byte, while the instructions `brset` and `brclr` are used to `test` specific bits within a byte, and branch conditionally, based

```
;;*******************************************************************
      bset  <offset>,<index_reg>,<bit_mask>
      bclr  <offset>,<index_reg>,<bit_mask>
      brset <offset>,<index_reg>,<bit_mask>,<go_to_address>
      brclr <offset>,<index_reg>,<bit_mask>,<go_to_address>
;;*******************************************************************
```

Figure 7.6: Syntax of bit-test instructions.

upon the result of that test.

In all four cases, the first two fields identify the address of the byte to be considered, using the conventions of the indexed addressing mode. the `<index_register>` must be `X` or `Y`, and the `<offset>` must be a single-byte value (between `00h` and `FFh` inclusive). The enhanced addressing mode is not possible with these instructions. (The direct mode is, although it is seldom of use in end-user programs.)

The `<bit_mask>` identifies the fields which are to be examined and/or altered. For example, with a bit mask of `42h`, which is `01000010` in binary, only bit `6` and bit `1` will be tested and/or altered. (Recall that bits are numbered `7` to `0`, left to right.) The `bset` instruction sets (to `1`) those bits of the addressed byte for which `<bit_mask>` contains a `1`. Similarly, the `bclr` instruction clears (to `0`) those bits of the addressed byte for which `<bit_mask>` contains a `1`. Other bits of the addressed byte are unaltered.

The instructions `brset` and `brclr` do not modify the addressed byte. Rather, if the bits identified by the bit mask are set (in the case of `brset`) or clear (in the case of `brclr`), the program counter is set to `<go_to_address>`, thus effecting a branch. The `<go_to_address>` is in fact a relative offset, so this instruction can only be used to branch to locations which are offset from it by no more than 127 bytes.

Armed with this information, take a look at the `brclr` in Figure 5, and convince yourself that it properly checks to see whether the `STAF` bit is set.

Because all registers (except CPU core registers) are found in a cluster beginning at memory location `1000h`, an alternative referencing convention is often used for indexed addressing , which is illustrated in Figure 7. The base of all registers (`1000h`) is given a name (`Reg_base`) in this case, and the various registers are then identified by their offsets. Which of the two representation conventions is most convenient often depends upon the particular code segment. Thus, it is most convenient to have `equ` statements identifying both the absolute address and the relative offset from the base, for each register. Remember that `equ` directives are statements of abbreviations which are used by the assembler. They are not assembled, and do not increase the size of a program. Use them liberally.

There is one further issue which has not been addressed. Namely, the system must be told where the interrupt handling routine is located. As described in Section 4, this is done via interrupt vectors, and things must be handled differently, depending

```
;;*********************************************************************
Reg_base     equ  $1000
PIOC_offset  equ  $0002
STAF_bit     equ  $40   ;; Interrupt-flag bit in PIOC register.
;;*********************************************************************
;;*********************************************************************
;; Service the STAF interrupt.
Svc_STAF_int:
        ldx   #Reg_base
        brclr PIOC_offset,X,STAF_bit,Rti_int
        bsr   Read_Temp
Rti_int:
        rti
;;*********************************************************************
```

Figure 7.7: Alternate interrupt service routine.

upon whether one is using the simulator or an actual EVB. Both versions are shown in Figure 8. Include this code in your program, and comment out the part which is not used. It is important to understand how the interrupt-vector addresses were

```
;;*********************************************************************
;; Interrupt vector definitions.
;; Use only one!!!  Comment out the one which is not used.
;;*********************************************************************
;; Interrupt vectors for simulator:
        org $FFF2
        dw  Svc_STAF_int
;;*********************************************************************
;; Interrupt vectors for EVB:
        org $00EE
        jmp Svc_STAF_int
;;*********************************************************************
```

Figure 7.8: Interrupt vector identification.

determined. As indicated in Table 9-2 of the Technical Data Manual, `FFF2h-FFF3h` is the vector address for interrupt $\overline{\texttt{IRQ}}$.[1] For the EVB, the *Interrupt Vector Jump Table* (Table 3-1 or 3-2 of the EVB User's Manual, depending upon the version), identifies the address of the associated jump table entry. Make sure that you understand how to look up these values.

[1] Parallel-port I/O shares an interrupt vector with the $\overline{\texttt{IRQ}}$ pin of the processor. An interrupt for this vector may be effected either by pulling the $\overline{\texttt{IRQ}}$ pin to logical zero, or else by latching data to a port. It is necessary to examine local flags to determine exactly which condition generated the interrupt.

7.5.2 Parallel I/O Interrupts and SIM11A

At this point, you should have the interrupt-driven version of your program ready for testing. It would hopefully be the case that it could be tested exactly as in Laboratory Exercise 6. When new data are latched at Port C and the STAF flag is set, if the STAI bit is also set, an interrupt should be generated. Alas, SIM11A does not handle such interrupts correctly. There are two solutions to this problem. The first is to use SIM11, which does handle them correctly, and which lets you use the address space C000h-DFFFh as well. Those students who have purchased the P&E Microcomputer Systems software will be given SIM11 for free, upon request.

The second solution is to employ a workaround for SIM11A. To understand the workaround, it is first necessary to understand what SIM11A does not do correctly. When the flag STAF is set, and the bit STAI in PIOC is also set, driving STRA from 0 to 1 is supposed to generate an interrupt on the associated vector. However, in SIM11A it does not. Fortunately, SIM11A does generate an interrupt in the case that the $\overline{\text{IRQ}}$ line is pulled low, and since $\overline{\text{IRQ}}$ and parallel I/O share the same interrupt vector, it is possible to simulate a parallel I/O interrupt using this line. The first step is to ensure that the $\overline{\text{IRQ}}$ line is configured to be *edge-sensitive* to interrupts; this means that an interrupt occurs as the result of a *transition* from high-to-low, rather than as a result of the *level* of $\overline{\text{IRQ}}$ being low. The appropriate code is shown in Figure 9. Include this

```
;;*******************************************************************
OPTION       equ  $1039  ;; Location of the configuration options register.
IRQE_bit     equ  $30    ;; IRQ interrupt mode bit for OPTION register.
;;*******************************************************************
        ldx  #OPTION
        bset 0,X,IRQE_bit ;; Edge-sensitive operation of IRQ for simulator.
;;*******************************************************************
```

Figure 7.9: Configuring $\overline{\text{IRQ}}$ for edge-sensitive operation.

code in the initialization section of your program. For an explanation of why it works, consult 9.1.5 of the Technical Data Manual.

Debugger commands for a simulated input of the temperature 25° are shown in Figure 10. The first three lines are identical to a simulated input in the polled version. The last two lines, involving the irq command, send the simulated $\overline{\text{IRQ}}$ line first high, and then low, simulating a falling edge. This sequence should be indistinguishable, to the program, from a properly generated interrupt at Port C. In using this approach, make sure that you configure $\overline{\text{IRQ}}$ for edge-sensitive operation; otherwise, the interrupt will not be cleared and as soon as the interrupt handler is exited, a new interrupt will be sensed.

Remember that if you are using SIM11, the irq commands are unnecessary, since

```
>inputc !25
>stra 0
>stra 1
>irq 1
>irq 0
```

Figure 7.10: Simulation of a Port C data latch, with interrupts.

the `stra` commands will effect the interrupt.

It should also be noted that both `SIM11` and `SIM11A` will complain and refuse to proceed if any needed registers and/or memory locations are uninitialized. In particular, the registers which are pushed onto the stack at an interrupt must be initialized. It is easy to do this. For example, in `SIM11A`, the command `IY 0` at the debug prompt will initialize register `IY` to `0`. With `SIM11`, use the command `Y 0`. This is not an issue with the EVB, since all registers and memory locations contain some value upon startup, even though this value may be meaningless.

7.5.3 Hardware Wiring and Experiments

The hardware wiring is *identical* to that of Laboratory Exercise 6. There is absolutely *no* difference. Test your program, making sure that it still responds faithfully to input settings, and that the inter-write delay still works properly. (Check for LED display flicker at the appropriate delay settings.)

7.6 Background – The Programmable Timer

The M68HC11 contains a multi-featured programmable timer. In this document, only information specifically relevant to the laboratory exercise will be discussed. For details, consult Chapter 8 of the Technical Data Manual, Addresses of registers are found in Table 3.1 of that same manual.

<u>The counter:</u> The core of the programmable timer is a 16-bit free-running counter. It counts endlessly from 0 to 65535; its current value is found in the 16-bit register `TCNT`. It is reset to `0` when the processor resets, and except in very special circumstances which are not used in ordinary programming, it is not possible to set its value otherwise.

<u>Counter frequency:</u> The frequency of the counter is a submultiple of the internal clock frequency (of 2 MHz.); the divisor of the internal clock frequency may be 1, 2, 4, or 16, depending upon the setting of bits `PR1` and `PR0` in register `TMSK2`. See 8.1.12 of the Technical Data Manual for details. In this experiment, the divisor will be set to one. This means that the counter cycles through its values every 0.5μsec. $\times$ 65536 = 32.768msec.

Counter overflow interrupt: Every time that the counter value goes from `FFFFh` to `0000h`, the timer overflow flag (`TOF`) is set. This flag is bit `7` in register `TFLG2`. If the timer overflow interrupt enable flag (bit `7` of register `TMSK2`), then an interrupt will be generated at overflow as well.

Output compare: The 68HC11 has five *output compare registers*, named `TOC1` through `TOC5`. These are writeable 16-bit registers, into which values may be stored. Bit `OCiI` ($1 \leq \mathtt{i} \leq 5$), in register `TMSK1` is a local mask for interrupts for output compare on `TOCi`; when bit `OCiI` is set, an interrupt will be generated when the value of the counter equals the value in register `TOCi`. The flag `OCiF` in register `TFLG1` is the corresponding flag register.

7.7 Procedure – Part 2: Using the Programmable Timer

In Laboratory Exercise 5, a data re-acquisition delay was implemented, so that once a temperature was read from Port C, another reading could not be processed until the delay had elapsed. Such a delay is necessary to prevent the display from rapidly switching from one value to another, resulting in an unreadable blur.

It was possible to implement this delay as a simple time-wasting loop in Laboratory Exercise 5 because the "display" was a row of simple LED's on the mini-lab. In Laboratory Exercise 6, in which the display was a pair of seven-segment LED's, each displaying one digit of the temperature, this delay had to be disabled. The reason is that the two seven-segment displays were time multiplexed, with a single port driving both of them. Were the processor in such a simple delay loop, the time-multiplexing would be suspended, and at least one of the seven-segment displays would be dark.

In this exercise, this delay will be reintroduced, with the aid of the programmable timer. The idea is very simple. The timer runs in the background, and generates an interrupt when the necessary time has elapsed. While the timer is running, the program can continue to drive the display.

7.7.1 Preliminary Programming

While programming of this timer-based delay loop is not difficult, it does requires substantial attention to detail. The overall strategy, as well as some of the details, will be described in this section. It will be up to you to fill in the remaining details. In constructing your program, you may need to refer to the Technical Data Manual for information on register addresses, bit locations within registers, and the like.

It is strongly suggested that you save a copy of the program which you developed for Section 5, and make the modifications described in this section on a separate copy.

Figure 11 shows the high-level representation for this program. The main program

```
Enable interrupts for Port C;
;; Note: Programmable timer interrupts are initially disabled.
While true do
   Compute and display temperature;
End while;

When interrupt for Port C
   Suspend execution of compute and display temperature;
   Disable further interrupts for Port C;
   Read new temperature from Port C;
   Initialize gross delay counter;
   If gross delay is nonzero
     then
       Enable interrupts for Programmable Timer;
     else
       Enable interrupts for Port C;
   End if;
   Resume execution of compute and display temperature;
 End when;

When interrupt for Programmable Timer
   Decrement gross delay counter;
   If gross delay counter = 0
      then
        Disable interrupts for the Programmable Timer;
        Enable interrupts for Port C;
   End if;
End when;
```

Figure 7.11: High-level representation for the program with delay counter.

is identical to that developed in Section 5.1. However, there are now two interrupt handlers; one for interrupts of Port C, and one for timer interrupts. The key feature to notice is that the handler for Port C disables further interrupts for that port before enabling timer interrupts. Interrupts for Port C are re-enabled only after the delay loop has run its course, and timer interrupts are disabled. Thus, at most one form of interrupt is unmasked at any one time. Remember that while the timer-based delay loop is running, we do not want to read new values from Port C. This is why its interrupts are disabled.

To augment your program, the first step which you should take is to augment the interrupt vector table (see Figure 8) of this program to include the timer interrupts. The format is shown in Figure 12. You need to replace the `????`'s in this code with the appropriate values. We will be using Timer Output Compare 1. Use Table 9-2 of the Technical Data Manual to determine the address of the interrupt vector, and use the *Interrupt Vector Jump Table* (Table 3-1 or Table 3-2, depending upon the version) of the M68HC11EVB Evaluation Board User's Manual to determine the jump table

```
;;*******************************************************************
;; Interrupt vector definitions.
;; Use only one!!!  Comment out the one which is not used.
;;*******************************************************************
;; Interrupt vectors for simulator:
        org $????
        dw  Svc_OC1_int
        org $FFF2
        dw  Svc_STAF_int
;;*******************************************************************
;; Interrupt vectors for EVB:
        org $????
        jmp Svc_OC1_int
        org $00EE
        jmp Svc_STAF_int
;;*******************************************************************
```

Figure 7.12: Interrupt vector identification.

entry. If you are doing this preparation outside of the laboratory and do not have the latter manual at hand, just leave the address to be filled in later. Call the interrupt service routine for Timer Output Compare 1 interrupts `Svc_OC1_int`.

The next task is to augment the handler of Figure 5 (or Figure 7) to manage the additional tasks identified in the first `When` block of Figure 11. This augmentation is shown in Figure 13. The first part of the code, up to and including the `bsr Read_temp`,

```
;;*******************************************************************
STAF_bit     equ  $80   ;; Interrupt-flag bit in PIOC register.
STAI_bit     equ  $40   ;; Interrupt-enable bit in PIOC register.
;;*******************************************************************
;;*******************************************************************
;; Service STAF interrupt.
Svc_STAF_int:
        ldx   #Base_Reg
        brclr PIOC_Offset,STAF_bit,STAF_false ;; Test for STAI enabled.
        bclr  PIOC_Offset,X,STAI_bit  ;; If so, disable STAI interrupts.
        bsr   Read_Temp               ;; process the new temperature,
        ldab  #delay_1
        bsr   Delay_init              ;; and begin a delay count.
STAF_false:
        rti
;;*******************************************************************
```

Figure 7.13: STAF interrupt service routine with timer-based delay.

should be easily understood in the light of Figure 5 and the high-level pseudo-code of Figure 11. Next, a delay count is loaded into register `ACCB`. This will be the number of

times the that timer will cycle through its 65536 values during the delay loop. Then, a subroutine named `Delay_init` is invoked to complete the realization of the first `When` block of Figure 11. This subroutine, with some details omitted, is shown in Figure 14. There are two lines which you need to expand. The current timer value is found in

```
;;*********************************************************************
;;*********************************************************************
Delay_cnt:
        ds  1       ;; Delay counter for delay routine.
;;*********************************************************************
;; Subroutine which initiates an interrupt-based delay
;; for a given period of time.
;; Number of 65536-tick blocks of delay passed as a parameter in ACCB.
Delay_init:
        tstb
        beq  No_delay   ;; Do nothing if delay is 0.
        stab Delay_cnt  ;; Save delay count in Delay_cnt.
;;      <Store current timer count in register TOC1>;
;;      <Enable interrupts on Output Compare 1> (use bset);
        rts
No_delay:
;;      <Enable STAI interrupts>;
        rts
;;*********************************************************************
```

Figure 7.14: Subroutine to initiate timer-based delay.

register `TCNT`; this value needs to be copied to `TOC1`. Remember that these are 16-bit values. To enable interrupts on Output Compare 1, use a `bset` instruction which sets the `OC1I` bit of register `TMSK1`.

The final task is to write the routine which services timer interrupts, corresponding to the second `When` block of Figure 11. A partial realization is shown in Figure 15. Realization of all of these command is straightforward, and follows patterns already described. There is one tricky part, however. To *clear* the `OC1F` bit of `TFLG1`, you have to issue an instruction which *sets* it, such as a `bset`. (See 8.1.11 of the Technical Data Manual.) Remember that this bit is not a "writeable" value in the usual sense, and that such flags have strange ways of wanting to be cleared. Compare this to the technique for clearing the `STAF` bit of `PIOC`, which requires a pair of reads on particular registers. As a general rule, always examine the manual carefully for nonstandard requirements for clearing interrupt flags.

7.7.2 Hardware Wiring and Experiments

Again, the wiring is *identical* to that of Laboratory Exercise 6.

```
;;*********************************************************************
;; Service OC1 interrupt.
Svc_OC1_int:
;;      <Exit if there is no Output Compare 1 interrupt>;
;;      <Decrement Delay_cnt>;
;;      <If Delay_cnt = 0
;;         then
;;           disable OC1 interrupts  (use bclr);
;;           re-enable STAI interrupts (use bset);
;;       End if;>
;;      <Clear the OC1F bit of TFLG1> ;; Use bset!!!!!!!!
OC1F_false:
        rti
;;*********************************************************************
```

Figure 7.15: Output Compare 1 interrupt service routine with timer-based delay.

Once you have your program working, use a wristwatch to determine the maximum delay time that can be obtained with `Delay_cnt` initialized to `FFh`. Change the temperature once, then change it again, and note the amount of time that it takes, from the point of the first change, for the second one to appear. Compare this to the theoretical delay time, which is determined by multiplying the clock-cycle time (0.5μsec.) times the number of cycles in one loop (2^{16}) times the number of loops (255). Do they agree within the resolution that you can obtain by looking at your watch?

7.8 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for two final programs, one embodying all concepts identified in 5.1 above, and a second adding the concepts identified in 7.1. Your programs code should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which includes the observations of the delay-time experiment, as well as general observations and conclusions.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 8

Basic Stepper Motor Control

8.1 Purpose and General Plan

The principal purposes of this exercise are the following:

- To gain familiarity with the properties of stepper motors.
- To gain familiarity with the generation of waveforms with a microcontroller, and the use of these waveforms to control a stepper motor.

8.2 Equipment

The equipment to be used in this experiment is the same as that used in Laboratory Exercise 6. Before proceeding, make sure that your group has all of these items available.

1. One PC running the P&E software `IASM11`, `SIM11A`, `SIM11`, and `EVB11`.
2. One Motorola M68HC11EVB evaluation board, with power supply.
3. One Motorola *M68HC11EVB Evaluation Board User's Manual*.
4. One serial port cable, for connecting the evaluation board to the PC.
5. One Knight Electronics Mini-Lab 200 analog-digital lab station.
6. One stepper motor.
7. One stepper-motor control parts kit, consisting of an MC3479P stepper motor controller chip, one 4.3 V. Zener diode, and one 47KΩ resistor,
8. Various wires for interconnection purposes.
9. One digital voltmeter.

10. One oscilloscope, with two coaxial probes..

8.3 Pre-Lab Preparation

It will be to your distinct advantage to prepare in advance for this laboratory. In addition to reading through this handout, you should do the following reading assignments.

Elementary stepper motors: Read the excerpt from the North American Phillips Controls Corp. *Stepper Motor Handbook*. You may safely gloss over the formulas concerning torque and ramping, but you should familiarize yourself with the main ideas in the other sections.

Stepper motor control: Read Section 7.6 of the text by Kheir.

Output compare: Read the material on pages 132-134 of Kheir on output compare, particularly Program 9. Also re-read the material in Section 8.1 of the Technical Data manual on output compare.

Interrupt Prioritization: Read the material in 9.2.4 and 9.2.5 of the Technical Data manual, which covers this topic. This material is also summarized on pages 104-105 of Kheir.

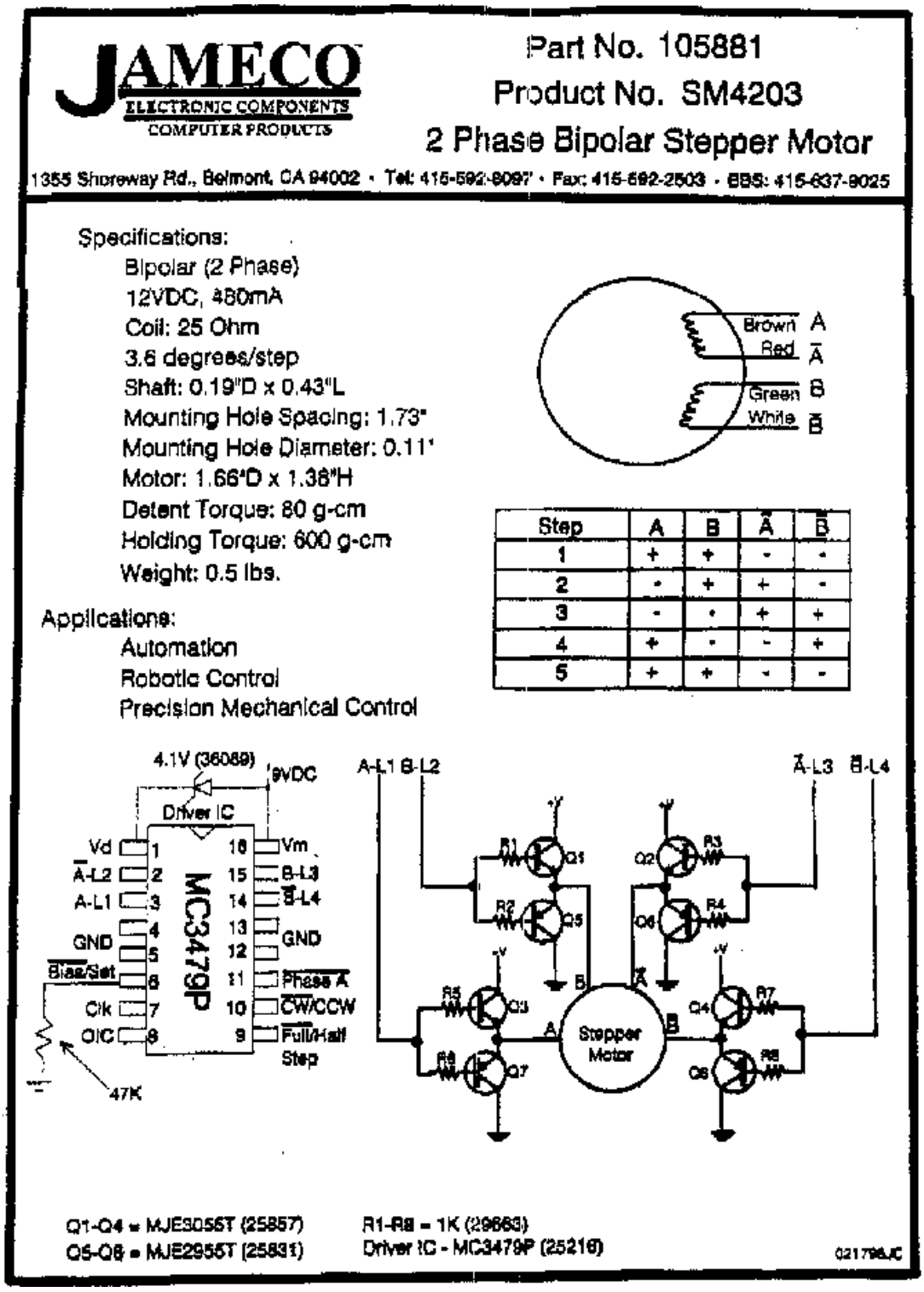
In addition, you should attempt the following.

Software: Implement the software described in Section 5.1 of this document, using either `SIM11A` or `SIM11` as a development platform,

8.4 Procedure – Part 1: Stepper Motor Configuration

The stepper motor to be used in this experiment is a two-phase bipolar unit from Jameco Electronics. The documentation for this motor is reproduced in Figure 1. The Motorola MC3479P stepper-motor driver chip will also be used as a smart interface to the motor, eliminating the need to generate the raw stepping signals within the microcontroller software. The procedure which is to be followed in this part of the exercise is given below, in a series of ten steps.

1. Familiarize yourself with the prototyping area. The prototyping area of the mini-lab consists of two beige platforms with wiring holes, into which wires and pins of chips may be inserted. They are situated in the center of the mini-lab, and are attached with velcro. They are not connected electrically to the mini-lab in any way. Pull one off and replace it, in order to convince yourself of this.



JAMECO
ELECTRONIC COMPONENTS
COMPUTER PRODUCTS

Part No. 105881
Product No. SM4203
2 Phase Bipolar Stepper Motor

1355 Shoreway Rd., Belmont, CA 94002 • Tel: 415-592-8097 • Fax: 415-592-2503 • BBS: 415-637-9025

Specifications:
- Bipolar (2 Phase)
- 12VDC, 480mA
- Coil: 25 Ohm
- 3.6 degrees/step
- Shaft: 0.19"D x 0.43"L
- Mounting Hole Spacing: 1.73"
- Mounting Hole Diameter: 0.11"
- Motor: 1.66"D x 1.38"H
- Detent Torque: 80 g-cm
- Holding Torque: 600 g-cm
- Weight: 0.5 lbs.

Applications:
- Automation
- Robotic Control
- Precision Mechanical Control

Step	A	B	Ā	B̄
1	+	+	-	-
2	-	+	+	-
3	-	-	+	+
4	+	-	-	+
5	+	+	-	-

Q1-Q4 = MJE3055T (25857)
Q5-Q8 = MJE2955T (25831)
R1-R8 = 1K (29663)
Driver IC - MC3479P (25216)
021796JC

Figure 8.1: Stepper motor documentation.

Each platform consists of a 14 (row) × 63 (column) array of holes. At the outside of each platform are two rows of 63 holes each, labelled `A` and `B`. All of the holes in each such row are connected to each other. However, elements in different rows (even when those rows are labelled by the same letter) are not connected. On the inside of the strips are two groups of five rows each. One group of rows is labelled `a` through `e`, and the other `f` through `j`. In each group, each set of five holes within a given column are connected to each other. Thus, for each column, elements `a` through `e` are connected to each other, and elements `f` through `j` are connected to each other. However, elements `e` and `f` are **not** connected. Elements in distinct columns of these inner rows are not connected.

Use the voltmeter and a pair of wires to test the connectivity of a platform, making sure that you understand these conventions.

2. Wire the circuit shown in Figure 2. Figure 2 shows the connections which are to be made to the MC3479P chip. This circuit corresponds fairly closely to that shown on

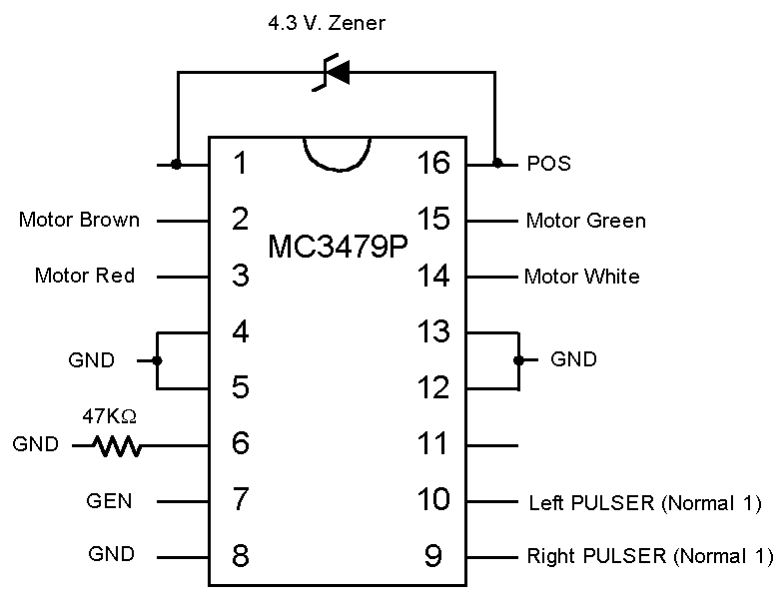


Figure 8.2: Wiring configuration for the stepper-motor driver.

the documentation of Figure 1. The major difference is that the four dual-transistor "H" servo driver circuits connected to the stepper motor in Figure 1 have been omitted. Rather, the leads of the stepper motor are connected directly to the driver chip. The purpose of these circuits is to increase the power available to drive the motor. For the level of power required in this experiment, the chip itself can supply sufficient power, and so the simpler configuration is adequate. For a real servo application, however,

one would almost certainly include such amplifier circuits.

Before you begin wiring, make sure that your mini-lab is shut off and unplugged. **Never wire a live circuit!!** While the voltage levels involved in this experiment are not dangerous, it is very easy to destroy sensitive components with incorrect wiring, even if the connection is made only for an instant.

Begin by installing the chip into the prototyping area, Install the chip so that one row of pins is in row `e` of the prototyping platform, and the other in row `f`. The holes in the prototyping platform are spaced correctly for such dual in-line integrated-circuit packages.

The pins on the chip are numbered `1` through `16`, going counterclockwise from the upper left. The top of the chip is identified by a special mark, which is usually a small notch in the case of the chip. This notch is depicted by a semicircle on the illustration of Figure 2.

The wiring will be done with sections of solid wire, stripped at each end. There should be plenty available, but if not, a spool of wire and cutters will be provided. The notations in capital letters (*e.g.*, `GND`, `GEN`, `POS`, and `PULSER`) refer to connectors on the mini-lab.

It is a good idea to tie all of the `GND` connections together on the prototyping platform, and then to run a single wire to a `GND` connector on the mini-lab.

Pins `9` and `10` should be connected to *separate* pulser switches. Each should be connected to the strip of the pulser labelled `1`, so that the untoggled logic level supplied is one.

The band on the Zener diode identifies the *cathode*; *i.e.*, the end with the zig-zag line pointed to by the triangle. (The left end of the diode in Figure 2.) Make sure that you have installed it correctly; if you reverse the connection and apply power, it is possible that the integrated circuit will be destroyed by the reverse voltage from the motor coils. The resistor is of course symmetric; it may be installed either way.

Pin `11` is provides a special output signal which will not be used, and may be left unconnected.

3. Disconnect the power lead Disconnect the lead connecting pin `16` to the `POS` connector of the mini-lab. Disconnect the end at `POS`, leaving the end which is connected to the prototyping platform and the integrated circuit connected. This ensures that, when the mini-lab is energized, no power will be delivered to the prototyping area.

4. Set `POS` to 9 V. Connect the common end of a digital multimeter to the `GND` connection of the mini-lab, and connect the voltage lead to `POS`. (You may need to use some ingenuity to make a good connection. Consult your laboratory instructor if you need help.) Set the multimeter to read voltage. After ensuring that you have performed step 3 above correctly, plug in and energize the mini-lab. Adjust the leftmost

potentiometer at the bottom of the lab (it is labelled for the range from `0` to `+18`) until the voltmeter reads approximately 9 volts. This value is not critical, but it should not be high by more than a volt or so. The higher it is, the more current the entire circuit must support, and so the hotter it will run.

5. Shut down, and reconnect the power lead. Switch the mini-lab off, and reconnect the power lead to the prototyping area.

6. Have the laboratory instructor check your circuit. Before proceeding, have the laboratory instructor check you wiring.

7. Configure for square waves. Set the multiplier knob in the upper left to 100, set the `ADJUST` knob to `1` (fully counterclockwise), and set the waveform knob just below it to square wave (fully counterclockwise).

7. Verify operation of the stepper motor. Turn on the power to the mini-lab. Slowly turn the the `ADJUST` knob clockwise; the motor should begin to turn and increase in speed as adjust is turned. If the motor does not turn, switch off the power immediately and check your wiring.

Notice that, as the frequency increases, so does motor speed, However, there is a maximum frequency, beyond which the motor will first start to stutter, and finally, with a slight further increase, stop. At this point, internal friction and moment of inertia within the motor prevent faster rotation.

Toggle the `PULSER` switch which is connected to pin `10`, and note that the motor reverses direction.

Toggle the `PULSER` switch which is connected to pin `9`, and notice that the motor doubles in speed.

8. Connect the oscilloscope. To analyze more systematically the relationship between input frequency and motor speed, it is necessary to measure both. We will use an oscilloscope to observe and to measure the input signal. While it is possible to measure the speed of the motor shaft with the appropriate sensors, we will use a much cruder measure of simply counting the number of revolutions over a given period of time.

Turn off the mini-lab, and connect an oscilloscope to measure the frequency of the signal at `GEN`. Use channel 1 for the signal display. You will find the triggering to be more stable if you use a separate trigger lead, and set the scope for external trigger. The red trigger lead should also be connected to `GEN`. Both black leads on the scope probes should be connected to `GND`.

Turn the power for the mini-lab back on, and observe the waveform on the screen. **There are several models of oscilloscopes, each with its own peculiarities. If you have trouble getting the scope to work, ask your laboratory instructor for assistance.**

9. Determine the relationship between frequency and speed. The relationship between input signal frequency and motor shaft speed should be linear. In this part of the experiment, you will verify the extent to which this is the case. You are to complete, experimentally, (a copy of) the table to the left in Figure 3, and then compute the values for the table to the right. In addition, prepare a plot of signal frequency vs. motor speed, for both half-step and full-step operation.

Recorded Data				
Signal Period (msec.)	**Motor Speed** Half-step Rev.	Half-step Sec.	Full-step Rev.	Full-step Sec.
1				
2				
5				
10				
15				
20				

Computed Data			
Signal Period (msec.)	**Signal Freq.** (KHz.)	**Motor Speed** (rpm) Half-step	Full-step
1			
2			
5			
10			
15			
20			

Figure 8.3: Prototype data tables for the frequency-speed relationship.

10. Determine the critical frequencies. In addition to the above measurements, determine the maximum input frequency which may be applied with the motor running smoothly. At a sufficiently high frequency, the motor will be unable to keep up with the signal, and will start to vibrate and miss steps. Record both this frequency and the unit number of the motor, which is written on the casing. (Because of mechanical differences, this critical frequency will be different for different motors, even of the same manufacturer and model. Usually, the specifications will give a minimum such frequency.)

8.5 Procedure – Part 2: Using the EVB to Control a Stepper Motor

8.5.1 Development of the Software

Before you arrive at the laboratory, you should write and test the program for this assignment, using `SIM11A` and/or `SIM11`. At this point in the course, you should have acquired a reasonable amount of competence in programming microcontroller applications on the 68HC11. Therefore, this section will provide somewhat less detail on how to implement the program than have previous laboratory exercise descriptions.

Make sure that you think through your software design, and that you have covered the usual routine tasks.

The primary function of the software for this assignment is to generate a square wave, which will be used to drive the MC3479 chip, which will in turn drive the stepper motor. The timer output compare unit of the processor, which was used in Laboratory Exercise 7 to implement the delay, will be used in this exercise as well. It is possible to enable an output compare unit so that whenever an interrupt occurs for that unit, the logic level of a pin on the processor is toggled. If the interrupts occur at regular intervals, the output will thus be a square wave!

To permit easy adjustment of the frequency of the wave, a coefficient which represents a multiple of the wavelength will be read from Port C. When a new value is read and latched at this port, the speed of the motor will change.

A high-level description of the algorithm for this exercise is shown in Figure 4.

```
Set Output Compare as a higher priority interrupt than Port C;
Enable interrupts for Port C;
Enable interrupts for Output Compare 2;
Initialize Wavelength_coefficient;
Start first half-cycle of the output wave;
While true do
    Nothing;  ;; Wait for interrupts!!
End while;

When interrupt for Port C
    Read new Wavelength_coefficient from Port C;
End when;

When interrupt for Output Compare 2
   Toggle the wave output line;
   Initiate a new half cycle;
End when;
```

Figure 8.4: High-level representation for the program.

The first five lines of the main program are initialization commands; we will return to those shortly. For now, look at the body of the program, which consists of a single while loop which does nothing but wait for an interrupt! The entire program is interrupt-driven. Whenever a timer interrupt occurs, the level of the output wave is toggled, and whenever a Port C interrupt occurs, a new frequency for the wave is computed and used.

The interrupt handler for Port C should be familiar by now. This handout will not provide any information on how to implement it, because you have already done it in Laboratory Exercise 7. All you need do is to port the appropriate code from that exercise to this one. Remember that this imported code must include specification of

the interrupt vector and jump table, as well as initial enabling of interrupts.

In this exercise, Timer Output Compare 2 will be used. Actually, any of the five timer output compare lines could be used for either this or the previous exercise. We use a different line than that used in Laboratory Exercise 7 for the sake of variety. The first step in configuring for use of this interrupt is to declare the interrupt vector and jump table entry. Consult Table 9-2 of the Technical Data manual, and the *Interrupt Vector Jump Table* (Table 3-1 or 3-2, depending upon the version of the manual) of the EVB user's manual for the correct addresses to use.

The next step is to initialize properly for interrupt handling. A skeleton specification is provided in Figure 5. First of all, as in Laboratory Exercise, we want the timer

```
;;*********************************************************************
;;  <set the OC2F bit of register TFLG1>; Enable output compare 2.
;;  <set the OL2 bit of register TCTL1> ;;Configure timer output-compare 2.
;;*********************************************************************
;;  <Enable interrupts on OC2> ;; Similar to OC1 case of previous lab.
;;*********************************************************************
```

Figure 8.5: Initialization for Output Compare 2.

to generate an interrupt whenever its value matches that of the appropriate output compare register. In this case, it is register `TOC2` which will be used, so we must set the `OC2F` bit of register `TFLG1`. Consult 8.1.11 of the Technical Data manual for details, including the location of the `OC2F` bit. Next, it is necessary to configure the Output Compare 2 unit so that its output pin (which is pin `PA6`)[1] is toggled each time that a successful compare occurs (between the timer value and the value in register TOC2). Upon consulting 8.1.9 of the Technical Data manual, it is seen that bit `OL2` of register `TCTL1` is to be set, and the rest of the bits are to be cleared. (Verify that this is correct.)

Now let us look at realization of the interrupt-service routine. A skeleton is shown in Figure 6. It is important to understand that the output pin `PA6` is toggled each time that the contents of register `TOC2` matches the timer value. It is not necessary to service the interrupt for this to happen. However, interrupt service is necessary in order to set the timer output compare register to the appropriate new value for the next interrupt. This function is performed by a subroutine named `Do_half_cycle`, which is sketched in Figure 7. The key idea is that the `Wavelength_coefficient`, which is the value read in from Port C, contains the eight most significant bits of the 16-bit value which is the number of clock cycles between toggles of the output line. Thus, if `Half_cycle` = 10,

[1] This provides a convenient means of observing the "square wave" generated by a program run on `SIM11A` or `SIM11`. Just configure the simulator to display the value of Port A, which is at location `1000h`. Bit 6 (second from the left) should toggle between `0` and `1` as the program runs. Of course, the frequency of the simulator will be much less than that of the real processor. Expect a slowdown on the order of 10^3.

```
;;*********************************************************************
;;*********************************************************************
;; Service TOC2 interrupt.
Svc_TOC2_int:
;;      <Exit if no OC2 interrupt occurred>;; Check OC2F bit of TFLG1.
        bsr   Do_half_cycle
;;      <Clear the OC2F bit of TFLG1>;;   Use bset!!!
        rti
;;*********************************************************************
```

Figure 8.6: Routine to service the TOC2 interrupt.

```
;;*********************************************************************
;; Subroutine which configures the timer for one-half cycle.
Do_half_cycle:
;;        <Half_cycle := Wavelength_coefficient * 256>
;;        <TOC2 := TCNT + Half_cycle>
        rts
;;*********************************************************************
```

Figure 8.7: Routine to initiate a half-cycle.

there will be $10 \times 256 \times 0.5\mu$sec = 1.28msec per half cycle, and so the frequency of the wave will be $1/(2 \times 1.28\text{msec/cycle}) = 391$Hz.

Notice that `Do_half_cycle` must be invoked once at the beginning of the program, as well as each time that a timer interrupt occurs.

To loop endlessly waiting for interrupts, the appropriate instruction to use is a branch to the current address, as shown in Figure 8, It is not appropriate to use the

```
;;*********************************************************************
         bra   $
;;*********************************************************************
```

Figure 8.8: An endless loop which waits for interrupts.

`wai` command, because once an interrupt is processed, the instruction after the `wai` will be processed. With the `bra $`, execution will return to the branch instruction upon completion of interrupt service.

There is one subtle point which we have not discussed. In this program, unlike that of Laboratory Exercise 7, two different interrupts are enabled at the same time. Thus, it is possible, for example, that a Timer Output Compare 2 interrupt will occur while an interrupt for Port C is being serviced. The question thus arises as to what

happens in such a situation. Is the interrupt for Port C processed to completion before the interrupt for Timer Output Compare 2 is serviced, or is processing of the Port C interrupt temporarily suspended in favor of the interrupt for Timer Output Compare 2? The answer is that there is a priority list for interrupts, and those with a lower listing will be suspended in favor of those with a higher listing, should a conflict occur. Sections 9.2.4 and 9.2.5, and particularly Table 9-6, of the Technical Data manual, provide detailed information. Table 5.1 of Kheir provides the same information. As a power-up and reset default, the entry labelled Reserved (Default to $\overline{\texttt{IRQ}}$) is given top priority. The pecking order then proceeds down to the bottom of the table and then back to the top and down to the entry just before the one of highest priority. The lowest priority interrupt is thus SCI Serial System.

This priority system may be altered by modifying the rightmost four bits (the right *nibble*) of register `HPRIO`. To make a given interrupt source that of highest priority, one places the bit pattern found in Table 9-6 of the Technical Data manual for that interrupt into the register. Shown in Figure 9 is the code necessary to make Timer Output Compare 2 the interrupt of highest priority. The appropriate bit pattern for

```
;;*********************************************************************

HPRIO        equ  $103C ;; Highest priority I interrupt register.
HPRIO_clear  equ $03    ;; Priority bits to clear,
HPRIO_set    equ $0C    ;; Priority bits to set.
;;*********************************************************************
        ldx  #HPRIO
        bset 0,X,HPRIO_set   ;; Set interrupt priority pattern
        bclr 0,X,HPRIO_clear ;; Timer output compare is highest.
;;*********************************************************************
```

Figure 8.9: Making TOC2 the interrupt of highest priority.

the right nibble of `HPRIO` is `1100`. Because the leftmost nibble contains settings which need not concern us, it is best to leave those bits alone. The code thus sets and clears only the bits in the right nibble. Make sure that you understand how it works.

It is important to understand that this setting may be changed only when global interrupts are masked (*i.e.*, the `I` bit of `CCR` is set). Typically, priorities are only altered in this fashion in the initialization phase of a program, before interrupts are enabled.

In the program for this exercise, it is desirable in principle to have the timer interrupt have a higher priority than the Port C interrupt. It is clearly undesirable for processing of data from Port C to delay proper generation of the square wave. However, since transition of the output pin occurs upon output compare match, and not upon processing of the interrupt, the wave will be generated properly provided that the timer interrupt can be serviced before the next transition of the output wave must be generated. Even with `Wavelength_coefficient` = 1, this time will

be $256 \times 0.5\mu$sec $= 128\mu$sec, which is loads of time for the microprocessor. Since the data to Port C are latched via a toggle switch, it is impossible to generate more than a few interrupts per second, due to mechanical limitations of the switch, so there will be plenty of time to process all interrupts. Nonetheless, to simulate a situation in which Port C could be pounded relentlessly with new data, you should install the prioritization code given above into your program, and you should definitely understand what it is doing.

8.5.2 Hardware Wiring and Experiments

Once the software has been developed and tested on the simulator, it is time to connect the EVB to the stepper motor circuit, and let the microcontroller control the motor. Here is the procedure to follow.

1. Connect `PA6`: Begin by disconnecting the lead from `GEN` to pin `7` of the MC3479, and connect pin `7` to pin `PA6` of the `EVB`. Also connect the ground pin of the EVB to `GND` of the mini-lab. This will enable you to use the EVB, rather than the mini-lab, as the source of the driving square wave to the stepper.

2. Connect Port C to the mini-lab Connect the eight pins of Port C to the eight switchable output lines of the mini-lab. just as you did in previous experiments.

3. Reconnect the oscilloscope: Connect both the channel `1` input and the trigger input of the oscilloscope to pin `7` of the MC3479, and set the scope to trigger externally. This will allow the waveform generated by the microcontroller to be displayed on the scope.

4. Test the circuit: Run the program, and observe the waveform and motor speed. If all is well, the motor should run, with lower values at Port C corresponding to faster speeds. Note that very low settings (typically three or less) may result in frequencies which will stall or stutter motor operation.

5. Frequency measurement: For several switch settings for Port C input, measure the frequency by observing the pattern on the oscilloscope. Draw up tables of the form shown in Figure 10, and compute the appropriate values. Plot both of the computed values from the second table as a function of the Port C input value, and comment on how well they agree. Note that since measurement of motor speed is not an issue here, quite accurate readings may be obtained.

For low values at Port C, the motor may stutter or lock. To avoid damaging it, you should disconnect the power connection to pin `16` of the MC3479 when making those measurements.

Recorded Data	
Port C Input Value	**Signal Period** (msec.)
1	
2	
4	
8	
16	
32	
64	
128	

Computed Data		
Port C Input Value (msec.)	**Signal Freq. from Port C Value** (KHz.)	**Signal Freq. from Measured Signal Period** (KHz.)
1		
2		
4		
8		
16		
32		
64		
128		

Figure 8.10: Prototype data tables for the Port C to frequency relationship.

8.6 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of a program source listing (`.asm` file) and an assembler listing (`.lst` file) for a final program embodying all concepts identified in 5.1 above. The code of your program should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which includes the data for the tables, the plots, and the answers for the questions associated with items 9 and 10 of Section 4, as well as for item 5 of Section 5.1. Make sure that your write-up includes the number of the stepper motor which you used. In addition, you should provide general observations and conclusions.

Since you should retain your copy of this laboratory handout for study purposes, it is best to use a copy of the tables of Figures 3 and 10, rather than the pages from this document.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Laboratory Exercise 9

Analog-to-Digital Conversion

9.1 Purpose and General Plan

The principal purposes of this exercise are the following:

- To gain familiarity with the use of the analog-to-digital converter on the MC68HC11.
- To implement a simple analog motor-speed control.

The experimental process contains two major steps.

1. In the first step, a simple digital voltmeter will be implemented and tested. A voltage will be applied to the A/D converter of the MC68HC11, and a number proportional to that voltage will be displayed, via Port B, on the LED's of the mini-lab. A potentiometer on the mini-lab will be used to control the voltage input to the A/D converter.
2. In the second step, the result of the first step will be combined with the configuration of Laboratory Experiment 8. In that experiment, the motor speed was set and altered via switches on the mini-lab which were connected to Port C. In this experiment, the input from Port C will be replaced by input via the A/D converter. Thus, the motor speed will be controlled by turning the knob of a potentiometer, rather than by discrete switch settings.

At this point in the course, you should have gained considerable expertise in developing programs for the MC68HC11. Therefore, this handout contains relatively little in the way of low-level details for the programs to be developed.

9.2 Equipment

The equipment to be used in this experiment is the similar to that used in Laboratory Exercise 8, with the addition of a single new resistor. Before proceeding, make sure

that your group has all of these items available.

1. One PC running the P&E software `IASM11`, `SIM11A`, `SIM11`, and `EVB11`.
2. One Motorola M68HC11EVB evaluation board, with power supply.
3. One Motorola *M68HC11EVB Evaluation Board User's Manual*.
4. One serial port cable, for connecting the evaluation board to the PC.
5. One Knight Electronics Mini-Lab 200 analog-digital lab station.
6. One stepper motor.
7. One stepper-motor control parts kit, consisting of an MC3479P stepper motor controller chip, one 4.3 V. Zener diode, and one 47KΩ resistor.
8. One resistor of approximately 1KΩ.
8. Various wires for interconnection purposes.
9. One digital voltmeter.
10. One oscilloscope, with two coaxial probes..

9.3 Pre-Lab Preparation

It will be to your distinct advantage to prepare in advance for this laboratory. In addition to reading through this handout, you should do the following reading assignments.

Analog-to-Digital Conversion: Read Section 5.9 of the text by Kheir, and Chapter 7 of the Technical Data manual for the MC68HC11.

In addition, you should attempt the following.

Software: Implement the software described in Sections 4.1 and 5.1 of this document, using either `SIM11A` or `SIM11` as a development platform,

9.4 Procedure – Part 1: Basic Analog-to-Digital conversion

9.4.1 Development of the Software

The software for this part of the experiment is extremely simple. As shown in Figure 1, all that the program does is to perform an analog-to-digital (A/D) conversion on the input to the A/D converter, and write the result of that conversion to Port B. You may be glad to learn that there are no interrupts or interrupt handlers in this program! The

```
Configure the A/D converter;
While true do
  Initiate an A/D conversion;
  Wait for the conversion to complete;
  Write the result to Port B;
End while;
```

Figure 9.1: High-level sketch for the first program.

only new task which must be mastered to realize this software is that of configuring and using the A/D converter on the MC68HC11.

The MC68HC11 has a built-in A/D converter, so there is no need for an external A/D chip, as there would be with many other microcontrollers. Actually, the MC68HC11 A/D converter has eight input channels; each channel is associated with a single pin on the processor. Input to channel `0` is on pin `AN0`, channel `1` on pin `AN1`, and so forth, with input for channel `7` on pin `AN7`. The `ANi` pins are shared with the pins of Port E, and so on the EVB, pin `ANi` is accessed by connecting to pin `PEi`.

In configuration of the A/D converter, there are two key registers: `OPTION` and `ADCTL`. Only the two leftmost bits of register `OPTION` are relevant to A/D converter operation. The leftmost bit, `ADPU`, turns the converter off and on. When it is set to `0` (the power-up default), A/D conversion is off; when it is set to `1`, A/D conversion is enabled. Bit `6`, the `CSEL` bit, is used to select the clock which is used to drive the A/D converter. For all work in this experiment, it is safe to leave this bit set to `0`, which is the power-up default. See Section 7.8 of the Technical Data manual for more information. Thus, the only bit of `OPTION` which you need to alter is `ADPU`.

The register `ADCTL` is the A/D status and control register, and is the key register for configuration and control of the A/D converter. Thus, it is necessary to understand the rôles of the bits of this register in some detail. The overall bit naming pattern is shown in Figure 2.

CCF	-	SCAN	MULT	CD	CC	CB	CA

Figure 9.2: Register `ADCTL ($1030)`.

In addition, there are four eight-bit A/D result registers, named `ADR1`, `ADR2`, `ADR3`, and `ADR4`. The A/D converter on the MC68HC11 is an 8-bit unit, and the the results of A/D conversions are placed in these registers.

With the A/D converter enabled (`ADPU` bit set), the conversion process is initiated whenever register `ADCTL` is written. Exactly which type of conversion process takes

place depends upon the settings of bits in `ADCTL`.

Number of channels: The A/D converter may be configured either to perform four consecutive conversions on a single channel, or else to perform four simultaneous conversions on four different channels. (Here, a *channel* is one of the eight input lines `ANi`). In either case, the results of these scans are placed into registers `ADR1` through `ADR4`. In this experiment, only single-channel conversion will be used, with the result of the first scan will be placed in `ADR1`, the second in `ADR2`, and so forth. The single-scan mode is selected by setting bit `SCAN` to `0`.

Channel selection: The channel which is scanned in single-channel mode is identified by the settings of bits `CD-CA`. Bit `CD` must always be `0`, and bits `CC` through `CA` must contain the channel number, in binary. For example, to scan channel 5 (*i.e.*, input line `AN5`), the bit pattern `0101` is used. Consult Table 7-1 of the Technical Data manual for more detailed information.

Type of scan: There are two types of scanning, continuous and single-cycle. In continuous scanning, A/D conversions proceed continuously, filling the `ADRi` registers in round-robin fashion. In single-cycle scanning, only one set of four conversions is taken, and the A/D converter then awaits a command to initiate a new scan. The type of scan is controlled by the `SCAN` bit. If it is set to `0`, single-cycle scanning is enabled. In this experiment, single-cycle scanning will be used.

Initiation and detection: When the A/D converter has been enabled by setting the `ADPU` bit of the register `OPTION`, a new scan is initiated whenever the register `ADCTL` is written. To detect that a scan has completed, the `CCF` bit may be examined. This bit will be reset by the controller whenever any bit of register `ADCTL` is written. There is no facility to generate an interrupt upon completion of a successful scan. However, it is ensured that a new scan will complete every 32μsec. so as an alternative to polling `CCF`, one may delay a sufficient amount of time and then read the appropriate `ADRi` register. It takes 128μsec. to complete one cycle of four scans. In single-scan mode, unless the input signal is changing extremely rapidly, all four values (`ADR1` through `ADR4`) will be very close, and your program may use just one (*e.g.*, `ADR1`).

With these ideas in hand, it is a simple matter to convert Figure 1 into a working assembly-language program. A slightly expanded version of Figure 1 is show in Figure 3. It is up to you to select the input channel. In principle, it does not matter which one you choose. However, it is possible that the EVB which you use will have one or two non-operational channels, so it is in your best interests to design your program so that the channel may be altered easily. More will be said about channel selection later, in the section on hardware wiring.

```
Turn on A/D conversion (use bset on register OPTION};
While true do
  Configure and initiate conversion of the A/D converter for:
        Single-channel, single-scan, channel ANi,
     by writing to ADCTL;
  Wait for the conversion to complete by polling CCF;
  Write the result to Port B;
End while;
```

Figure 9.3: Expanded high-level representation for the first program.

9.4.2 A/D Conversion and the Simulators

Both `SIM11A` and `SIM11` allow the user to enter values into the `ADRi` registers, to simulate the process of A/D conversion. Neither allows entry of an analog value which is to be converted, however.

In `SIM11A`, a 256-byte queue of "converted" A/D values is maintained. This queue is initially empty; values may be added with the `addi` commands at the debug prompt. Only one value may be added per command; thus, to enter four values, for separate commands must be issued as illustrated in Figure 4. If these commands are issued, and

```
>addi !11
>addi !22
>addi !33
>addi !44
```

Figure 9.4: Simulating A/D input with `SIM11A`.

then a single-cycle A/D conversion is initiated, upon completion of the conversions `11` will appear in register `ADR1`, `22` will appear in `ADR2`, `33` will appear in `ADR3`, and `44` will appear in `ADR4`. If only the first command (`addi !11`) is issued, `11` will be placed in register `ADR1`, with `ADR2-ADR4` left unfilled. Even if a new round of conversions is initiated and new command (*e.g.*, `addi !22`) is issued, the new value will be placed in register `ADR2`. This can be somewhat confusing. To ensure that you understand what the simulator is doing, it is strongly suggested that you declare `ADR1` through `ADR4` as variables to be displayed in the `Variables` window. Watch the values which each assumes as A/D conversions are performed. Typing the command `addi` with no arguments will pop up a window which shows the current values in the queue, and allows these values to be modified as well. An arrow points to the next element in the queue.

Despite the fact that the documentation terms this queue "round-robin," values are not re-used. New values must be entered to initiate new conversions. Apparently,

there is no way to select an A/D channel with `SIM11A`. The command `addc` clears the queue. A/D conversion will work in either single-chip or expanded mode, since the port replication chip is not involved. However, you will still need to use single-chip mode to access Port B.

Simulation of A/D conversion in `SIM11` is simpler in some respects. Figure 5 shows a debug command session. The commands `analog0` through `analog7` provide a sim-

```
>analog0 !11
Analog Inputs = 0B XX XX XX XX XX XX XX
>analog3 !44
Analog Inputs = 0B XX XX 2C XX XX XX XX
>analog
Analog Inputs = 0B XX XX 2C XX XX XX XX
```

Figure 9.5: Simulating A/D input with `SIM11`.

ulated converted value for the corresponding channel, and then displays the value for each of the eight channels. All four conversions on that channel will be simulated with that input value, so there is no need to enter more than one value, as there is with `SIM11A`. The command `analog` displays the current values for each channel, without altering any. The values of register `ADCTL`, as well as the four registers `ADR1` through `ADR4`, are shown in a green window just above the debug window.

9.4.3 Hardware Wiring and Experiments

The principal hardware interface characteristics of the A/D converter on the MC68HC11 are summarized in the two items below.

Input lines As mentioned previously, the input line for channel `i`, $0 \leq$ `i` ≤ 7, is pin `ANi` of the MC68HC11. Pin `ANi` coincides with pin `PEi`, the `i`th line of Port E, and it is the latter labelling which is used on the pin-out diagram of the connector on the EVB. The voltage level of the input is always taken to be relative to ground, the `GND` pin on the EVB connector.

Voltage reference points For use of the A/D converter on the MC68HC11, it is necessary to define the *low reference voltage* V_{RL} and the *high reference voltage* V_{RH}. There are pins on the MC68HC11, and on the EVB connector, for each. Both voltages are relative to ground. V_{RL} is the voltage which causes the A/D converter to deliver its minimum value, `00h`, while V_{RH} is the voltage which causes it to deliver its maximum value, `FFh`. It must be the case that $0 \leq V_{RL} < V_{RH} \leq 6$ volts. It is usually the case that the V_{RL} line is tied to

ground, so that $V_{RL} = 0$, although this need not be the case. It is also necessary that $V_{RH} - V_{RL} \geq 2.5$ volts.

A simple circuit which will be used in the experiment to deliver a user-controllable voltage to the A/D converter is shown in Figure 6. The resistor R_{Lim} is a current-

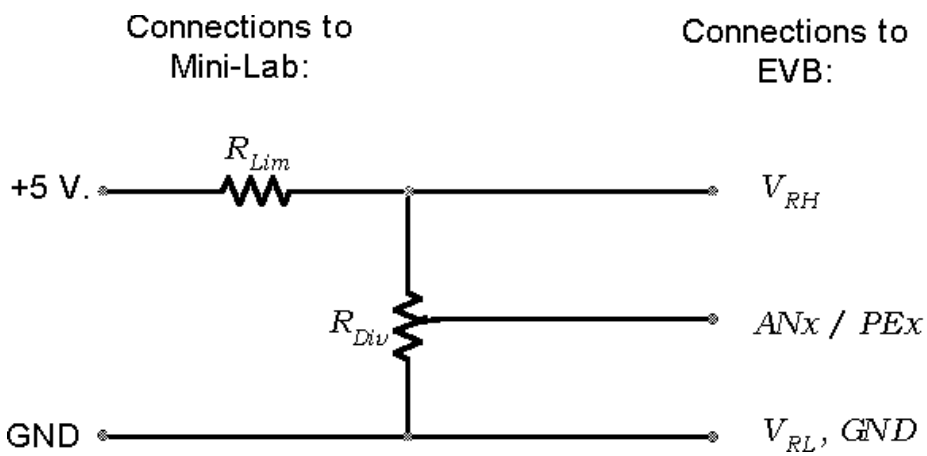


Figure 9.6: Voltage driver wiring.

limiting resistor, and will be taken to be approximately 1KΩ. (The exact value will be determined by equipment availability.) R_{Div} is the 1KΩ potentiometer[1] on the mini-lab. It is at the bottom of the unit, and is labelled `1K`. The connectors for it are to its immediate right.

The formal procedure for wiring the experiment is as follows.

1. Wire the circuit shown in Figure 6. Connect the lead labelled `ANi` / `PEi` to the pin corresponding to the A/D channel which you have selected for use in your program. For example, if you have selected channel three, connect it to pin `PE3`.

[1] A *potentiometer* is a resistor with a variable center tap. Analog volume controls on radios and (older) televisions are perhaps the most familiar uses of such devices. A knob is used to move the center tap along the fixed resistor; the full range of motion is perhaps 270° to 300°. The resistance rating is that which is measured between the two end connections. When the knob is fully counterclockwise, the resistance between the center tap and the left connection is 0, while the resistance between the center-tap and the right connection is the full resistance of the unit. As the knob is rotated clockwise the resistance between the left connection and the center tap increases, while the resistance between the right connection and the center tap decreases, with the sum of these two resistance remaining equal to the full resistance of the unit. The *taper*, *i.e.*, the plot of shaft angle versus resistance, may be either linear or logarithmic, depending upon the application.

2. Connect `GND` on the EVB to `GND` on the mini-lab.
3. Connect the eight Port B leads of the EVB, `PB0` through `PB7`, to the eight LED lines on the mini-lab.

Once you have wired things correctly, and the laboratory instructor has checked your wiring, it is time to test your program. Turn on the mini-lab, the EVB, and load and start your program. If all is working properly, the light pattern on the LED's should change as you turn the `1K` potentiometer knob. The value displayed should be `00h` when the knob is fully counterclockwise, and `FFh` when it is fully clockwise. If the reverse is the case, switch the connections to the end connectors of the potentiometer. The transition of values should be linear in the position of the potentiometer knob, indicating that it has a linear taper.

If things do not work, begin by verifying that the LED's are displaying the value which the EVB reports for Port B. Next, verify that the A/D registers of the EVB are recording the actual input values. Use the `MD` command to display their locations (`1031H` through `1034h`). Change the input voltage to the A/D converter, step the EVB so that at least 32 clock cycles have elapsed, and see if the value of `ADR1` has changed. After 128 clock cycles. all four registers should reflect approximately the same value. If all are showing a very small value (*e.g.*, `04h` or less) regardless of potentiometer setting, something is wrong. First make sure that the port which you are driving is the same one which you have selected with the value in bits `CD-CA` of register `ADCTL`. If you are reasonably certain that your program is operating properly, try using a different A/D channel. Move the wire to a new `PEi` pin, and change the value written to bits `CD-CA` of register `ADCTL` to reflect this new port selection. If you discover a defective port, please inform the laboratory instructor, so that a note may be made, and the equipment repaired.

Before proceeding, demonstrate the operation of your program to the laboratory instructor.

9.5 Procedure – Part 2: Analog Motor Speed Control

In this part of the experiment, the setting of the potentiometer will be used to control the speed of the stepper motor.

9.5.1 Development of the Software

In essence, you have already written all of the software for this part of the experiment. All that you need do is to take the program developed for Part 1 of this experiment,

and combine it with the software for motor-speed control which you wrote for Laboratory Exercise 8. Whereas in Laboratory Exercise 8 Port B and the toggle switches on the mini-lab were used to control the speed of the motor, in this experiment the value obtained from A/D conversion will be used. Thus, you will use the value in register `ADR1`, rather than `PORTCL`, as the wavelength coefficient. A high-level description of the process is summarized in Figure 7. Each line of this description, save one,

```
Enable interrupts for Output Compare 2;
Initialize Wavelength_coefficient;
Start first half-cycle of the output wave;
Turn on A/D conversion (use bset on register OPTION);
While true do
  Configure and initiate conversion of the A/D converter for:
        Single-channel, single-scan, channel ANi,
     by writing to ADCTL;
  Wait for the conversion to complete by checking CCF;
  Write the result to Port B;
  Complement the result (One's complement using coma);
  Write the result to Wavelength_coefficient;
End while;

When interrupt for Output Compare 2
   Toggle the wave output line;
   Initiate a new half cycle;
End when;
```

Figure 9.7: High-level representation for the second program.

should be familiar either from Figure 3 of this exercise, or else from Figure 4 of Laboratory Exercise 8. The only new line is the one which commands to complement the result to obtain the `Wavelength_coefficient`. This step is taken because a low value or `Wavelength_coefficient` corresponds to a high frequency, and hence a high motor speed. This complementation will ensure that higher values obtained via A/D conversion will correspond to higher motor speeds.

9.5.2 Hardware Wiring and Experiments

The hardware wiring for this part consists of a combination of the wiring for Part 1 of this experiment and part of the wiring for Part 2 of Laboratory Exercise 8. You should wire up everything from Laboratory Exercise 8 except the connections from Port C to the toggle switches (since the wavelength coefficient is now obtained via the A/D port.) The connections to the pulser switches may also be omitted.

Once you have made these connections, and your instructor has checked the wiring, it is time to test the configuration. The potentiometer should control the motor speed. You will note that most of the effective speed control lies in a range of a few degrees

of turn of the knob. In your write-up, explain why this is the case, and make some suggestions as to how this problem might be overcome.

In the course of debugging your work, you may find it useful to use the oscilloscope to observe the stepper-motor control signal generated by the EVB. However, you need not do this if things work properly.

9.6 Extra credit

If you are interested in extra credit, combine this program with that of Laboratory Exercise 7 so that the value read in from the A/D port is displayed on the 7-segment LED unit. Since the value lies between `00h` and `FFh`, you should display hex "digits." This will require you to extend the segment-pattern table for the LED's from ten to sixteen entries. Consult with the laboratory instructor before pursuing this option. It should be attempted only after you have completed each of the nine regular experiments.

9.7 Submission Requirements

Submission of this assignment requires two components, a written submission and a demonstration. The exercise is not considered to have been submitted until both components are completed.

Written submission: Your written submission must consist of program source listings (`.asm` files) and assembler listings (`.lst` files) for a programs embodying all concepts identified in 4.1 and 5.1 above. The code of your program should be clear and well documented. **Clarity is far more important than cleverness. You will not gain points for saving a few microseconds or a few bytes, but you will lose points if your code is difficult to follow.**

In addition to the program, you must submit a short write-up which includes an answer to the question posed at the end of 5.2.

Demonstration: The laboratory instructor will evaluate the operation of your programs by running them. He will let you know how this is to be done; by an in-person demo, or by submitting a diskette.

As usual, the cover sheet must also be submitted with your written component.

Part III

Appendices

Appendix A

P&E Software: Installation and Configuration Notes

In both the laboratory work and in some of the homework of EE101 and EE134, software from P&E Microcomputer Systems will be used. While this software is very well suited to the needs of these courses, it is unfortunately the case that it is distributed with very little documentation. The purpose of this document is to provide some basic information on how to install and configure this software on a PC, as well as how to access the documentation which is provided.

This document is not a tutorial on how to use the software. Information on how to develop programs using the software will be provided separately, as parts of the laboartory and homework handouts.

A.1 Components

All of the software runs on IBM-PC compatible computers. (See the next section for specific information on system requirements.) The software package consists of two main components, the *integrated development environment* (IASM11) and the *68HC11 simulator* (SIM11A).

IASM11 is a program which contains an editor (for writing programs), as well as a cross assembler which assembles 68HC11 programs and generates binary image files, the latter of which may be downloaded to either the simulator or the evaluation board for execution. The IASM11 program also contains a simple communications interface for the evaluation board, but this interface will not be used. (A more sophisticated interface, the P&E EVB interface, will be used instead.)

The 68HC11 simulator is a program which emulates the behavior of the 68HC11 microprocessor. With it, one is able to run and debug programs entirely on the PC, without the need for an actual 68HC11 processor. This simulator provides many features for the detailed study of the execution of a program, including the ability to step

through one instruction at a time, to examine registers and memory locations as the program executes, and even to modify memory locations.

The programs which are part of the distribution are the following.

`IASM11.EXE` This is the executable program which is the integrated development environment.

`IASMINST.EXE` This is the initialization program for `IASM`. This program is run to set certain operational parameters of `IASM`. It is discussed in more detail in Section 4.1.

`IASM11.HLP` This is the help file for the `IASM11` program. It is not a text file, but must be read by commands issued from within `IASM11`, or else by using the `PRHELP` utility.

`SIM11A.EXE` This is the executable program which is the 68HC11 simulator.

`SIM11A.HLP` This is the help file for the `SIM11A` program. It is not a text file, but must be read by commands issued from within `SIM11A`, or else by using the `PRHELP` utility.

`PRHELP.EXE` This program is used to generate a text file from the `.HLP` files, and to print it on the printer. Its use is discussed in Section 6.

`PRHELP.TXT` This file contains information on how to find out how to use the `PRHELP` utility.

`CASM.EXE` This is a command-line version of the assembler which is built into the `IASM11` utility. It will not be used explicitly in the course.

`README.ASM` This file contains a short description of the contents of the distribution diskette.

There is one other program from P&E Microcomputer Systems which will be used in the course laboratories, but which is not part of the student distribution. This program is useful only in conjunction with a Motorola evaluation board, and so not needed for work outside of the laboratory. The associated files are the following.

`EVB11.EXE` This is the executable program which is used to communicate with the Motorola Evaluation board.

`EVB11.DOC` This file contains some documentation for `EVB11`.

A.2 System Requirements

The software runs on IBM-PC compatibles. By current standards, it is exceptionally undemanding. It is DOS-based, which means that no Windows environment of any

form is required. It will run straight from the DOS command line. It was tested successfully on an old 80286-based machine with only 1 Mb. of memory. It will probably run on an 8086-based machine with 640 Kb. of memory, but it was not tested on such a platform.

The software occupies only about 650 Kb. of disk space, and will fit on one floppy disk. In its most basic modes, it requires no special installation, and will run off executables which reside on a floppy disk.

It is much easier to develop and test programs for the 68HC11 using this software if task switching is available, thus allowing switching back and forth between the simulator and the integrated development environment, without having to terminate one before starting the other. This means that some form of Windows (Windows for Workgroups, Windows 95, or Windows NT) must be used. The requirements are again very undemanding. The software was tested successfully on an 80386-based system with only 4 Mb. of memory, running Windows for Workgroups 3.11, with successful task switching between the development environment and the simulator.

The 68HC11 simulator responds to some elementary mouse operations (although the mouse is not necessary for its use), while the integrated development environment does not use the mouse at all. However, it is tedious (to say the least) to use any form of Windows without a mouse, so a mouse is virtually a necessity unless this software is to be run entirely from the command line.

A.3 Preliminary Installation

First and foremost, it should go without saying that the distribution diskette should never be written. Before doing anything else, make sure that it is write protected. (The moveable tab should be positioned "up," so that there is a "light hole" beneath it which goes straight through the diskette.) Also, make a copy of the distribution, either to another diskette (using the `diskcopy` command), or else to an archive such as a tape. Do not risk losing your only copy.

In its most basic form, installation of the software is trivial. Just create a directory (*e.g.*, `c:\pemicro`), and copy the contents of the distribution diskette to it. To run the software, change to this directory (using the `cd` command), and type the name of the appropriate executable (*e.g.*, `IASM11` or `SIM11A`).

A.4 Essential Customization

To use the software efficiently, it is advisable to perform some customization. This customization will ensure that the programs initialize with key parameters set properly, thus relieving the user of the burden of having to set them manually at each invocation.

A.4.1 Customization of IASM11

IASM11 is customized by using the IASMINST command. The latter program will lead the user through a long sequence of questions, to be answered one-by-one. It is important to know ahead of time which answers to provide. To view the parameters which must be changed, start up IASM11, Hit the F10 key, and then the A key. The following menu should appear.

```
Assemble   F4
Object     off
Listing    off
Debug map  off
Cycle cntr off
Macros     hide
Includes   hide
```

These are not the settings which are most appropriate for the development of course software. (It is not necessary to understand what these setting mean at this point.) The appropriate settings are:

```
Assemble   F4
Object     .s19
Listing    on
Debug map  on
Cycle cntr on
Macros     view
Includes   view
```

While these settings may be changed within IASM11 by highlighting the relevant entry and then toggling through the possibilities using the Enter key, settings selected in this fashion will only persist until IASM11 is exited. To install settings which apply to each new invocation of IASM11, the IASMINST utility must be used.

Upon invoking IASMINST, one will first be prompted for the name of the IASM to install. Respond with IASM11, and hit Enter. A sequence of questions involving settings will be presented, with the default values shown in rectangular brackets (*e.g.*, [Y]). The default value is selected by striking the Enter key. For this first screenful of questions, it is safe just to strike Enter in response to each query, although it may be preferable to modify some of these parameters later, once familiarity with the use of the utility is increased. The second screenful of queries poses the questions which will modify the parameters noted above. Answer the following questions with the answers shown in square brackets.

```
Do you want a listing file automatically created?  [Y]
Do you want an object file automatically created?  [Y]
Type S for s19 files or H for hex files ...  [S]
Do you a map file automatically created?  [Y]
Do you want cycle counts shown in the listing file?  [Y]
Do you want macros expanded in the listing file?  [Y]
Do you want include files expanded in the listing file?  [Y]
```

Upon answering the question about include files, a new screen of questions, pertaining to the communication window, will appear. The default values may be left intact here. Next, a "snow test" will be run, which should be answered appropriately. Next a color map will presented. At this point, the colors used may be changed. If, for example, it is difficult to read highlighted entries in the menu for the assembler options, change the colors used. As each type of text is selected, a sample is displayed. Change the colors as per preference. (In some cases, depending upon the color map of the computer, this may be necessary to render menus readable.) When the color-map modification is finished, hit the `Esc` key. At the next prompt, hit `Esc` again, and customization is finished. If the wrong key is struck and the keyboard modification utility entered, just hit `N`, then `Esc`, and then `Q` to quit.

It is impossible to back up through the sequence of menu items. If a mistake is made, just run out the option sequence by selecting the default values, and then restart the utility.

A.4.2 Customization of `SIM11A`

When `SIM11A` is started for the first time, two parameters must be selected via menu. The first is a the processor type, which must be selected from the following list.

```
HC11A0
HC11A1
HC11A7
HC11A8
HC811A2
```

The most appropriate one to select is `HC11A1`, although for most assigned programs it does not matter. The second parameter which must be selected is the mode; there are two choices:

```
Single Chip
Expanded
```

The appropriate choice is `Expanded`. The choices of these parameters are saved in a file named `SIM11A.CFG`, and upon subsequent invocations of `SIM11A`, values for the processor type and mode are not requested; rather they are taken from the `SIM11A.CFG` file. If a mistake is made in the selection, or if it is desired to make a change for any other reason, just delete the `SIM11A.CFG` file before starting `SIM11A`.

The default colors used in `SIM11` are also recorded in the `SIM11.CFG` file. Just type the command `colors` at the `>` prompt in the `DEBUG` window and follow the directions.

In addition to via the `SIM11.CFG` file, `SIM11` may also be customized via a file named `STARTUP.11A`. This file will be used later in the course to set advanced parameters, but it is not needed for initial use of the software.

A.5 User-Friendly Installation

In this section, some information on customizing an installation to make it more user friendly is given. While nothing in this section is essential to the use of the software, employing some or all of these ideas will make use of the software much easier in the long run.

A.5.1 Some simple hints

Window display option A DOS-shell running under Windows may be displayed in one of two modes: as a "full-screen" which covers the entire screen, or as a smaller, moveable window within the Windows environment. In both Windows for Workgroups and Windows 95, one may toggle between these two modes by using `Alt-Enter`. (Hold down an `Alt` key and strike the `Enter` key.)

Window size Sometimes, all of the text of a DOS shell in window mode will not be displayed. Rather, arrows and scroll bars will be displayed along the periphery. To see the entire screen, it must be scrolled. For software such as `IASM` and `SIM11A`, this can be extremely annoying. To resize the window, just position the mouse cursor over an edge until it is displayed as a double-headed arrow, and then drag the edge to make the window larger. It will not resize beyond dimensions which permit display of the entire screen.

If the window covers more of the screen than is desired, adjust the font size to make the window smaller. (See below.)

Font size The size of a DOS-shell window may be modified by adjusting the font size. In Windows 95, click on the `A` button a the top of the screen, and then select an alternate font. The `Apply` button will put this font in effect without closing the interactive properties window, so that various possibilities may be tried and the effect observed.

In Windows for Workgroups, first click on the "–" in the upper-left corner, and then select `Fonts...` from the menu. Finally, select the desired font and click on `OK`.

Termination of a DOS window Occasionally, a process running within a DOS window (such as `IASM` or `SIM11A`) will "hang," and it then becomes necessary to terminate it from the operating system. This is easy to do. Start by putting the process in "window" mode by using `Alt-Enter` if necessary. Next, in Windows 95, just click on the box marked `x` in the upper right of the window. Click on "Yes" in the resulting dialog box, and the process is terminated. The technique is slightly different in Windows for Workgroups. First click on the "–" box in the upper-left corner of the window. Then select the `Settings...` menu item.

Click on the resulting `Terminate` box, and finally on `OK` in the subsequent dialog box.

Remember that when a process is terminated in this way, all information contained internally is lost. So, for example, any unsaved editing within `IASM` is gone forever. Thus, termination in this fashion should only be used as a last resort, when there is no other way out.

A.5.2 Setting Up for Invocation via Icons and Menu Items

Most modern software for Windows installs automatically with icons and menu shortcuts, which enable the user to invoke the software directly, without having to start a DOS shell first. With the software from P&E Microcomputer Systems, this is not the case. Rather, such shortcuts must be installed manually, if they are desired. In this section, directions on how to install such shortcuts are described. These techniques are, by necessity, somewhat involved. Furthermore the procedure for Windows 95 differs substantially from that for Windows for Workgroups, so each is presented separately.

A.51 Windows 95 Installation

In Windows 95, there are two avenues to shortcuts for a program. One is via the `Programs` submenu of the `Start` menu, and the other is via a shortcut icon placed directly on the desktop. To install such a shortcut in either case, begin by opening up a `Windows Explorer` window, by selecting that program from the `Programs` submenu of the `Start` menu, or by clicking on its desktop icon. Next, navigate down to the directory containing the program to which a shortcut is desired (*i.e.*, the directory containing the P&E software). Now, right-click (*i.e.*, select and click with the right mouse button) on the item to which a shortcut is desired, and select `Create Shortcut`. A shortcut will be created and will appear in the directory. This shortcut may be dragged to the desktop, and may also be renamed. Use the `Rename` option from the right-click menu to effect a rename (*e.g.*, to remove "shortcut to" from the name). It is easiest to rename after dragging the shortcut to its new home, since it is not allowed to have two entries of the same name in the same directory.

To place the shortcut in the `Programs` menu, proceed as follows. Select the `Settings` option from the `Start` menu, and then select `Taskbar`. In the resulting window, click on the `Start Menu Programs` tab, and then click on `Advanced`. A Windows Explorer window will appear, which is rooted at the start menu. Select the `Programs` menu. Now, pull down the `Files` menu, and select `New` and then `Folder` (by sliding the mouse off the right end of the box). A new folder will appear, which initially will be named `New Folder`, but which may be renamed by right-clicking on it and selecting `Rename`. Next, select the new folder by double clicking on it. An empty right window of the Explorer should appear. Just drag the shortcut which was created in the P&E directory

to this location, and it will appear in start-up menu.

A shortcut has many properties, some of which may be very useful. To access the properties list, right click on the shortcut, and select the `Properties` menu item. A tabbed window will appear. Many of the properties of the shortcut may be changed from this window. Some of the most useful are the following:

Close on exit Found under the `Program` tab, placing a check in the box will cause the window to disappear when the underlying process is exited. This is the appropriate setting for both `IASM11` and `SIM11A`.

Argument prompt Under the `Program` tab lies the `Cmd line:` field, which contains the path to the command to be executed. If a `?` is placed after the command name (*e.g.*, `c:\pemicro\iasm11.exe ?`), then whenever the shortcut is invoked, a dialog box will appear, in which command-line arguments may be placed. The shortcuts on the laboratory machines are set up in this fashion.

Screen size The choice of whether the process begins in full-screen mode or as a window is selected under the `Screen` tab.

Font The initial font size for a window is selected under the `Font` tab.

A.52 Windows for Workgroups Installation

In Windows for Workgroups, there is only one form of shortcut, and that is via an icon in a group window. To set up such a shortcut, from the `File` menu of the `Program Manager`, select `New`, and then select `Program Group`. Give the new group an appropriate name (such as "68HC11"); the `Group File` entry may be left blank. Click on `OK` and a new group will appear. Now entries may be added to this group. It is here that things are more complicated than in Windows 95.

Start by going to the `Main` group and selecting the `PIF Editor`, which has a yellow sale tag as its icon. (`PIF` stands for *Program Information File.*) If this icon cannot be found, then go to the `File` menu of the Program Manager, select `Run`, and type `c:\windows\pifedit.exe`. A PIF Editor window should appear. Enter the path to the program in the `Program Filename` box (*e.g.*, `c:\pemicro\iasm11.exe`). If it is desired to have a prompt for command-line arguments (as is done in the laboratory installations), place a `?` in the `Optional Parameters` box. Finally, enter an appropriate title for the window (*e.g.*, `IASM11`).

At the bottom of the screen, two appropriate options may be selected. First of all, one may choose as to whether the display initializes as `Full Screen` or as `Windowed`. Similarly, one may choose to have the window close upon exit, so that the window disappears when the program finishes. It is recommended that both `Windowed` and `Close Window on Exit` be selected.

Finally, it is time to save the PIF. From the `File` menu, select `Save As`, and then give a path to a name with extension `.pif`; *e.g.*, `c:\pemicro\iasm11.pif`. When this is completed, close the PIF editor by selecting `Exit` from the `File` menu.

To finish, it is necessary to place an icon for this PIF in the new group. Select the group which was created earlier, and from the `File` menu of the program manager, select `New` and then `Program Item`. The `Command Line` entry should be the path to the PIF; *e.g.*, `c:\pemicro\iasm11.pif`. Provide a `Description` as appropriate (*e.g.*, `IASM11`), an also provide a working directory, which may be either the directory in which the software resides, or else a temporary system directory (*e.g.*, `c:\tmp`). Finally, it is nice to select an interesting icon by clicking on the `Change Icon` button, followed by `Browse`.

A.6 Known Bugs

A.6.1 Bugs in `IASM11`

There is one major bug which has been found in `IASM11`. The `F2` key is used to save the editor contents to a file. If no file has been selected previously, the user is prompted for one. If the path given is illegal, the response `Path not found - Press <Esc> to continue` will be displayed. However, there does not appear to be any way to override this and save the file along a different, legal path — the buffer contents will be lost. The workaround is to do one of the following:

1. Start up `IASM11` by providing the file name on the command line (*e.g.*, `IASM c:\ee101\programs\foo.asm`). Note that in the laboratory installations this argument may be provided in response to a prompt when the software is started. Section 5.2 above provides instructions on how to configure a personal system similarly.

2. Before doing any editing, immediately hit the `F2` key and provide a file name. If an error is made, nothing is lost, although `IASM11` must be exited and restarted.

A.6.2 Bugs in `SIM11A`

One major bug has also been found in `SIM11A`. If the `load` command is issued in the `DEBUG` window without any argument, a pop-up menu appears in the center of the screen, which supposedly allows the user to select the file to load. However, if an inappropriate selection is made, a "`DOS Error #204`" message appears, and the simulator has apparently crashed and must be restarted. The workaround is always to provide an argument to the load command (*e.g.*, `load foo.s19`).

A.7 Documentation

A.7.1 Accessing the On-Line Help

Although there is no user manual per se, there is on-line documentation for both `IASM11` and for `SIM11A`. This documentation is contained in the files `IASM11.HLP` and `SIM11A.HLP`, and is normally accessed internally from the executables. To access the documentation from `IASM11`, press the `F1` key, and then select the desired item from the menu. Note that there is no mouse control in `IASM11`, so the arrow keys must be used to highlight the item for which further information is desired. Once the desired item is highlighted, hitting the `Enter` key will pop up a subwindow containing help text for the selected topic. If the text will not fit in a single window, the `Page Up` and `Page Down` keys may be used to scroll the window. To exit a subwindow, strike the `Esc` key, and to exit the help utility, strike `Esc` again.

Help is accessed similarly from `SIM11A`, by striking the `F1` key. In this case, however, the mouse may be used to selected the desired item. Note also that the help menus are hierarchical, so selecting a given item may yield another menu.

A.7.2 Printing the On-Line Help

The utility `PRHELP` may be used to print the contents of the `.HLP` files, as well as to create a plain text file containing their contents. Running the `PRHELP` command without any arguments will provide a screen of information on how to use the command. Basically, running the command with a single help file as an argument (*e.g.*, `PRHELP IASM11.HLP`) will print the contents of that help file on the printer connected to the printer port of the PC. If an "`F`" is postpended to the command (*e.g.*, `PRHELP IASM11.HLP F`), instead of printing directly, the output will be directed to a file, the name of which will be requested by the program. When `PRHELP` is used to print a help file, simple printing is invoked in the default font of the printer. If it is desired to use a utility to print (*e.g.*, Microsoft *Write* or Microsoft *Word Viewer*), then it is necessary first to write to a file, and then to use the utility to print the file. The `PRHELP` utility will always paginate the output (with the size of the pagination adjustable — run `PRHELP` with no argument to see how). Unfortunately, it does not appear to be possible to generate a file with no pagination, which might be appropriate choice when using utilities such as Write or Word, which create their own pagination.

Appendix B

Debugger and Interrupt Subtleties

B.1 Interrupts Used by Debugging Commands

The debugging commands found in `EVB11` (and in the underlying Motorola BUFFALO monitor as well), such as setting breakpoints and single-stepping, are implemented via a complex interaction of a PC-based monitor and monitor firmware on a ROM chip on the EVB. A major goal of such a design is to provide these services with a minimum of disruption of execution of the program being debugged. Unfortunately, since part of the debugger program must run on the MC68HC11 itself, it is not possible to make these services totally transparent to execution of the user program. In this section, the restrictions which the debugging routines place upon the user program are sketched.

It is important to remember that these restrictions apply only to use of the Motorola EVB and its firmware; they do not apply to `SIM11` or `SIM11A`.

Breakpoint management uses `SWI` The MC68HC11 instruction `SWI` executes a *software interrupt*; *i.e.*, an interrupt executed by the software. It generates an interrupt by its very execution, rather than as a consequence of an external event. Breakpoints are implemented in the debugging package with the aid of `SWI` instructions. To set a breakpoint at a particular address, the monitor replaces the instruction at that address with an `SWI`. When execution of the program reaches the given address, the interrupt handler for software interrupts is invoked. This handler is provided by the monitor program. The handler takes care of the details of breakpoint support, and it then replaces the `SWI` instruction with the original instruction, so, to the user, it is not apparent that an `SWI` was ever executed.

Of course, this strategy implies that programs which themselves contain `SWI` instructions cannot be debugged using breakpoints, since the interrupt handler of the user would necessarily replace that of the system. On the other hand, a software interrupt is not maskable, and has a higher priority than any interrupt other than a reset. Thus, one need not worry about an interrupt of higher priority disrupting this process.

Occasionally, when debugging goes awry for other reasons, an `SWI` instruction may

be found at a location which should contain something else. In this case, the easiest strategy is to reload the program from disk.

The stepper uses Timer Output Compare 5 The stepping routine in the debugger makes use of Timer Output Compare 5. To single step an instruction, register `TOC5` is set to a value which is one greater than the value of the timer at the beginning of the instruction to be stepped. A timer interrupt is thus generated during execution of the instruction, and the handler for Timer Output Compare 5 manages the details of providing the display information.

Obviously, this strategy implies that the program cannot use Timer Output Compare 5 itself, but this is seldom a problem, since the MC68HC11 has four other timer output compare units. A more significant limitation regards the enabling of interrupts within interrupt handlers. When an interrupt handler is entered, the `I` flag of the `CCR` register is set, thus masking all maskable interrupts. This action is taken to ensure that whatever critical actions that handler must take may be performed without suspension by another interrupt. The user program may execute a `CLI` within the handler if so desired. However, when the interrupt handler is exited via an `RTI` instruction, the old value of `CCR` is restored. Except in the rare case that an interrupt occurred during the execution of a `STI` command, this will mean that interrupts will be re-enabled upon exit of the handler.

This convention masking interrupts poses a problem to the stepper, since stepping makes use of interrupt-driven timer output compare. If interrupts are disabled within a handler, the stepper cannot operate. Thus, the stepper must enable interrupts at any point to which it steps, including an interrupt handler. This may or may not pose a problem, depending upon what the program is doing. The programmer needs to be aware of this action, however. See particularly the discussion in the next section regarding the clearing of timer flags.

Similarly, if one steps a program from the beginning, interrupts will be enabled, even before a `CLI` instruction is executed.

B.2 Clearing Timer Compare Flags

As indicated in the write-up for Laboratory Exercise 7, the flags for timer output compare are cleared by writing a `1` to the corresponding bits. The obvious way to do this is via a `bset` instruction, and the strategy shown in Figure 1 was suggested for clearing bit `OC1F` of `TFLG1` in Laboratory Exercise 7. Unfortunately, this strategy will not always work correctly. Due to subtleties in the way that the `bset` instruction works, it may wind up clearing other bits in `TFLG1`. (This limitation applies only using `bset` on registers `TFLG1` and `TFLG2`, and not in general.) Clearing other bits in `TFLG1` will not pose a problem as long as only one timer compare function is being used. However, if the stepper of the debugger is being used, then as noted above, Timer

```
;;*******************************************************************
TFLG1        equ  $1023 ;; Timer Flag Register 1.
OC1F_bit     equ  $80   ;; Timer Output Compare 1 flag.
;;*******************************************************************
Svc_OC1_int:
;;      <Service the interrupt>
        ldx   #TFLG1
        bset  0,X,OC1F_bit  ;; Clear the interrupt flag
        rti                 ;; and return.
;;*******************************************************************
```

Figure B.1: The dangerous way to clear a timer flag.

Output Compare 5 is also being used. Stepping over the `bset` instruction in Figure 1 will clear the `OC5` flag as well! The stepper will thus be run until the next Timer Output Compare 5 match, which will be 65536 clock ticks later! This explains the strange behavior which some have observed while trying to step through the Timer Output Compare 1 interrupt handler.

Figure 2 illustrates a reliable way to clear a timer flag. Figure 3 illustrates another

```
;;*******************************************************************
TFLG1        equ  $1023 ;; Timer Flag Register 1.
OC1F_bit     equ  $80   ;; Timer Output Compare 1 flag.
;;*******************************************************************
Svc_OC1_int:
;;      <Service the interrupt>
        ldaa  #OC1F_bit
        staa  #TFLG1       ;; Clear the interrupt flag
        rti                ;; and return.
;;*******************************************************************
```

Figure B.2: A safe way to clear a timer flag.

way. except the leftmost. Admittedly, both are rather cryptic, in that neither alters the value of any bit of `TFLG1`, while both set this bit. Again, these techniques apply to `TFLG1` and `TFLG2` only. The reason why they works is subtle and will not be discussed here. Consult Section 10.2.4 of the *Motorola MC68HC11 Reference Manual* for more information.

```
;;*********************************************************************
;;*********************************************************************
TFLG1         equ  $1023 ;; Timer Flag Register 1.
OC1F_bit_comp equ  $7F   ;; Timer Output Compare 1 flag.
;;*********************************************************************
Svc_OC1_int:
;;      <Service the interrupt>
        ldx   #TFLG1
        bclr  0,X,OC1F_bit_comp     ;; Clear the interrupt flag
        rti                         ;; and return.
;;*********************************************************************
```

Figure B.3: Another safe way to clear a timer flag.