Software Development in Assembly Language: 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

A curious distinction has evolved in university education in computer science, in comparison to education in engineering. In engineering, the curriculum consists of lectures covering conceptual material, which are complemented by formal, scheduled laboratory work. In these laboratories, the student is given a detailed written description of what to do, and the approach to take. During the laboratory meeting, a skilled instructor first makes a presentation to the students on some of the finer and more difficult points, and the student then proceeds with the investigation, with the laboratory instructor close at hand to provide assistance and encouragement. Upon completing the investigation, the student is expected to submit a detailed report on the procedures and conclusions of the investigation, which is then evaluated by the laboratory instructor.

In computer science education, this approach is rare. While many, if not most, courses involve a substantial amount of practical investigation in the form of software development, this activity is typically viewed as one which is ancillary to the organization of the course meetings. In lieu of formal laboratory meetings, the student is expected to pursue and complete the software exercises on his or her own, often with only the vaguest of descriptions of what is to be done, and with little more than user manuals (sometimes even those are not available) describing the use of the supporting software. Not surprisingly, the student often invests far more time developing the software than studying the concepts, while instructors express amazement at the inefficient and inelegant approaches embodied in the student's work.

In short, most computer science courses seem to consist of two components: formal class meetings which cover conceptual material, and practical experience in the form of software development for which little or no formal course support is provided. I have come to term the latter components *hidden labs*. In the course of teaching many computer science courses at all levels over the past twenty years, it has become my firm conviction that, at least for lower-level courses in which the students are still learning the principles of software development, hiding the laboratory experience as an ancillary component of the course, without specific, formal laboratory meetings, results in an ineffective mode of instruction for the vast majority of students. This laboratory manual is the product of an effort to conduct instruction in an introductory course in machine organization and assembly-language programming using the engineering-style explicit laboratory, with detailed instructions for laboratory work and formal laboratory meetings staffed by skilled instructors. It presents a sequence of ten software assignments, each of which is intended to be pursued at one or two laboratory meetings.

In harmony with the engineering approach, the description of each software experiment contains a substantial amount of detail, not only on what the completed product it to do, but also on how development of the solution is to be approached. I have striven to identify a balance in which the student is provided with enough information to establish a firm basis for organizing an approach to attacking the problem, while still being required to display a significant amount of creativity. Ideas which are new to a given software assignment are expanded in some detail, while those which are embodied in earlier ones are merely suggested as appropriate, or even left to be identified by the student.

It must be emphasized that the student is not expected to complete the entire software assignment in one laboratory meeting. Rather, the laboratory meetings are intended to provide fora in which the more difficult aspects of the exercise are clarified by the laboratory instructor, and in which the student may seek individual help. Students must still dedicate outside time to the completion of the exercises. However, with the combination of these relatively detailed task descriptions and the guidance of dedicated laboratory instructors, the software exercises become much more an integral part of the course. It has been my sense that the level of student frustration is far less with this arrangement than it has been with courses in which the students are expected to pursue software development entirely outside of formal class meetings. Nonetheless, there is no particular reason why the experiments of this manual could not be used in a course without a formal laboratory, although it would probably be unreasonable to expect the average student to be able to complete more than half of them under those circumstances.

The experiments are designed to be performed using (version 6.11 of) the Microsoft MASM assembly-language development system, From the start, the experiments introduce the student to the use of the entire Programmer's WorkBench (PWB) development system, including the integrated editor, debugger (CodeView), and make facility. The package which Microsoft makes available to students is the full professional version, and not a stripped-down shell. The students thus have access to all of the powerful features which the professionals do.

The experiments are keyed to both the current textbook which I use for the course [Irv93], and the three main MASM/PWB reference manuals [MAS93], [MAS92b], and [MAS92c]. (The MASM package belongs to the "old school" of software distribution, and includes approximately 1600 pages of printed documentation.) However, the experiments may easily be adapted for use with another textbook.

Appendix A provides some rather detailed notes on the use of MASM and PWB. While much of this information is specific to the installation at the University of Vermont EMBA Computing Facility, it also includes a fair amount which is relevant to MASM/PWB in general.

Content Overview

In this section, an overview of the experiments is provided, for the convenience of the instructor developing a course, as well as for the interested student. The ten experiments are divided into three groups, with the following emphases.

Part 1 — Introductory software exercises: The overall goal this part, which consists of the first three software assignments, is to introduce the student to software development using MASM and PWB. Software Assignment 1 simply requires the student to enter, assemble, debug, and run a program which is given in full. Its main purpose is to provide an introduction to the use of the development environment. In engineering style, the student is led to try many of the features of the system, and to observe how they are used in software development. The second and third experiments provide the student's first experience in writing an assembly-language program, and focus upon simple input from the keyboard and output to the display. They begin with line-at-a-time output, and then progresses to more complex behavior, including character-at-a-time output, proper processing of the backspace input character, and unechoed input and masked output. The final product is a password-validation module, which behaves much as a real password entry interface would, by echoing asterisks as the password is typed and responding with a statement as to whether or not the correct password was typed.

Part 2 — Program organization and development techniques: In the second group of experiments, consisting of Software Assignments 4 and 5, the student is introduced to two of the most important concepts in software engineering: procedures and separately assembled modules. Often, assembly-language programming is taken to be the last bastion of true hackery, in which the disciplines considered to be the core of software engineering have no place. It is my position that quite the opposite is the case. Since assembly language provides fewer language-specific tools for disciplined software development than do higher-level languages, it becomes paramount to take full advantage of those which are present. Software Assignments 4 and 5 are specially designed to familiarize the student with the powerful software management tools and functional capabilities of MASM and PWB. Software Assignment 4 introduces the student to the notion of a procedure in assembly language, while Software Assignment 5 provides a first experience with the module-management facility (the make facility) of Programmer's WorkBench, in which the program may be broken into several modules, which are separately assembled. The latter notion is particularly critical for software engineering in 8086 assembly language, because there is no facility for locality of reference within a given module. Rather than requiring the student to build new and different software, the fundamental focus of Software Assignments 4 and 5 is to re-engineer the work of Software Assignments 2 and 3, using modern software tools. In this way, the student may focus upon the ideas at hand – the use of new tools for solving old problems – and may thus see the comparative advantages of the various approaches. Furthermore, a set of libraries of common system and procedure calls is developed in Software Assignment 5, and is used throughout the remainder of the projects.

Part 3 — selected software projects: Upon the completion of the first five software assignments, the student will have a firm foundation for software engineering in assembly language. The purpose of the third group of experiments, consisting of Software Assignments 6 through 10, is to provide selected excursions into more advanced topics in software development in assembly language and interfacing to the operating system. Throughout these exercises, the student is required to make use of techniques learned in the introductory experiments, such as use of the make facility, as well as of the system call libraries developed in Software Assignment 5.

Software Assignment 6 focuses upon the development of a service which higher level languages provide, but which must be programmed explicitly in assembly language — the translation of integer representations between internal and external formats. Upon the completion of this assignment, the student should have a firm understanding of how integers (positive and negative) are represented in various formats, and what higher-level languages must do when reading or writing an integer value.

Software Assignments 7 and 8 constitute a two-part exercise which has proven to be one of the most popular amongst the students. In it, a directory-listing utility, similar in behavior to “`ls -l`” of UNIX, is developed. In these assignments, the student must not only issue the appropriate system calls to obtain the information about the selected directory; this information must also be unpacked and translated into a format suitable for display. Thus, in addition to system calls, the use of logical and shifting commands plays a central rôle, and an opportunity to re-use portions of the software developed in Software Assignment 6 is provided. Furthermore, the use of assembler macros to generate both code and data structures is introduced as a required component. Finally, Software Assignment 8 provides a rather thorough coverage of the program segment prefix (PSP), and how to extract command-line arguments from it.

Software Assignments 9 and 10 constitute a two-part exercise in which the student develops a simple XOR file encryption program. While this form of encryption is hardly secure, its realization does involve several new ideas, including particularly random file access. Furthermore, the techniques of developing the programs include constraints in which procedures must pass parameters via the stack.

In developing these exercises, a number of compromises had to be made. Perhaps the most difficult was the exclusion of an exercise which would link a procedure written in assembly-language to one in a higher-level language, such as C. In the modern systems environment, it is becoming less and less common to write entire programs in assembly language. Rather, most of the software is written in a higher-level language, with only a few key components developed in assembler. Linkage to programs in other languages is thus a critical notion. Unfortunately, such linkage is only possible (without great effort) when a standard object-file format is available. In plain English, this means that that the assembler and compiler must come from the same vendor. Unfortunately, at the time that these assignments were under development, the use of Microsoft C was not a viable option, and so such an experiment had to be eschewed. For those working in an environment which includes Microsoft C, the addition of a multi-language linkage exercise would be most appropriate.

Another compromise which was made was the limitation of the exercises to 8086-compatible code; instructions specific to higher-level processors were excluded. In terms of "ordinary" programming, this is not much of a compromise, since in protected mode, the additional instructions are largely 32-bit versions of existing ones. The real power of the 80386-compatible processors is the ability to support virtual mode, which is used largely in the development of operating systems. In addition to the advanced nature of such programming, modern operating systems, such as Windows 95, will not permit a user program to execute such commands. For these reasons, such topics were viewed as too advanced for an introductory course.

Acknowledgments

Ronald Williams taught the course in machine organization and assembly-language programming the semester before I assumed responsibility for it, and graciously provided a preliminary but well-thought-out sequence of experiments based upon that offering, some of which evolved into those found in this manual. He also served as a critical listener for my ideas on the design of experiments, and provided much-needed encouragement for the formal-laboratory approach to the course.

The success of a laboratory-based course depends in no small measure upon the contributions of the laboratory instructors. Both Gregg Stadtlander and David Noel have served in this rôle, and have contributed in no small measure to the success of the course, through their dedicated service as instructors and as well as through constructive comments on preliminary versions of the written descriptions which comprise this manual.

Finally, numerous students who have taken the course have provided valuable feedback.

Trademarks

Part I

Introductory Software Exercises

Software Assignment 1

Becoming Acquainted with MASM and PWB

1.1 Purpose

The purpose of this laboratory exercise is to help you gain familiarity with the MASM development environment, including the Programmer's Workbench (PWB) and the CodeView debugging environment.

Bring your copies of the MASM manuals to the laboratory session. You may need them, and copies will not be available in the laboratory.

1.2 Procedure

As you work through this first part of the procedure, you may wish to reference Chapters 2 through 7 of the *Environment and Tools* manual [MAS93], which documents these concepts.

It is suggested that you "check off" each step in this laboratory exercise as you have completed it.

1. Begin by installing MASM/PWB properly on your system. This includes in particular the initialization files. See Appendix A, and in particular A.4.1, for more information.
2. Start up PWB, either by issuing the PWB command at an MS-DOS prompt, or else by double-clicking on the appropriate Windows icon.
3. Check the option settings, as outlined in Appendix A, Section A.5.1..
4. Go to the `Files` menu, and select `DOS Shell`. This will open a subshell for you. Create an appropriate directory in which you will save your programming

assignments for the course. If you are using an EMBA-CF microcomputer, issue the command `mkdir M:\assign`. This will create a directory named `assign` in the file space of the account which you are using.

5. Type the command `exit` to return to MASM/PWB.
6. Go to the `Files` menu, and select `New`. A new window, entitled something like `Untitled.001`, will appear.
7. Go to the `Files` menu again, and select `Save As`. Now backspace the cursor at the `File name` field at the top of the new command window which has appeared on the screen, to erase that field. Then, type `M:\assign\soft1.asm` in that field. Now click on `OK` at the bottom of the screen. Notice that the name which appears at the top of the file window is now the name which you have given the file.

   Note: Do not type just a file name, such as `soft1.asm`, as the name of the file. If you do this, the file will be saved in the default directory. If you started MASM/PWB directly by selecting MASM from the Programs menu of the Start button, the file will be saved in the system Windows directory, `C:\Windows`. If you are using your home system, this is a nuisance. If you are using an EMBA-CF system running Windows 95, this is not only a nuisance; it is also not proper etiquette for use of a shared system.[1] It clutters up a key system directory. Furthermore, if you leave your program sitting there in the Windows directory, someone else can easily copy it. If you started MASM/PWB by typing `pwb` on the command line of a DOS shell, then the default directory will be whatever directory you were in when the shell was started.
8. Windows in PWB can be moved around easily. Put the cursor over the double horizontal bar at the top of the window, and move the mouse. Notice that the whole window can be moved in this fashion. Similarly, one can put the cursor over the single vertical bar on the left side of the window and move it around.
9. Windows can also be resized easily. Put the cursor over the short horizontal bar at the lower right corner of the window. Moving the cursor changes the height of the window. Similarly, place the cursor over the short vertical bar at the lower right of the window, move the cursor, and note that the width of the window changes.
10. Put the cursor on the arrow which points down which is at the upper right corner of the window. Click the left mouse button, and note that the window has disappeared. Actually, it has not really disappeared. Rather, it has been iconified, but it may now be hidden by another window. By *iconified*, we mean that it has been shrunk to a little window with just a name (in this case a number) inside. To get it back, select the `Window` menu item from the top of the screen. Towards

[1]If you are using an EMBA-CF system running Windows NT, it will not let you write to the directory `C:\Windows`, so you must choose another directory.

the bottom of the menu which appears, you should find the name of this window. Select this item to restore the window. When you click on the window name, it will reappear iconified. Just click on this small box to get the full display back. Notice also that the window has a number, which appears both on this menu and to the left of the name on top of the window.

11. Put the cursor on the arrow which is pointing up, located at the upper right corner of the window. Click the left mouse button. Notice that the window fills the entire screen. To restore it to its previous size, click on the double-headed arrow at the upper right of the window.

12. If there is another window on the screen, notice that the same command works with it, and that each window has a number. Any window may be selected by clicking on it, or by selecting it from the `Window` menu.

13. Now it is time to start programming! For a starter, you will enter a program which has already been written for you. Type in the program which is found in Figure 1.1 on page 12. In doing so, you will gain some experience in using the editor which comes with MASM/PWB. Just type the text of the program into the window for `soft1.asm` which you created.

    Unfortunately, Microsoft has chosen not to document systematically the operation of this editor in the massive manuals which come with MASM/PWB. Some information is contained in Chapter 7 of the *Environment and Tools* manual, but one really has to dig.

    There is limited documentation on line. Try it as follows. Select `Editor Settings` from the `Options` menu. Now scroll through the `Switch List` and select one of them, say `tabalign`. Click on `Switch Help` to get information on what that switch is about. Click on `Cancel` to return to the `Editor Settings` menu. This sort of documentation is limited to switch settings, however. (Get out of the options menu by clicking on `Cancel` again.)

    You can find out the name of the command which a particular control key is bound to by typing `Ctrl-T` followed by that key. Try it on a few keys, including `Ctrl-L` (search). One can find out what a command does by looking it up in Chapter 7 of the `Environment and Tools` manual [MAS93].

    While the editor is reasonably simple to use, it does have some quirks. The tab key skips to a new field, and does not insert a tab. This can be frustrating if one formats a line incorrectly, and later wishes to insert a tab. The easiest way around this is to insert by using the correct number of spaces. Another quirk is that the delete key will not delete an end-of-line. To delete an end-of-line, go to the beginning of the next line and type backspace (or use the visual `Cut` command).

    Notice that at the right lower corner, the line and column number is indicated.

Also, some letters may be shown. The letter `O` denotes overwrite mode, is toggled with the insert key. The letter `R` denotes read only, and is toggled from the `Read Only` entry in the editor menu. The letter `M` indicates that the file has been modified. See page 62 of the *Environment and Tools* manual [MAS93] for complete information on the display at the bottom of the screen.

The `Stream Mode`, `Box Mode`, and `Line Mode` entries in the `Edit` menu determine the type of capture that occurs when text is selected with a sweep of the mouse. Try it. The other menu items, such as `Cut`, `Copy`, and `Paste`, are reasonably self-explanatory. Surprisingly, search is not on the menu, but may be invoked with `Ctrl-L`.

It is wise to save frequently. To save a file, one can either select the `Save` entry from the `File` menu, or else type `Shift-F2` — that is `Shift` plus the `F2` key — or just hit the `SF2` key if your keyboard has one.

Finally, notice that the editor window may be scrolled both vertically and horizontally by dragging the mouse on the movable rectangles on the right and bottom of the window, by clicking on the arrows along the bottom and right side.

Before you go to the next step, make sure that you have typed in the program.

14. Once you have completed typing in the program, it is time to assemble and link it. To do this, go to the `Project` menu and click on `Build:` or `Rebuild All`. One of two things will happen. If you typed in the program correctly, a window giving a message indicating zero errors will appear, and you will be given a number of options. Select `Run Program` and it should print the message on the screen and then prompt you to type any key to return to PWB. Do this.

    If you typed in the program incorrectly, you will get a message that there are some errors. Select `View Results` and an error window will appear at the bottom of the screen. By examining the error messages, it is often possible to get a good idea as to what the error is. If you can't figure it out now, do not be concerned. Just check the text carefully for typing errors.

    Before you go on to the next step, make sure that your program works.

    If you get a message such as

    ```
    NMAKE : fatal error U1045: spawn failed :  Arg list too long
    ```

    see item 6 of Section A.5.1 of Appendix A to see how to declare a virtual drive, with a resulting shorter argument list. To recover from this error, you will have to open a new editor window, with your source file accessed via the virtual drive.

15. Notice that in the project menu, in addition to `Compile File: <filename>.asm`, there is also `Build: <filename>.exe` and `Rebuild All`. The menu item

`Compile File: <filename.asm>` just assembles the file, while the `Build` command both assembles and links and loads the file. Since this is a one-module program, no linking is necessary. *Loading* refers to the process of taking the assembler output and configuring it for actual execution. (We will discuss this process in more detail in the course. For now, it is best to avoid using the `Compile File:` option by itself, because it will not link and load — it generates a new `.obj` file, but not a new `.exe` file, and so you may inadvertently execute an old version of the program.) The `Rebuild All` command differs from `Build` when there is more than one file in the project. It rebuilds them all, and not just the one named. Again, we will discuss this in more detail as the course progresses.

Once a program is assembled, the `Execute` entry may also be selected from the `Run` menu.

It is now time to learn a little about the CodeView debugger. As you work through this part of the exercise, you may wish to refer to Chapters 8 through 12 of the *Environment and Tools* manual [MAS93].

16. From the `Run` menu, select the `Debug` option. This will place you in the CodeView debugging environment. If you have set the options properly, the screen should switch to 50-lines mode. If you would like the PWB main screen to be in 50-line mode as well, follow the directions in Section A.5.4 of Appendix A.

17. CodeView provides a host of windows, including two `Source` windows, two `memory` windows, a `help` window, a `watch` window, a `locals` window, a `register` window, an `8087` window, and an `command` window. The existing window layout should provide a good view, but if not, try to lay things out so that the following windows are visible without too much overlap: `source1`, `memory1`, `command`, `watch`, `reg`, and `locals`. It will be most informative if you can view all of these windows simultaneously.

18. The `source1` window provides a view of the source code of the program. (If it does not, that probably means that you have not set the `Use Debug Options` in the `Build Options` menu of PWB.) Note that the `F3` key toggles the format of the `source1` window amongst three distinct formats: a disassembly of the executable, a listing of the source, and mixture of the two. Notice that the first executable instruction is highlighted. We can step through the program, step-by-step. To do so, use the `F8` key. This is known as the *trace function*. Each time that the `F8` key is pressed, the program advances one step. Notice that the highlighted line changes at each step. The `F10` key invokes the *step function*, which is similar, except that it skips functions which are called (excluding DOS calls). Since there are no such functions in this program, `F8` and `F10` behave identically here.

    The `source2` window operates identically. It is of use when one is working with a program with more than one source file.

Notice that the screen flashes to the MS-DOS output screen at each step. To turn this off, select the `Screen Swap` entry of the `Options` menu. This will toggle this option. Now, stepping will not cause the screen output to be displayed at each step. However, whenever output is generated, a message box will appear indicating so; click on `OK` to continue. Also, the output may be viewed at any time by selecting the `View Output` item from the `Windows` menu.

To repeat this process, select the `Restart` option from the `Run` menu. This will restore the program to a starting configuration.

After selecting `Restart`, try selecting the `Animate` function from the `Run` menu. The program will step automatically, at a slow rate. The speed of the animation may be changed by selecting the `Trace Speed` item from the `Options` menu. Try it.

19. The `memory1` window provides a view of memory. The display changes dynamically as the program steps through its instructions. There are twelve different formats for memory display, with the `Shift+F3` key (or `SF3` if you keyboard has it) toggling the display. Select the `Memory1` window and try toggling the format.

    One problem with using a memory window is that often it is not entirely obvious just which data items one is viewing. How do we find out the address at which a particular data item is stored? The answer is to use the `command` window. The `X` command, with a suffix, will provide the addresses of data items. Try selecting the command window, and then, at the prompt `>`, type `XM`. This will display all data items in the current module. It may look something like this:

```
>xm
1915:0044   {,SAMP1.ASM,SAMP1.exe} var1
1915:0008   {,SAMP1.ASM,SAMP1.exe} message
1914:0000   {,SAMP1.ASM,SAMP1.exe} main
>
```

    This tells us where the variables are stored. For example, `var1` is stored at location `1915:0044`. So, to follow its value, we have to make it visible in the `Memory1` window. The problem is that the memory window might use a different segment. For example, `Memory1` might display as follows.

```
1914:0000  B8 15 19 8E D8 BA 08 00 B4 09 CD 21 83  8UYNX:H.4.M!C
1914:000D  06 44 00 01 B0 00 B4 4C CD 21 00 54 68  FD.A0.4LM!.Th
1914:001A  69 73 20 6D 65 73 73 61 67 65 20 73 68  is message sh
1914:0027  6F 75 6C 64 20 62 65 20 64 69 73 70 6C  ould be displ
1914:0034  61 79 65 64 20 77 68 65 6E 20 74 68 65  ayed when the
1914:0041  20 70 72 6F 67 72 61 6D 20 69 73 20 72   program is r
1914:004E  75 6E 2E 0A 0D 24 00 00 4E 42 4E 42 30  un.$....NBNB0
1914:005B  38 90 02 00 00 00 00 00 00 01 00 43 56  8PB......A.CV
1914:0068  01 00 00 00 00 00 00 00 17 00 00 00 23  A.......W...#
```

```
1914:0075  44 3A 5C 43 4F 55 52 53 45 53 5C 43 53  D:\COURSES\CS
1914:0082  31 30 31 5C 50 52 4F 47 52 41 4D 53 5C  101\PROGRAMS\
```

Yet, the address we have for `var1` uses segment 1915. We could compute the equivalent address in segment 1914 manually, but that would be time consuming and prone to error. Instead, we should let the machine do it. To change the segment, just go to the `memory1` window and rewrite the first 1914 to be 1915. (Just type over the numbers — that is all that there is to it.) We then see the following in the `memory1` window.

```
1915:0000  01 B0 00 B4 4C CD 21 00 54 68 69 73 20  A0.4LM!.This
1915:000D  6D 65 73 73 61 67 65 20 73 68 6F 75 6C  message shoul
1915:001A  64 20 62 65 20 64 69 73 70 6C 61 79 65  d be displaye
1915:0027  64 20 77 68 65 6E 20 74 68 65 20 70 72  d when the pr
1915:0034  6F 67 72 61 6D 20 69 73 20 72 75 6E 2E  ogram is run.
1915:0041  0A 0D 24 00 00 4E 42 4E 42 30 38 90 02  $..@@NBNB08PB
1915:004E  00 00 00 00 00 00 01 00 43 56 01 00 00  ......A.CVA..
1915:005B  00 00 00 00 00 17 00 00 00 23 44 3A 5C  .....W...#D:\
1915:0068  43 4F 55 52 53 45 53 5C 43 53 31 30 31  COURSES\CS101
1915:0075  5C 50 52 4F 47 52 41 4D 53 5C 53 41 4D  \PROGRAMS\SAM
1915:0082  50 31 2E 6F 62 6A 01 00 00 00 0A 00 05  P1.objA.....E
```

Now, we know that the value of `var1` is represented as the fourth two-hexdigit number on the line labelled `1915:0041`.

Perform a similar analysis for your version of the program. (The addresses will probably be different, but the process is the same.) Now, step through the program, watching the value of `var1`. Note that it is highlighted at the first step after its value changes.

The `memory2` window operates identically to `memory1`.

21. We can also modify data in the data window. Find an appropriate display mode, so that you can locate the message "This data should be displayed when the program is run." Restart the program, and then type over the word "should," replacing it with "oughta." After running the program, select `View Output` from the `Windows` menu, and note that the message has been changed. Hit any case to return to the CodeView display. Of course, the source program is not changed. In fact, if you restart the program and run it again, "should" will reappear. Try it.

21. There is another way to keep an eye on the values of critical variables as a program progresses — by using a watch. Select the `command` window, and type `w? var1` at the prompt. Notice that `var1 = 0` appears in the `watch` window. Now restart the program (select `Restart` from the `Run` menu), and step through the program again. Notice that the value of `var1` changes when the instruction `add var1,1` is executed.

Watches may also be added and deleted by using the `Add Watch` and `Delete Watch` items from the `Data` menu. Try this with the `var1` data item.

Watches may be set for processor registers. Go to the `command` window once again, and type `w? ax` at the prompt. Now `ax =` appears in the watch window, and the value changes as the `ax` register does.

Consult the *Environment and Tools* manual [MAS93], pages 436 and 442–443, for more information on these commands.

22. All of the registers and flags of the processor may be monitored in the `reg` window. Restart the program and step through it again, this time observing the changes displayed in this window. Note that the entries which have just changed are highlighted.

23. From the `Options` menu, choose `Locals Options`, and then mark `Modules` by placing an `X` between the appropriate brackets. Notice that `var1` and `message` appear in the `locals` window, along with their addresses. (If the addresses do not appear, go back to the `Local Options` menu and select `Show Addresses`.) This is yet another way to view data objects, and to find out their addresses as well. The values will change dynamically as the program is traced.

24. A final use of the debugger which we will examine is the use of breakpoints. There are actually many types of breakpoints which may be set; see pages 339–341 of the *Environment and Tools* manual [MAS93]. We will try one simple kind — a location break. From the `Data` menu, choose `Set Breakpoint`. Now, in the grey box that appears, choose `Break When Expression has Changed`, and then type `var1` in the expression field. Now select `Restart` and then `Animate` from the run menu. The program will step automatically until it executes the statement `add var1,1`. It will stop at the next statement, but may be resumed by using `F8` or `F10` or by invoking `Animate` again.

    The `Edit Breakpoints` item from the `Data` menu allows one to both view the details of existing breakpoints and to modify them. Select this item and look at the information which you are provided.

25. There is one more debugger command worth learning about at this time. Restart the program, select the `command` window, and type a backslash, followed by a return. The output screen will appear. To get back to the debugger environment, hit any key.

26. To return to the Programmer's Workbench, select `Exit` from the `Files` menu of CodeView.

27. As a final exercise, print out the source (`.asm`) file, the listing (`.lst`) file, and the map (`.map`) file. Consult Appendix B for information on how to print the `.lst` file properly.

28. If you are using an EMBA-CF machine, make sure that you remove any source files from the `C:` drive, and that you have not left any "garbage" files around, except possibly in the `TMP` directory. Save your work to a floppy disk, as well as to the `M:` drive, for safety.

```
;; CS101 Software Assignment 1.
;; Author: Stephen J. Hegner.
;; Date Due: February 5, 1996.
;; Purpose: This is just a demonstration program, used to illustrate
;;          the use of MASM/PWB.
;;
               .MODEL small, farstack
               .STACK 100h

               .DATA
CR              equ     0dh
LF              equ     0ah
ENDTEXT         equ     '$'
display_string  equ 9h
return_code     equ 0
exit_to_dos     equ 4ch
dos_call        equ 21h
message         db 'This message should be displayed when the program is run.'
                db      CR, LF, ENDTEXT
var1            dw      0

               .CODE
main           proc
;; Identify the default data group.
               mov     ax, @data
               mov     ds, ax
;; Display the message on the screen.
               mov     dx, OFFSET message
               mov     ah, display_string
               int     dos_call
;; Increment a data item, just to see it work with the debugger.
               add     var1, 1
;; Return to DOS.
               mov     al, return_code
               mov     ah, exit_to_dos
               int     dos_call
main           endp

               end     main
```

Figure 1.1: Text of Program 1.

Software Assignment 2

Basic Input and Output: Part 1

2.1 Purpose

The purpose of this laboratory exercise is to help you gain an initial familiarity with writing assembly-language programs which interface to the MS-DOS operating system.

Bring your copies of the MASM software manuals to the laboratory session. You may need them, and copies will not be available in the laboratory.

2.2 Procedure

In this exercise, you are to write two distinct programs. Each program will write a prompt to the display, request simple input from the user, and then provide a response based upon that input. The programs will differ in the way that they obtain and echo that input.

2.2.1 The First Program

In this part of the exercise, you are to write a program which does the following.

1. The program displays the message

   ```
   Please type your name, followed by a return:
   ```

2. The cursor of the display remains on the same line as the above message, positioned with one space between it and the final colon. At this point, the system is waiting for input from the user.
3. The system reads the name which the user types, collecting the characters until a return is typed. These characters are echoed as the user types them.

4. The system then responds with the message `Hello`, followed by the name which the user typed in, followed by a comma, a space, and then `I hope that you are having a nice day.`, all on the same line. Thus, a sample dialog might be as follows. The text which the system prints is shown in typewriter font, while the text which the user types is shown in italics.

   ```
   Please type your name, followed by a return:   Charlie Brown
   Hello Charlie Brown, I hope that you are having a nice day.
   ```

5. The program then returns control to MS-DOS.

This program is very simple to write, but in so doing, you will learn about the basic MS-DOS calls for string-at-a-time input and output to the display. To obtain this input, use the `int 21h` MS-DOS call `0ah`. See pages 119–120 of the text by Irvine for details on how it is used [Irv93]. Basically, the command writes into a buffer, which may be declared as follows.

```
buffer          equu    $
buffer_size     db      maxsize
buffer_count    db      ?
buffer_data     db      maxsize dup (?)
```

The symbol `$` in the assembler evaluates to the address at which the next item will be assembled. Thus, in the above example, `buffer` is set to the starting address of the buffer. The buffer is, in effect, represented as a record. The byte `buffer_size` records the maximum number of data bytes in the buffer; `maxsize` is a constant which must be defined in the program. The next byte, named `buffer_count`, will be filled by the system routine with the number of characters actually read by the routine. (The final carriage return which the user types is not counted.) The symbol `?` is used in the value field to signal the assembler that the value need not be initialized. The final part of this "record" is in fact an array of bytes, with the first byte located at the address named by `buffer_data`. The number of bytes in the array is `maxsize`. The data entry `maxsize dup (?)` tells the assembler to replicate the value `?` (no value) `maxsize` times. In other words, the array is uninitialized.

Once the program has received the input, it must, among other things, write it out again. The `int 21h` call `09h` may be used for this purpose. This call writes out bytes until it encounters one whose value is the ASCII representation of `$`. Thus, to write out the string which has been read in, it is necessary to place a `$` at the end. The `$` must be placed in the first available byte in `buffer_data`, which is located at `buffer_data + value(buffer_count)`, with `value(buffer_count)` denoting the value (not the address) of the byte stored at offset `buffer_count` in the data segment.

A way to store a value into this dynamically computed address is needed. The most effective technique is that of *indexing.* An example will illustrate how this is done, assuming that `ENDTEXT` is a symbol whose value is `'$'`.

```
mov al, buffer_count
mov ah, 0
mov si, ax
mov buffer_data[si], ENDTEXT
```

The `si` register is used for indexing here. (The register `di` may also be used.) The value of the *index* is placed into `si`, and then the *indexed address* `buffer_data[si]` is used. We cannot use the instruction

```
mov si, buffer_count
```

directly because `si` is a 16-bit register and `buffer_count` is a byte value. (The sizes of the arguments in a `mov` instruction must match.) The rest of the process is very straightforward, and similar to that of the program of Software Assignment 1.

2.2.2 The Second Program

In the second program, instead of echoing the characters which the user types at the prompt, the program echos asterisks. Thus, a typical session with this new program might appear as follows.

```
Please type your name, followed by a return: *************
Hello Charlie Brown, I hope that you are having a nice day.
```

This sort of "disguised echoing" is used in the validation program which prompts you for your password when you log onto an EMBA-CF microcomputer. To achieve this behavior, the `int 21h` call `0ah` cannot be used. Rather, the `int 21h` call `08h`, which reads one character at a time, without echoing the character, should be used. The high level process may be described as shown in Figure 2.1.

An index register, such as `si`, should be used to store the `buffer pointer`. Using the same record structure as for the first program described above, the `buffer count` is just the byte stored at `buffer_count`.

To realize `if` and `repeat` constructs in assembly language, we make use of two operations. The first is the comparison operation `cmp`. This operation compares two quantities, and sets certain status flags in the flags register as a result. The second is the set of operations known as *jump* instructions. A transfer of execution to a new address is effected conditionally upon the value of a status flag. Here is an example.

```
DONE := false;
Repeat
  Read a character C without echo;
  If C is the return character
     then
       DONE := true;
     else
       If the buffer is full
          then
            DONE with error;
          else
            write an asterisk to the display;
            put C into the input buffer;
            increase the buffer count by one;
            move the buffer pointer ahead by one;
       end if;
  end if;
until DONE;
```

Figure 2.1: High-Level Description of Disguised Echoing.

```
loop:  <some computations>
       cmp    ax, 0
       jne    loop
```

The mnemonic `jne` stands for *jump on not equal*, and so this program fragment is operationally equivalent to the control expressed in the following pseudocode.

```
repeat
 <some computations>
until ax=0
```

Here `<some computations>` is just some set of instructions, whose exact nature is not important here. Of course, these instructions must eventually set the value in the `ax` register to zero if the loop is to terminate. Similarly,

```
        cmp    ax, 0
        jne    ahead
        <some computations>
ahead:  <more computations>
```

is operationally equivalent to

```
if ax=0
  then
    <some computations>
  end if;
<more computations>
```

The full array of jump instructions is given on pages 149–151 of the text by Irvine [Irv93].

Using these ideas, you should be able to realize, in assembly language, the high-level process outlined above.

Software Assignment 3

Basic Input and Output: Part 2

3.1 Purpose

The purpose of this laboratory exercise is to help you gain additional familiarity with writing assembly-language programs which interface to the MS-DOS operating system, with emphasis upon the use of indexing in simple linear buffers.

Bring your copies of the MASM software manuals to the laboratory session. You may need them, and copies will not be available in the laboratory.

3.2 Procedure

In this exercise, you will write two distinct programs. Since they will extend the programs of Software Assignment 2, they will be referred to as the *third program* and the *fourth program*, respectively. This write-up also makes use of the notation and ideas of the write-up of Software Assignment 2.

3.2.1 The Third Program

In the second program, in prompting for the name of the user, instead of echoing the characters which were typed, the program echoes an asterisk for each character typed. Thus, if your name is Calvin, and you type `Calvo^Hin`, with the `^H` denoting a backspace (to erase the erroneous `o`), all seven characters will be placed in the buffer, including the backspace character `^H` (which is 08 hexadecimal in the ASCII character set). In other words, the sequence of bytes in the data part of the buffer looks like

```
C  a  l  v  o  ^H  i  n
```

In the second program, this is not a problem, since the backspace character is displayed by the `int 21h` *display character* call `02h` as a backspace of the terminal cursor. The `o` is overwritten so quickly by the `i` that it is imperceptible to the human eye (unless the program is slowed down via stepping or animation).

For other uses of the buffer (one of which will be illustrated in the fourth program) however, it is far better to process the backspace characters "on the fly" as editing characters, so that the final buffer contents would be

```
C a l v i n
```

Simultaneously, the display of asterisks should reflect the number of characters in the buffer, and not the total number of characters typed. The critical sequence of values may be visualized as follows, with the underscore `_` representing the flashing cursor on the display screen.

Typed Input	Buffer Data	Display
`Calvo`	`C a l v o`	`*****_`
`Calvo^H`	`C a l v`	`****_`
`Calvo^Hi`	`C a l v i`	`*****_`

The *Buffer Data* column shows only the characters up to and including the position indicated by the buffer pointer. The `o` is not really erased or replaced by a space, but it is eventually overwritten by the `i`. To realize this, we must process the backspace character specially. A high-level description of the algorithm appears in Figure 3.1 on the following page. This description is an extension of that given for the second program in Software Assignment 2. Notice two things in particular.

1. When the buffer is empty, a backspace is ignored. This makes sense in terms of a user interface, since one cannot back up beyond the beginning of the input.
2. When a backspace is encountered and the buffer is not empty, the asterisk immediately to the left of the cursor must be "erased," and the cursor must be backed up one position. To do this, first echo a backspace (to back up the cursor), then echo a space (to replace the asterisk in the current cursor position with a space character), and then echo another backspace (to compensate for the cursor moving ahead one position due to writing the space character). This may seem a bit complex, but try it — you will see how it works.

3.2.2 The Fourth Program

In the fourth program, the user will be prompted not only for a name, but for a password as well. A sample dialog might begin as follows.

```
DONE := false;
Repeat
  Read a character C without echo;
  If C is the return character
     then
       DONE := true;
     else
       if C is a backspace
         then
           if buffer is not empty
              then
                decrease buffer count by one;
                move the buffer pointer back by one;
                echo a backspace;
                echo a space;
                echo a backspace;
           end if;
         else
           If the buffer is full
             then
               DONE with error;
             else
               write an asterisk to the display;
               put C into the input buffer;
               increase the buffer count by one;
               move the buffer pointer ahead by one;
           end if;
       end if;
  end if;
until DONE;
```

Figure 3.1: High-level description for processing input.

```
Please type your name, followed by a return:  Charlie Brown
Hello Charlie Brown, I hope that you are having a nice day.
Please type the secret password, followed by a return:  *****
```

This program echoes the user name, just as the first program did. (You may and should use your code from the first program for this part.) The program then prompts for a secret password, echoing asterisks as the candidate password is typed. The actual password is declared in the program using a `db` command, such as

```
passwd db '01234'
pwdsize equ $-passwd
```

The first line declares the password to be the byte sequence `01234`, and the second line reflects a little trick to compute the number of characters in the password. The password must contain at least one character, and no more than sixteen. The program should provide one of three possible responses, depending upon the user's response.

1. If the user types the correct password, followed by a return, the system responds with

   ```
   You gave the secret password.  You are so wise!
   ```

   and then returns to the operating system.

2. If the user types an incorrect password, followed by a return, the system responds with

   ```
   That is not the secret password.  You are so stupid!
   ```

   and then returns to the operating system.

3. If the user types a candidate password which is longer than the size of the buffer which is used to store it, the system responds with

```
The secret password is no longer than 16 characters. You are so stupid!
```

   and then returns to the operating system. It does this as soon as the candidate password is longer than sixteen characters, and does not wait for a return to be typed.

Otherwise, the program should not reject a candidate password until the user has typed a return. In other words, it should allow the user the option of backspacing to replace incorrect characters. The only exception is the case in which the user has typed in a candidate which is longer than sixteen characters, in which case the program should reject that candidate immediately.

To realize this final part, the candidate password is read into a buffer. After the user types a return, the contents of this buffer must be compared, character by character, to the actual password. This is done in two steps.

1. If the candidate password is either shorter or else longer than the actual password, the match fails immediately. This is checked by comparing the buffer count with the value of `pwdsize`, as declared above.

2. If the candidate password and the actual password have the same length, then the characters are compared, byte-by-byte. This is realized via a simple loop, using an index register.

Part II

Program Organization and Development Techniques

Software Assignment 4

Procedures in Assembly Language

4.1 Purpose

The purpose of this laboratory exercise is to introduce the concept of procedures in the assembly language environment.

4.2 Procedure

In most high-level languages, one can write procedures which are reusable definitions of algorithm fragments. It is possible to write and to call procedures in assembly language as well, although, as you might expect, far less is done automatically than in high-level languages, and one must pay attention to a substantial amount of detail. There are, in fact, many facets to procedure calls in assembly language. First and foremost, there is the notion of a procedure versus a macro. Roughly speaking, a *procedure* is a single block of instructions which may be entered and exited many times, while a *macro* is a declaration of a set of instructions which may be copied automatically to several distinct places in the program. In this exercise, we will deal solely with procedures. Software Assignments 7 and 8 will introduce programs with macros.

Within the context of procedures, there are at least two fundamental issues: the way in which parameters are passed, and whether the procedure is separately assembled and linked or just included in the calling program. In this exercise, we will write procedures which (a) pass their parameters in a very basic way, via registers; and (b) which are included in the calling program, so that issues of separate assembly and linking are not considered. In later exercises, we will look at relaxing these constraints.

You will modify the program which you wrote for Software Assignment 3. This will be done in two steps.

4.2.1 Background — The Form of Procedures

The basic syntax for a procedure declaration is as follows:

```
proc_name        proc
       <procedure code>
proc_name        endp
```

while the basic syntax for procedure invocation is

```
call      proc_name
```

In the above, `proc_name` is the name of the procedure. Let us illustrate with a simple example. A common `int 21h` DOS call is `08h`, which gets a character from the keyboard without echoing it to the display. Let us write a procedure call to realize this function. First of all, we need to make some decisions about how parameters are passed. This procedure has no input parameters, and it has one output parameter; namely, the character which has been read. Let us adopt the convention that the character will be returned in the `al` register. Here is a first try at such a procedure.

```
get_char_necho  proc
;; Get a character from the keyboard without echoing to the display.
;; input: none
;; output: al <- character typed.
                mov  ah,08h
                int  21h
                ret
get_char_necho  endp
```

Note that the `ret` command is used to return control of the procedure to the calling program. Often, it is the last statement (lexically) of the procedure, but it need not be. Beyond that, there is relatively little which is absolutely essential to know. The handling of the return address is done automatically — the `call` command pushes the return address onto the stack, and the `return` command pops it off. To use this procedure in a program, we need simply write a line such as

```
call get_char_necho
```

and the `al` register will contain the character which has been typed, after execution of the procedure.

There is a weakness in the procedure, as it is written, however. Namely, it uses and overwrites the `ah` register, without warning the user. Thus, if a call to the procedure is executed in a program which expects the `ah` register not to change, the program may not work. As a matter of good practice, it is prudent to protect the contents of registers which the procedure uses. The way to do this is to save the values of these registers on the stack when the procedure is called, and to restore them when it finishes. For this procedure, this becomes a bit complicated. An "obvious" solution is the following.

```
get_char_necho  proc
;; Get a character from the keyboard without echoing to the display.
;; input: none
;; output: al <- character typed.
                push ah
                mov  ah,08h
                int  21h
                pop  ah
                ret
get_char_necho  endp
```

However, this is not legal assembly language. The reason is that the arguments to `push` and `pop` instructions must be *words*, and not bytes. However, if we replace `push ah` and `pop ah` with `push ax` and `pop ax`, respectively, then the return value contained in `al` will be lost. The solution is to use a second register, as follows.

```
get_char_necho  proc
;; input: none
;; output: al <- character typed.
                push bx
                push ax
                mov  ah,08h
                int  21h
                pop  bx
                mov  ah,bh
                pop  bx
                ret
get_char_necho  endp
```

Notice that neither the contents of `bx` nor the contents of `ah` are altered by this procedure.

4.2.2 Program — Part 1

In this part, you are to write procedures which exhibit the following behavior.

1. A procedure named `display_string` which uses the `int 21h` call `09h` to display a string on the monitor.
2. A procedure named `get_string` which uses the `int 21h` call `0ah` to get a string from the keyboard.
3. A procedure named `display_char` which uses the `int 21h` call `02h` to display a character on the monitor.
4. A procedure named `exit_to_dos` which uses the `int 21h` call `04ch` to return control to the operating system.

Then, you are rewrite the last program of Software Assignment 3 (which reads both the name and the password, and then verifies the correctness of the password), using these four procedures, and the procedure `get_char_necho` given above. The main part of your program must not make any direct `int 21h` calls; all of these services **must** be realized via procedure calls.

There is one further point which needs to be mentioned. All of the procedures are declared within the same `.CODE` segment of the program. One of these procedures is the main program, which is executed when the program begins, and all of the others are procedures to be called. The main program is indicated by the statement

```
end  main_prog_name
```

with `main_prog_name` identifying the name of the procedure which is the main program.

4.2.3 Program — Part 2

In this part, you will write a procedure with somewhat more complex parameter passing conventions. Consider the problem of comparing the contents of two buffers, as was necessary in the last program of Software Assignment 3 during password verification. For consistency, we assume that we have two buffers, declared as follows.

```
b1        equ  $
b1_size   db  bufmax
b1_count  db  ?
b1_data   db  bufmax dup (?)
b2        equ  $
b2_size   db  bufmax
b2_count  db  ?
b2_data   db  bufmax dup (?)
```

We wish to write a procedure which will compare the two buffers, and report back as to whether they have the same contents. The procedure must be able to compare two *arbitrary* buffers of the above form, and not just `b1` and `b2`. Thus, the arguments `b1` and `b2` must be passed to the procedure as parameters. A high-level pseudocode description of this procedure is shown in Figure 4.1 on the following page..

To realize this procedure in assembly language, several points must be considered. First of all, it is not possible to pass the actual buffers to the procedure in registers. Rather, the starting address of the buffer is passed to the procedure in registers. This is a low-level realization of the so-called *pass-by-reference* mode of parameter transmission (called `var` parameters in Pascal, and realized by passing pointers in C.) Let us adopt the convention that `cx` will point to the first buffer, and `dx` will point to the second. (That is, they will contain the offset addresses for these buffers.) Second, the return

```
procedure compare_buffers(buf1, buf2: buffer): Boolean;
  integer: index;
  Boolean: flag;
  if buf1.count <> buf2.count
      then
        flag := false;
      else
        index := buf1.count;
        flag := true;
        while ((index >= 0) and flag) do
           if buf1.data[index] = buf2.data[index]
             then
               index := index-1;
             else
               flag := false;
            end if;
   end if;
   return flag;
 end procedure.
```

Figure 4.1: Pseudocode for `compare_buffers`

value must be passed back in a register. Let us adopt the convention that this value will be returned in `al`, with true represented by `01h` and false by `00h`. An example invocation of this procedure would then be as follows.

```
mov  cx, OFFSET b1
mov  dx, OFFSET b2
call compare_buffers
cmp  al, 01h
je   bufs_equal
jmp  bufs_not_equal
```

A useful declaration of the procedure is as follows.

```
compare_buffers proc
;; Comments on how to invoke the procedure.
buf1_ptr   equ   cx
buf2_ptr   equ   dx
   <procedure code>
compare_buffers endp
```

The `equ` statements provide explicit and meaningful names for the data objects passed to the procedure. Most of the code is relatively straightforward, and similar to that which was used in Software Assignment 3. However, there are a few points for which may give some difficulty. The most salient point is addressing within the buffers. In Software Assignment 3, to access the i^{th} position in `buf1`, you probably put i into the `si` register, and used something like the following

```
    cmp   ah,buf1_data[si]
```

In the procedure, it is not so straightforward. We do not have access to the names; all we have is an address. We might try

```
    cmp ah,buf1_ptr[si+2]
```

which is equivalent to

```
    cmp ah,cx[si+2]
```

Unfortunately, the above line is not legal assembly code. The term `cx[si+2]` makes no sense, since `cx` refers to the contents of the `cx` register, and not the memory location whose offset within the data segment is the value of `cx`. Fortunately, the assembler almost allows us to write `[cx]`, which means the location indexed by register `cx`, so that

```
    cmp  ah,[cx][si+2]
```

is *almost* what we want. Unfortunately, the above is not legal assembly code either. Only certain registers may be used for indexing in this way, and `cx` is not one of them. The only one which may be used safely in this situation is `bx`. Thus, we may write

```
    mov  bx,buf1_ptr
    cmp  ah,[bx][si+2]
```

to compare the byte in `ah` to the `si`th data byte in the buffer `buf1`. Equivalently, we may write

```
    mov  bx,buf1_ptr
    cmp  ah,[bx+si+2]
```

In studying this code fragment, recall that the data part of the buffer begins with the third byte. The `+2` term is used to skip to the `data` field, which we cannot easily refer to by name here.

With these tools in hand, it should be relatively straightforward to write this program.

Make sure that your procedure restores the values of registers which it uses. Once you have written and tested your procedure, integrate it into the program which you developed in Part 1 above. In other words, the part of the program which tests for the validity of the password must use the procedure `compare_buffers` which you have written. Thus, your final program must use procedures not only for all of the `int 21h` calls, but for buffer comparison as well. Also, for this program, you may enter the actual password and its size manually into a buffer; you need not do this automatically.

One final word of advice is in order. Because these procedures are all declared within the same module, they share a common name space. Thus a name (*e.g.*, `buffer`) used within a procedure is also visible within any other procedure. Be careful to avoid name collisions.

Software Assignment 5

Multiple Source Files and the Make Facility

5.1 Purpose

The purpose of this laboratory exercise is to introduce the ideas of placing procedures in separate files, and of separately assembling these files using the `make` facility.

5.2 Procedure

In Software Assignment 4, the idea of a *procedure* in assembly language was introduced. Many of the procedures developed in that assignment, including the DOS `int 21h` calls, could be reused in many different programs. However, there are several disadvantages to the approach developed in that assignment, among them the following.

1. The code for the procedures must be inserted manually into each program in which the procedures are used.
2. The names used in the code for the procedures are global to the entire program. Therefore, care must be taken to ensure that the same name is not used more than once.

We will now take a look at a few techniques which overcome these difficulties. In so doing, the program of Software Assignment 4 will be modified.

5.2.1 Using Simple `INCLUDE` Directives

The semantics of the `INCLUDE` directive are very simple. If a statement of the form

```
INCLUDE support_file.inc
```

is placed in an assembly language program, then the contents of the file `support_file.inc` are included in the program, *exactly at the point at which the* `INCLUDE` *statement occurs.* This provides an extremely simple way of packaging procedures or groups of procedures so that they do not need to be copied explicitly into each program in which they are used. In the first part of this software assignment, you will try out this technique.

Modify a copy of your program of Software Assignment 4 as follows.[1] Call this copy `soft5a.asm`. Create a file named `doscalls.inc`. From the copy of your program for Software Assignment 4, move the lines which define the five procedures `get_char_necho`, `display_string`, `get_string`, `display_char`, and `exit_to_dos`, to the new file `doscalls.inc`. In the old file, at exactly the position where these procedures once were, put the single line

```
    INCLUDE doscalls.inc
```

Now, repeat this process for another new file, named `compbufs.inc`. That is, delete the definition of the procedure `compare_buffers` from the old file and place it in the new file. Place the single line

```
    INCLUDE compbufs.inc
```

at the exact place where the definition of `compare_buffers` once was. These new files must be placed in a directory which MASM will find. The best place is in the same directory as the program.

Now, assemble, load, and run the new program. It should work! To the assembler and linker, it is exactly the same as the old one. If it does not work, check carefully to make sure that you have not made any typographical mistakes.

After you have run the program, go into the CodeView debugger for your program `soft5a.asm`. Scroll through the `Source1` window. Notice that only the text for the main program, `soft5a.asm`, is visible. Now step through the program. Note that, when the program executes statements in the file `doscalls.inc`, the window `Source1` automatically switches to that file.

What does one do, though, when it is desired to set a breakpoint in an included file? Proceed as follows. Restart your program. (Select `Restart` from the `Run` menu.) Now open the window `Source2` (from the `Windows` menu), and select the file `doscalls.inc` by using the `Open Source` item from the `File` menu. Pick a line in this file which will be executed; say a line in the `exit_to_dos` procedure. Set a breakpoint on this line by placing the cursor on it and double-clicking the left mouse button. The text of the line will become highlighted. Now, return to the `Source1` window by clicking the left

[1] Always keep the programs which you submitted for the previous software assignments intact. This is your insurance in case a question regarding the grading arises.

mouse button in it. Hit the F5 key to run the program. Notice that the breakpoint is captured in the Source1 window, even though it was set in Source2. Breakpoints are set in the program, and are not localized to a particular window.

5.2.2 Separate Assembly and the Make Utility

While the technique just described has its place, it is somewhat unsatisfactory for the management of software libraries and multi-component programs, for a number of reasons, some of which are listed below.

1. Since the technique is functionally equivalent to having one large program, all identifiers are global to the whole program. As a program becomes larger, greater care must be taken to ensure that the same name is not inadvertently used for two or more purposes. This problem becomes particularly acute when a number of libraries are all included in a program; the possibility of conflicts between procedures from different libraries becomes great.
2. Data objects are not protected. Any part of the program can modify any data object. In modern terminology, *information hiding* is impossible.
3. Whenever any changes are made to the program, the whole thing must be re-assembled. For large programs, this can be quite time consuming.

Fortunately, there is a simple technique which overcomes all of these difficulties. The idea is to partition the program into *separately assembled modules*. Each such module has its own local data area and its own code area. The same identifier may be used in each module, as such names are local by default. Furthermore, these modules may be assembled and saved separately. Thus, when a change is made to one of them, only that module need be re-assembled.

Of course, if the modules are to communicate. not everything can be local. Data objects which are to be shared must be declared to be so explicitly. There is a number of ways to accomplish this sharing; Section 8.4 the textbook by Irvine [Irv93] describes the EXTERN/PUBLIC approach which was used in version 5 of MASM, and which is still available in version 6. Chapter 8 of the MASM *Programmer's Guide* [MAS92b] describes a number of techniques; we will use the most modern — the EXTERNDEF approach. For a data object which is to be shared, an EXTERNDEF statement is placed in each file which will share the object, including the file in which it is defined. Thus, to share the procedure exit_to_dos, the statement

```
EXTERNDEF exit_to_dos:near
```

is placed in both the file containing the definition of the procedure, and in every file which uses it. (For now, include the keyword near with every such procedure declaration. We will discuss its significance later in the course.)

Now, let us put this technique into practice. A program source file `soft5b.asm` will be created. It will be linked to two other programs, `doscalls.asm` and `compbufs.asm`, which we will also create.

To start, copy the contents of your `doscalls.inc` file to a new file named `doscalls.asm`, and then modify the new file as follows. Since this new file will be a separately assemblable entity, it will need some header material. Here is what it should look like.

```
;; Documentation for the package -- IMPORTANT
        .MODEL small,farstack
        INCLUDE doscalls.dec
        .CODE
   <Definitions of the procedures (from the old file)>
        end
```

Notice that the file ends with an `end` declaration. **Every** `.asm` file **must** end with an `end` declaration. If it is forgotten, a fatal error message will be generated.

Rather than enter the `EXTERNDEF` statements manually, it is most convenient to place all of them, for a given module, into an `INCLUDE`able file. Let use adopt the convention that, for a file of the form `name.asm`, the associated `INCLUDE`able file is named `name.dec`. Thus, the file `doscalls.dec` will include the `EXTERNDEF` declarations which are necessary for the use of this package. Here is the contents of such a file for this application.

```
EXTERNDEF       display_string:near
EXTERNDEF       get_string:near
EXTERNDEF       get_char_necho:near
EXTERNDEF       display_char:near
EXTERNDEF       exit_to_dos:near
```

Now, repeat this process for the existing file `compbufs.inc`, creating new files `compbufs.asm` and `compbufs.dec`.

The main program must also be modified to reflect these changes. Copy the program file `soft5a.asm` to a new file `soft5b.asm`. Then edit `soft5b.asm` as follows.

1. Delete the lines

   ```
   INCLUDE doscalls.inc
   INCLUDE compbufs.inc
   ```

   since these files will not be used.

2. Insert the lines

   ```
   INCLUDE doscalls.dec
   INCLUDE compbufs.dec
   ```

at the beginning of the program, right after the `.STACK` directive. These lines communicate the critical linking information.

The programs are now ready to go. However, to assemble and link them, we must communicate to PWB that all of these procedures are involved. One way (which is not recommended), is to do this manually, as described in Section 4.2 of the text by Irvine [Irv93]. The preferred approach is to use the make utility of PWB. This utility allows you to define a *project*, and to maintain information about which files are used in which project. Here is how to proceed.

1. From the `Project` menu, choose `New Project`.
2. At the prompt, enter the file name `T:\soft5.mak` as the name of the make file. (Here `T:\` is the directory in which you are placing your work for this course.)
3. When the next window appears, leave the `Runtime Support` choice to be `None`, but select `DOS EXE` for the `Project Template` by highlighting that option, and then select `<OK>`. When the next window appears, select `<OK>` again.
4. When the `Edit Project` window pops up, you need to select the files which comprise the project. You can do this either by double clicking on their names, or else by typing in the names. You need to select `soft5b.asm`, `doscalls.asm`, and `compbufs.asm`. If you inadvertently include the wrong file, use the `Add/Delete` option to the right of the box to remove it. (Select only the `.asm` files; do not select the `.dec` files. The latter files are included by the `INCLUDE` directives in the program modules, and are not separately assembled.)
5. Once you have selected the correct files, click on `Save List`, and you will be returned to the top level of PWB.
6. If you need to change anything, use the `Edit Project` item from the `Project` menu.

Now everything is set up to use the project. You can just select `Build: soft5.exe` from the `Project` menu, and the whole ensemble should be assembled and linked. It should run, and behave, exactly as did the program of Software Assignment 4.

Once you have things working, open the file `doscalls.asm` and save it, just to tell the system that it has been changed. Now select `Build:soft5.exe` again, and look at the results screen. Notice that only `doscalls.asm` has been re-assembled. The make utility keeps track of the last update time of all files, and only re-assembles those which are out-of-date. If you wish to force the re-assembly of all files for the project, use the `Rebuild All` item from the `Project` menu.

Object files, rather than assembled files, may also be included in the project description. Select `Edit Project` from the `Project` menu, and then delete `doscalls.asm` and `compbufs.asm` from the list of files, replacing them with `doscalls.obj` and `compbufs.obj`, respectively. Run `Rebuild All` and note that things still work.

There is one further point of which you should be aware. Whenever PWB is exited, all projects are closed. When you re-enter PWB, you need to re-open the project. This can be a bit confusing. Suppose, for example, that you were working on this assignment, and exited with `soft5b.asm` as the active edit window. When you restart, PWB and go to the `Project` menu, you will be invited to compile, build, *etc.*, the file `soft5b.asm`. This is not what you want!! Resist the urge!! Instead, select `Open Project` from the `Project` menu, and give `T:\soft5.mak` as the project name. Once you have multiple source files, you must work with projects. Of course, if you do inadvertently try to rebuild `soft5b.asm`, you will be greeted with a host of messages indicating unresolved references. This causes no harm; just open the project and rebuild it.

Part III

Selected Software Projects

Software Assignment 6

Number Representation and Integer Arithmetic

6.1 Purpose

The purpose of this laboratory exercise is twofold. The first is to gain insight into the techniques necessary to translate integer numbers between internal format and display format. The second is to gain additional insight into the development of programs containing separately assembled procedures.

6.2 Procedure

When using a high-level language, one often takes for granted the process of reading in or printing out a number. However, there is a significant amount of processing which takes place during this process. These routines are not a part of assembly, language; they must be supplied by the programmer, or else taken from a software library.

Chapter 8 of the textbook by Irvine [Irv93] includes a development of such routines. However, there are substantial differences, both in method and in generality, between the programs of Irvine and those required for this assignment. Thus, while it will be helpful to refer to the programs developed by Irvine, they are not solutions to this exercise.

6.2.1 Representation of Number in Various Bases

In this exercise, software will be developed which has the capability of displaying numbers in any base between 2 and 36 inclusive. Using the representation of numbers in hexadecimal as an example, it is a straightforward generalization to use the entire alphabet for numbers between 10 and 35 inclusive. That is, `A` represents 10, `B` represents

11, ..., `F` represents 15, `G` represents 16, `H` represents 17, ..., `Y` represents 34, and `Z` represents 35. Of course, only those letters which are necessary for a given base are used. Thus, for a number in base 25, the digits 0 through 9, and the letters `A` through `O` are used, while for number is base 8, only the digits 0 through 7 are used. Here are a few examples, to crystallize the concept.

(i) The string 1574 represents, in base 8, the number $1\times 8^3+5\times 8^2+7\times 8+4 = 892$.

(ii) The string 1JM3 represents, in base 25, the number $1 \times 25^3 + 19 \times 25^2 + 22 \times 25 + 3 = 28053$.

Clearly, the principle of this idea may be extended beyond 36, but this value has been chosen as a convenient upper limit on the size of the base because there are exactly 36 digits and upper case letters. In all of the software for this assignment, lower-case letters and upper-case letters will represent the same value. Thus, both `P` and `p` represent 25.

6.2.2 The Primary Package to Be Developed

The primary goal of this assignment is to write two procedures, one which reads a number typed on the keyboard and translates it into internal representation, and one which takes an internal number and displays it on the monitor. Here are more detailed descriptions of their behavior.

`display_number` This procedure operates as follows. There are two input quantities. The first is a sixteen-bit quantity, passed in the register `ax`, and interpreted as a signed integer. (Thus, the value of the number is between −32768 and 32767 inclusive.) This is the number whose value is to be displayed. The second input value is an eight-bit quantity, passed in the register `bl`. This value is interpreted as the base of the number to be displayed, and must be between the values 2 and 36, inclusive. The output goes to the display; the representation of the number in `ax`, in the base given in `bl`, is written to the display. Shown below are some examples of the behavior of this procedure. (All register values are shown in internal hexadecimal format.)

Inputs		**Outputs**
Register `ax`	Register `bl`	Display
`00F3`	`10`	`F3`
`00F3`	`0A`	`243`
`00F3`	`14`	`C3`
`00F3`	`02`	`11110011`
`FF0C`	`0A`	`-243`
`0044`	`24`	`1W`

If the number is positive or zero, no sign is written. However, if the number is negative, the string written begins with a minus sign. Leading zeros should not be displayed; however the number zero should be displayed as a single 0. The output of this procedure will not contain carriage returns, line feeds, or any other padding. Only the number will be written to the display. This procedure corresponds to Irvine's `WRITEINT` [Irv93, Sec. 8.2], but differs in many details, not the least of which is the ability to display numbers in any of thirty-five bases.

If the value passed in register `bl` is not between 2 and 36, inclusive, then the value of the number displayed is not specified. However, the program must display something; it must not crash or lock up or display a screenful of garbage, even if the value of the base is not within range.

`get_number` This procedure reads a number from the keyboard. Other than the string which is typed on the keyboard, there is one input quantity; the base in which the input number will be presented is given in the register `bl`. This base, represented in internal format (as an unsigned number), is assumed to be between 2 and 36, inclusive. The procedure reads a string from the keyboard, interprets it as a signed number in the base given in `bl`, and places the signed internal representation in the register `ax`. Shown below are some examples of the behavior of this procedure. (All register values are shown in internal hexadecimal format.)

Inputs		**Outputs**
Display	Register `bl`	Register `ax`
`1234`	`0A`	`04D2`
`-1234`	`0A`	`FB2E`
`+1234`	`0A`	`04D2`
`1234`	`08`	`029C`
`1234`	`19`	`423A`
`-000`	`24`	`0000`
`000`	`24`	`0000`

The input string may begin with an optional sign (`'+'` or `'-'`), and must consist entirely of legal "digits" for the input base thereafter. It must allow the user to edit the input string by using backspacing. The string is only accepted and processed once a return is typed. This procedure corresponds to Irvine's `READINT` [Irv93, Sec. 8.3], but differs in many details, including that it must handle any base between 2 and 26 inclusive.

If the value passed in register `bl` is not between 2 and 36, inclusive, then the value of the number placed in register `ax` is not specified. However, the program must display something; it must not crash or lock up, even if the value of the base is not within range.

These procedures are to be packaged in in a module named `numio.asm`. The procedures are declared in `EXTERNDEF` statements, so that they are available to other procedures. They are to be the *only* procedures defined in `numio.asm` which are available to other programs. Of course, `numio.asm` may use procedures from other packages which you have written; indeed, you will find it convenient to use the package `doscalls.asm` which you developed in an earlier assignment.

6.2.3 The Demonstration Program

To demonstrate the behavior of the `numio.asm` package, you are to write a demonstration program which prompts for an input base, a number in that input base, and an output base, and translates the number from the input base to the output base. Figure 6.1 below provides a sample of a session with this program.

```
Welcome to the base translation program.
Type a ^C to exit the program at any time.

Base of input number, 2 <= base <= 36: 10
Input number, -32768 <= n <= 32767: 100
Desired base of result: 16
The number in base 16 is: 64

Base of input number, 2 <= base <= 36: 8
Input number, -32768 <= n <= 32767: -100
Desired base of result: 2
The number in base 2 is: -1000000

Base of input number, 2 <= base <= 36: 30
Input number, -32768 <= n <= 32767: pq
Desired base of result: 10
The number in base 10 is: 776

Base of input number, 2 <= base <= 36: 16
Input number, -32768 <= n <= 32767: +400
Desired base of result: 36
The number in base 36 is: SG
```

Figure 6.1: An example run of the program.

Once the main routines `get_number` and `display_number` have been developed, it is a simple matter to write this demonstration program, which is largely a formatted interface to the routines of the `numio.asm` package . Note that the bases of the input and output numbers are entered as *decimal* values.

6.2.4 Support Programs and Development Hints

The complexity of this program is substantially greater than that of the previous software exercises for this course. Thus, it is highly inadvisable just to jump in and start coding. Rather, it is strongly recommended that you plan the program, by first describing how it will operate in a high-level pseudo-language. It is also extremely helpful to break down the design of the procedures `get_number` and `display_number` into further, internal procedures. Particularly, for this assignment, it is required that the package `numio.asm` contain two additional procedures, named `char_to_base` and `base_to_char`. These two procedures are **not** to be external. Rather, they are to be internal to the package, and used solely as support routines for the external procedures. Here is further description of their behavior.

`char_to_base` This support routine converts a single character code to an internal format number. The input character is contained in the register `dl`, and the output number is also returned in the register `dl`. Figure 6.2 provides a high-level description of how this procedure behaves.

```
if ord('0') <= ord(input_char) <= ord('9')
   then output_number := ord(input_char) - ord('0');
else if ord('A') <= ord(input_char) <= ord('Z')
   then output_number := ord(input_char) - ord('A') + 10;
else if ord('a') <= ord(input_char) <= ord('z')
   then output_number := ord(input_char) - ord('a') + 10;
else error_in_input;
end if;
```

Figure 6.2: High-level description of character-to-base translation.

The notation `ord(-)` is taken from Pascal, and denotes the equivalent numerical value of the character code of the argument. (In assembly language, this is what you get automatically. You do not need to write a function named `ord`.) This procedure takes advantage of the fortunate fact that the digits between `0` and `9`, the upper-case letters between `A` and `Z`, and the lower-case letters between `a` and `z`, are each represented contiguously in the ASCII character set.

Irvine [Irv93, Sec. 8.1] uses the `xlat` instruction, together with a translation table, as an aid in realizing this procedure. You must not use `xlat`. Rather, you should implement the translation algorithm directly, as outlined in the pseudocode of Figure 6.2. **You will lose points if you use `xlat`.**

`base_to_char` This procedure is the inverse of `char_to_base`. It converts a number between 0 and 35 inclusive to the equivalent character code, using upper-case letters for values between 10 and 35, inclusive. The input value is contained in the register dl, and the output character is also placed in register dl. Figure 6.3 gives a high-level description of how this procedure behaves.

```
if 0 <= input_number <= 9
   then output_character := chr(input_number + ord('0'));
else if 10 <= input_number <= 35
   then output_character := chr(input_number - 10 + ord('A'));
else error_in_input;
end if;
```

Figure 6.3: High-level description of base-to-character translation.

The Pascal notation `chr(-)` is used to denote the character code corresponding to a given character. Again, in assembly language this translation is implicit, and need not be explicitly coded.

As with `char_to_base`, Irvine uses the `xlat` instruction, together with a translation table, as an aid in realizing this procedure. Again, you must not use `xlat`. Rather, you should implement the translation algorithm directly, as outlined in the pseudocode of Figure 6.3. **You will lose points if you use `xlat`.**

When writing larger programs, it is essential to test small routines, one-by-one. An easy way to do this is to write a test routine, and then to let the debugger help you see how it is working. Figure 6.4 provides an example of a test routine for `char_to_base`.

The two `INCLUDE` statements are the standard ones for the libraries being developed. The `EXTERNDEF` statement is temporary, and provides a way for the test program to interface to the routine to be tested. An identical statement is **temporarily** included in the file `numio.asm`.

A key feature of this simple program is that it is indeed simple. It took only minutes to write. It does not do any output. Rather, to observe the behavior of the procedure, one uses CodeView and examines the register display as the program is stepped. To test the program on another input, it is not necessary to re-edit and re-assemble. Rather, this can be done right in CodeView. Just step the program until the line `mov dl,'c'` is highlighted. Then, switch to the command window and type an `a`, followed by a return. This is the command to begin assembly. Type `mov dl,'D'`, and then type two returns. (After the first return, you may type additional instructions to be assembled. The second return tells CodeView that you are done with the assembly specification.) Now, when you step through the program, the instruction executed will be `mov dl,'D'`, and not `mov dl,'c'`. This only applies for the current stepping session. Once you restart the program, it reverts to the original assembled code. In this way, you can try several different input parameters without the need to exit CodeView

```
;; CS101 Software Assignment 6.
;; Test stub.
;; Author: Stephen J. Hegner.
        .MODEL small, farstack
        .STACK 1000h
        INCLUDE doscalls.dec
        INCLUDE numio.dec
        EXTERNDEF char_to_base:near

        .DATA

        .CODE
main    proc
        mov     ax,@data
        mov     ds,ax
        mov     dl,'c'
        call    char_to_base
        call    exit_to_dos

main    endp
        end     main
```

Figure 6.4: A test routine for `char_to_base`.

and re-assemble. It is strongly suggested that you employ this technique for testing all of your routines.

Once you have written and tested the routines `char_to_base` and `base_to_char`, it is time to work on the main routines, `get_number` and `display_number`. Here are the key points of difference, compared to similar routines found in the text of Irvine [Irv93], which you should bear in mind.

(i) In `get_number`, you should not read and process the characters on the fly. To do so deprives the user of the ability to do simple editing of the input. Rather, your procedure should start by invoking the procedure `get_string`, which you have already written for the package `doscalls.asm`. The buffer which you use to store the input string should be local to the `numio.asm` module. That is, it should be declared in that module, and should be internal to it. There is no reason for users of `get_number` to have access to this buffer. It is an internal data structure, local to the procedure.

(ii) In `get_number`, you must allow for the possibility that the user will type a `+` or a `-` at the beginning of the number. This is completely straightforward to handle.

(iii) In Irvine's `READINT` [Irv93, Sec. 8.3], character-to-number translation is performed right in the procedure. In your routine `get_number`, you will invoke the utility `char_to_base` which you have already written. Similarly, your routine `display_number` will invoke the routine `base_to_char`.

(iv) In developing `display_number`, use the stack to reverse the order of the computed digits. The natural algorithm for producing the character-by-character equivalent of a binary number will produce the digits from right-to-left (*i.e.*, least-significant first). However, you must print them left-to-right (*i.e.*, most-significant first). To accomplish this, push the digits onto the stack as you compute them, and pop them off as you print them. If you use the buffering technique of Irvine instead, you will lose some credit. **You must use the stack here.**

(v) It is required that you use a make file to manage your project. This technique was introduced in Software Assignment 5.

Software Assignment 7

A UNIX-Style Directory Listing Program: Part 1

7.1 Purpose

This assignment has multiple purposes.

(1) To gain some practice in the use of elementary assembler macros.

(2) To gain some experience in the use of bit-wise operations to extract data from packed representations.

(3) To gain further practice with DOS calls, particularly ones that extract information about the environment in which the program runs.

7.2 Procedure

7.2.1 The Overall Programming Project

The final result of this software assignment, together with Software Assignment 8, will be an assembly-language program which produces a formatted file listing, in the general spirit of the UNIX `ls -l` command, although it is not as full featured. Figure 7.1 is an example of the actual operation of this program, in response to the command `ls101 d:\testdir\*.*` on a particular system. The first column of entries in Figure 7.1 for each file gives the values for each of five attributes. Their meanings are given in Figure 7.2.

With the exception of the directory bit, there is nothing particularly sacred about these status bits. For a given file, their values may be changed by using the DOS `attrib` command. Of course, the settings of some of these bits determine the access rights on the file. Consult DOS documentation for more details.

```
w..d.      0 10/23/95 00:28:58 - .
w..d.      0 10/23/95 00:28:58 - ..
r...m  56081 01/26/94 07:00:00 - COMMAND.COM
w...m    318 09/27/95 01:06:24 - AUTOEXEC.BAT
rhs..  27880 10/23/95 00:32:24 - IBMBIO.COM
w..d.      0 10/23/95 00:33:08 - FOODIR
```

Figure 7.1: Sample output the program of Software Assignments 7 and 8.

r/w The `r/w` attribute tells whether or not the file is writable. If a `w` is shown, it is writable. If an `r` is shown, it is read-only.

h The `h` attribute tells whether or not the file is *hidden*. If a `h` is shown, it is hidden. If a dot is shown, it is not hidden.

s The `s` attribute tells whether or not the file is a *system file*. If an `s` is shown, it is a system file. If a dot is shown, it is not a system file.

d The `d` attribute tells whether or not the file is a directory. If a `d` is shown, it is a directory. If a dot is shown, it is an ordinary file.

m The `m` attribute (called the *archive attribute*), tells whether or not the file has the *modification bit* set. If an `m` is shown, then the file has the modification bit set. If a dot is shown, this bit is not set.

Figure 7.2: Meaning of the file attribute bits.

The second column of entries in Figure 7.1 gives the size of the file, in bytes. A directory always has `0` in this column, regardless of how much space it actually uses.

The third and fourth columns give the date and time of the last modification of the file, respectively, in `mm/dd/yy hh:mm:ss` format. Finally, after a purely cosmetic "`-`", the name of the file is given.

Development of this project is divided into two stages, which together comprise Software Assignments 7 and 8, respectively. In Software Assignment 7, you will develop a program which generates one line of such a listing of files, while taking the argument from a string declared inside the program. In Software Assignment 8, you will extend this program, by enabling it to list all of the files satisfying a certain constraint, and by enabling it to take the constraint specification as an argument on the command line.

Section 11.1 of the textbook by Irvine [Irv93] gives an overview of the DOS file system. Although this handout is reasonably self-contained, you should also read

Section 11.1 of the textbook in order to obtain an overview from an alternative point of view. Also, Section 11.8 of the textbook by Irvine contains a program which displays file names and dates. Although this program has quite a bit in common with the programs of these assignments, it is sufficiently different that you will be far better off if you write your own program, rather than trying to adapt the program of Irvine. Nonetheless, you should study Figure 11.9 of the textbook to see how he accomplishes similar tasks.

7.2.2 Using `Find First Matching File` to Retrieve File Parameters

For each file, the DOS operating system maintains, in an internal format, all of the information necessary to build a line of the program described above. To access this information for a specific file, the command to use is function `4Eh` of `int 21h`. That is, put the value `4Eh` into register `ah`, and perform an `int 21h` instruction. This function goes by the name `Find First Matching File`.[1] Here is a description of how arguments are passed to it.

`4Eh`-(a) The register pair `ds:dx` contains the address at which an ASCIIZ file descriptor may be found. This is the string which identifies the file whose attributes are to be returned.

`4Eh`-(b) The information about any matched file is returned to a buffer beginning at the so-called *disk-transfer address* (DTA). This address must be set by a separate instruction.

`4Eh`-(c) The register `cx` contains the *file attribute mask*. This mask tells which kinds of files are to allowed to be matched.

`4Eh`-(d) The *carry flag* is set if the find fails, and is cleared if the find succeeds.

`4Eh`-(e) Of course, the `ah` register must contain the function code, `4Eh`, when the `int 21h` call occurs.

There are several new concepts in this description, which will now be elaborated.

First of all, passing an address to a function via the `ds:dx` register pair is familiar. Usually, the `ds` register identifies the current data segment, so all that need be done is to put the offset in register `dx`. This mode of argument transfer has been used in several previous assignments. Assuming that `ds` already identifies the current data segment, either of the commands shown below will set the beginning address of the ASCIIZ file descriptor to the byte at offset `fspec`.

[1]The textbook by Irvine [Irv93, p. 361] contains a summary of how this call works.

```
mov   dx, OFFSET fspec
lea   dx, fspec
```

The name of the file must be passed as an ASCIIZ string. An *ASCIIZ string* is a string which is terminated by a zero byte — that is, a byte whose value is 0. (This is not to be confused with a byte containing the character code for zero, which is `30h`.) So, if the file name to be submitted as argument is `c:\programs\*.lst`, then the string would be declared in MASM as follows:

```
fspec     db  'c:\programs\*.lst',0
```

The zero byte, which is also known as an *ASCII null character*, is used to identify the end of the string. The DOS operating system maintains an area known as the *disk transfer address* (or *DTA*) for each program. This is the area which is used by certain programs (such as `Find First Matching File`) as the location to return strings of result data. When a program is started, the DTA is set to offset `80h` of the so-called *program segment prefix* (*PSP*). We will say more about the PSP later. For now, it suffices to say that what we want to do is to reset the DTA to a more amenable location. This is accomplished via the DOS function call `1Ah`, which is known as `Set Disk Transfer Address`.[2] To invoke this function, which is an `int 21h` call, we put the new DTA in the `ds:dx` pair, and do the `int 21h` call. Thus, assuming that `ds` has already been set to point to the current data segment, the following simple code will do the trick.

```
mov   ah,1Ah
lea   dx, dta_buffer
int   21h
```

Here, `dta_buffer` is the offset of the new DTA. For this software assignment, you are to write a *macro* called `set_disk_transfer_address` which sets the DTA using the above code. Invoke this macro to set the DTA for your use of the other DOS calls.

Associated with each file is its *attribute vector*. This is a one-byte vector, with the bits having the following significance. The bit numbering is left-to-right.

Bit	Significance
7	reserved by DOS
6	reserved by DOS
5	archive bit
4	directory bit
3	volume label bit
2	system file bit
1	hidden file bit
0	read-only file bit

[2]This function is summarized on page 362 of the book by Irvine.

The values of bits 7, 6, and 3 need not concern us here. The other five bits contain information about the file which is represented in the first field (column) of the desired format of a directory listing, as illustrated at the beginning of this document. For example, if the attribute byte has value `23h` = `00100011`, this means that the file has the archive bit set, it is not a directory, it is not a system file, it is a hidden file, and it is read-only. So, the first column of the listing program should be displayed as `rh..m` for such a file.

A *mask* of this format is used as an argument of the `Find First Matching File` function `4Eh`. This argument is passed to the function in the `cx` register. (Apparently, the contents of `ch` is relevant, and only register `cl` is used, but all manuals explicitly identify `cx` as the appropriate register.) The mask works for the following attributes: hidden (bit 1), system (bit 2), and directory (bit 4); bits 0 and 5 are ignored. For example, if bit 1 is set to zero, only a *non*-hidden file is retrieved, while if bit 1 is set to 1, both hidden and non-hidden files are retrieved. On the other hand, both modified and non-modified files are returned, regardless of the value of bit bit 5. In any case, for this assignment, a mask of all ones (`0FFFFh`) in register `cx` will suffice, since we want to retrieve information on any file which matches the file descriptor.

The file whose description is returned is the first file which matches the ASCIIZ file descriptor. Here *first* means the first file which DOS finds in its internal representation; this may have nothing to do with alphabetic order or any other obvious file attribute. Thus, if the file descriptor is `c:\programs\*.lst`, then if there is some file in the directory `c:\programs` with the extension `.lst`, and the file meets the mask test described in the previous paragraph, a call to `Find First Matching File` will find such a file. If there is no such file, then the carry flag will be set, and the function fails. If there is more than one such file, information on one of them will be returned. Of course, if there can be only one match (such as with the descriptor `c:\programs\soft7.lst`), then either information on that file will be returned (if the file exists), or else the call will fail and set the carry flag. In case a call succeeds, the carry flag is reset. The `jc` instruction may be used to execute a conditional jump, based upon the value of the carry flag.

Let us now take a closer look at the format of the result which will be placed at the DTA by a call to `Find First Matching File`. The information is packed into fields of a 43-byte record-like structure, with layout as follows.

Offset	File Information
00h-14h	Reserved by DOS for its use
15h	Attribute bit-vector
16h-17h	Time stamp
18h-19h	Date stamp
1Ah-1Dh	File size (double word format)
1Eh-2Ah	File name in ASCIIZ format

The first thing that must be done is to declare a buffer to receive this data. Using the above description, the layout is as shown in Figure 7.3.

```
dta          equ     $
dta_dos      db      15h dup (?)
dta_attr     db      ?
dta_time     dw      ?
dta_date     dw      ?
dta_size_l   dw      ?
dta_size_h   dw      ?
dta_fname    db      0Dh dup (?)
```

Figure 7.3: Declaration of the receiving buffer for file parameters.

In your program, for this assignment, you **must** do this with a macro. That is, you **must** write a macro, call it `make_dta_buffer`, which creates such a buffer when invoked. The macro must take exactly one argument, which is the name of the buffer prefix. Thus, the directive `make_dta_buffer dta` will create exactly the buffer of Figure 7.3, while the directive `make_dta_buffer foo` will create the following buffer.

```
foo          equ     $
foo_dos      db      15h dup (?)
foo_attr     db      ?
foo_time     dw      ?
foo_date     dw      ?
foo_size_l   dw      ?
foo_size_h   dw      ?
foo_fname    db      0Dh dup (?)
```

Now let us take a closer look at the format of the data which is returned by function `4Eh` to such a buffer. As the name suggests, the first 21 bytes (recall that 15h is 21 in decimal) are used by DOS. For the problem at hand, you need not be concerned with the values placed in these bytes. The next byte contains the *attribute vector*. The meaning of each bit has already been described.

The time of last update of the file is contained in bytes `16h-17h`, at the location identified by `dta_time`. The format is as follows, with bits numbered from left to right.

Bit	Information Stored
15-11	hours
10-5	minutes
4-0	seconds (in multiples of two)

So, suppose that the two-byte value representing the time is `0111101100110110` in binary. Then, the hours field is `01111`, which is 15 (or 3pm in US format), the minutes field is `011001`, which is 25, and the seconds field is `10110`, which is 22, representing 44 seconds. The time is thus 15:25:44, or 3:25:44pm in US format. (Note: In your listing program, all times should be displayed in 24-hour format; *e.g.*, `15:25:44`.)

The date of last update of the file is contained in bytes `18h-19h`, at the location identified by `dta_date`. The format is as follows, with bits numbered from left to right.

Bit	Information Stored
15-9	year, offset from 1980
8-5	month
4-0	day

So, suppose that the two-byte value representing the date is `0111101100110110` in binary. Then, the year field is `0111101`, which is 61, representing the year 1980+61 = 2041; the month field is `1001`, which is 9, representing September; and the day field is `10110`, which is 22. The date is thus September 22, 2041. (Note: In your listing program, all dates should be displayed in mm/dd/yy format; *e.g.*, `09/22/41`. Each field is two digits long, with leading zeroes displayed.)

The file size is represented in double-word format, so that files whose sizes range from 0 to $2^{31} - 1$ bytes may be represented. However, for this program, it will suffice to show only the file size modulo 2^{16}. In other words, we will use only the two least-significant bytes. Remembering that the 80x86 architecture is little-endian, the least-significant word is the first one, at `dta_size_l`.

The name of the file which is retrieved is stored in the final twelve bytes, in ASCIIZ format. Note that only the name, and not the entire path, is returned. Thus, if the argument given to the function is `c:\program\*.lst`, and the file found is `c:\program\soft7.lst`, then the string `SOFT.LST` will be found in locations `1Eh-26h` offset from the DTA, and the byte at location `27h` offset from the DTA will contain a `00h`. The remaining four bytes have indeterminate values. (The string returned will always use upper-case letters for letters in the file name.)[3]

7.2.3 Formatting the Retrieved Data for Display

To display the retrieved data, first of all, an output buffer needs to be built. It takes the form shown in Figure 7.4 on the following page.

Notice that the hardwired punctuation symbols which separate the fields are declared as part of the buffer. These never change, so we put them in when the buffer is declared. Also, six bytes have been allocated for the representation of the file size, since $2^{16} - 1 = 65535$ is the limit.

[3]In DOS, the character "`$`" is legal in filenames. If you use `int 21h` call `09h` (string output), a `$` in a file name will be interpreted as the end of the string to be printed. In your program, you need not deal with this case; however, you may enhance your program to print filenames with `$`'s in them for extra credit. See Section 2.4 for more information.

```
ls_buf  equ     $
ls_w  db      ?
ls_h  db      ?
ls_s  db      ?
ls_d  db      ?
ls_m    db      ?
        db      ' '
ls_size db 6 dup (?)
        db      ' '
ls_mo   db      2 dup (?)
        db      '/'
ls_day  db      2 dup (?)
        db      '/'
ls_yr   db      2 dup (?)
        db      ' '
ls_hh   db      2 dup (?)
        db      ':'
ls_mm   db      2 dup (?)
        db      ':'
ls_ss   db      2 dup (?)
        db      ' - '
ls_fn   db      12 dup (?)
        db      CR, LF, ENDTEXT
```

Figure 7.4: Declaration for the one-line output buffer.

The high-level algorithmic description which you should follow takes the form shown in Figure 7.5 on page 55.

For this software assignment, each one of the decoding steps must be realized as a separate (near) procedure, and the collection of these procedures must be housed in a library file which is separate from that of the main program. Name these procedures as follows:

```
get_file_attr
get_file_time
get_file_date
get_file_size
get_file_name
```

You must adopt a workable parameter-passing mechanism for these procedures. Since they are defined in a file separate from the main program, they cannot depend upon global names. Rather, parameters must be passed to them by the mechanisms which have been studied, such as passing values or addresses in registers. It is not allowable to pass parameters by using global naming of data objects via `EXTERNDEF` or similar declarations.

```
Set DTA to the fspec address;
Find the first matching file;
If carry flag is set
  then
    output nothing;
  else
    decode and format for output file attributes
       (fill in the ls_w, ls_h, ls_s, ls_d, and ls_m fields);
    decode and format for output file size
       (fill in the ls_size field);
    decode and format for output file date
       (fill in the ls_mo, ls_day, and ls_yr fields);
    decode and format for output file time
       (fill in the ls_hh, ls_mm, and ls_ss fields);
    decode and format for output file name
       (fill in the ls_fn field);
    display the formatted file-description string;
end if;
```

Figure 7.5: High-level description of the overall process.

Let us take a look at how to design one of these; namely, `get_file_time`. A section of the code is shown in Figure 7.6 on page 56.

Notice how the translation takes place. First, the entire time word is logically anded with a mask which preserves only the bits in the `hours` field. Then, the register is shifted so that these bits are right justified. Finally, the number is translated into internal format.

The procedure `buffer_number` is a support function which is used by several of the `get_` procedures. A header description is shown in Figure 7.7 on page 57.

This procedure is similar to the `display_number` procedure developed for Software Assignment 6. However, it writes to a buffer, rather than to the display. In fact, it is simpler, because it prints all leading zeroes, and since it writes to a buffer, it need not reverse the order of the digits. Furthermore, it deals with unsigned numbers, since that is what we need. You may use the code of your `display_number` procedure as a starting point if you wish. In particular, you may find that the auxiliary procedure `base_to_char` is a useful support routine. For this reason, it is required that you include this procedure in the same library file as your other numerical input/output routines. Do not forget to include an appropriate `EXTERNDEF` in the associated include file. (Note: The procedure only need translate into base 10, although if you adapt your `display_number`, you may find it easiest to retain the ability to choose the base.)

This procedure will be used to display the numbers in the date and time fields, as well as the size field. For the size field, it is necessary to suppress leading zeroes.

```
;; *****************************************************************
get_file_time PROC
;; Unpack the last-update time of a file, putting it into ASCII format.
;; Inputs: dx = pointer to beginning of eight-byte buffer area.
;;    ax = attribute word in packed format.
;; Outputs: buffer filled with appropriate values.
;; pkbuf is a local word-sized buffer, declared in the data area.
hours_mask equ 1111100000000000b
minutes_mask equ 0000011111100000b
seconds_mask equ 0000000000011111b
hours_shift equ 11
minutes_shift equ 5
push ax
push bx
push cx
push dx
mov pkbuf,ax         ;; save the packed word for future use.
;;       <process hours field>
;; Process minutes field:
and ax,minutes_mask  ;; mask out the other fields.
mov cl,minutes_shift ;; shift length must be in cl.
shr ax,cl            ;; right justify.
mov bl,10            ;; base 10 conversion parameter.
mov cl,2             ;; two digits of output parameter.
call buffer_number
;;      <process seconds field>
pop dx
pop cx
pop bx
pop ax
ret
get_file_time ENDP
;; *****************************************************************
```

Figure 7.6: Part of the code for the procedure `get_file_time`.

This is best done by using `buffer_number` as is, and replacing the zeroes by spaces afterwards.

Here is a summary of the files and procedures which you are to turn in for Software Assignment 7.

(a) The main program, which will print out a one-line summary of the attributes of a file, as described above.

(b) A collection of macros, which include at least the following.

 (i) A macro `make_dta_buffer`, of one argument, to create a data buffer for the DTA. The argument is the name of the buffer prefix.

```
;; ***********************************************************
buffer_number proc
;; Buffer a display-format number in a given base format.
;; Inputs: number in ax (unsigned).
;;    base in bl, 2 <= b <= 36.
;;    cl = size of buffer in bytes.
;;    dx = offset address of first byte of buffer.
;; Outputs: ASCII representation of number into buffer.
;; No error checking.
;; Leading zeroes are displayed.
;;       <code goes here.>
buffer_number endp
;;*************************************************************
```

Figure 7.7: Part of the code for the procedure `buffer_number`.

(ii) A macro `find_first_matching_file` of no arguments which invokes DOS function `4Eh` when it is called.

(iii) A macro `set_disk_transfer_address` of one argument which sets the DTA to the address identified by its argument, by using DOS call `1Ah`.

These macros must be in a separate file, communicated to the main program via an `include` directive.

(c) A collection of procedures, in a separate file, which handle the unpacking and formatting of the file attributes. These procedures must include at least the following near procedures, which have already been described.

(i) `get_file_attr`

(ii) `get_file_time`

(iii) `get_file_date`

(iv) `get_file_size`

(v) `get_file_name`

Associated with this package must be a declaration file which contains the `include` directives necessary to link this file to the main program.

(d) An augmented version of a collection of procedures for numerical I/O. The new procedure is to be named `buffer_number`, and has been described above. The associated declaration file must also be augmented.

Software Assignment 8

A UNIX-Style Directory Listing Program: Part 2

8.1 Purpose

This assignment extends Software Assignment 7, and has two principal purposes.

(1) To show how to retrieve information on *all* of the files satisfying a given descriptor.

(2) To learn how to configure an assembly-language program to obtain an argument from the command line.

8.2 Procedure

The software developed for Software Assignment 7 has at least two drawbacks. First of all, it only returns information on one file, regardless of how many files match the descriptor. Second, the argument must be hardwired into the program, and to change the argument, the program source file must be edited and re-assembled and linked. In Software Assignment 8, the program will be extended to overcome these shortcomings.

8.2.1 Finding all Matches to a File Descriptor

Extending the program to display the information on all matching files, rather than just on the first one found, is extremely easy, thanks to a DOS function named `Find Next Matching File`. It is invoked by placing `4Fh` in the `ah` register and executing an `int 21h` instruction.[1] It may only be invoked after the function `Find First Matching File` has. Each time that it is called, it finds a new file matching the descriptor for

[1]Page 362 of the text by Irvine [Irv93] contains a summary of the use of this function.

the last call to `Find First Matching File`. It places the information about the file which is found in a buffer beginning at the DTA, just as `Find First Matching File` does. Note: A new descriptor is **not** passed to `Find Next Matching File`. Rather, the system remembers the descriptor used in the last call to `Find First Matching File`, and it uses that. When `Find Next Matching File` cannot find any more new files, it signals failure by setting the carry flag. Thus, the high-level algorithm for this part is as shown in Figure 8.1 below.

```
Set DTA to the fspec address;
Find the first matching file;
If carry flag is set
  then
    output nothing;
  else
   repeat
     decode and format for output file attributes
        (fill in the ls_w, ls_h, ls_s, ls_d, and ls_m fields);
     decode and format for output file size
        (fill in the ls_size field);
     decode and format for output file date
        (fill in the ls_mo, ls_day, and ls_yr fields);
     decode and format for output file time
        (fill in the ls_hh, ls_mm, and ls_ss fields);
     decode and format for output file name
        (fill in the ls_fn field);
     display the formatted file-description string;
     find the next matching file;
   until carry flag is set;
end if;
```

Figure 8.1: The high-level algorithm for finding all matches.

Note that this is a simple extension of the description on the bottom of page 8. It should take only a few minutes to implement this improvement. Make sure that you define a macro called `find_next_matching_file` to implement the `1Ah` call.

8.2.2 Obtaining Program Arguments from the Command Line

Extending the program to take its argument from the command line, while not quite as trivial as the above extension, is not difficult, and it goes without saying that knowing how to do this is an extremely useful tool to have in one's bag of tricks for assembly-language programming in the DOS environment.

In the final executable module for a program, DOS attaches a 256 byte segment called the *Program Segment Prefix*, or *PSP* for short.[2] The PSP is mostly of historical interest, but it still does serve one very useful purpose. Namely, starting at address 80h of the PSP, the tail of the command line is stored. The *command tail* is that which is left after the command name is stripped away. Thus, if we have a program called `foo`, and we invoke it with argument `arg1`, `arg2`, and `arg3` with the command

```
foo arg1 arg2 arg3
```

then the command tail is `arg1 arg2 arg3`. Actually, a little more information is stored. In the byte at location 80h of the PSP is stored the number of characters in the command tail. Furthermore, the command tail is terminated by a carriage-return character, which is not part of this count. Thus, for the above example, assuming that the user types a return immediately after `arg3`, the character count would be 15 (the space before each argument is counted), and the sequence of byte values, in hexadecimal, starting at location 80h in the PSP would be

```
0Fh 20h 61h 72h 67h 31h 20h 61h 72h 67h 32h 20h 61h 72h 67h 33h 0Dh
 15  SP   a   r   g   1  SP   a   r   g   2  SP   a   r   g   3  CR
```

Just below the hexadecimal byte values are shown the character equivalents. Here `SP` denotes the space character, and `CR` denotes the carriage-return character.

We still have not said how one finds the PSP. When a program is started both the data segment register `ds` and the extra segment register `es` point to the PSP. While it is common custom to change the `ds` register to point to the current data segment when a program is started, the `es` register is not touched in our programs. To access the PSP, we use `es`. For example, the following code will put the character count for the program tail into the register `al`.

```
mov   si,80h
mov   al,es:[si]
```

In other words, just use `es:` as a prefix whenever you wish to access a memory location in the PSP.

To realize this final part of the assignment, first declare the `fspec` buffer to be large enough to accept any argument. A command line cannot be longer than 80h characters in DOS, so this should do it. (Remember that we will now get `fspec` from the tail of the command, rather than by hard wiring it into the program.)

```
fspec  db   82h dup (?)
```

[2]Section 11.5 of the text by Irvine [Irv93] contains information on the PSP.

You will need to write one more procedure, `get_psp_filespec`. This procedure will take the first argument from the tail of the command line and place it in the buffer pointed to by the `ds:dx` pair. In writing `get_psp_filespec`, keep in mind the following points.

(a) The filespec will be the *first* argument on the command line, as long as there is at least one argument. Any arguments beyond the first are discarded.

(b) If the command tail is empty (no arguments), then the resulting value of `fspec` should be set to `*.*`. (This is done so that a command without arguments will list the files in the current directory.)

(c) If it is nonempty, the command tail will begin with at least one space, and may contain more (if typed in by the user), before the actual arguments appear. Make sure that these lead spaces are stripped away, and not included in `fspec`.

(d) The command tail may consist entirely of spaces. In this case, your program should produce `*.*` as the value of `fspec`, just as in the case in which the tail is empty.

(e) The command tail ends with a carriage return. This byte must not be included in the value of `fspec`.

(f) The contents of `fspec` must be terminated by a byte with value `00h`.

To move the command tail to another buffer, the most direct approach is to use the two indexing registers, `si` and `di`. Initialize `si` to the starting point of the source buffer, and `di` to the starting point of the target buffer, and repeatedly move and increment. Here is rough code of the body of a basic copy loop.

```
loop:
   mov   al,es:[si]
   mov   [di],al
   inc   si
   inc   di
   cmp   si,source_limit
   jle   loop
```

In testing your program, note that both PWB and CodeView give you the opportunity to enter command-line arguments for a program, and run it with those arguments directly in the development environment. From the `Run` menu in PWB, use the `Program Arguments...` option. From the `Run` menu of CodeView, use the `Set Runtime Arguments` option. However, it is certainly a good idea to test your final program by running it directly on the DOS command line, with a variety of test arguments. This is what your laboratory instructor will do during the demonstration evaluation.

Here is a summary of the files and procedures which you are to turn in for Software Assignment 8.

(a) The main program, which will print out a multi-line summary of the attributes of a set of file, as described above.

(b) A collection of macros, which include at least the following, in addition to the macros for Software Assignment 7.

   (i) A macro `find_next_matching_file` of no arguments which invokes DOS function `4Fh` when it is called.

   These macros must be in a separate file, communicated to the main program via an `include` directive.

(c) A collection of procedures, in a separate file. These procedures must include at least the following near procedure, in addition to those constructed for Software Assignment 7.

   (i) `get_psp_filespec`

   Associated with this package must be a declaration file which contains the `include` directives necessary to link this file to the main program.

(d) An augmented version of a collection of procedures for numerical I/O. The new procedure is to be named `buffer_number`, and has been described above. The associated declaration file must also be augmented. This is the same collection as for Software Assignment 7.

Note: If you finish both Software Assignment 7 and Software Assignment 8 before the due date, then the logical thing to do is to submit just one package, since Software Assignment 8 essentially subsumes Software Assignment 7. If you do this, make sure that you mark your submission clearly that it is a combination of both assignments, and attach both cover sheets. However, if you do not finish Software Assignment 8 on time, but you do finish Software Assignment 7, then it is to your advantage to submit the latter separately.

8.2.3 Extra Credit

If you are ambitious and want to gain extra credit, here are a few improvements that you can make. The first two apply to Software Assignment 7 as well as to this software assignment. All extra credit options are listed here because you should finish both assignments before tackling any extra credit extensions.

(a) As noted previously, if `int 21h` call `09h` is used to print each line, file names which contain the character `$` will not be displayed properly. Rather, the line will end abruptly at that point, since `$` is interpreted as the end of string character by that DOS call. You may obtain extra credit by writing your own output routine, which must display the character `$` properly. The way to do this is to use `int 21h` call `02h`, which displays one character at a time. You have already done something similar in Software Assignments 2 and 3.

(b) The program does not behave exactly as one would want when the path identifies a directory. For example, if `foodir` is a directory, the call `soft8 foodir` will produce a one-line listing giving information on `foodir` as a file, rather than providing a listing of all files in `foodir`. Provide a modified program which tests to see whether the argument contains wild-card characters (`*` or `?`), and if it does not, and the single file returned is a directory, then the listing will be of the files in the directory.

(c) Sometimes it is useful to override the modification identified in (a) above and produce a listing about a directory. Modify the program of (a) above so that if `-d` occurs as the first argument on the command line, then if the next argument is a directory, a single-line listing about that directory as a file will be produced. In other words, the call `soft8 -d foodir` will produce the same result as the original (no extra credit) version of soft8.

If you decide to do the extra credit, make sure that you also submit the non-extra-credit versions, and that you clearly label each.

Software Assignment 9

Simple File Encryption: Part 1

9.1 Purpose

This assignment has multiple purposes.

(1) To gain some practice in performing random file input and output in assembly language.

(2) To gain some experience in passing parameters to procedures via the stack.

(3) To work with a simple technique of file encryption.

9.2 Procedure

9.2.1 The Overall Programming Project

The final result of this Software Assignment will be an assembly-language program which *encrypts* a file. To encrypt the file, a secret *encryption key* is provided. After the file is processed with the encryption key, its contents are encoded and cannot be read or executed without first being decrypted. Decryption requires the same encryption key.

This assignment uses a particularly simple type of encryption known as *XOR-encoding*. While it is not likely to deter the NSA, it will prevent the casual snoop from reading your files. Here is how it works. Suppose that we have a text file which contains the following ASCII sequence.

```
Contents of a file.
```

This contents of this file is 19 bytes long. In XOR encoding, we start with a 16-bit (word-length) key. We XOR each pair of bytes with the key. An example will clarify this. Suppose that we use the key `12345`, which as an unsigned 16-bit number is `3039h`. The first two bytes of the file are the character codes for `C` and for a, which has the combined value `436Fh`. When `436Fh` is XORed with the key `3039h`, we get `7A5Fh` (Assume that the key is stored in a register, and the words to be encrypted are stored in memory. Remember that the 8086 is a little-endian architecture, so the key is effectively byte-swapped!) The byte pair (`7Ah`,`5Fh`) then replaces the byte pair (`43h`,`6Fh`) = (`C`,`o`) in the file. We then proceed to the next byte pair, (`n`,`t`), which has the representation (`6Eh`,`74h`). Upon XORing this with `3039h`, we get (`57h`,`44h`). This process is repeated for each pair of bytes in the file. The final result is depicted below.

C	o	n	t	e	n	t	s		o	f		a		f	i	l	e	.
43	6F	6E	74	65	6E	74	73	20	6F	66	20	61	20	66	69	6C	65	2E
7A	5F	57	44	5C	5E	4D	43	19	5F	5F	10	58	10	5F	59	55	55	17

The first line shows the initial sequence of characters, while the second line shows the equivalent ASCII character code. The third line shows the result after using XOR encoding with `3039h`.[1]

If we replace the original contents of the file by the byte sequence shown in the third line in the above display, the contents of the file has been encrypted without losing any information. Interestingly, to recover the original contents, we just repeat the process. By XORing each byte pair with the key `3039h` once again, the original contents is recovered. Indeed, XORing any quantity with the same value twice yields the original value. XORing once flips the bits in the positions in which the key has a `1`, and doing this again, *with the same key*, yields the original value back. Thus, the encryption program will be, at the same time, a decryption program.

Note that it is important to pick a "good" key. For example, any key which is less than `256` (`100h`) will leave every other byte in the file unchanged, since the XOR of `00h` with any byte leaves that byte unchanged. Similarly, a key which is a multiple of `256` will leave every other byte unchanged.

In this software assignment, you will write a program which accomplishes this form of encryption. The program will prompt for required values. An example session is shown in Figure 9.1.

The messages generated by the system are shown in typewriter font, while the user response is shown in italics. The first two prompts are self-explanatory; note that the encryption key is entered in decimal. The third line, requesting a buffer size, stipulates the number of bytes which will be read and then rewritten in one i/o operation. This value affects the performance, but not the correctness, of the program.

[1] Notice that the file contains an odd number of characters — 19. For the last byte pair, we just append some arbitrary byte, since its value is not used. The last encoding is thus the XOR of (`2Eh`,`??h`) and (`30h`,`39h`), which is (`17h`,`??h`), the `??` denoting a don't-care value, which is not written to the encrypted file.

```
Welcome to the encryption program.
File to be encrypted: z:\temp.foo
Key, 0 <= k <= 65535: 12345
Buffering size, 2 <= b <= 1024 (default 512): 128
Encryption completed.
```

Figure 9.1: Example session of the encryption program.

When this program is run, it will encrypt the file with the given key. It will overwrite the file with an encrypted version, rather than create a separate encrypted file. Running the program on the same file again, with the same key, restores it to its initial contents.

Software Assignment 10 will add several bells and whistles to the basic program, including the ability to take arguments from the command line.

Before you move on to developing these programs, make sure that you understand how XOR encoding works.

9.2.2 File Input and Output in DOS

The programs of these assignments require that files be read and written. In this section, the appropriate DOS calls will be summarized. These functions are described in some detail in Section 12.1 of the text by Irvine [Irv93].

In DOS, a file must be *opened* before it can be read or written. It may be opened for input only, for output only, or for both input and output simultaneously. When work with a file is completed, it should be *closed*. Whenever a file is opened, it is assigned a unique *handle* by the operating system. Once opened, all further directives to the operating system regarding the file (*e.g.*, read, write, position, close) are given using the handle, and not the file name, as the identifier of the file. Here is a summary of the DOS calls which are needed for this assignment. All are `int 21h` calls, and the function code (which is passed in register `ah`) is shown next to the name of the function. All have the common characteristic that success or failure of the operation is indicated by the value of the carry flag. If the operation is successful, the carry flag is cleared, and if it is unsuccessful, the carry flag is set.

Open File (3Dh) This function opens an existing file. It may be opened in one of three modes: input-only, output-only, or input-output. The mode is determined by the value in register `al` passed to the function, as shown in the table below.

al	**Mode**
0	input-only
1	output-only
2	input-output

The only other input parameter to this function is the name of the file. The register pair `ds:dx` must point to an ASCIIZ string which contains the path of the file to be opened.

If the command is successful, the handle for the file is placed in register `ax` by the function. If the open is not successful, an error code is placed in in `al`. The possible error codes are given in the table below.

Error code	**Meaning**
1	cannot share file
2	file not found
3	path not found
4	too many open files
5	access denied

Error 1 means that some other program (such as an editor) is already using the file. Error 2 means that there is no file of the name specified, but that the path of a drive and directories leading to the file name is valid. Error 3 means that either the drive letter or directory path which was specified in the ASCIIZ string is invalid. Error 4 means that the system cannot open any more files. To remedy this, increase the value on the right-hand side of the `FILES=` statement in your `config.sys` file. Error 5 usually means that the file is read-only, or that it is a directory.

Close File Handle (3Eh) To invoke this function, just place the file handle in register `bx`, place `3eh` in register `ah`, and do an `int 21h` call. The only possible error (which results in error code `6` being placed in register `al`) is that the handle is invalid.

All files should be closed when processing on them is done. Closing a file flushes all system buffers associated with the file, and the effect of a write on file contents is not guaranteed to take place until the close is effected. Also, the close command releases the file buffers for other files to use. The maximum number of files which may be open at any one time is specified by the `FILES=` command in the `config.sys` file.

Read From File or Device (3Fh) This function requires the following input parameters.

Register	**Contents**
`bx`	file handle
`cx`	number of bytes to read
`ds:dx`	pointer to beginning of input buffer

Note that a file name or path to read from cannot be specified. The handle of a file which has been opened must be specified.

The carry flag is set if an error occurs. The error codes (passed in register `al`) include 5 (access denied) and 6 (invalid handle). If no error occurs, then register `ax` will contain, after the read, the number of bytes actually read. This is a convenient way to determine whether or not the end of the file has been reached. If the number of bytes read is less than the value passed in register `cx`, then the last read must have hit the end of the file.

Write to File or Device (40h) The input parameters to this function are similar to those for the read function:

Register	**Contents**
`bx`	file handle
`cx`	number of bytes to write
`ds:dx`	pointer to beginning of output buffer

The carry flag is set if an error occurs. The error codes (passed in register `al`) include 5 (access denied) and 6 (invalid handle). If the carry flag is not set, then register `ax` will contain the number of bytes actually written. The only way that this value can be less than the requested number (in register `cx`) is if some sort of i/o error occurred, such as a full disk.

Move File Pointer (42h) This function is central to random access of files, since it permits us to position the file pointer (which determines where the file will be read and/or written) at an arbitrary position within the file. The next read and/or write to a file begins with the byte pointed to by the file pointer. The inputs to this function are as follows.

Register	**Contents**
`al`	method code
`bx`	file handle
`cx`	high word of 32-bit offset
`dx`	low word of 32-bit offset

The *offset* (not to be confused with the offset of a segment:offset pair) is the distance that the file pointer should be moved from some known location. Because files may be large, this offset is represented as a 32-bit quantity, in the register pair `cx:dx`. Register `cx` contains the sixteen most significant bits, while register `dx` contains the sixteen least significant bits. The number may be interpreted either as an unsigned quantity, or as a signed (32-bit two's-complement) quantity. The *method code* determines this interpretation, and also how the file position is encoded. There are three possibilities, as shown below.

Register al	**Interpretation of register cx:dx**
0	absolute offset from beginning of file
1	signed offset from current file position
2	signed offset from end of file

In this assignment, we will only use method code 0. When using this code, the file pointer is positioned `n` bytes from the beginning of the file, with `n` the value of the `cx:dx` pair, when interpreted as an unsigned 32-bit quantity. Thus, to position the file pointer at byte `100000` in the file, the value `1` is placed in register `cx`, and the value `34464` is placed in register `dx`, since $100000 = 65536 + 34464 = 2^{16} + 34464$.

There are other useful file functions, such as **Create File** (function `3Ch`). However, since we will not use these functions in this assignment, they will not be described further.

At this point, you should write simple procedures for all of the file i/o calls identified above, and place these functions in your *doscalls.asm* file (or whatever you have named it) with all of your other basic `int 21h` calls. Remember to put the appropriate `externdef` statements in the associated `doscalls.dec` file. Figure 9.2 provides an example for the move file pointer function.

9.2.3 Developing Software Assignment 9

In this assignment, you will be expected to integrate the program-development skills which you have acquired in this course. Therefore, this handout will provide less of the step-by-step instructions than earlier ones did. You will be expected to do more of the design on your own. In so doing, keep in mind the following principles.

1. Begin by planning the overall project and by developing high-level pseudo-code for the overall program.
2. Develop and test individual modules in a bottom-up fashion. Test each module before going on to the next. Do not try to get the whole thing running in one fell swoop. There is simply too much attention to detail necessary in assembly-language programming. If you do not break the task into small subtasks, each developed and tested individually, you will be overwhelmed.

```
;; *********************************************************
move_file_pointer       proc
;; Move the file pointer for random i/o.
;; input: method code in al: 0 = absolute.
;;                           1 = signed offset from current.
;;                           2 = signed offset from end.
;;       bx = file handle.
;;       cx = offset, high.
;;       dx = offset, low.
        mov     ah,42h
        int     21h
        ret
move_file_pointer       endp
;; *********************************************************
```

Figure 9.2: Procedure for DOS function 42h.

3. Try to re-use or to adapt modules and procedures which were developed in earlier assignments, whenever possible.

First of all, Section 6.2 of the textbook by Irvine [Irv93] contains an extremely simple XOR file-encryption program. However, the program which you are to write for this assignment has many more features. Therefore, while it is a good idea to study the program on page 153 of the text by Irvine, you should develop your own program. The program of Irvine lacks many of the features which the program of this assignment requires.

Let us begin by writing a high-level specification for the program. Such a specification is shown in Figure 9.3.

```
begin
  1. Issue the welcome line to the display.
  2. Prompt for name of file to be encrypted.
  3. Prompt for the encryption key.
  4. Prompt for the buffer size.
  5. Convert the file name to an ASCIIZ string.
  6. Encrypt the file.
  7. Issue the success line to the display.
end
```

Figure 9.3: High-level description of the overall process.

Now, the first thing to notice is that parts of this program are very similar to those of previous programs. In particular, in Software Assignment 6 you developed assembly code which prompted for values and which converted numbers from external to internal format. So, you should be able to write the routines corresponding to lines 1, 2, 3, 4, and 7 in no time at all. In line 3, when obtaining the encryption key, you will need to translate an *unsigned* number from display to internal format. In Software Assignment 6, you wrote a procedure *get_number* which translated a *signed* number from display to internal format. Add a new function *get_number_unsigned* to your *numio.asm* library which performs the same translation on unsigned numbers. You will probably find that you can realize this function with an almost trivial modification to *get_number*.

In developing the code for line 4, get buffer size, you need to make sure that you substitute the default value 512 if no value is given, or if the value 0 is given. This should be very easy, since your `get_number_unsigned` function, which may be used for this part as well, will probably return 0 as the equivalent of the number represented by the empty string. In any case, you can easily modify it to make it behave in this fashion.

The code for line 6, which converts the file name to an ASCIIZ string, is very simple. You should write a separate procedure named `make_asciiZ` to accomplish this task. The address of the buffer which contains the file specification string should be passed to the procedure in registers `ds:dx`. Remember that the DOS call which reads a string (`0Ah`) writes to a buffer which includes the number of bytes read in the second byte of the header. So, let `make_asciiZ` take advantage of this information. All that this function has to do is to tack a zero byte onto the string, just after the last character. The position at which this zero byte is placed is easily computed from the header information.

Before proceeding on, make sure that your program can do all of these “supplementary” tasks. Write a partial program, including all parts except for line 6, and make sure that they all work. Test them by going into CodeView and making sure that the correct values are placed in the appropriate buffers. Resist the temptation to work on the “main” part of the program until you have this support shell working perfectly.

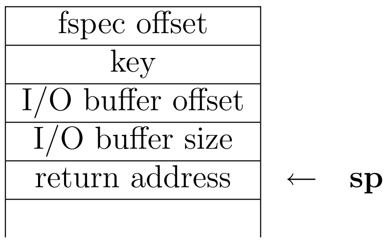
Once the support shell is working, it is time to tackle the main procedure of line 6, which encrypts the file. This task is also best approached by first developing a high-level plan, and then building and testing the components in a bottom-up fashion. First of all, **it is required for this assignment that this encryption routine be developed as a separately assembled procedure named `encrypt_file`, and thus stored in a separate file.** Let us identify the parameters which this procedure must receive.

1. The address of the buffer containing the specification (path) of the file to be encrypted. This buffer will have been prepared by the procedure `make_asciiZ`.
2. The encryption key.

3. The address of the i/o buffer which is to be used. The calling program must supply a buffer which this routine uses for input from and output to the file. This buffer is the "work area" for the encryption.

4. The size, in bytes, of the i/o buffer. This value will be determined from user input (the buffering size field). Note that you must reserve space for the largest admissible buffer, which is 1024 bytes. However, the user may specify a smaller buffer. The extra space is simply not used.

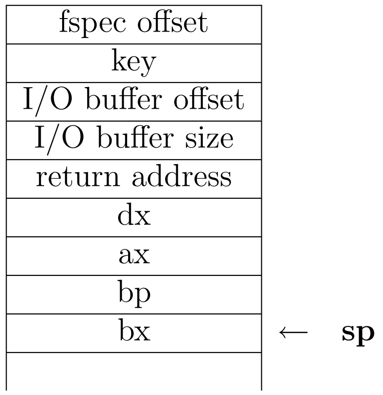
It is required that this procedure receive its parameters via the stack, and not via registers. The passing convention is to push the parameters onto the stack in the order specified above. Thus, immediately after a call to the procedure *encrypt_file*, the top part of the stack will appear as follows.



Now, in addition to the parameters, there are surely to be registers which are saved during the execution of the procedure. For example, the procedure might begin with the code

```
push dx
push ax
push bp
push bx
```

so that the stack would actually appear as follows after these `push`es.



```
;;*******************************************************************
encrypt_file    proc
;; encrypt a file with simple word-level XOR encoding.
;; inputs: pushed onto stack:
;;          offset of address of fspec buffer.
;;          key
;;          offset of address of i/o buffer.
;;          size of i/o buffer.
ef_pushsize      equ 10  ;; 4 registers + return address pushed
                         ;; after arguments on the stack.
ef_fspec         equ  ef_pushsize+6
ef_key           equ  ef_pushsize+4
ef_buffer        equ  ef_pushsize+2
ef_iosize        equ  ef_pushsize
        push    dx
        push    ax
        push    bp
        push    bx
             .
             .
        pop     bx
        pop     bp
        pop     ax
        pop     dx
        ret     8
encrypt_file    endp
;;*******************************************************************
```

Figure 9.4: Managing stack values in `encrypt_file`.

With so much information on the stack, it becomes difficult to avoid clerical errors. Here is a technique which helps quite a bit. Look at the code of Figure 9.4.

Notice what has been done here. The value `ef_pushsize` records the total number of bytes on the stack which have been pushed *after* the parameters. This number includes the return address offset. Since four registers have been pushed, the total number of bytes used is ten. The parameters may then be accessed relative to `ef_pushsize`. Thus, the key is four bytes offset from the beginning of the parameter list, and so `ef_pushsize` $+ 4 = 10 + 4 = 14$ bytes from the top of the stack. If we consistently use descriptive addresses, such as `[bp+ef_key]`, rather than absolute computations such as `[bp+14]`, then is the code not only much more readable, but changes are easily made as well. For example, if we discover that we need to save another register on the stack (say `si`), then rather than having to go through the entire code and change all references such as `[bp+14]` to `[bp+16]` (which is highly prone to error and misinterpretation), all that we need to is to change the value of `ef_pushsize` to 12. Everything else stays as it is.

Now, let us return to the issue of designing the actual high-level algorithm. Figure 9.5 below is a high-level description of what the procedure `encrypt_file` must do.

```
begin
  Open the file for i/o;
  Repeat
     Read a block from the file into the buffer;
     Encrypt the block in the buffer;
     Write the block back to the file;
     Move file pointer ahead one block;
  until no more blocks;
  Close the file;
end
```

Figure 9.5: High-level description of encrypt_file.

Now let us take a closer look at some of the details of this procedure. In building it, it is strongly suggested that you first develop everything but the block encryption step. In other words, first develop the part which opens the file, reads in a block, writes it back out, reads in the next block, writes it out, and so forth. Make sure that you have this part working perfectly before you dive into the encryption step. Otherwise, it is all too easy to get lost in a sea of details.

Opening the file for i/o uses as simple call to the DOS function `3Dh`. Upon doing this, save the file handle in a local variable. Closing the file is similarly trivial.

Reading a block into the buffer is also quite easy. The size of the block is specified by the user via the last prompted value during the initialization dialog. Make sure that this value is at least two (obviously!), and no larger than 1024. (Reserve a 1024-byte buffer for this operation.) Make sure that the size of the block actually read (as reported in register `ax` by DOS call `3Fh`) is recorded. This information must be passed along both to the encryption process, and to the process which writes the buffer back to the file. Remember that this number may be smaller than the buffer size when the last block (at the end of the file) is read.

Both the read operation and the write operation will advance the buffer pointer. When doing both input and output, it is easiest to specify the position to read and to write as an absolute offset to the file. So, maintain the 32-bit offset in a pair of words. Every time that a read-write pair is performed, increment the value represented by this pair by the size of the previous read/write pair. Since many files have size larger than 64K bytes, this offset must be represented as a 32-bit quantity. Start out with both the high offset and the low offset set to zero. Each time through the loop, add the previous read/write size to the low offset to get the new address. If there is a carry then add one to the high offset as well.[2] If the file size is smaller than 64K, then the high offset will stay at zero. However, test your program on larger files, because your laboratory instructor surely will!

The step which encrypts a block of text is best realized as a separate procedure. You just step through the array of bytes in the buffer, one word at a time, and XOR that word with the key. With the experience that you gained in previous assignments, you should be able to accomplish this without difficulty. You will probably want to use base indexing, with register `bx` representing the base address of the array, and register `si` or `di` representing the offset from the base.

9.2.4 Test conditions for Software Assignment 9

A working version of Software Assignment 9 should satisfy the following conditions.

1. It should not lock up or crash the machine, even on bad input, although it need not work correctly in such a case. However, it should not irreversibly corrupt a file. For example, if the user specifies a buffer size larger than 1024 bytes, it should round down to 1024 rather than write over memory areas which are not part of the buffer.
2. It must work for files which are larger than 64K bytes.
3. It must default to the 512 byte buffer size in the case that the user types 0 or 1 or just return at the prompt for that value.
4. It must not tack "garbage" onto the end of the file, or lop off the last few bytes. Make sure that the size of the encrypted file is exactly the same as the size of the original file, and that running the file through the encryption program twice yields exactly the same result. (Make a copy of the file before testing, and then use the DOS command `fcomp` to test for differences.)
5. It must work for any sixteen-bit key, and for any buffer size between 2 and 1024 bytes inclusive.

Needless to say, you should not test your program on your only copy of a valuable file. Always test on a throwaway copy, in a test directory where nothing else of value can be clobbered.

[2] Since we are working with *unsigned* numbers, check for a carry and not an overflow when deciding whether to increment the high offset. A carry signifies that an addition of unsigned numbers would not fit in the word, and that a bit must be carried to the next place, which is the low-order bit in the high word of the offset. An overflow signifies that signed arithmetic needs to extend to an extra place, which is not what is wanted here.

Software Assignment 10

Simple File Encryption: Part 2

10.1 Purpose

This assignment has the following principal purposes.

(1) To gain proficiency in working with simultaneously open (for I/O) files.

(2) To gain further experience in obtaining arguments from the command line.

10.2 Procedure

The core of this assignment is to add some bells and whistles to the previous program. Here are the features to add.

1. Enable the program to take its arguments from the command line, rather than from an interactive dialog. The command-line call should use the standard "/" option designator of DOS. Here is an example invocation, corresponding to the interactive example given earlier in this document.

   ```
   encrypt  z:\temp.foo /k12345 /b128
   ```

   The executable has been named `encrypt`. The first argument on the command line is the file to be encrypted. The `/k` argument gives the key, and the `/b` argument gives the buffer size. These arguments should be able to be given in any order. The `/k` argument is mandatory, while the `/b` argument is optional.

2. Enable the program to write the encryption to a different file. Use the syntax illustrated in the following example.

```
encrypt z:\temp.foo /oz:\temp.enc /k12345 /b128
```

> Here the `/o` option specifies the output file. The original file (`z:\temp.foo` in this case) remains unchanged. Arguments should be able to be given in any order.

In this final assignment, you are left to use the skills learned in this course to realize these features. No further hints or information is provided.

Appendices

Appendix A

Notes on the Use of the Software and the System

A.1 The Components

MASM MASM is an acronym for the Microsoft *Macro ASseMbler*. This is the basic program which translates assembly language into executable machine language programs. The current version is 6.11.

CodeView CodeView (CV) is the *debugger* which is used to monitor the execution of programs closely. With CodeView, one can watch a program execute step-by-step, monitoring memory locations and register values along the way. One can also set *breakpoints* and the like. The current version is 4.

Programmer's Workbench The Programmer's Workbench (PWB) is a *development environment*. It contains an editor, as well as means to invoke the assembler and the debugger. It also contains a fairly sophisticated *make facility*, which allows one to rebuild an entire set of interacting programs. While one can develop assembly language programs without using the Programmer's Workbench, by invoking the assembler, editor, debugger, and other utilities, PWB renders management of software development much simpler.

Actually, the Programmer's Workbench is not just for assembly language; it may also be used with Microsoft C/C++. However, we shall not use it in this way in the course.

Utilities There are a number of utilities which come with the MASM package. The most important for our purposes are the *linker*, named *LINK*, and the make utility, named *NMAKE*. The purpose of the linker is to combine assembled programs into one executable module. The purpose of the make utility is to ease the management of building large programs which contain many modules.

Both of these utilities may be invoked effortlessly from PWB, although they may be invoked separately from the command line as well.

Another utility which you may find useful is the *help* utility. Again, it may be invoked effortlessly from PWB.

A.2 Documentation

The MASM/PWB software package comes with four manuals, containing a total of approximately 1600 pages. It is useful to have a general idea of which topics are covered by each manual. Here is a short synopsis for each manual, in order of thickest to thinnest.

Environment and Tools [MAS93] This manual contains information on the Programmer's Workbench and it supporting utilities, including the debugger and the make utility. It does not contain information on the assembler or assembly language itself.

Programmer's Guide [MAS92b] This manual contains detailed information on the assembler (MASM) itself, including the use of macros and separately assembled modules,

Reference [MAS92c] This manual contains detailed information on the instruction set of the 8086 family of processors, and how those instructions are invoked within MASM.

Getting Started [MAS92a] This manual contains information on installing and configuring the MASM/PWB software.

In addition to these manuals, there is extensive on-line documentation which may be invoked directly from PWB and CodeView. For some information (such as editor key bindings), the on-line documentation is far more extensive than the printed.

A.3 The MASM/PWB Environment Variables

Before you use MASM/PWB (Macro ASseMbler / Programmer's WorkBench) for the first time, you must set certain MS-DOS environment variables. These variables tell MASM/PWB where certain crucial files may be found and/or stored. If you have installed MASM on your own personal computer, then the installation procedure will place commands defining these variables in your `autoexec.bat` file. However, if you are running MASM/PWB on a laboratory machine at the university, the values of these variables may be set somewhat differently. While they should be set properly, it is a good idea to have a basic understanding of them, so that if problems arise, they can be tracked down quickly.

A.3.1 Overview

The files which are associated with MASM and the Programmer's Workbench may be divided into eight categories.

Libraries: There are several libraries associated with MASM and the PWB development system. In addition, you may develop some of your own libraries. The MS-DOS environment variable `LIB` indicates where these libraries may be found. The value of this variable may be a *sequence* of paths, separated by semicolons. (It follows the same format as the familiar `PATH` environment variable of MS-DOS.) This allows you to access the system-wide libraries, and to access your own as well.

Include files: Include files are analogous to ".h" files for the programming language C. The MS-DOS environment variable `INCLUDE` indicates where these files may be found. The value of this variable may be a *sequence* of paths, separated by semicolons. (It follows the same format as the familiar `PATH` environment variable of MS-DOS.)

System binaries: The binaries are the executable files which are run in connection with the use of the MASM/PWB system. There is not a special environment variable for these files; rather, the path to them should be included in the `PATH` environment variable. When there is a call to a program of a given name, DOS searches the directories in the `PATH` environment variable, in order, until it finds a program with the correct name. It is therefore critical that you configure the `PATH` variable so that it finds the correct program.

Initialization files: The initialization files contain the information which customizes your particular sessions of MASM/PWB. It contains information on where the files are that you have recently worked on, as well as customization information for your development environment. These files reflect things that you do as an individual user, so on a shared system such as the EMBA-CF, it is critical that each user have a private copy of these files. We will say more on how this is done later.

The location of the initialization files which are used by a particular session of MASM/PWB is given by the value of the MS-DOS environment variable `INIT` when PWB is started.

User files: User files are the programs that you will write. There is no special MS-DOS environment variable to identify where they are stored. Rather, you will specify this information when you save the file with the PWB editor.

Scratch files: MASM/PWB must generate a number of "scratch" files as it goes along. These are files which are only needed temporarily, and which may be discarded once a particular development session is terminated. The MS-DOS environment variable `TMP` indicates to the system where these files may be stored.

Help files: MASM/PWB has an extensive on-line help system. The location of the files supporting this help system is identified by the MS-DOS environment variable `HELPFILES`.

A.3.2 Environment Variables on a Personal System

If you install MASM on you personal computer, the `setup` program will install statements in your `autoexec.bat` file which define their values properly. These statements are then executed automatically whenever you boot your computer. If you use the default installation directory which the installation program suggests, the values will be the following:

```
SET LIB=C:\MASM611\LIB
SET INCLUDE=C:\MASM611\INCLUDE
SET INIT=C:\MASM611\INIT
SET HELPFILES=C:\MASM611\HELP\*.HLP
SET ASMEX=C:\MASM611\SAMPLES
```

(The `ASMEX` environment variable defines the location of example programs; we have not mentioned this above.) In addition, the installation program will add `C:\MASM611\BIN` to your `PATH` variable.

A.3.3 Environment Variables on an EMBA-CF Microcomputer

MASM is installed a bit differently on the EMBA-CF systems. When an EMBA-CF machine boots up, it defines these environment variables as follows.

```
SET LIB=C:\MASM611\LIB
SET INCLUDE=C:\MASM611\INCLUDE
SET INIT=M:\MASM611\INIT
SET HELPFILES=C:\MASM611\HELP\*.HLP
SET ASMEX=C:\MASM611\SAMPLES
```

In addition, `C:\MASM611\BIN` is included in the `PATH` variable. Notice that there is one difference from the personal system installation — namely, the `INIT` directory resides on the `M:` drive. On the EMBA-CF system, the `M:` drive is the file space associated with your account; it is not shared with other users. Thus, for MASM/PWB to work properly on this system, it is necessary for you to create the directory `M:\MASM611\INIT` and to install initialization files there. We will say more about this below.

The `M:` drive, rather than the `C:` drive, is used to store the initialization files so that these files are customizable for a given user. MASM/PWB has a large number of user-settable options, and your choice of settings may be different than those of other users. In addition, the initialization files contain information on the names and locations of files which have been recently edited and built, and this information is clearly local to a given user. The above modification thus keeps the settings which you choose in MASM/PWB from being altered by other users. Each time that you start MASM/PWB on an EMBA-CF system, the settings will be exactly as you left them.

A.4 Running MASM/PWB on an EMBA-CF Microcomputer

A.4.1 One-Time Initialization

Before running MASM/PWB on an EMBA-CF microcomputer for the first time, it is necessary to install the initialization files. Here are the steps that you should follow.

1. If you are in windows, start an MS-DOS shell by double clicking on the *MS-DOS Prompt* icon in the *main* group.
2. Type the command `Q:\cs101\bin\reinit.bat`. This program will do the following:
   - (i) It will create the directory `m:\masm611\init` if it does not already exist.
   - (ii) It will copy the system files (contained in the directory `Q:\masm611\init`) named `tools.ini` and `current.sts` to the directory `m:\masm611\init`. To provide some protection, if files named `tools.ini` and/or `current.sts` already exist in the directory `m:\masm611\init`, they will be moved to files in that directory named `tools.sav` and `current.sav`, respectively. However, any previous `.sav` files will be overwritten.

This step should be performed only once, before you use MASM/PWB for the first time. These initialization files will contain information about your work as you proceed through the semester, and reinstalling them will result in this information being lost. (Such a loss is annoying, but not catastrophic. Actual programs are not lost, just information on where they reside, and associated command options.)

For your home system, default initialization files will be installed. However, you may wish to copy the files contained in `Q:\masm611\init` to a floppy disk and install these on your home system, as they already have options set which will be of use to you. Of course, you may also set the appropriate options yourself.

A.4.2 Certifying the Installation on an EMBA-CF Microcomputer

Important Note: As this document goes to press, the EMBA-CF is in the process of upgrading one of its laboratories to machines which run the operating system Windows NT. While the operating systems MS-DOS and Windows 95 support only the *File Allocation Table* file system (*FAT*, for short), Windows NT supports both the FAT file system and the newer and more sophisticated *New Technology File System*, or *NTFS* for short. Unlike the FAT file system, NTFS supports user-specific file protection which enables the EMBA-CF to install files on the local drive of the microcomputer in such a way that ordinary users may read and execute them, but not modify them. If NTFS is used as the principal file system for these new microcomputers, then the material in A.4.2 which follows will probably not be applicable to those machines. Ask your laboratory instructor for up-to-date information.

As mentioned previously, all of the files associated with MASM/PWB on an EMBA-CF microcomputer reside on the local `C:` drive. If you are using a system running Windows NT with NTFS (the advanced file system of Windows NT), then these files should be protected against modification by users. Unfortunately, the FAT file system (which may be used with Windows NT, and which is the only file system available for Windows 95 and for MS-DOS) does not support multilevel protection. As a consequence, any user may modify any system files on such file systems. While well-behaved users should not modify these files, there is no guarantee that they have not. Therefore, unless you are using a Windows NT system with NTFS, it is a good idea to reinstall MASM/PWB each time that you log in.[1] It takes only a minute. To reinstall the software, proceed as follows.

1. If you are not already in an MS-DOS shell, select and double-click on the *MS-DOS prompt* from the `Programs` listing of the `Start` button menu.
2. At the command prompt, type the command `Q:\cs101\bin\pwbinstf`. This will reinstall all of the MASM/PWB files to the `C:` drive, and will set the associated environment variables correctly.

At this point, you have two options.

a. You can exit the MS-DOS shell by typing `exit`, which will kill the shell and return you to Windows. You can then select MASM6.1 from the `Programs` entry of the `Start` button menu.

[1]This procedure will not work with Windows NT and NTFS; an error message will appear if it is attempted. However, with that system, the files are protected, and so the procedure is not necessary.

There is only one shortcoming to this approach. It is possible, but highly unlikely, that someone has modified the system initialization file which sets the MS-DOS environment variables. The definitions made by the `pwbinst` program are lost when the MS-DOS shell in which they were made is exited. If someone has modified the system initialization file, this may cause a problem. (One way to make sure that this has not happened is to start up an MS-DOS shell by double clicking on the *MS-DOS prompt* icon, and then to type the *set* command, which will display the paths.)

b. A safer approach is as follows. After you have run the `pwbinst` program, do **not** exit the MS-DOS shell. Rather, type the command `pwb` at the command prompt.[2] This will start up Programmer's Workbench, just as clicking on the menu item does, but since you have not exited the shell, the environment variable settings made by the `pwbinit` program will remain in force. You may still return to the Windows environment, to run an additional program, by typing `Ctrl+Esc`.

A.4.3 Further Security Issues

As we have noted several times, files on the `C:` drive of and EMBA-CF microcomputer running Windows 95, or running Windows NT with the FAT file system, are unprotected. Therefore, any user can change and/or delete them. It is highly unlikely that a non-malicious modification of the `C:` drive by another user will cause anything but a failed program. However, it is wise not to discount the possibility that someone will attempt a malicious action. We suggest the following steps to minimize the possibility of loss of work under these circumstances.

1. After each working session on an EMBA-CF, save all source files on a floppy disk. Never depend upon files on your `M:` drive remaining unchanged from session to session.

2. When using and EMBA-CF microcomputer for CS101, use only your CS101 account. Do not use a personal account with many other valuable files. If you want to access a personal account, it is strongly suggested that you use UNIX and one of the X-stations in rooms 227 or 229 of Votey building. Security when working from these X-stations is far better than the security from a microcomputer.

[2]If this does not work, try typing the full path `C:\masm611\bin\pwb`.

A.4.4 Proper Etiquette When Using an EMBA-CF Microcomputer

1. Needless to say, you should never do anything malicious which will steal another user's password, or damage that user's account. If you do so and are caught, you will be in very serious trouble, and may very well be dismissed from the university.
2. If you do something inadvertently which destroys or changes common files on the `C:` drive of a microcomputer, please notify the course instructor or an assistant, or someone working for the EMBA-CF. The integrity of a local drive can easily be restored. You will not be disciplined for an honest mistake. Indeed, you will be thanked for pointing it out rather than forcing us to discover it the hard way.
3. Do not leave files which you have created in inappropriate places, such as the root directory. For temporary files, use the directory pointed to by the `TMP` environment variable.
4. Do not leave copies of the source files for your CS101 software assignments on the `C:` drive, where others can access them. Do not make it easy for another student to steal your work.

A.4.5 Logical Drives on an EMBA-CF Microcomputer

We have made several references to drive names on the EMBA-CF microcomputers. Here is a summary of these drives, and the directories which are particularly relevant to CS101.

A: The `A:` drive is the 3.5 inch floppy disk drive. This is standard for just about every MS-DOS system, whether running on a home system or an EMBA-CF one.

B: The `B:` drive, if it exists, is the 5.25 inch floppy disk drive. These drives exist on some of the older 80486 systems in room 206 Votey, but not on any of the newer Micron Pentium systems in room 210/246 Votey.

C: The `C:` drive is the local hard disk on the machine. Except for systems using NTFS, there is absolutely no protection of any files on this drive, and **anyone** may modify things there. **As we have already emphasized, be very careful when trusting the integrity of files that you find there, and never leave copies of your work there for others to steal.**

Q:, R:, S: The `Q:`, `R:`, and `S:` drives are system drives. The actual computer on which these drives reside is a network server. It is not in the physical computer box that you are using. Generally, you may read files on these drives and execute programs on them but you may not add, modify or delete files there. Particularly important directories on these drives include `Q:\cs101`, which contains files specific to CS101, and `Q:\masm611`, which contains a copy of the complete installation files for MASM/PWB (which are downloaded to the `C:` drive with the `pwbinst` command and to the `M:` drive with the `reinit` command.)

M: The `M:` drive is your private drive. The actual computer on which this drive resides is a network server. It is not in the physical computer box that you are using. You can be reasonably assured that other users will not have access to the files on this drive, unless you do something to give them such access. (Unfortunately, a malicious program on the `C:` drive can lead to such access.) It is the same personal file space that you may access by using the UNIX operating system. Always save your work, and keep personal customization files, on this drive.

A.5 Other Issues Relevant to MASM/PWB

A.5.1 Option Settings in PWB

The material in this section applies both to installations on your personal computer and to installations on an EMBA-CF computer. All of the options listed below originate from the `options` menu item of PWB. These should be pre-set properly when using the initialization files which are installed on the EMBA-CF system, but may not be set properly on your home system if you use the installation defaults.

1. From the `Build Options` subitem, select `Use Debug Options`. This tells the system always to build your software so that the CodeView debugger can analyze it.
2. From the `Language Options` subitem, and `MASM Options` selection, the following settings are important.
   (i) On the main menu, make sure that `Warnings Treated as Errors` is selected. For `Warn Level`, select `Level 2` or `Level 3`.
   (ii) Click on `Set Debug Options`. Make sure that the `Debug Information` item has `CodeView` selected. Otherwise, critical information necessary for the full use of CodeView will not be generated by the assembler and linker. Also, make sure that `Generate Listing File` is selected, and that the suboptions `List Generated Instructions`, `Generate Symbol Table`,
   `Include All Source Lines`, and `Include Instruction Timings` are selected. It is best that the other suboptions of `Generate Listing File` not be selected.

3. From the `Link Options` menu item, choose the following.

   (i) Make sure that the `Debug Options` item is tagged. Also, make sure that the `CodeView` item is tagged.

   (ii) Click on the `Additional Debug Options` item. From the table which appears, make sure that the item `Pack Executable File` is **not** selected. (If this item is selected, the system may generate object files which will cause protection faults. If you get protected mode errors when you try to run a program which you have built, check the setting of this option.)

   (iii) While remaining in the `Debug Options` menu, select the `Full` option for the `Map File` option. Select `OK` to exit this submenu.

4. Click on `NMAKE Options`. It is best if the `Halt on any Errors` option is selected.

5. Click on `CodeView Options`. You will find it most convenient to use CodeView if you select the `50-line mode` for `Screen Size`.

6. Click on `Customize Project Template`. Put the cursor within the rectangle which appears in the middle of the new window (beneath the text `Build Rule List`. Scroll down to the entry which begins with `macro "AFLAGS_D..."`, and click on it. Now, move the cursor to the one-line rectangle labelled `Build Rule`. The text of the `macro "AFLAGS_D..."` appears in this window. Move the cursor to the end of this line, which should read `/Fl`. Replace `/Fl` with `Fl$*.lst`. Now, click on `Set Build Rule` near the bottom of the box. Do not click on `OK` first, or this change will be lost.

   The purpose of this change is to tell PWB to put the listing file in the same directory as the source file. By default, it will put that file in the directory in which PWB is invoked, which will be your windows directory (often `c:\windows`), which is probably not what you want. One problem that may occur with this new configuration is that the command line for assembly may be too long. If so, you will get an error message when you attempt to compile. This is a “feature” of MS-DOS, and cannot be fixed. To address this problem, you must work with shorter paths. Here is a useful trick. If you are using your home system, put the line `lastdrive=z` in your `config.sys` file. If you are using an EMBA-CF system, you don’t need to do this (nor can you). Now, enter an MS-DOS shell and issue the command such as `subst T: M:\masm611\programs`. Here `T:` is the name of a new *logical drive*. Use any letter which does not show up when you run the windows file manager utility. The `M:\masm611\programs` is the directory in which you have your program files. Use whatever path is appropriate. Now, you may use the name `T:` as an abbreviation for this longer path. When you start up PWB, instead of opening the file `M:\masm611\programs\soft1.asm`, say, open `T:\soft1.asm`. It is the same file, but the path name is shorter in this *alias*. Note that to use this trick, you must start PWB in a shell in which you have run the `subst` command. On an EMBA-CF system, you cannot do it any other way. On you home system, you could put the `subst` command in your `autoexec.bat` file, and reboot. Then, Windows will know about this path, and so you can just click on the PWB icon.

A.5.2 General Notes

The following hint applies whenever you run MASM/PWB on any system.

1. When you are finished using PWB, always quit PWB and exit back to Windows or MS-DOS before you log off of your computer. If you do not, some important information will not be saved in your initialization files, and when you start up MASM/PWB the next time, you will have to redo certain things.

A.5.3 Running MASM/PWB on a Home System

There are a few issues which you should be aware of.

1. Despite what the installation manual says, MASM/PWB will not run with only 4 megabytes of main memory. It does run with 8 megabytes. We do not know if it runs with intermediate values, such as 6 megabytes; such configurations are uncommon. Bottom line: You need 8 megs of main memory to run this software.
2. Some of the utilities, including the linker and nmake, seem to be very fussy about memory management, and will complain and crash when run under MS-DOS with certain DPMI (DOS Protected Mode Interface) managers. If you have any problems, we strongly suggest that you run PWB from Windows. MASM/PWB works flawlessly with the Microsoft Windows DPMI (which is installed automatically when running windows).

A.5.4 Fifty-Line Mode in PWB

To be able to see more at one time, you may wish to set PWB to 50-line mode. To do so, proceed as follows.

1. From the `Options` menu, select `Editor Settings`.
2. In the upper right corner of the window, set `Switch Type` to `Numeric`.
3. Put the cursor in the `Switch List` box in the middle of the window, and move the cursor to highlight the `Height` entry, which will probably read `Height:25`. You may need to scroll the `Switch List` to get to this entry.

4. The text `Height:25` should appear in the `Switch` window at the very top of the `Editor Settings` window. Type over the `25`, replacing it with a `50`.
5. Click on the `Set Switch` box at the bottom of the window.
6. Click on `Save`, and then on `OK` in the new window.
7. Click on `OK` to exit the `Editor Settings` window.

All windows in the main PWB screen should now appear in 50-line mode.

Appendix B

Submission Rules and Evaluation Criteria

B.1 Content of a Submission of a Software Assignment

The completion of a given software assignment includes two components, a written submission, and a real-time demonstration of the behavior of the program. A submission is not complete until both components have been completed, and to avoid late penalties, both must be completed before the due date of the assignment.

B.1.1 The Written Submission

Cover sheet For each software assignment, a cover sheet will be distributed. This sheet will describe the conditions which must be followed in preparing the assignment. For a grade to be recorded for a given assignment, the associated cover sheet must be included in the submission.

Documented source code Every submission must contain a printout of all source code. This means all assembly sources (`.asm` files), as well as any include files (`.inc` and `.dec` files). All source code must be adequately documented, including in particular what each procedure and macro does, and how parameters are passed to and from it. Further issues surrounding proper documentation will be discussed in the laboratory meetings.

Assembler listing files Every submission must include an assembler listing file (`.lst` file) for each `.asm` file. Since listing files are wider than 80 columns, normal portrait-style printing utilities will not provide a complete printout. To remedy this, use the special utility `q:\cs101\bin\smallprt`. It uses a smaller font, which permits the entire file to be displayed without truncation. Alternately, if you need a bigger typeface, you may use `q:\cs101\bin\landsprt`, which will print in landscape mode on the page. These utilities are available on the EMBA-CF systems. If you use your home system and printer, you will need to use some other appropriate utilities.

Map file listing For each program, submit a printout of the map file (extension `.map`) which is generated by the loader.

Additional items Some programming assignments may stipulate that specific output or other items be submitted as well.

B.1.2 The Demonstration

Each program must be demonstrated to the laboratory instructor. During this demonstration, the program will be tested using a test suite of inputs designed to determine whether it handles all cases properly, and will make note of any problems encountered. Generally speaking, the best time to demonstrate programs is during the laboratory sessions. However, demonstrations may also be arranged at other times, such as during the laboratory instructors office hours. Except by explicit prior agreement, a diskette containing the program may **not** be submitted in lieu of an in-person demonstration.

B.2 Restrictions

Any restrictions which pertain to a specific assignment will be distributed separately. The following restriction applies to all assignments.

No External Libraries Unless stipulated explicitly to the contrary, you may not use external libraries of procedures or macros to support your programs. This includes in particular the libraries which are distributed on the diskette which comes with the course textbook.

B.3 When and Where to Submit Software Assignments

Information on when and where software assignments may be submitted, as well as penalties for lateness, will be provided separately.

Bibliography

[Irv93] Irvine, K. R., *Assembly Language for the IBM-PC*, Prentice-Hall, second edition, 1993.

[MAS93] *Environment and Tools: Microsoft MASM Assembly Language Development System Version 6.1*, Microsoft Corporation, 1993.

[MAS92a] *Getting Started: Microsoft MASM Assembly Language Development System Version 6.1*, Microsoft Corporation, 1992.

[MAS92b] *Programmer's Guide: Microsoft MASM Assembly Language Development System Version 6.1*, Microsoft Corporation, 1992.

[MAS92c] *Reference: Microsoft MASM Assembly Language Development System Version 6.1*, Microsoft Corporation, 1992.