CHAPTER 3. USING THE F83 SYSTEM

I suppose that you are now ready and eager to use the F83 system to do some programming on your own. In this chapter I will try to give you some tips on how to use F83 to write programs and save them on disk for posterity. It is not my job to teach you how to programming in Forth. There are too many books on this subject in the book stores. What I want to do is to discuss many useful commands in the F83 system which are very helpful to generate code, test and debug the code, and save them on the disk in files. Most of them are specific to the F83 system. You will find all of them in the source code form and commented in the shadow screens if you study diligently the entire F83 listings. Rather than postpone the pleasure in exploring this system until you learn all about it, I think you will appreciate a few tips to get started immediately and do something useful.

Again I have to remind you to make backup copies of your original F83 system disks. If you are through with the viewing facility and do not need all the F83 source files, you may want to use a formatted blank disk in the currently logged disk drive for the exercises we will do here. Since all the Forth words are loaded into the dictionary in RAM memory, you really do not need those source files unless you want to copy and use some of the screens. When we invoke the editor, we will make permanent changes on the disk. You would certainly want a good F83 system backed up so you have something to fall back to.

3.1. CREATE YOUR OWN FILE

To write your own programs, the first thing you have to do is to find some space on the disk to store your program. You can get disk space in three different ways:

1. Open an existing file and use screens in it. You will destroy some of the information in this file.
2. Extend an existing file and use the increased space at the end of this file to store your code.
3. Create your own file and do whatever you want with it.

If you wanted to modify F83 to suit your own computer or to make it perform better, you probably will use the first approach. Just be sure that the disk is not write-protected so that you will be able to save your program. Simply OPEN the file you want to use and select an screen by the command n LIST or n EDIT. Then you can use editor commands to enter new code into the screen or edit its contents. We will discuss the editor commands later.

If you choose the second approach to extend an existing file, you should first open the desired file and use the command MORE to add more screens to this file. For example,

```
10 MORE
```

will add 10 screens to the end of the current file. All the added screens are filled with blank characters. Now, the command CAPACITY will return the total screen number on the stack. From this number you can select any of the added screens to enter your program. One problem with this approach is that the shadow screens in the file will not match with the corresponding source screens.

To create a new file on the disk for your private use, you have to use the command CREATE-FILE. Following is an example:

```
30  CREATE-FILE  MYFILE.BLK
```

where MYFILE.BLK is the name of the new file, and the length of the file is 30 Forth blocks or 30K bytes. After a file is created this way, it can be opened by the OPEN command:

```
OPEN  MYFILE.BLK
```

and now we can use the command LIST or EDIT to select one screen in this file to enter new text or other information. The file name must conform to the rules of the disk operating system. Usually the name can contain up to 8 characters with a three character extension.

F83 allows you to open 2 files at a time so that screens can be copied from one file to the other. The command FROM opens the second file which can be read while the current file can be written. For example,

```
FROM  YOURFILE.BLK  1 10  COPY
```

opens YOURFILE.BLK file as the input file and keeps MYFILE.BLK file as the current file. Of course, you have to create YOURFILE before you can open it. The command COPY which normally would copy screen 1 to screen 10 in MYFILE.BLK will now copy screen 1 from YOURFILE.BLK file to screen 10 in MYFILE.BLK. When a file is OPENed, it is made both the current file and the input file. When a file is opened by the command FROM, the file becomes the input file only and can be used with the current file.

To copy a range of screens in the current file from one place to another, the command to use is CONVEY. First you have to tell the F83 system how many screens you want to skip over during copying. If you want to copy screens 1-6 to screens 12-17, you should give the following commands:

```
        11  HOP   1 6  CONVEY
or      1  6  TO  12  CONVEY
```

If the number before HOP is negative, the range of screens will be moved towards the beginning of the file.

If you want to copy screens 1-20 in YOURFILE.BLK to HISFILE.BLK and put them down as screens 11-30, the commands are:

```
        OPEN HISFILE.BLK  FROM YOURFILE.BLK
and     10  HOP  1  20  CONVEY
or      1  20  TO  11  CONVEY
```

You should be very careful about these commands because two files are involved. You always read from the input file and write to the current file. In case that you want to copy screens from the current file to the input file, you must use the command SWITCH to exchange the current and the input files before issuing COPY or CONVEY command.

3.2. THE EDITOR

There are two text editors in the F83 system: a regular line editor and a screen editor. The line editor processes the text one line at a time and can be used with any type of terminal. Since most terminals can display 24 80 column lines, there are more than enough spaces on the terminal screen to display an entire Forth screen in the 16 by 64 block format, with ample space to enter editing commands. The line editor is adequate for all editing and programming purposes. The only drawback is that after entering 8 lines of commands, the listed screen starts to roll off the top of the terminal display and you will have to re-list it to keep it in the view. The screen editor keeps the listed screen on the top of the terminal display and refreshes its contents after any editing command is executed.

The line editor is compatible with the editor described in Brodie's book 'Starting FORTH'. The more often used editing commands are summarized in Table 3.1.

TABLE 3.1. EDITOR COMMANDS

1. Block Editing Commands:

n LIST	Display screen n and make it the current screen.
L	Display the current screen.
N	Display the next screen.
B	Display the previous screen.
A	Toggle between the current screen and its shadow screen.
UPDATE	Mark the current screen to be saved to the disk file.
SAVE-BUFFERS	Write all updated screens to their respective disk files.
FLUSH	SAVE-BUFFERS and DE-allocate the buffers.
n LOAD	Interpret the text in screen n.

2. Line Editing Commands

n T	Select line n as the current line for editing.
P xxxx	Put the string xxxx in the current line.
U xxxx	Insert the string xxxx under the current line.
X	Delete the current line.
n NEW	Input multiple lines of text starting at line n.

3. String Editing Commands

F xxxx	Find string xxxx from the current cursor position and place the cursor at the end of xxxx.
D xxxx	Find string xxxx and delete it.
I xxxx	Insert string xxxx after the current cursor and move the cursor to the end of xxxx.
TILL xxxx	Delete all text till the end of string xxxx in the current line.

J xxxx Delete text till the beginning of string xxxx. (Justify).

To use the line editor, you first have to select a screen as the current editing screen by the command:

```
1 LIST EDITOR
```

Then, you can use the line editing commands to enter text into this screen. The NEW command is especially useful in entering several contiguous lines into the screen. If screen 1 is a blank screen, you probably will start with:

```
0 NEW
```

and follow with up to 16 lines of text. Two consecutive carriage returns will terminate the NEW command. If you find any error in the entered text, you can use the string editing command to find text strings, delete strings, and also insert strings. When you are satisfied with the contents of the screen, you can interpret or compile the screen using the command:

```
1 LOAD
```

and start to debug your program entered in screen 1. Usually you will find some more errors or bugs in the program, causing the interpreter to abort during the loading process or giving wrong results when you execute words defined in this screen. You will then have to find the cause of the problem and fix the bug, again using the editor.

F83 has a generic screen editor. However, this screen editor must be customized to run on your terminal. An example to install a screen editor for the ADM-3A dumb terminal is shown in the README files which is also a good example on the command sequence in using the line editor. Terminal characteristics are specified in four words:

AT	Move the display cursor to a specified screen coordinate.
DARK	Clear the screen and home the cursor.
-LINE	Delete current line and scroll up the lines below by one line.
BLOT	Erase till the end of line.

These four words have to be vectored to the word definitions which perform these functions on the terminal you are using.

There are some new features in the F83 screen editor. An automatic ID stamping utility inserts an identification string on the top right of every screen being edited. This is very convenient to keep the date and person doing the entry and modifications. FIX xxxx will locate the source definition of xxxx and display the screen of this definition. It also invokes the editor to let you edit the definition.

WHERE is also a very useful command during program developing. When an error causes the text interpreter to abort, executing WHERE will call EDIT to display the screen where the error occurred while loading and the cursor will be pointing right at the word causing trouble. All the editing commands can then be used to fix the problem.

To use the screen editor, you have to select a screen as your current editing screen. Instead of the LIST

command as used in the line editor, you should use the EDIT command:

```
23  EDIT
```

invokes the screen editor to edit screen 23. The screen editor first checks the ID field at the end of line 0. If this field is blank, it will ask you to input a ten character string as a stamp to fill the ID field. The ID stamp helps you to keep track of the modifications you make on this screen. The contents of the screen are then displayed in the screen window on the top of the terminal display. You can now enter any of the line editing commands and the results will be shown immediately in the screen window. The command dialog will be scrolled in the command text window at the bottom of the display screen while the screen text is stationary in the screen window on the top of the display.

After you have completed the editing work and decide to leave the screen editor, you should type:

```
DONE
```

and the editor will save this screen to the disk file if you made any modification. The terminal display will be returned to its normal scrolling mode. To re-enter the screen editor to edit the same screen you just parted, you can use:

```
ED
```

without specifying the screen number as you would using EDIT.

3.3. LOADING AND TESTING PROGRAM

Screens of 1024 bytes are about the optimal size for writing and testing programs. The limited size forces you to modularize your program and eases the tasks of testing and debugging the program.

A source screen may contain three types of information: words to be interpreted or executed, new word definitions to be compiled to the dictionary, and comments. The text interpreter treats the source text the same way as it treats text entered from the keyboard. The command to ask the text interpreter to interpret source text in a screen is LOAD:

```
1  LOAD
```

will cause the system to fetch screen 1 from the current file and interpret its contents.

F83 has a few other commands to load source screens. They are collected in Table 3.3. here.

TABLE 3.2. LOADING COMMANDS

n LOAD	Interpret source text in screen n in the current file.
n m THRU	Interpret sequentially the source text in screens n to m.
n +LOAD	Load the screen that is n blocks from the current screen.

n m +THRU	Load a range of screens n blocks offset from the current screen.
-->	Exit the current screen and load the next screen immediately.

F83 system allows you to open two files at the same time by the command FROM. After you open a file using FROM, the LOAD command will load a screen from the from file instead of the current file. This way you can load utility programs from other files while still maintaining the file you are using as the current file. However, LOAD restores the current file to be the input file at the end of its operation, so that you will be able to refer to the current file. Thus you can only load one screen from the from file with the LOAD command.

3.4. MEMORY DUMP

LIST allows you to display text data in a screen. However, screens can also be used to store binary data or object code. Binary data cannot be listed on the terminal or printed by a printer. If binary data are accidentally sent to the terminal or printer, usually the terminal or printer will print garbage with lots of form-feeds. Sometimes they can be locked up by some binary code and you will have to turn off the entire system and re-boot. F83 provides a few commands to let you examine binary data in memory or in disk files.

The dump utility gives you a formatted hex dump with the ASCII text corresponding to the hex bytes, on the right hand side of the screen. Three words are available to specify the desired dumping actions. DUMP requires an address and a byte count on the stack to display the contents of a range of memory. The dump is always in hex, but the current base is not disturbed. DU dumps 64 bytes at the specified address. The address is incremented by 64 to facilitate dumping the next memory range of 64 bytes.

Examples of using these commands are:

```
HEX   100 80 DUMP   ( Dump 128 bytes starting from 100H.)
DECIMAL  256 DU DU DROP   ( Do the same thing as above.)
```

DL dumps the specified line on the current screen, with the line number as the input on the stack. This dump is useful in detecting nonprintable characters in the screen which disturb interpretation and compilation.

```
13 DL   ( Dump the 13th line in the current editing screen.)
```

A dumping example is shown in Fig. 3.1.

3.5. DEBUGGING YOUR PROGRAM

A program usually does not work and you will have to find out why it doesn't work and try to fix it. The advantage of Forth is that you can fix bugs quickly because loading a screen and testing the definitions in a screen is simple and fast. It allows you to experiment and try out ideas and methods to solve your problem. If you limit yourself writing programs one screen at a time and test the definitions in the screen fully, the problem can be solved very efficiently. Of course, you should make it easier for

yourself by writing short definitions which are easy to test, and if there is a problem, easy to spot the problem and fix it. Choosing good names for your definitions and commenting the stack effects will make the program more readable and easier to modify or update.

F83 system gives you a powerful debugging tool in case you cannot spot the bug by staring at the source code long and hard. The F83 debugger allows you to single step through a colon definition and show you the contents of the data stack at each step. By examining the data stack at each step, it is a simple matter to find when and how the bug gets into your program. During the tracing, you can jump back into Forth to poke around and change things like the data stack before continue the tracing. These functions in the debugger really help a Forth user to produce clean code.

There are two steps in using the debugger. First you have to prepare the word you want to trace use the DEBUG command. Then, you have to execute the word in the normal way it is used, with necessary data stack parameters. The word is then executed in steps. The computer displays the name of the word in the definition to be executed next and the contents of the data stack. You have to press a key on the keyboard for it to step to the next word. For example, we would like to debug the word LIST:

```
DEBUG  LIST
```

sets up the definition LIST so that it will be executed in steps. Then we can debug LIST by listing screen 1 using the patched LIST:

```
1  LIST
```

Figure 3.1 Memory dump

Now, LIST will be executed one step a time to allow you the examine the data stack at each step. The sequence of commands are shown in Fig. 2.7.

During single stepping, you can use three keys to divert the stepping action:

F Temporarily enter Forth so that you can use regular Forth commands to change stack values and anything else you care to do. Executing RESUME allows you to come back to the proper place to continue single stepping.
C Executing the rest of the definition continuously to the end without pausing.
Q Quit the debugger immediately and restore the debugged word to it original state.

If you keep your definitions simple and thoroughly test them as they are defined, you may not need this debugger. However, every once a while you will find that the capability in single stepping through a definition is very helpful in spotting some obscure bugs.

3.6. THE 8086 ASSEMBLER

F83 is available for 8080/Z80, 8086/88, 68000, and also 6502 CPUs. Each version of F83 comes with an assembler to assemble code routines in the machine code of the host CPU. The assembler is useful if you want to write machine code routines to speed up the execution of your program or to utilize special hardware features in your computer system. Since Forth is fast and quite efficient, and has the words to access memory and i/o directly, it is not necessary to dip into the machine code in normal circumstances. However, there are occasions that you have to optimize the program. As the assembler for your CPU is provided free in F83, we might just as well learn to use it for the fun of it. Since this book is devoted to the version of F83 for IBM PC, I will only discuss the 8086 assembler.

Another reason to describe the assembler in detail is that the kernel of the F83 system is written using the same assembler. Therefore, it is mandatory that you are familiar with the 8086 assembler if you want to dig into the F83 system and to apply it to do useful work.

8086 has 12 registers in its CPU. All these registers can be referred to in the assembler. However, the F83 system uses four of the registers to implement the virtual Forth machine. These registers used by Forth have special names to indicate their special functions in Forth, and they should be preserved inside the code definition. Table 3.3. shows the 8086 registers and the mapping to Forth registers.

In code routines, RP and IP must be restore if they have to be used. SP is used for data stack and must be maintained to pass parameters between words. AX, CX, DX, and DI can be used freely. W points to the code field of the word currently being executed. If this address is not needed, W register can also be used freely. The segment pointers can be used to address memory outside of the 64K code space, but they must also be restored to the original state before the end of the code routine.

The 8086 registers can be used in a number of different addressing modes. The addressing modes defined in the Forth 8086 assembler are shown in Table 3.4.

TABLE 3.3. 8086 REGISTERS AND FORTH REGISTERS

8086	Forth	Function of register
AX		Scratch, accumulator
CX		Scratch, counter
DX		cratch, I/O register
BX	W	Current word pointer
SP	SP	Data stack pointer
BP	RP	Return stack pointer
SI	IP	Instruction pointer
DI		Scratch
ES		Extra segment pointer
CS		Code segment pointer
SS		Stack segment pointer
DS		Data segment pointer

TABLE 3.4. REGISTER ADDRESSING MODES AND MNEMONICS

Addressing Mode	Mnemonics
8 bit register mode	AL CL DL BL AH CH DH BH
16 bit register mode	AX CX DX BX BP SP SI DI ES CS SS DS W RP IP
Indirect registe rmode	[SI] [DI] [BP] [BX] [RP] [IP] [W]
Indirect with index	[BX+SI] [SI+BX] [BX+DI] [DI+BX] [BP+SI] [SI+BP] [BP+DI] [DI+BP]
Immediate	#
Immediate address	#)
Inter-segment address	S#)

All the 8086 machine instructions are implemented in this F83 8086 assembler, making it rather complicated. It is not appropriate to discuss all the possible combinations of the instructions and the addressing modes. Here I shall discuss a few important aspects in making use of this assembler. You should read Chapter 24 on the assembler to study how the machine instructions are assembled and how the addressing modes are processed to put together complete machine instructions. It is also a good idea to study the code definitions in the Forth kernel, where we have hundreds of fine examples of code definitions. The best way to write a code definition is to find one of similar function in the kernel and make modifications to the existing code to put in the function you need.

A code definition must be started with the command CODE which creates a header in the dictionary and invokes the ASSEMBLER vocabulary to do the assembly work. Then a sequence of assembler mnemonic words are executed to assemble machine instructions into the body of the code definition. At the end of the code definition, there must be a command to return control to the Forth interpreter and a command to terminate the code definition:

```
CODE <name>
<assembly commands>
<return command>
END-CODE
```

where <name> is the name of the new code definition. CODE only creates the header. The Forth text interpreter is still in control after CODE, and the sequence of assembly commands is executed. Executing an assembly command causes a machine instruction to be assembled to the dictionary. END-CODE terminates the assembly process and makes the new code definition available for searching and compiling.

An assorted collection of assembly commands is shown in Table 3.5. Note that the assembler syntax is still reverse Polish: the operands are put before the assembly command. The operands, whether they are addresses, immediate values, registers, or specification of addressing modes, are all pushed on the data stack for the assembly command to consume and build the final machine code into the dictionary.

TABLE 3.5. 8086 ASSEMBLY COMMANDS, FORTH STYLE.

Assembly Command	Function
>NEXT #) JMP	Jump to address >NEXT.
1 # AX MOV	Move value 1 into AX register.
6 # RP ADD	Add 6 to the return stack pointer. (Intention: Pop three 16 bit numbers off the return stack.
8000 # BX ADD	Add 8000 to the contents of BX.
0 [IP] DX MOV	Copy the contents of memory pointed to by IP into DX register.
0 [W] W MOV	Copy the memory pointed to by W register into W register.
PDO JNE	Jump to address PDO if the zero flag in status is not set.
REP BYTE MOVS	Move a range of bytes in memory. Source pointed to by SI, destination pointed to by DI, and count in DX.
0 [BX] POP	Pop data stack into memory pointed to by BX register.
CX PUSH	Push contents of CX on the data stack.
AX LODS	Move memory pointed to by IP into AX register and increment IP by 2.
SP RP XCHG	Exchange data and return stack pointers.

There are three return commands which assemble jump instructions to the inner interpreter which returns control of the CPU to the next word to be executed or to the text interpreter.

TABLE 3.6. RETURN COMMANDS

NEXT	Assemble a jump to >NEXT routine so that the next word pointed to by the IP will be executed.
1PUSH	Assemble a jump to APUSH routine which pushes AX on data stack and then falls into >NEXT.
2PUSH	Assemble a jump to DPUSH routine which pushes DX on data stack and then falls into 1PUSH.

F83 uses a centralized return mechanism by which all code definitions eventually execute the code routine at >NEXT, the inner interpreter. This single return point makes it possible to implement the powerful debugger which patches >NEXT to a debugging routine to single step through the execution sequence.

CODE generates executable machine instruction routines which behave the same externally as high level colon definitions. The code routines are easy to test under the Forth operating system. However, to write a piece of code which can be shared by many code definitions, it is necessary to build subroutines. The command to define a subroutine is LABEL, which is similar to VARIABLE in the sense that it returns an address when invoked. This address can be used in other code definitions to assemble a CALL instruction doing the subroutine calling. A LABEL routine cannot be executed or tested directly. It can only be called inside a code definition.

Within a code definition, the execution path can be altered or repeated using structure words IF...ELSE...THEN, BEGIN...UNTIL, and BEGIN...WHILE...REPEAT, similar to those used in colon definitions. The principal difference is that the test conditions required by IF, UNTIL, and WHILE are not taken from the data stack but from the status register. These conditional commands must be preceded by machine code conditionals so that proper conditional branching instructions can be assembled. The machine code conditionals are listed in Table 3.7.

TABLE 3.7. MACHINE CODE CONDITIONALS

Forth Conditional	Assembled Code
0=	JNE/JNZ
0<>	JE/JZ
0<	JNS
0>=	JS
<	JNL/JGE
>=	JL/JNGE
<=	JNLE/JG
>	JLE/JNG
U<	JNB/JAE
U>=	JB/JNAE
U<=	JNBE/JA
U>	JS
OV	JNO

The Forth conditionals are reverse of the assembler code because the IF, UNTIL, and WHILE are skip-on-contition, not jump-on- condition as in the machine instructions. A machine instruction preceding the jump instruction sets the flags in the status register in 8086 CPU for the jump instruction to select the next instruction to be executed. An example of the branching structure in Forth style is:

```
5 # CX CMP   0< IF   AX BX ADD   ELSE   AX BX SUB   THEN
```

The structure commands IF, ELSE, etc., are different from those used in the colon definitions. The assembler structure commands are in the ASSEMBLER vocabulary and those in colon definitions are in the FORTH vocabulary. Their behaviors are quite different.

A very good example is the code definition of UPPER which converts a string of Ascii characters to upper case characters:

```
CODE   UPPER        ( addr length --- )

        CX POP   BX POP          Get parameters from F83 stack into CPU registers.
        BEGIN                    Setup the loop.
                CX CX OR         Is the count in CX zero?
        0<> WHILE                Exit the loop if CX is zero.
                0 [BX] AL MOV    Get one character to AL.
                >UPPER #) CALL   Call a subroutine to convert the character to upper case.
                AL 0 [BX] MOV    Put the converted character back into the string.
                BX INC           Address of the next character.
                CX DEC           Character count.
        REPEAT                   Convert the next character.
        NEXT              Done.  Return.
        END-CODE
```

CX CX OR sets up the status flag and 0<> WHILE sets up the conditional jump instruction to exit the loop. REPEAT assembles a jump instruction back to CX CX MOV to continue processing the next character until the character string is exhausted.

Code definitions are difficult to debug and are machine dependent. They are defined only as the last resort to squeeze performance out of your computer and should not be considered lightly. In most programs, there are only a few critical words which are executed very often. You should try to identify these words and convert only these words to code definitions.

3.7. MULTI-TASKER

The early Forth systems implemented by Charles Moore were multi- tasking systems and could support many users or terminals to operate simultaneously. The implementors of fig-Forth neglected this feature in the fig-Forth Model. Because of the popularity of fig-Forth, Forth has been regarded by most users as a system only suitable for single user without the multi-tasking capability. The Forth architecture was designed to be able to run many tasks at the same time. The authors of F83 restored this important feature of Forth to the F83 system without too much effort. The source code to put the multi-tasker back consumes only about six screens. Try to do it in PASCAL!

Commands to create one or more tasks are already defined in the FORTH vocabulary. A task must be first created by the command TASK:, which allocates space in the dictionary needed by a task, including areas for user variables, return stack, and data stack. When the task is later activated by the command ACTIVATE, the task will execute a sequence of words and share the CPU with the regular FORTH system in a round robin task switching scheme. Each task uses the CPU until it voluntarily gives it up by executing PAUSE or STOP and releases the CPU to the next task in the round robin chain. All system I/O words have PAUSE's embedded to switch tasks automatically. In the word sequence give to a task to execute, the command PAUSE or STOP must be placed properly so that the task will not hold on to the CPU indefinitely.

Background tasks are defined by the word BACKGROUND: . Examples are:

```
BACKGROUND:   SPOOLER   1 CAPACITY SHOW STOP   ;
```

SPOOLER is defined as a background task which lists a complete screen file to the printer while you still have full control over the Forth computer via your keyboard.

Once the SPOOLER is defined as a task, it can be re-assigned to perform other chores. For example:

```
: SPOOL-THIS   SPOOLER ACTIVATE   3 15 [ SHADOW ] SHOW STOP ;
```

will print the screens 3 to 15 with their shadow screens in a six screens per page format.

Another example is to maintain a counter counting the cycles around the round-robin task switching loop. A global variable COUNTS is defined to keep the counts so that you can examine it any time you want to.

```
VARIABLE  COUNTS
BACKGROUND:   COUNTER   BEGIN PAUSE 1 COUNTS +! AGAIN  ;
```

which increments the variable COUNTS indefinitely at the background. You can run the computer in the foreground, doing any thing you care to. Occasionally, you can read COUNTS to see how many times the computer runs around the task switching loop.

After background tasks are defined, the command MULTI starts the multi-tasker running. The command SINGLE stops the multi-tasker. Individual background tasks can be stopped or restarted by the following commands:

```
SPOOLER  SLEEP   ( Put spooler on hold.)
SPOOLER  WAKE    ( Restart the spooler.)
```

The multi-tasker is fun to play with. You have to try it yourself to appreciate it. Windows are built this way.

3.8. SAVE A SYSTEM IMAGE

Suppose you have developed an application using F83 and also found a company interested in buying this program from you. It would be nice if the user of this program can simply insert a disk into his drive, type a magic word, and have the entire program running immediately. He just wants to use the program and does not have any interest on how this thing is written or how it works. If you sell this program to the company, you may not want to reveal to them all the source code you developed so that they will short cut you. The best solution is to give them an executable object file which can be loaded into the computer and run, but is very difficult to decipher and modify.
F83 gives you a tool to generate an executable object file from the dictionary of your running Forth system. When this object file is loaded into the memory through the operating system, it restores the computer to the same state as your current system. The command to do it is SAVE-SYSTEM:

```
SAVE-SYSTEM  GISMO.COM
```

copies the entire dictionary into a file named GISMO.COM. When you boot your system in DOS or CP/M, typing GISMO will load this file into memory and start the Forth system. Now, you can execute the highest level word in your application and run it.

3.9. THE META-COMPILER

Using SAVE-SYSTEM you make an object file with your application program overlay on the top of the F83 Forth system. It is fine this way if you allow the user open access to F83 system. However, allowing an unsophisticated user unlimited access to Forth can be disastrous for the health of his computer and to your profit. Another problem is that there is lots of code in the F83 system not needed by your application, but it is also saved in the object file doing nothing but occupying RAM space. The meta-compiler in F83 allows you to strip out the dead wood and build an application package best tailored to the application. It recompiles the entire Forth system with your application. During the compilation, you have the option to omit any definition not needed by your application. If the target computer is not identical to the host computer you used to develop the application, you can also modify the source code in the kernel so that the application can be run on the target computer. The ability to generate a new Forth system from a Forth system is called meta-compilation. Meta-compilation is sometimes referred to as the 'extensibility of the third kind', which is the highest level of programming activity on a computer.

F83 system was itself generated by the meta-compilation process, and all the source code needed for its self-generation is included in it. The command sequence to create the executable object image F83.COM is as follows:

1. Prepare a disk with F83.COM file on it. This F83.COM file is the Forth system which will do the meta-compiling. Insert this disk into drive A.
2. Prepare a disk with METAnn.BLK and KERNELnn.BLK files on it. METAnn.BLK contains the meta-compiler and KERNELnn.BLK contains source code for the bare-bones Forth system. Insert this disk into drive B.
3. Log on to drive B, and type the following command:

```
A:F83  METAnn.BLK
```

1 LOAD

F83 is now loaded and builds a bare-bones Forth system, which is stored on the disk in drive A with a name KERNEL.COM.

4. Type BYE to return to DOS or CP/M.
5. Copy KERNEL.COM to a new disk with three files EXTENDnn.BLK, CPUnnnn.BLK, and UTILITY.BLK. Insert this disk in drive A.
6. Log onto drive A and type the following command:

   KERNEL EXTENDnn.BLK
   1 LOAD
   All the extensions and utilities are now loaded and a new version of F83.COM is created on drive A.
7. Type BYE to return to DOS.

nn above stands for 80, 86, or 68 depending upon the host computer running the F83 system, and nnnn stands for 8080, 8086, or 68000.

The above procedure was used to generate the F83 system. If you want to modify the system and add your own programs to the system, you have to prepare the files and load them in a sequence similar to the above sequence. I must remind you that meta-compiler is very complicated because it has to compile the target code to a virtual memory space where the code cannot be executed. There are many important conditions which must be satisfied for a successful meta-compilation, such as forward references, proper initialization of system variables, and the initialization of all the vectored execution routines. These subjects will be discussed in detail when we study the code in the meta-compiler. You have to understand the meta-compiler fully before attempting your modifications.