Preface to the Fourth Edition

It is thirty years since F83 was first released by Mike Perry and Henry Laxen. This book was out of print a long time ago. Yet, I still receive requests for the printed copies. The book was originally produce on a CP/M microcomputer, with a Diablo daisy wheel printer. All the text was preserved, but the figures and tables could not be reproduced. Earlier, I released an electronic edition with figures and tables scanned from the original book. It was embarrassing, because it exceeded the 25 MB limit for email delivery, due to the scanned images. I think it is probably time to do a better job, taking advantages of Microsoft Words with its fonts and formatting capabilities.

I am using the 12 point Times New Roman font for all narration, and the 8 point Courier New font for all source code and documentation. Code and documentation are presented in two columns. Left column is for code and right column for documentation. As the left column is generally 1.5 inches wide, I allow only 1 character space for each level of indentation. It is hard to see the nested levels, but I think is adequate if you do want to inspect the code in detail.

The only significant modification is on the term of ‘words’. In the original book, Forth commands were called Forth words, Forth definitions, and sometimes Forth commands. Now I have decided to call things which are executed by the host computer machine instructions or just instructions. What’s executed by the Virtual Forth Computer are commands, which are mostly colon commands and code commands. I think it is now very consistent in the narration. However, in the documentation, words and definitions are not changed.

Many figures were listings produced by an old IBM PC which was sent to dump yard long ago. I was extremely please that I could get the 8086 F83 v.2.10 to run in a small CMD window on my old desktop PC with Windows XP. The listings were produce by redirecting text output to a file. F83 is still alive! It reads and writes to a floppy disk in the A: drive. Boy! I am lucky that I keep this PC which still has a 3.5” floppy drive. I almost threw it away, as it developed some intermittent disk problems.

Seeing a working F83 system in front of me, it’s like seeing a good old friend coming back after 30 years. It makes me feel encourage, and gives me hope that this new edition of Inside F83 may still be useful for some friends out there. It will also save some trees.

C. H. Ting

July 2013
San Mateo, California
Preface to the Third Edition

It is almost eight years since F83 was first released by Mike Perry and Henry Laxen. It has been widely distributed by many shareware and freeware distributors, as well as through many bulletin boards. It also has found its way into many real applications and useful products. Although we have seen many better public domain Forth systems brought out over the years, F83 still stands out because of its high quality and because it is available on three very popular microprocessors: 8080, 8086, and 68000.

The quality of F83 is testified by the fact that over the years, we have found only one bug in the 8086 F83 system (DOS Version 2.10). This bug was discovered by Mike Yantis at Maxtor Corp. The ENTRY cell in the user area contains 90H (NOP) and E9H (JMP) when the task is asleep. E9H causes a jump to the next task, thus skipping the current task. These two bytes are changed to CDH and 80H (INT 80) when the task is to be waken. INT 80H wakes up this task when the CPU control is passed to the task. This scheme works fine in most instances. This mechanism falls apart only if the waking up routine is activated by an interrupt, and if the interrupt hits when the CPU just finishes executing the NOP (90H) instruction and is ready to execute the JMP (E9H) instruction. Unfortunately, the waking up routine secretly changed E9H to 80H, whose behavior at this point is unpredictable and in most cases crashed the CPU. The probability of this occurring is very small, only about once in 100,000 interrupts, which were enough to bother Mike Yantis. Mike fixed this bug by choosing INT E9H to start the wakened tasks.

Discussing this bug in detail is meant to be a compliment to Mike Perry and Henry Laxen in their efforts producing the F83 system. It took a bug of such a oblique nature to escape Mike and Henry's tight grips.

I take this opportunity to revise the book and produce it using a laser printer. I am always amazed at how a laser printer can transform lies into truth. In spite of the laser, I like to give special thanks to Jay McKnight in reviewing the text and corrected many of my grammatical and technical errors.

F83 is one among the very few Forth systems which are useful while still understandable. Inside F83 had helped many people gain the privilege to peek inside a fully functional Forth system. I hope it will help you also. Not just take a peek, but use it as a key and open a whole new field to yourself.

C. H. Ting

June 1991
San Mateo, California
Preface to the Second Edition

After I implemented my first Forth system on a Data General Nova computer and got the 'OK' message, I went back home and told my wife: "I just promoted myself from an applications programmer to a system programmer!" I was so excited and my brain was so completely filled with the intricate details of the Forth fabric that the only way to get hold of myself was to dump everything on paper. That was the Systems Guide to figForth. I xeroxed it and brought a boxful to the then Northern California FIG meeting and it was sold out immediately. Apparently I had struck a chord in the Forth community which was desperately in need of documentation and instruction on Forth internals.

I was fortunate that a polyForth on LSI-11 computer was available at work. I tried to convince Forth, Inc. to publish a similar book on polyForth and obtained some support to proceed. I sent a draft manual, titled Systems Guide to polyForth to Forth, Inc. Somehow, Forth, Inc. decided not to publish or promote it, and had left it on their bookshelf. I heard that it found its way into the underground Forth circle in Southern California. PolyForth is concise and powerful, and it deserves better system documentation than what is provided. I was very impressed as I went through it screen by screen. I was delighted in picking a great mind, that of Chuck Moore himself. It was like poetry.

When Mike Perry and Henry Laxen released their public domain F83 system, I bought a listing to read. It was a very worthy product, with lots of tools and utilities. The best part is that it is complete with on-line documentation in the form of shadow screens. I thought there was little for me to contribute. As F83 was spreading wider, we started to hear more complaints about the difficulties in learning and using it. I reached a conclusion that Forth screens are good medium for programming, but a screen is too small a window for viewing and learning a large Forth system, even with shadows. In reading the source code, it is necessary to look at many screens at the same time, quickly moving from one screen to another while keeping everything in plain view. We have all been conditioned to read things in the printed form, making the best use of our visual system with instant zooming and panning capability. The visual system is very difficult to emulate with a 24 by 80 character screen.

Then Wil Baden came to one of the FIG meetings and showed the completely sorted index of F83 words in all vocabularies. I rushed to the front table and grabbed a copy of his handout with the index, which was the tool I needed to navigate through the F83 system. With the help of the index, lots of midnight oil, and ignoring my wife's orders to clean up my room, I was able to rearrange the source code of F83 in a form more tangible to mortal souls. Most of the work was simply rearranging the source code from the horizontal to the vertical format and fill the right hand side of the page with words taken from the shadow screens. I collected related source code and present it in a logical sequence, which often does not coincide with the loading sequence of the source screens. Once the code is ordered logically, you will find it is much easier to comprehend this very large and seemingly intimidating system.

F83 provides a very extensive and solid foundation for professional programmers to build application packages. It is also a very useful source for beginners to learn Forth programming style and techniques. Its problem, as in any large Forth system, is the fragmentation of functions
in a multitude of words. With more than 1000 words, it is very difficult to have a firm handle on F83. However, functions a user needs to program a computer application or to use a computer application are not that many. Once you are familiar with those top level utility words, you can dig into the underlying low level words and use them to build your own castles. This book, I hope, will serve the purpose of showing you the power and the beauty behind the Forth language.

We are all indebted to Mike Perry and Henry Laxen for releasing the F83 system into the public domain. It certainly sets a higher standard for commercial Forth systems and forces Forth vendors to provide more powerful systems and better user support. Anything less than F83 will not be acceptable anymore. Thanks are also due to Dr. S. Y. Tang and Mr. John Peters for reading the manuscript and making numerous suggestions and corrections. The Chinese brush painting on the covers was provided by my mother, Mrs. I-Jean Hwang Ting. My father, Mr. C. W. Ting, was the editor and also managed the production of this book. This is a traditional Chinese family business, small but efficient and very Forth-like.

C. H. Ting

May 1985
San Mateo, California
## INSIDE F83

### Contents

**Part I  Introduction to F83 system**

1. **The heritage of F83**
   1.1 The roots of the F83 1
   1.2 Advancements in Forth-83 Standard 3
   1.3 Creators of F83 system 4
   1.4 Features of F83 system 6

2. **Browsing F83 system**
   2.1 Listing the word names 9
   2.2 Vocabulary 10
   2.3 Viewing source code of word definitions 13
   2.4 Shadow screen documentation 15
   2.5 Files in F83 16
   2.6 Printing utility 17
   2.7 Debugger 19

3. **Using the F83 system**
   3.1 Create your own file 21
   3.2 The editor 23
   3.3 Loading and testing your program 25
   3.4 Memory dump 26
   3.5 Debugging your program 28
   3.6 The 8086 assembler 29
   3.7 Multitasker 33
   3.8 Save a system image 34
   3.9 The meta-compiler 35

**Part II  The Forth kernel**

4. **Interface to the host computer**
   4.1 Virtual Forth computer 37
   4.2 Forth computer hosted on 8086 38
   4.3 Inner interpreters 41
   4.4 Interpreters for in-line data and strings 46
   4.5 Interpreters for control structures 48

5. **The Forth nucleus**
   5.1 8086 assembly language in Forth 51
   5.2 Code definitions in Forth nucleus 52
   5.3 Examples of code definitions 53

6. **Terminal input and output**
   6.1 The BDOS I/O calls to the operating system 56
   6.2 Terminal output commands 57
   6.3 Interpreting control characters 57
   6.4 More sophisticated input commands 58
   6.5 String commands 60

7. **The virtual memory** 62
### 15. Text editors

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>String utility</td>
<td>121</td>
</tr>
<tr>
<td>15.2</td>
<td>Terminal dependent deferred words</td>
<td>121</td>
</tr>
<tr>
<td>15.3</td>
<td>The cursor commands</td>
<td>123</td>
</tr>
<tr>
<td>15.4</td>
<td>Editing buffers</td>
<td>123</td>
</tr>
<tr>
<td>15.5</td>
<td>Line editing commands</td>
<td>125</td>
</tr>
<tr>
<td>15.6</td>
<td>String editor commands</td>
<td>127</td>
</tr>
<tr>
<td>15.7</td>
<td>Screen editor</td>
<td>128</td>
</tr>
<tr>
<td>15.8</td>
<td>The screen display commands</td>
<td>129</td>
</tr>
<tr>
<td>15.9</td>
<td>The screen editor commands</td>
<td>131</td>
</tr>
<tr>
<td>15.10</td>
<td>Configuring the terminal</td>
<td>133</td>
</tr>
</tbody>
</table>

### 16. Viewing source screens

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>The view field</td>
<td>136</td>
</tr>
<tr>
<td>16.2</td>
<td>The view files</td>
<td>136</td>
</tr>
<tr>
<td>16.3</td>
<td>The viewing command</td>
<td>137</td>
</tr>
</tbody>
</table>

### 17. WORDS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1</td>
<td>Output formatting commands</td>
<td>140</td>
</tr>
<tr>
<td>17.2</td>
<td>WORDS</td>
<td>140</td>
</tr>
</tbody>
</table>

### 18. Disk file utility

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1</td>
<td>Displaying screens in a file</td>
<td>142</td>
</tr>
<tr>
<td>18.2</td>
<td>Disk buffers</td>
<td>142</td>
</tr>
<tr>
<td>18.3</td>
<td>Single block copying</td>
<td>143</td>
</tr>
<tr>
<td>18.4</td>
<td>Multiple block copying</td>
<td>144</td>
</tr>
<tr>
<td>18.5</td>
<td>Multiple file copying</td>
<td>144</td>
</tr>
</tbody>
</table>

### 19. Memory dump

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1</td>
<td>The dumb DUMP</td>
<td>147</td>
</tr>
<tr>
<td>19.2</td>
<td>The smart DUMP</td>
<td>147</td>
</tr>
</tbody>
</table>

### 20. Decompiler

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1</td>
<td>Positional case defining word</td>
<td>149</td>
</tr>
<tr>
<td>20.2</td>
<td>Associative defining word</td>
<td>149</td>
</tr>
<tr>
<td>20.3</td>
<td>Decoding different classes of words</td>
<td>150</td>
</tr>
<tr>
<td>20.4</td>
<td>Sorting and execution tables</td>
<td>151</td>
</tr>
<tr>
<td>20.5</td>
<td>Decompiling different word classes</td>
<td>152</td>
</tr>
<tr>
<td>20.6</td>
<td>Word classification</td>
<td>153</td>
</tr>
<tr>
<td>20.7</td>
<td>The decompiler SEE</td>
<td>154</td>
</tr>
</tbody>
</table>

### 21. Printing utility

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1</td>
<td>Variables and setup</td>
<td>157</td>
</tr>
<tr>
<td>21.2</td>
<td>Printing two screens side by side</td>
<td>157</td>
</tr>
<tr>
<td>21.3</td>
<td>Printing 6 screens on a page</td>
<td>158</td>
</tr>
<tr>
<td>21.4</td>
<td>SHOW</td>
<td>159</td>
</tr>
</tbody>
</table>

### Part IV 8086 Specific utilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.</td>
<td>Debugger</td>
<td>163</td>
</tr>
<tr>
<td>22.1</td>
<td>Low level supporting words</td>
<td>163</td>
</tr>
<tr>
<td>22.2</td>
<td>High level trace commands</td>
<td>164</td>
</tr>
<tr>
<td>23.</td>
<td>Multitasker</td>
<td>166</td>
</tr>
<tr>
<td>23.1</td>
<td>Multitasking</td>
<td>166</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>23.2</td>
<td>User variables and the user area</td>
<td>166</td>
</tr>
<tr>
<td>23.3</td>
<td>PAUSE and RESTART</td>
<td>168</td>
</tr>
<tr>
<td>23.4</td>
<td>The multitasker</td>
<td>170</td>
</tr>
<tr>
<td>23.5</td>
<td>Task definition</td>
<td>170</td>
</tr>
<tr>
<td>23.6</td>
<td>Background tasks</td>
<td>171</td>
</tr>
<tr>
<td>24.</td>
<td>8086 Assembler</td>
<td>173</td>
</tr>
<tr>
<td>24.1</td>
<td>Assembly tools</td>
<td>173</td>
</tr>
<tr>
<td>24.2</td>
<td>8086 register definitions</td>
<td>174</td>
</tr>
<tr>
<td>24.3</td>
<td>Addressing mode operators</td>
<td>176</td>
</tr>
<tr>
<td>24.4</td>
<td>Defining words to generate opcodes</td>
<td>180</td>
</tr>
<tr>
<td>24.5</td>
<td>Special opcodes</td>
<td>183</td>
</tr>
<tr>
<td>24.6</td>
<td>Structures in code definitions</td>
<td>185</td>
</tr>
<tr>
<td>25.</td>
<td>Metacompiler</td>
<td>188</td>
</tr>
<tr>
<td>25.1</td>
<td>Concept of metaccompilation</td>
<td>188</td>
</tr>
<tr>
<td>25.2</td>
<td>Vocabularies for metaccompilation</td>
<td>189</td>
</tr>
<tr>
<td>25.3</td>
<td>Accessing memory in the target system</td>
<td>191</td>
</tr>
<tr>
<td>25.4</td>
<td>Branching constructs</td>
<td>192</td>
</tr>
<tr>
<td>25.5</td>
<td>Forward referencing</td>
<td>194</td>
</tr>
<tr>
<td>25.6</td>
<td>Compiling new words to target system</td>
<td>195</td>
</tr>
<tr>
<td>25.7</td>
<td>Transition compiler directives</td>
<td>196</td>
</tr>
<tr>
<td>25.8</td>
<td>Defining words in metacompiler</td>
<td>198</td>
</tr>
<tr>
<td>25.9</td>
<td>User variables</td>
<td>199</td>
</tr>
<tr>
<td>25.10</td>
<td>Vocabulary</td>
<td>199</td>
</tr>
<tr>
<td>25.11</td>
<td>Resolving forward references</td>
<td>200</td>
</tr>
<tr>
<td>25.12</td>
<td>Redefining host words</td>
<td>201</td>
</tr>
<tr>
<td>25.13</td>
<td>Running the metacompiler</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>198</td>
</tr>
</tbody>
</table>
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Forth family tree</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>The standard bearer</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>IBM-DOS files in F83 system</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Forth commands</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Assembler and DOS commands</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Commands in other vocabularies</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>VIEW and SEE</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>File and directory commands</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Debugging LIST</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Memory dump</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>The virtual Forth computer</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory map of F83 system</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Representation of strings</td>
<td>61</td>
</tr>
<tr>
<td>7.1</td>
<td>The file control block</td>
<td>64</td>
</tr>
<tr>
<td>7.2</td>
<td>Disk buffer management</td>
<td>68</td>
</tr>
<tr>
<td>8.1</td>
<td>Structure of a Forth command</td>
<td>75</td>
</tr>
<tr>
<td>8.2</td>
<td>Vocabularies and dictionary structure</td>
<td>77</td>
</tr>
<tr>
<td>8.3</td>
<td>Four-way threading in a vocabulary</td>
<td>80</td>
</tr>
<tr>
<td>9.1</td>
<td>Input and output number conversion</td>
<td>83</td>
</tr>
<tr>
<td>10.1</td>
<td>Parsing with WORD</td>
<td>91</td>
</tr>
<tr>
<td>12.1</td>
<td>The interpreter and the compiler</td>
<td>101</td>
</tr>
<tr>
<td>13.1</td>
<td>Numeric data structures</td>
<td>107</td>
</tr>
<tr>
<td>13.2</td>
<td>The string literals</td>
<td>108</td>
</tr>
<tr>
<td>13.3</td>
<td>The control structures</td>
<td>111</td>
</tr>
<tr>
<td>15.1</td>
<td>The editing buffers</td>
<td>126</td>
</tr>
<tr>
<td>15.2</td>
<td>Screen editor display</td>
<td>131</td>
</tr>
<tr>
<td>16.1</td>
<td>The view field and the view files</td>
<td>137</td>
</tr>
<tr>
<td>20.1</td>
<td>Decoding different types of commands</td>
<td>152</td>
</tr>
<tr>
<td>20.2</td>
<td>Decompile different commands</td>
<td>156</td>
</tr>
<tr>
<td>21.1</td>
<td>Two printing formats</td>
<td>160</td>
</tr>
<tr>
<td>23.1</td>
<td>The round robin task scheduler</td>
<td>169</td>
</tr>
<tr>
<td>24.1</td>
<td>Register addressing mode constants</td>
<td>175</td>
</tr>
<tr>
<td>24.2</td>
<td>8086 instruction types</td>
<td>179</td>
</tr>
<tr>
<td>25.1</td>
<td>The chicken-egg cycle of meta-Forth</td>
<td>285</td>
</tr>
<tr>
<td>25.2</td>
<td>Supporting vocabularies for metacompilation</td>
<td>191</td>
</tr>
</tbody>
</table>
## Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Vocabularies in F83</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Editor commands</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Loading commands</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>8086 registers and Forth registers</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Register addressing modes and mnemonics</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>8086 assembler commands, Forth style</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Return commands</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Machine code conditionals</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>8086 Register assignments for Forth</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>String commands</td>
<td>60</td>
</tr>
<tr>
<td>9.1</td>
<td>Data representation</td>
<td>81</td>
</tr>
<tr>
<td>23.1</td>
<td>User variables</td>
<td>167</td>
</tr>
</tbody>
</table>
Part I. Introduction to F83 System

Chapter 1. The Heritage of F83

1.1. The Root of F83

Forth was invented by Charles Moore in the 60's as he developed specialized programming tools for various software projects and crystallized them into a language-operating system. Forth spread to many continents following the radio telescopes originally programmed by Mr. Moore when he was with the National Radio Astronomy Observatory. Forth was so prevalent in the astronomy communities that the International Astronomical Union formally adopted Forth as their standard programming language in 1974.

Mr. Moore and some of his colleagues left NRAO and formed the Forth, Inc. to market Forth systems and services in 1972. Over the years, a series of Forth implementations were produced for commercial minicomputers and microcomputers. These products evolved into poly-Forth, which contained many advanced features such as interrupt drive I/O, multitasking-multiprogramming, data base management, transcendental functions, and meta- or target compilation. It remains the most comprehensive Forth system in the Forth marketplace.

Forth users in Europe organized a user's group, called EuropeanForth users Group (EFUG). To encourage the exchange of Forth programs and information, EFUG published a list of Forth commands with standard commands, commonly known as the Forth-77 Standard. It documented the then often used Forth commands in an effort to prevent these commands from mutation as more Forth systems were installed.

The Forth Interest Group was organized in 1978 to encourage the use of Forth on small personal computers. Two major activities sponsored by FIG in 1978 were the publication of figForth Model and the organizing of the Forth Standards Team. Because of the low costs of the figForth source listings and the quality of these figForth implementations, figForth became the de facto standard of Forth on small computers. The product from the Forth Standards Team, the Forth-78 Standard, however, was not as successful. It was soon orphaned by FIG. The Forth Standards Team went back to the drawing board and produced the Forth-79 Standard which was much more
precise in wording and consistent in the naming of standard commands. Many vendors including Forth, Inc. made genuine efforts to adopt this standard into their products.

Several problems kept the Forth Standards Team working. Among them, the more serious ones are the state dependence of many commands, the loop structure, the representation of falsehood, integer division with negative divisor, and the naming of many commands. These problems were
resolved in the publication of the Forth-83 Standard in early 1984. The exhausted Forth Standards Team decided that no new Forth standard will be considered in the near future to let Forth-83 Standard some time to establish itself in the Forth community.

1.2. Advancements in Forth-83 Standards

Major improvements in Forth-83 Standards over previous Forth standards are briefly discussed here. Exhaustive discussions have appeared in Forth Dimensions, authored by the Secretary of the Forth Standards Team, Dr. Robert L. Smith. Some of the more significant features in the Forth-83 Standard are summarized here.

Mono-Addressing
Four addresses are used to address different fields in a command in the dictionary, the name field address, the link field address, the code field address, and the parameter field address. To allow maximum implementation flexibility and code portability, the 83-Standard uses only one address, the compilation address. It is equivalent to the code field address in the figForth model. The compilation address is the one returned by ‘ and FIND, compiled to colon commands, and used by EXECUTE to run the command. Only one extra command is provided in the standard to access information stored in the parameter field: >BODY.

The importance of the compilation address cannot be overstressed. Mono-addressing recognized this characteristics of Forth. It is also beneficial that the compilation address serves as the focal point in locating information stored in the commands.

Eliminating State Smart Commands
Many Forth commands in the early standards execute differently depending upon whether the system is in the execution mode or compiling mode, like LITERAL, ‘ (tick), ." , etc. In Forth-83 the state smart commands are either eliminated or separate commands are defined for interpreting and compiling states, making the system less ambiguous and faster.

Old ‘ is split into two commands: ‘ and [] . Old ." is split into .( and ." .

Improved DO-Loop
The DO-LOOP structure went through a major overhaul in Forth-83. The range of index is extended to 64K so that full memory range can be addressed in the loops.

LEAVE is made to terminate the loop immediate upon its execution rather than wait until LOOP is executed.
**Improved Division**
Division is now floored towards negative infinity instead of rounded towards zero. It is more useful in that the quotient and modulus have a smooth change between positive integer and negative integer domain when either divisor and/or dividend is negative.

**Representation of True Flag**
A true flag generated by the Forth-83 system is represented by -1 instead of 1 in the older standards. -1 is more useful than 1 in doing bit-wise logic operation.

Consequently, NOT can now be defined as one's complement operation instead of being simply an alias of 0=.

**Zero-based PICK and ROLL**
Top of the data stack is treated as the based of a memory area and addressing into this area is zero based like addressing regular memory areas.

**WORD Returning an Address**
Word buffer was generally assumed to be at the top of the dictionary. However, this is implementation dependent. With WORD returning the address of the word buffer, the word buffer can be assigned to other memory areas. Practical usage of WORD always requires the address of the WORD buffer. Including the word buffer address in the WORD function is a convenience to you. A slight speed advantage can also be realized.

1.3. Creators of the F83 Systems

F83 is a very extensive language and operating system created by Henry Laxen and Mike Perry, two professional Forth programmers in Berkeley, California. Both of them have been active in the Forth Interest Group since its beginning, and participated in the work of the Forth Standard Team to develop Forth-79 Standard and Forth-83 Standard. They have published papers and written tutorials on Forth in Forth Dimensions and in the FORML (Forth Modification Laboratory) proceedings. As Forth-83 Standard evolved, they felt the need of a complete Forth system based on this standard to carry the standard to Forth users and community, in the same way the figForth implementations on the 6 popular microprocessors carried the figForth model. F83 system was the result of their efforts.
The F83 system was designed to use the CP/M operating system as its host for terminal input-output and disk interface so it is rather straightforward to be transported to a variety of microcomputers using the CP/M system. It has been implemented for 8080-Z80, 8086-8088, and 68000 CPU’s. Laxen and Perry put this system in the public domain, according to the tradition of the Forth Interest Group, as a vehicle to distribute the newly established Forth-83 Standard. Wil Baden at Los Angles ported it to the Apple II computer and named his system F83X.

F83 Version 1.0 was released by Perry and Laxen in September, 1983, shortly after the Forth-83 Standard was published. They also organized an F83 working group in the Northern California to evaluate the F83 system for practical applications. The F83 system, over the period of about a
year, was enhanced and upgraded several times. The latest version, Version 2.1 was released in the summer of 1984. The authors promised that this version will not be modified in the near future, and it should be stable for you to get familiar with it and to be distributed to a wider audience.

This book is meant to be a reference manual to the F83 system. It was originally written for the F83 system Version 1.0 for the 8086 processor on a CP/M86 operating system. Since the release of Version 2.1, it was also upgraded to this version. Due to the overwhelming spread of the IBM Personal Computers and its compatible models, it was also modified for using with the F83 system for the MS-DOS system. As for the other F83 systems, it is useful as a reference for all the high level Forth commands. For low level machine code commands, you will have to refer to the source code and documentation coming with the specific F83 system.

1.4. Features of F83 System

F83 is not a toy language like most other public domain and some commercial Forth system. It contains all the necessary utilities and tools for you to develop application programs conveniently and efficiently. Both executable object codes and the source codes are provided and distributed on floppy diskettes in the machine readable form. Although Laxen and Perry do not intend to provide support and consultation on the F83 system, as they called their publishing firm No Visible Support, Inc., the systems they distributed are of excellent quality and can stand on their own strength. Extensive documentation are provided in the form of shadow screens and in-line comments.

Laxen and Perry intended that F83 should demonstrate and bring out the best features in Forth as a professional programming language. Many utilities and tools were included in this system which are not generally available even in the best of the commercial Forth systems. Some of these utilities are listed here:

- Editor
- Assembler for the host CPU
- Full BIOS/BOS interface
- Multiple file accessing
- Four-way threaded dictionary
- Dynamically defined vocabulary search order
- Source code viewing
- Decompiler
- Debugger
- Memory dumping
- Multitasking
- Shadow-screen documentation
- Source and documentation printing
- Forward referencing
- Metacompiler
- Huffman code compression and expansion
In realizing all these functions, F83 has more than 1000 commands in its dictionary, comparing with about 300 commands in the figForth model and 130 commands in the required command set of Forth 83-Standard. Casual users may not need to know the details of all these commands. However, this whole F83 system is a huge reservoir of Forth programming examples from which serious Forth programmers can study and find ready solutions to many of their programming problems.

FigForth became the de facto standard of Forth on small computer systems because of the quality and the availability of the source listings distributed by the Forth Interest Group. The figForth system is complete in the sense that it includes all the necessary system functions so that it can be implemented on real life microprocessors. The Forth 79-Standard, on the other hand, had not attained the popularity of figForth in spite of the intensive lobbying efforts within and without the Forth Interest Group. The principal reason is that 79-Standard has to be supported by a system to be useful. Forth Interest Group expected that the support to Forth 79-Standard would come from vendors of commercial Forth systems, and it did not provide executable systems in supporting the standard.

Forth-83 Standard is a refinement on the 79-Standard. Many ambiguities in the 79-Standard were resolved and all required commands are defined in better precision. The DO-LOOP structure was overhauled. However, Forth Interest Group maintained the policy to let Forth vendors to provide the system support to the 83-Standard. Because of the reservation they had in the capability of vendor supports, Laxen and Perry built the F83 system as the bearer of the 83-Standard. They also realized that the Forth community has matured over the years and a minimal system like figForth will not satisfy the needs of Forth users in building applications and systems. To be the standard bearer, F83 had to go beyond the figForth model and provides you with a complete program developing environment.
Chapter 2. Browsing F83 System

I assume that you have either followed the instructions as described in the README.TXT file on the original disk you obtained from Henry Laxen or Mike Perry and expanded the compressed files to the full length files comprising the F83 system, or somebody did the expansion for you and you have a set of floppy disks ready to be used to explore this interesting and powerful Forth operating system and language. If you did not have an expanded system, please read the instructions in the README file and then run the executable file RUNME. You will be guided step by step to create a set of disks which will contain all the files to be used by the F83 system, as show in Figure 2.1.

Volume in drive A has no label.
Volume Serial Number is 0000-0000

Directory of A:\
02/09/1985 09:23 PM  12,288 CLOCK.BLK
02/09/1985 09:23 PM  53,248 CPU8086.BLK
02/09/1985 09:23 PM  30,720 EXTEND86.BLK
02/09/1985 09:23 PM  26,368 F83.COM
02/09/1985 09:23 PM  4,993 F83-FIXS.TXT
02/09/1985 09:23 PM  43,008 HUFFMAN.BLK
02/09/1985 09:23 PM  190,592 KERNEL86.BLK
02/09/1985 09:23 PM  49,280 META86.BLK
02/09/1985 09:23 PM  112,640 UTILITY.BLK
06/24/2013 05:55 PM  0 2-2.TXT
10 File(s)         523,137 bytes
0 Dir(s)          916,480 bytes free

Figure 2.1. IBM PC-DOS files in F83 system

In this chapter, I would like to show you what are contained in the F83 system and also the files on the disks and help you to get familiar with this system. All the commands and exercises mentioned in this chapter can be used freely to exercise the system so that you will gain certain degree of confidence to use them later when you will do programming. These commands will in no way disturb the information stored on the disks. The best way to learn them is to type them in on the keyboard and observe the results on the CRT terminal.

F83 system is very large comparing to earlier public domain Forth system like figForth. It has about 1000 commands or commands in its dictionary. However, most commands are defined to support other high level commands and are seldom used for ordinary programming purposes. Only a very limited number of commands are used often and these are commands that a user must learn and be fluent in them to use Forth productively. Included in this set of commands are the required command set defined in the Forth-83 Standards, which is a minimum set of commands allowing you to compose solutions to a wide range of programming problems, and the set of utility commands in this F83 system which allows you to use the specific resources provided by your computer. I further assume that you have already had some knowledge on Forth by reading some
textbook like Leo Brodie's 'Starting Forth', or its equivalent, and used a Forth system from some other source. Therefore, I will not try to explain in details the elementary functions and commands common to most Forth system and only discuss those commands unique in the F83 system. The purpose is to get you to know this system well enough so that you will be able to use it as a basis to build your application or your new Forth system.

In this chapter, all the words or commands discussed are non-destructive. They will allow you to browse through the entire system and explore its riches without writing anything to any of the files. You must try them all and get to know them well before we get to the next chapter where we will try to edit files and make permanent changes on disks. However, it is recommended that you make some backup copies of the disks with the expanded files and only use the copies for the exercises, just to be safe.

2.1. Listing the Command Names

Words or commands in Forth are very powerful constructs. They have the essence of subroutines in FORTRAN, procedures in PASCAL and PL/I, characters in APL, macro instructions in assembly, and command files in operating systems. Because they are resident in a dictionary in the RAM memory of a computer, they are available for immediate execution or for compilation into other high level commands. Commands in the dictionary are arranged in the form of a linked list so that the execution procedure associated with a particular command can be located quickly by the Forth operating system. A very useful utility command is defined to go through this linked list and print the names of all the commands in the dictionary. It is called WORDS in Forth-83 Standard, a remarkable improvement over the old computerese name VLIST in the figForth model. Typing:

```
WORDS
```
on your keyboard will generate a long list of command names on the terminal, as shown in the following figures. On the list of Forth commands, you will find all the regular Forth commands for arithmetic operations like +, -, *, /, and other division and ratio operators; the stack operators like DUP, DROP, SWAP, OVER and ROT; the memory operators like @, !, C@, and C!; etc. In fact, all the Forth-83 standard commands are included in this list some where.

WORDS have few equivalent in other language or operating system. The Forth computer can tell you all the commands it knows, which are available for your use, any time you care to browse. In other language or operating system, you have to go look them up in thick manuals and can never be sure that they are really in your system. WORDS reveals the current state of the dictionary. If you add more commands to the dictionary, they will appear at the top of the name list. It is very handy when you are extending the system by defining new commands and add them to the dictionary. In this case you will be interested in the commands on the top of the dictionary and
not the rest of the long listing. You can stop the name listing by pressing any key on the keyboard.

2.2. Vocabulary

Fig. 2.2 shows you how to list the commands in the Forth vocabulary. The dictionary in Forth is usually not a single linked list of commands or commands, but contains a number of logically independent linked lists of commands called vocabularies. The purpose of vocabulary is three-fold: to shorten the time needed to search through the dictionary, to group functional related commands together, and to allow different commands to share the same name. There are nine vocabularies defined in the F83 system. The names of these vocabularies can be displayed by typing the following command:

```
VOCS
```

and nine vocabulary names will be displayed on the terminal. The function and contents of these vocabularies are summarized in Table 2.1.

### Table 2.1. Vocabularies in F83

<table>
<thead>
<tr>
<th>NAME</th>
<th>FUNCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT</td>
<td>Words to assign vocabulary searching order. All vocabulary must be defined in this vocabulary.</td>
</tr>
<tr>
<td>FORTH</td>
<td>The main trunk vocabulary for all standard and system words.</td>
</tr>
<tr>
<td>EDITOR</td>
<td>All editing commands.</td>
</tr>
<tr>
<td>ASSEMBLER</td>
<td>All words needed to define low level machine code routines.</td>
</tr>
<tr>
<td>DOS</td>
<td>Words to use the underlying DOS utility.</td>
</tr>
<tr>
<td>USER</td>
<td>User variables.</td>
</tr>
<tr>
<td>SHADOW</td>
<td>Words to support shadow screens for comments and documentation.</td>
</tr>
<tr>
<td>BUS</td>
<td>Words to support F83 debugger.</td>
</tr>
<tr>
<td>HIDDEN</td>
<td>Miscellaneous supporting words not useful to the user.</td>
</tr>
</tbody>
</table>

Executing a vocabulary command makes the specified vocabulary the 'context' vocabulary. The system will search the context vocabulary first to locate a command entered by you. The command WORDS displays only the list of commands in the context vocabulary. Since normally the context vocabulary is the Forth vocabulary, executing WORDS usually displays the command names in the Forth vocabulary as shown in Fig. 2.2. Executing WORDS after a vocabulary command will list the command names in that vocabulary, as shown in the examples in Fig. 2.3-4.
**Figure 2.2** Forth commands

- `?DEFINE FILE` - Define a file
- `MAKE-FILE` - Create a new file
- `WRITE` - Write data to a file
- `READ` - Read data from a file
- `DELETE` - Delete a file

**Figure 2.3** Assembler and DOS commands

- `?DEFINE` - Define a command
- `FILE:` - Define a file
- `WRITE-FILE` - Write to a file
- `READ-FILE` - Read from a file
- `SET-IO` - Set input/output modes
- `REC-WRITE` - Record write
- `REC-READ` - Recomposed read
- `MAXREC#` - Maximum record size
- `LABEL` - Define a label
The vocabulary structure in F83 is significantly improved as compared with the vocabulary structure in the figForth Model. It is more flexible in that you can dynamically change the vocabulary searching sequence and specify up to eight different vocabularies in the searching sequence. The speed of dictionary searching is also much faster than that in the figForth Model, because all vocabularies in the dictionary are hashed into four threads. In order to locate a command, only a quarter of a vocabulary needs to be scanned. This hashed searching greatly improves the speed of text interpretation and program compilation.

Two commands are used to manage the vocabulary searching sequence: ONLY and ALSO.

ONLY initializes the searching sequence and makes ROOT as the only vocabulary available for
searching. In the ROOT vocabulary, all the other vocabulary names must be defined so that they are accessible. After ONLY is executed, executing any other vocabulary command will make that vocabulary the context vocabulary which becomes the first vocabulary to be searched during text interpretation. Executing ALSO pushes the context vocabulary on the top of a vocabulary stack and makes it the first resident vocabulary. Other resident vocabularies already in the vocabulary stack are pushed down so that they will be searched in order after searches in the context and the first resident vocabularies failed to locate a command.

The context vocabulary, for all practical purposes, is the equivalent to the context vocabulary in figForth. The resident vocabularies are extensions of the context vocabulary to allow you to specify the number and the order of vocabularies to be searched in runtime.

To arrange the searching order as DOS-EDITOR-ASSEMBLER-Forth, one has to execute the following command sequence:

```
ONLY   FORTH ALSO   ASSEMBLER ALSO   EDITOR ALSO   DOS
```

Here DOS becomes the context vocabulary and EDITOR is the first resident vocabulary to be searched. If a command cannot be located in either DOS or EDITOR, the ASSEMBLER and the FORTH vocabularies will be searched in turn.

Another command ORDER will list the context and the resident vocabularies on the terminal. It is a useful command to assure yourself the context environment you are in at any time. If you executed the above string of vocabulary commands, typing

```
ORDER
```

results in the following display on the terminal:

```
Context: DOS EDITOR ASSEMBLER FORTH ROOT
Current: FORTH ok
```

indicating the desired vocabulary search order. The current vocabulary, in this case FORTH, is the vocabulary to which new commands are added. It will be discussed later.

The command WORDS behaves similarly to VLIST in figForth. However, WORDS only lists the names of commands in the context vocabulary; therefore, WORDS must be preceded by the name of the vocabulary you wish to examine, like FORTH WORDS, DOS WORDS, etc. Since the list of command names always starts with the command defined last, it is often used to see which was compiled last. If there were any error during disk file loading, you can find quickly where compilation stopped.

### 2.3. Viewing Source Code of Command Definitions

Since there are so many commands in the F83 system, it is impossible for anybody to remember the
meaning and the function of all these commands. Although the compiled object code of a
command in the dictionary contains all the information about this command, it is not readily usable
to casual users. F83 system provides a very interesting and powerful tool set which permits you to
see the source code of any command in the system. This magic command is named 'VIEW'. If
you wanted to see how the command LIST was defined, you should type:

```
VIEW LIST
```

and the F83 system will open the file in which LIST was defined and display the screen containing
the definition of LIST. On the top of the displayed screen, you will also find the named of the file.
This is shown in Fig. 2.4.

To use the viewing facility, you must have all the source files on disks and have them properly
inserted into appropriate disk drives. For some computers, the files fit on a single floppy disk.
This is the ideal case because you don't have to worry about where a particular file is. For
computers with smaller disk drives, the files must be spread over two or more drives. If the
required file is not on the disk of your current disk drive, you have to log on to the drive where the
file is located and repeat the viewing command, or insert the proper disk in the the log-on drive and
repeat the viewing command.

Commands are grouped by their functions and by the order of compilation into six major files in
the F83 system:

- **METAnn.BLK**: The meta-compiler
- **KERNELnn.BLK**: The trunk FORTH system. Nucleus, interpreter and compiler.
- **EXTENDnn.BLK**: Vocabulary and file words.
- **CPUnnn.BLK**: Assembler and CPU dependent words.
- **UTILITY.nn.BLK**: Editor, debugger, decompiler, printing and other utility.
- **HUFFMAN.BLK**: Huffman compression.

where nn or nnnn identifies the CPU for which the F83 system is hosted. 80 for 8080 and Z80, 86
for 8086 and 8088, and 68 for 68000.

If you have to choose which files to put on a disk for viewing, I suggest that you put
UTILITY.BLK, KERNELnn.BLK, and EXTENDnn.BLK on one disk and use it for viewing,
because they comprise the majority of useful commands that you might be interested in browsing.

F83 also comes with a built-in decompiler which can regenerated the source code from the object
code in the dictionary. The decompiler command is 'SEE', followed by the name of the command
you want to decompile. For example:

```
SEE LIST
```

will display the sequence of commands which define the function of LIST on the terminal. The
displayed sequence of commands does not match exactly the sequence in the original source code,
because the control structures are not decompiled but simply represented by the corresponding runtime routines. Nevertheless, the decompiled sequence does reveal the composition of the source code faithfully. The advantage of the decompiler over the viewing facility is that the decompiler is always available for you to browse commands, even without the disk files.

The result of SEE LIST is also shown in Fig. 2.5.

```
ok
ok
VIEW LIST is in A:UTILITY.BLK screen 4
Scr # 4         A:UTILITY.BLK
0 \ Managing Source Screens 22Mar84map
1 : .SCR (S -- ) ." Scr # " SCR ? 8 SPACES FILE? ;
2 : LIST (S n -- )
3 1 ?ENOUGH CR DUP SCR ! .SCR L/SCR 0
4 DO CR I 3 .R SPACE
5 DUP BLOCK I C/L * + C/L -TRAILING >TYPE KEY? ?LEAVE
6 LOOP DROP CR ;
7 : TRIAD (S n -- )
8 12 EMIT (form feed) 3 / 3 * 3 BOUNDS DO I LIST LOOP ;
9 : .LINE0 (S n -- )
10 DUP 3 MOD 0= IF CR THEN CR DUP 3 .R SPACE
11 BLOCK C/L -TRAILING >TYPE ;
12 : INDEX (S n1 n2 -- )
13 2 ?ENOUGH 1+ SWAP DO I .LINE0 LOOP CR ;
14 : IND (S n -- )
15 BEGIN DUP .LINE0 1+ KEY? UNTIL DROP ;
ok
ok
ok
ok
SEE LIST
: LIST 1 ?ENOUGH CR DUP SCR ! .SCR L/SCR 0 (DO) 38 CR I 3 .R SPACE DUP
BLOCK I C/L * + C/L -TRAILING >TYPE KEY? (?LEAVE) (LOOP) -34 DROP CR ; ok
ok
ok
```

Figure 2.5   VIEW and SEE

2.4.   Shadow Screen Documentation

Since most people think that a Forth screen of 1024 bytes is too small to put inline documentation with the code in the same screen, the shadow screen technique was developed to give you an extra screen to write comments and documentation for each source screen. This documentation screen is the shadow of the source screen.

F83 divides a screen file into two equal parts: the first half will be used for source code and the second half for documentation. one can toggle between a source screen and its shadow screen with the commands A and L. After viewing the source code in a source screen, you can type A L and switch to the shadow screen to see the comments and documentation. Documentation thus provided in the F83 system is quite extensive, and you are encouraged to examine the shadow screens with their respective source screens. The shadow screens generally bring out the purpose and over-all function of commands which are not obvious in the source definition.
2.5. Files in F83

F83 uses MS-DOS or CP/M operating system to access the terminal and the disk files. Using a readily available operating system to host the F83 system has the advantage that it can be transported to a large number of computers with that operating system. It also allows the partitioning of the F83 system into several named files which are easier to handle than simply blocked disk. Within a file, however, F83 system still deals with program or data in the 1024 byte block format as required by the Forth-83 standard. Most of the elementary file functions are defined as Forth commands. However, you only need a few high level commands to use files to store and to retrieve programs and data.

Three simple Forth commands have functions similar to their DOS or CP/M counterparts: DIR lists on the terminal all the files on the current disk drive, A: makes drive A the current drive, and B: makes drive B the current drive. All file activities are processed for files on the current drive.

All the Forth commands using the disk mass storage, such as BLOCK, BUFFER, FLUSH, etc., access the current file on the current disk. A file becomes the current file when it is opened by the command OPEN <filename>, and subsequent disk commands are directed to this file. In our previous example of the command VIEW, which displays the screen containing the source code in a file, the command VIEW actually opens the file containing the command and displays the requested block on the terminal. If you want to examine or to modify data or source in a specific file, you have to open it explicitly. Once a file is opened, you can display any block within that file.

The size of a file is usually specified when the file was created. The size in number of 1024 byte blocks can be recalled by the command CAPACITY. Execute CAPACITY and the number of blocks in the current file is returned on the stack. Source code files in the F83 system with the extension BLK are arranged to have the source code in the first half of the file and the shadow documentation in the second half.

To examine the contents of a BLK file, you can use the command INDEX to display the first lines in a range of screens. For example, to display the first lines of all the source code screens, you can type:

```
0 CAPACITY 2/ INDEX
```

and the first lines of those screens will be displayed on the terminal. By convention, the first line in a screen should always be a comment to the contents of this screen. Thus INDEX gives us the
After you identify any screen of your interest, you can examine the detailed contents of this screen by the command LIST, preceded by the screen number:

```
1 LIST
```

It will display the first text screen in a file. In all the F83 source code files, screen one is the load screen of the file, i.e., it contains commands that will load or compile the source screens in the rest of the file. There are also some comments in screen one indicating the packaging of the screens in the file.

To display the shadow documentation of any source screen, you should type:

```
A L
```

The command A uses CAPACITY to calculate the screen number of the associated shadow screen and makes it the current screen. The command L displays the current screen. Executing A and L again will display the source screen again.

### 2.6. Printing Utility

To make hard copy of the source screens and shadow screens, a simple method is to let the printer follow the terminal display. In the CP/M systems you can type the control P code on the keyboard to turn on the printer. Any character hereafter displayed on the terminal will also be printed. Now you can use any of the listing commands discussed in the last section to print index of a file or individual screens. However, you do not have control on the printing format. F83 provides some utility commands to print source code and shadow screens. If you have an EPSON printer capable of printing in condensed format, you can print the source screens side by side with their shadows on single 8.5" by 11" paper, which is very convenient when studying the source code.

The print utility allows you to print a range of screens on a printer. It must be properly initialized for your printer. If you do have an EPSON printer you have to initialize it by the following commands:

```
* EPSON IS INIT-PR
```

which initialize the vectored command, INIT-PR. The print format is 6 screens to an page with two 3 screen columns. The printer must be able to print 132 characters per line to fit two screens side by side.
The command to print a range of screens is SHOW:

```
1 30 SHOW
```

will print screens 1 to 30.

There are two versions of SHOW in F83. The version in Forth prints 6 screens per page and the version in the SHADOW vocabulary prints 3 screens of source with their corresponding shadow screens:
prints source screens 1 to 30, 3 screens to a page with 3 shadow screens.

If your printer cannot handle 132 columns per line, you will have to use the command TRIAD to print three screens on a page.

To obtain the complete listing of a file in the source-shadow format, there is a simple command LISTING. LISTING was used to generate all the source listings as distributed with the F83 systems, with file name, page number, and footing.

2.7. Debugger

The debugger is designed to let you single stepping through the execution sequence of a high level command. To invoke the debugger to trace a command, issue the following command:

```
DEBUG <name>
```

where `<name>` is the command to be debugged. Nothing happens at this point. DEBUG sets things up so that when the command is executed you will get a single step trace showing the command within `<name>` that is about to be executed and the contents of the parameter stack.

While single stepping through a command, the name of the next command to be executed and the contents of the parameter stack are displayed on the CRT terminal. The debugger then waits for a key stroke on the terminal keyboard. Any key will cause the next command to be executed and the debugging information displayed. Three special keys, C, F, and Q, have the following functions:

- Q: Quit the debugging process and restore the debugged word to its original state for normal execution.
- C: Turn off the single stepping mechanism and let execution run to completion.
- F: Temporarily return to Forth system so that you can execute other Forth commands, for example, to change the data stack items. You must type RESUME to come back and continue the debugging process.

An example to single step through the execution of 1 LIST is shown in Fig. 2.7. Typing Q at the bottom of the list terminates the execution.
Figure 2.7   Debugging LIST

DUP    -->    1    1
SCR    -->    1    1    2998
!      -->    1
.SCR   -->    Scr #    1    A:UTILITY.BLK    1
L/SCR  -->    1    16
0      -->    1    16    0
(DO)   -->    1
CR     -->
       1
I      -->    1    0
3      -->    1    0    3
.R     -->    0    1
SPACE  -->    1
DUP    -->    1    1
BLOCK  -->    1    64000
I      -->    1    64000    0
C/L    -->    1    64000    0    64
*      -->    1    64000    0
+      -->    1    64000
C/L    -->    1    64000    64
-TRAILING -->    1    64000    64
>TYPE   --> \\ Load Screen to Bring up Standard System    07Apr84map    1
KEY?    -->    1    0
(?LEAVE) -->    1
(LOOP)  -->    1
CR     -->
       1
I      -->    1    1
3      -->    1    1    3
.R     -->    1    1
SPACE  -->    1
DUP    -->    Unbug
ok
ok
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 commands 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 a screen by the command \texttt{n LIST} or \texttt{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 chose 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 OPEN'ed, 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 space 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 most often used editing commands are summarized in Table 3.1.

Table 3.1. Editor Commands

<table>
<thead>
<tr>
<th>Block Editing Commands</th>
<th>Line Editing Commands</th>
<th>String Editing Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>n LIST</td>
<td>n T</td>
<td>F xxxx</td>
</tr>
<tr>
<td>n L</td>
<td>P xxxx</td>
<td>D xxxx</td>
</tr>
<tr>
<td>A L</td>
<td>U xxxx</td>
<td>I xxxx</td>
</tr>
<tr>
<td>B L</td>
<td>X</td>
<td>TILL xxxx</td>
</tr>
<tr>
<td>UPDATE</td>
<td>n NEW</td>
<td>J xxxx</td>
</tr>
<tr>
<td>SAVE-BUFFERS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **n LIST**: Display screen n and make it the current screen.
- **n L**: Display the current screen.
- **n N L**: Display the next screen.
- **n B L**: Display the previous screen.
- **A L**: 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.
- **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.
- **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 errors or bugs in the program, causing the interpreter to abort during the loading process or giving wrong results when you execute commands 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 file which is also a good example on the command sequence in using the line editor. Terminal characteristics are specified in four commands:

- **AT** Move the display cursor to a specified screen coordinate.
- **DARK** Clear the screen and home the cursor.
- **-LINE** Delete current line and roll up the rest of the screen.
- **BLOT** Erase till the end of line.

These four commands have to be vectored to the commands 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 command of xxxx and display the screen of this command. It also invokes the editor to let you edit the command.

WHERE is also a very useful command during program development. 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 command 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
```

It 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: commands to be interpreted or executed, new commands 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 3.2. here.
Table 3.2.  Loading Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>Interpret source text in screen n in the current file.</td>
</tr>
<tr>
<td>n m THRU</td>
<td>Interpret sequentially the source text in screens n to m.</td>
</tr>
<tr>
<td>n +LOAD</td>
<td>Load the screen n blocks from the current screen.</td>
</tr>
<tr>
<td>n m +THRU</td>
<td>Load a range of screens n blocks offset from the current screen.</td>
</tr>
<tr>
<td>--&gt;</td>
<td>Exit the current screen and load the next screen immediately.</td>
</tr>
</tbody>
</table>

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 maintain 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 commands 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.
Memory dump

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0001</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0002</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0003</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0004</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0005</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0006</td>
<td>0x00000006</td>
</tr>
<tr>
<td>0007</td>
<td>0x00000007</td>
</tr>
<tr>
<td>0008</td>
<td>0x00000008</td>
</tr>
<tr>
<td>0009</td>
<td>0x00000009</td>
</tr>
<tr>
<td>000A</td>
<td>0x0000000A</td>
</tr>
<tr>
<td>000B</td>
<td>0x0000000B</td>
</tr>
<tr>
<td>000C</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>000D</td>
<td>0x0000000D</td>
</tr>
<tr>
<td>000E</td>
<td>0x0000000E</td>
</tr>
<tr>
<td>000F</td>
<td>0x0000000F</td>
</tr>
<tr>
<td>0010</td>
<td>0x00000010</td>
</tr>
<tr>
<td>0011</td>
<td>0x00000011</td>
</tr>
<tr>
<td>0012</td>
<td>0x00000012</td>
</tr>
<tr>
<td>0013</td>
<td>0x00000013</td>
</tr>
<tr>
<td>0014</td>
<td>0x00000014</td>
</tr>
<tr>
<td>0015</td>
<td>0x00000015</td>
</tr>
<tr>
<td>0016</td>
<td>0x00000016</td>
</tr>
<tr>
<td>0017</td>
<td>0x00000017</td>
</tr>
<tr>
<td>0018</td>
<td>0x00000018</td>
</tr>
<tr>
<td>0019</td>
<td>0x00000019</td>
</tr>
<tr>
<td>001A</td>
<td>0x0000001A</td>
</tr>
<tr>
<td>001B</td>
<td>0x0000001B</td>
</tr>
<tr>
<td>001C</td>
<td>0x0000001C</td>
</tr>
<tr>
<td>001D</td>
<td>0x0000001D</td>
</tr>
<tr>
<td>001E</td>
<td>0x0000001E</td>
</tr>
<tr>
<td>001F</td>
<td>0x0000001F</td>
</tr>
</tbody>
</table>

Figure 3.1 Memory dump
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 commands 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 commands in the screen fully, the problem can be solved very efficiently. Of course, you should make it easier for yourself by writing short commands which are easy to test, and if there is a problem, easy to spot the problem and fix it. Choosing good names for your commands 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 command and shows 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 command you want to trace use the DEBUG command. Then, you have to execute the command in the normal way it is used, with necessary data stack parameters. The command is then executed in steps. The computer displays the name of the command in the command 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 command. For example, we would like to debug the command LIST:

```
DEBUG LIST
```

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

```
1 LIST
```

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 commands short 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 command 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 commands 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 referenced in the assembler. However, the F83 system uses four of the registers to implement the Virtual Forth Computer. These registers used by Forth have special names to indicate their special functions in Forth, and they should be preserved inside the code command. Table 3.3. shows the 8086 registers and the mapping with 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 commands. AX, CX, DX, and DI can be used freely. W points to the code field of the command 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**

<table>
<thead>
<tr>
<th>8086</th>
<th>Forth</th>
<th>Function of register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Accumulator</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td>Counter</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>I/O register</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>W</td>
<td>Current word pointer</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>Data stack pointer</td>
</tr>
<tr>
<td>BP</td>
<td>RP</td>
<td>Return stack pointer</td>
</tr>
<tr>
<td>SI</td>
<td>IP</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>DI</td>
<td>Scratch</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td>Extra segment pointer</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4.  **Register Addressing Modes and Mnemonics**

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bit register mode</td>
<td>AL CL DL BL AH CH DH BH</td>
</tr>
<tr>
<td>16 bit register mode</td>
<td>AX CX DX BX BP SP SI DI ES CS SS DS W RP IP</td>
</tr>
<tr>
<td>Indirect register mode</td>
<td>[SI] [DI] [BP] [BX] [RP] [IP] [W]</td>
</tr>
<tr>
<td>Indirect with index</td>
<td>[BX+SI] [SI+BX] [BX+DI] [DI+BX] [BP+SI] [SI+BP] [BP+DI] [DI+BP]</td>
</tr>
<tr>
<td>Immediate #</td>
<td>#</td>
</tr>
<tr>
<td>Immediate address $#</td>
<td>$#</td>
</tr>
</tbody>
</table>

All 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 the chapter 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 commands in the Forth kernel, where we have hundreds of fine examples of code commands.  The best way to write a code command 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 command must start 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 commands are executed to assemble machine instructions into the body of the code command.  At the end of the code command, there must be a command to return control to the Forth interpreter and a command to terminate the code command:

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

where <name> is the name of the new code command.  CODE only creates the header.  The Forth text interpreter is still in control after CODE, and the sequence of assembly commands are 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 command
available for searching and compiling.

An assorted collection of assembly commands are 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.**

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

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

**Table 3.6. Return Commands**

<table>
<thead>
<tr>
<th>NEXT</th>
<th>Assemble a jump to &gt;NEXT routine so that the next word pointed to by the IP will be executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1PUSH</td>
<td>Assemble a jump to APUSH routine which pushes AX on data stack and then falls into &gt;NEXT.</td>
</tr>
<tr>
<td>2PUSH</td>
<td>Assemble a jump to DPUSH routine which pushes DX on data stack and then falls into 1PUSH.</td>
</tr>
</tbody>
</table>

F83 uses a centralized return mechanism by which all code commands 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 commands. 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 commands, it is necessary to build subroutines. The command to define 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.
commands 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 command.

Within a code command, the execution path can be altered or repeated using structure commands IF-ELSE-THEN, BEGIN-UNTIL, and BEGIN-WHILE-REPEAT, similar to those used in colon commands. 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

<table>
<thead>
<tr>
<th>Forth Conditional</th>
<th>Assembled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=</td>
<td>JNE/JNZ</td>
</tr>
<tr>
<td>0&lt;&gt;</td>
<td>JE/JZ</td>
</tr>
<tr>
<td>0&lt;</td>
<td>JNS</td>
</tr>
<tr>
<td>0&gt;=</td>
<td>JS</td>
</tr>
<tr>
<td>&lt;</td>
<td>JNL/JGE</td>
</tr>
<tr>
<td>&gt;=</td>
<td>JL/JNGE</td>
</tr>
<tr>
<td>&lt;=</td>
<td>JNLE/JG</td>
</tr>
<tr>
<td>&gt;</td>
<td>JLE/JNG</td>
</tr>
<tr>
<td>&lt;</td>
<td>JNB/JAE</td>
</tr>
<tr>
<td>U&gt;=</td>
<td>JB/JRAE</td>
</tr>
<tr>
<td>U&lt;</td>
<td>JNBE/JA</td>
</tr>
<tr>
<td>U&gt;</td>
<td>JS</td>
</tr>
<tr>
<td>OV</td>
<td>JNO</td>
</tr>
</tbody>
</table>

The Forth conditionals are reverse of the assembler code because the IF, UNTIL, and WHILE are skip-on-condition, 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 commands. The assembler structure commands are in the ASSEMBLER vocabulary and those in colon commands are in the FORTH vocabulary. Their behaviors are quite different.

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

```
CODE UPPER ( addr length -- )
  CX POP BX POP Get parameters into registers.
  BEGIN Setup the loop.
  CX CK 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.
```

32
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 commands 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 commands which are executed very often. You should try to identify these commands and convert only these commands to code commands.

### 3.7. Multitasker

The early Forth systems implemented by Charles Moore were multi-tasking systems and could support many users or terminals to operate simultaneously. The implementers of figForth neglected this feature in the figForth Model. Because of the popularity of figForth, 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 area 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 commands 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 up by executing PAUSE or STOP and releases the CPU to the next task in the round robin chain. All system I/O commands have PAUSE's embedded to switch tasks automatically. In the command sequence give to a task to execute, the command PAUSE or STOP must be place properly so that the task will not hold on to the CPU indefinitely.

Background tasks are defined by the command 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 your self 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 command, 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 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

It 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 command in your application and run it.
3.9. The Metacompiler

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 command 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 metacompilation. Metacompilation 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 metacompilation process, and all the source code needed for its self-generation are 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 metacompiling. Insert this disk into drive A.
2. Prepare a disk with METAnn.BLK and KERNELnn.BLK files on it. METAnn.BLK contains the metacompiler and KERNELnn.BLK contains source code for the bare-bone 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-bone 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 metacompilation,
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 metacompiler fully before attempting your
modifications.
Part II. The Forth Kernel

Chapter 4. Interface to the Host Computer

Source code discussed in this chapter is in the file KERNEL86.BLK, Screens 3 to 15.

4.1. Virtual Forth Computer

The Virtual Forth Computer is a program loaded into the memory of a real computer. It partitions the computer memory into areas of specific functionality and enables the real computer to process Forth command streams. Fig. 4.1 is a schematic representation of the functional parts in a Virtual Forth Computer. It consists of a dictionary, two stacks, a terminal input buffer, and a number of disk buffers. These are the essential parts in a Virtual Forth Computer.

The Virtual Forth Computer uses a set of registers to keep the most vital information and to control the execution sequences. They are:

- `SP` Data Stack Pointer
- `RP` Return Stack Pointer
- `IP` Interpretive Pointer
- `W` Current Word Pointer
- `PC` Program Counter

The program counter PC and the return stack pointer RP are usually registers in the host CPU. The data stack pointer SP, the interpretive pointer IP, and the current command pointer W can reside in CPU as registers or implemented in memory if the host CPU does not have enough registers.

The dictionary is a linked list of commands. Each command consists of five fields: The name field and the link field allow commands to be linked into a linear list which can be searched by the text interpreter for a command by its name. The code field contains the address of the inner interpreter for this command and the parameter field contains necessary information specific for the task defined for this command. The view field contains information on where the source code of the command is located on disk to help user in locating the source code and detailed documentation on the command.

Two stacks are needed by the Virtual Forth Computer. The return stack contains a list of addresses of commands which are waiting to be executed, or addresses to be returned to after procedure calls. It is similar to the return stacks used in most modern computers. The other
stack is called data stack, holding a list of numeric and logic parameters to be passed between commands. Separating numeric parameters and return addresses into two stacks allows procedure calls without passing parameters through explicitly parameter lists. It greatly simplifies the syntax of Forth and cuts down the overhead for procedure calls.

Buffers are used to reduce time and efforts to process data transferred between the Forth computer and the I/O devices. Since the two major I/O devices in a typical computer system is the disk for mass storage and the terminal for operator control, two buffer areas, a disk buffer area and a terminal input buffer, are allocated to handle the I/O data.

The kernel of the F83 system is the part of the dictionary which contains commands defined in the machine code of the host computer, and transforms the host computer into a Virtual Forth Computer so that the computer can accept and act upon Forth commands give to it either through a keyboard or through text loaded from a disk. It is the elementary Forth operating system, which can be expanded by loading utility programs and application programs, and then executing those programs.

4.2. Forth Computer Hosted on 8086

The Virtual Forth Computer is a hypothetical computer or an ideal structure of a computer which can process (interpret or compile) Forth command set. So far, we have yet to see a computer built based upon the Forth architecture. However, this architecture can be implemented on any CPU worthy of the name. As a matter of fact, most of the commercial CPU's have at least one version of Forth on them, including all popular microprocessors, minicomputers, and many mainframes. Because Forth is simple and relatively small, it can be implemented on a computer with about one man-month's effort. This is very short comparing to other operating systems or high level languages.

F83 has three versions: one for 8080, one for 8086/8088, and one for 68000. Because the very small number of registers are available in the 8080 CPU, many Forth registers will have to be simulated in memory. Most numbers processed in Forth are 16 or 32 bits in width. They have to be manipulated in 8 bit chunks in 8080. Forth code commands in 8080 machine instructions are thus very messy and very difficult to explain. Between 8086 and 68000, my personal preference is 68000 which has a much cleaner architecture and more orthogonal instruction set. Nevertheless, I feel very feeble in raising a voice against the infinite wisdom of IBM, who picked 8088 for PC. Since there are apparently more people using 8088 than 68000, it is better to write this manual in terms of the 8086/8088 F83 model if I want to sell more copies of this manual.
First of all, let see how the Forth registers are assigned in the 8086 CPU:

### Table 4.1. 8086 Register Assignments for Forth

<table>
<thead>
<tr>
<th>8086 Reg.</th>
<th>Forth Reg.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>A</td>
<td>Accumulator</td>
</tr>
<tr>
<td>CX</td>
<td>Scratch</td>
<td>Counter</td>
</tr>
<tr>
<td>DX</td>
<td>Scratch</td>
<td>I/O control</td>
</tr>
<tr>
<td>BX</td>
<td>W</td>
<td>Current word pointer</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>Data stack pointer</td>
</tr>
<tr>
<td>BP</td>
<td>RP</td>
<td>Return stack pointer</td>
</tr>
<tr>
<td>SI</td>
<td>IP</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>DI</td>
<td>DI</td>
<td>Scratch</td>
</tr>
<tr>
<td>ES</td>
<td>ES</td>
<td>Extra segment</td>
</tr>
<tr>
<td>CS</td>
<td>CS</td>
<td>Code segment</td>
</tr>
<tr>
<td>SS</td>
<td>SS</td>
<td>Stack segment</td>
</tr>
<tr>
<td>DS</td>
<td>DS</td>
<td>Data segment</td>
</tr>
</tbody>
</table>

8086 has only one stack, with SP as stack pointer, which allows pushes and pops. Other registers do not have automatic incrementing or decrementing facilities, and increment/decrementing must
be done explicitly. An interesting exception is the SI index register. When SI is used in the supposed string instruction LODS and STOS, it is incremented by 1 or 2 bytes to point to the next string element. It is ideal for the interpretive pointer. The stack pointer SP is used to implement the Forth data stack, while the Forth return stack is simulated using the BP register. Popping and pushing on the return stack have to be done explicitly by incrementing and decrementing the BP register. The command pointer W is simulated by the BX register. Lacking automatic incrementing and decrementing facility, the W register has to be left pointing to the code field after NEXT. It has to be incremented in code so that it will point to the parameter field. We will have to use indirect JMP instructions through the code field to control the program flow.

Other 8086 registers, like AX, CX, DX, and DI, can be used freely in code routines. However, parameters and other information cannot be passed from one command to another through these registers. They have to be initialized appropriately before use, but they do not have to be restored before the end of a code command. The W register or the BX register contains the code field address of the command under execution. If this address is not needed in the code command, this register can be also used without restoring. The SP, RP, and IP registers, however, have to be restored to the original values if they have to be used in a code command.

All information in the F83 system is contained within a single 64K byte memory segment, and the four segment registers ES, CS, SS, and DS are initialized to the same segment. They can be changes to address other segments of memory, but must be restored before the end of a code command. F83 system does not use them.

**Memory Map**

RAM memory in the host computer, as used by the F83 system, is a contiguous 64 Kbytes of memory. Forth separates this memory space into a few regions, each dedicated to specific function. The lowest memory is used to hold the interrupt vectors which is used by the hardware to service external interrupts or software interrupts. Immediately above the vector region is the dictionary, holding all the executable codes of the commands. Above the dictionary is a free space for you to define new commands. On the top of the memory map is the region for disk buffers. F83 allocate 4 Kbytes for the buffers, enough to hold 4 blocks of data from/to the disk. Under the disk buffers is an area storing user variables which are essential parameters for the Forth system to work. Below the user area is the return stack, sharing its space with the terminal input buffer, which is used to store characters received from the terminal keyboard before processed by the text interpreter.

Below the terminal input buffer is the data stack, growing downward into the free memory space
above the dictionary. The space just above the dictionary are used to store temporary text data. We can identify a word buffer, a text buffer called PAD, an insert buffer and a delete buffer used by the text editor. A video buffer of 1 Kbytes is also assigned if the screen editor is invoked. These buffers float on the top of the dictionary, moving to higher memory as new commands are added to the dictionary. Data stored in them have to be used before new commands are defined.

Fig. 4.2. schematically shows the arrangement of various regions in a typical F83 Forth system. Most Forth system are arranged similarly.

4.3. Inner Interpreters

Inner interpreters in Forth are a set of execution procedures, usually in the machine code of the host computer, which execute various Forth commands by processing the information stored in their parameter fields. The address of such a procedure is stored in the code field of a command. Forth commands of the same class have the same address in their code fields. Two major inner interpreters are used to process code commands, defined by machine instructions, and colon commands, defined in terms of other existing Forth commands. Many other minor inner interpreters are used in F83 system to process constants, variables, user variables, and other types of data and structures.

Code Interpreters

Forth commands defined by host machine instructions are executed by one of two routines, EXECUTE or NEXT. EXECUTE can be used to start executing any Forth command, given that the code field address of the Forth command is placed on the data stack before calling EXECUTE. EXECUTE is a regular Forth command which can be executed interactively or called from the text interpreter. At the end of all code commands, there must be a jump to the machine routine labeled NEXT, which transfer control to execute the next command in the execution sequence. In F83, NEXT is centralized so that every command must return to it through a JMP instruction. NEXT assumes that the code field address of the next command to be executed is stored in the IP register.
Figure 4.2   Memory map of F83 system
CODE EXECUTE ( cfa -- ) Execute the word whose execution address is on the data stack.

W POP Pop execution address into W.
0 [W] JMP Make an indirect jump through W. W is left pointing to the END-CODE code field. It must be incremented if parameter field must be addressed.

LABEL DPUSH A label in the target system.

LABEL APUSH Another label.

LABEL >NEXT The principal point of return for all code definitions. The execution address of the next word to be executed in IP register.

AX LODS Load the next execution address from IP into AX and increment IP.

AX W MOV Copy execution address into W.
0 [W] JMP Indirect jump through W.

The codes of EXECUTE appears in Screen 36 of META86.BLK, and the codes of NEXT is in Screen 25 in the F83 source. The fact that these codes are scattered in many separated blocks makes it difficult for the reader to put a whole picture together. It is the purpose of this manual to present you a well organized system description, and to help you understand the F83 model fully.

: NEXT  A macro definition to assemble a jump to >NEXT instruction at the end of a code definition.
>NEXT #) JMP Assembler the jump instruction.
;
: 2PUSH Another macro definition.
DPUSH #) JMP Jump to DPUSH, push DX on stack.
;
: 1PUSH Macro definition pushing AX on the data stack before NEXT.
APUSH #) JMP
;

These commands are defined in the metacompiler as shown in the source code. Let us not worry about them here.

Address Interpreter

The address interpreter is used to execute a high level Forth command whose parameter field contains a list of execution address. It processes this list by executing commands at these addresses sequentially.

The address interpreter is not an executable Forth command. It is a machine code routine labeled NEST:

LABEL NEST IP has the address to return and W has the code field address of the colon definition to be called.

W INC W INC Increment W to point to the parameter field of the callee.
RP DEC RP DEC Decrement return stack pointer and prepare for a push.
IP 0 [IP] MOV Push contents of IP, the return address on the return stack.
W IP MOV Copy the first execution address into IP, to start the called colon definition.
NEXT Assembler >NEXT #) JMP here.

In 8086, W and RP have to be incremented twice because these registers are byte pointers. All
code routine ends up with the code >NEXT #) JMP. This is very convenient in debugging the system or changing the behavior of the code interpreter to include new features in the Forth system.

CODE EXIT
Terminate a colon definition and return to the caller routine whose address is on top of the return stack.
0 [RP] IP MOV
Pop the return address back to the IP register.
RP INC RP INC
NEXT
Return.

CODE UNNEST
' EXIT ' EXECUTE !

The standard command EXIT is vectored to UNNEST which is the reverse of NEST. NEST is equivalent to the high level subroutine call in FORTRAN, and UNNEST is the equivalent of RETURN.

Variable Interpreter

Variable interpreter uses the W register to point to the parameter field of the variable command and returns the parameter field address on the data stack for the subsequent commands to access the parameter field. This inner interpreter can be shared by other types of commands which uses the parameter fields to store various types of data, like strings, double integers, floating point numbers, or even large arrays. When a new command is created in the dictionary, the compiler assumes that the command is of this type, unless a new inner interpreter is defined. Instead of the old name DOVAR in figForth, F83 uses the generic name DOCREATE:

LABEL DOCREATE
W INC W INC
Increment W to point to the parameter field.
W PUSH
Push pfa on the data stack.
NEXT
Return.

W register as returned by NEXT contains the code field address of the variable command. To get the parameter field address, W has to be incremented here. This is called the post-incrementing NEXT. In many other Forth systems, the W register is incremented inside NEXT so that it points to the parameter field at the end of NEXT. The pre-incrementing NEXT is more desirable than the post-incrementing NEXT, because the post-incrementing NEXT requires that the code field is two bytes ahead of the parameter field. Because F83 uses the W register to make the indirect jump to the inner interpreter, the W register cannot be incremented before the jump.

Constant Interpreter

Constant interpreter is very similar to variable interpreter. The only difference is that constant interpreter returns the contents of the parameter field while variable interpreter returns the address of the parameter field.
The constant interpreter first copies the contents of the parameter field into the AX register, and then jumps to the APUSH routine which pushes AX on the data stack before falling into NEXT.

**User Variable Interpreter**

User variables are defined for a multitasking or multiuser Forth system. These variables are not addressed by their parameter field addresses, but by an offset into a memory area unique to the current user, a user variable area whose starting address is stored in a register or a variable UP. The user variables define the operating environment for a user at any point of its operation. Since each user has its own user variables preserved in a unique memory area, users or tasks can be switched very conveniently with minimal house keeping.

The user variable interpreter in F83 is defined as:

```
VARIABLE UP       The user area pointer is defined as a variable.
LABEL DOUSER-VARIABLE
  W INC  W INC     Point W to the parameter field.
  0 [W] AX MOV      Fetch contents in the parameter field to AX register.
  1PUSH JMP         Push AX on the data stack and then return.
```

The parameter field of a user variable stores the offset value of the user variable in the user area. This offset value is added to the starting address of the user area as stored in the variable UP. The address returned on the data stack is the address of the user variable of the current user who is controlling the Forth system at this moment.

With all the viable system parameters saved in the user variable areas, the task switching in a Forth multitasking system is very easy and very efficient. The multitasker only has to save and restore the IP, RP, and SP in between two tasks. We will get into this in detail later. F83 has a very interesting multitasker which is a good demonstration of the power and the versatility of Forth as a system and as a language.

**High Level Inner Interpreter**

Inner interpreters are preferably coded in the host machine instructions, because they are the actual routines executed by the host computer. However, F83 does provide the CREATE ... DOES>
construct for users to define inner interpreters using high level Forth commands. These high level inner interpreters are easy to develop and eminently transportable across different host computers. The mechanism which allows this type of inner interpreters to execute correctly is DODOES:

```
LABEL DODOES
W INC  W INC  Point W register to parameter field which may contain data.
W PUSH  Push W on the data stack.
NEXT    Return to execute the high level interpreter while the top item on the data stack
        points to the parameter field of the current word.
```

Using this DODOES, the new commands defined by the CREATE-DOES> structure is almost identical to those defined by the CREATE-;CODE structure. DODOES must be the first command to be executed in the high level inner interpreter.

**Deferred Command Interpreter**

F83 uses a special technique to handle forward references, which is normally not allowed in a regular Forth system. A deferred command is created with a blank parameter field. When the contents of the deferred command is finally compiled, the parameter field in the deferred command is then patched with a pointer pointing to the beginning of the compiled codes so that the deferred command can be executed. Before the contents of a deferred command is defined, however, the deferred command can be referred to by the compiler and be compiled as other regular commands even if it cannot be executed. This technique is useful, especially during metacompilation, where commands have to be referred before their functionality can be precisely defined by the commands compiled after them.

The deferred command interpreter fetches the address in the parameter field and makes an indirect jump through it:

```
LABEL DODEFER
W INC  W INC  Get the parameter field address.
0 [W] W MOV  Replace W with contents of the parameter field.
0 [W] JMP    Make an indirect jump through it.
```

The deferred address can also be stored as a user variable so that each user may have its own version of the execution procedure to be referred to by the same name.

**4.4. Interpreters for In-Line Data and Strings**

In the parameter field of a colon command there is normally a list of execution addresses, which is
scanned sequentially by the address interpreter and executed. However, there are many instances that the execution sequence must be change in runtime or that some special data have to be included in-line with the execution addresses, like literal numbers and character strings. A set of special commands are defined to take care these conditions at runtime, when the colon command is being executed. Although these special commands are given names like other commands and can be found by both the text interpreter and the colon compiler, they are not meant to be invoked by either. They are compiled into colon commands by a corresponding set of immediate commands or compiler directives. To indicate their associations with corresponding compiler directive and that they are not to be directly invoked, they are assigned names with enclosing parentheses. Executing them interactively from a terminal is the most convenient way to crash a Forth system. Be warned of it!

```
CODE (LIT) ( -- n ) Push the contents in next cell on the data stack.
AX LODS Load the contents of next cell, pointed to by IP, into AX. Increment IP to skip over the numeric literal.
1PUSH Push the literal number on the data stack and return.
```

LODS is an interesting 8086 instruction. It is used to access character strings in memory using the SI register as a pointer. After the memory fetching, SI is automatically incremented. It happens that the SI register is the IP register in Forth virtual computer and the incrementing is exactly what we wanted in (LIT). It makes an extremely simple code command for (LIT). APUSH pushes the contents of AX register on the data stack before falling into the NEXT routine.

(LIT) thus overrides the natural tendency of the address interpreter to interpret data as execution addresses and forces the interpretation of the contents in the next cell as an in-line literal. This is the way literal numbers are compiled in a colon command, preceded by (LIT), so that in runtime, the number will be pushed on the stack and not to be mistaken for an execution address.

```
: (""") ( -- ) Print the next character string to the terminal.
R> RP is pointing to the next cell where the string starts. Pop the string address to data stack.
COUNT Get the string address and character count on the stack.
2DUP + EVEN Compute the address of the next executable word after the string.
>R Replace the next execution address back on the return stack.
TYPE ; Now, type out the string.
```

("") and the character string following it are compiled by the immediate command .", in-line with the other execution addresses in a colon command. When the colon command is executed, ("") will pull this string out of the execution sequence, print it on the terminal, and then pass the control to the command after the string. This is the way we let a colon command print messages on the terminal to facilitate the user-computer interface. Computer can be made much more friendly this way if proper messages are printed timely.

```
: (*) ( -- addr n ) Leave the address and the character count of the following string on the stack and continue execution after the string.
```
4.5. Interpreters for Control Structures

BRANCH and ?BRANCH

We all think Forth is a totally structured programming language, even saying: "Look Mom, no GOTO's!" GOTO's are replaced by structures like IF-ELSE-THEN, BEGIN-UNTIL, and DO-LOOP, etc. Well, the hard truth is that Forth does have GOTO's, disguised in names like BRANCH and ?BRANCH, and many other commands. If you learned how to use them, you could jump anywhere you wanted and create really messy spaghetti codes. Novices are made to believe Forth is GOTOless because they are shielded from the dark side of Forth.

BRANCH and ?BRANCH take the contents in the next cell as the address of the next executable command and direct the address interpreter to that address to start a new execution sequence. This can be done simply by manipulating the interpretive register IP.

```
CODE BRANCH ( -- ) Perform an unconditional jump to the address in the next cell.
LABEL BRAN1
0 [IP] IP MOV Copy next cell into IP, thus
NEXT effecting the branch.
END-CODE

CODE ?BRANCH ( f -- ) If the flag on stack is false, branch to the next address; otherwise, skip the
next cell and continue the execution sequence.
AX POP Pop the flag into AX register.
AX AX OR Set the CPU status register.
BRAN1 JE Branch if flag is false.
IP INC IP INC Skip the jump address if flag is true.
NEXT
END-CODE
```

BRANCH is compiled by ELSE, REPEAT, and AGAIN. ?BRANCH is compiled by IF, WHILE, and UNTIL. The cell immediately following BRANCH or ?BRANCH is the address of the next executable command in memory, and it directs the conditional or unconditional branching, deviating from the normal sequential execution path favored by the address interpreter.

The New F83 Loops

The DO-LOOP structure experienced a major surgery in the birth of Forth-83 Standard, drastically
deviated from the DO-LOOP structure as Charles Moore invented. The basic reasons behind the new DO-LOOP structure were to eliminate the discontinuity of indexing through the 8000H boundary and to leave the loop immediately at LEAVE. F83 provides a solution by using three numbers on the return stack to handle the indexing and looping. The number at the bottom of the three is the address of the command right after LOOP, providing LEAVE with the return address to terminate the looping. The second number is the loop limit, offset by 8000H so that the index range from 0 to FFFFH becomes contiguous. The top number is the difference between the index and the limit, also offset by 8000H. At the end of the loop, LOOP increments the top number on the return stack by either one or the amount specified in the case of +LOOP, and tests for overflow from bit 14 to bit 15. The overflow condition occurs when the 8000H boundary is crossed from either direction. Therefore, both the positive and negative increments are handled correctly with a single run-time loop routine. Since the address of the command after LOOP is carried on the return stack, LEAVE can use this address to jump out of the loop.

CODE  DO
( limit index -- )
Push the in-line exit address and the modified loop limit and scan range on the
return stack.
AX POP Get the index.
BX POP Get the limit.
LABEL PDO
RP DEC RP DEC Make room on the return stack.
0 [IP] DX MOV Get the in-line address following DO.
DX 0 [RP] MOV Push the exit address on the return stack.
IP INC IP INC Pointing IP to the next executable word.
8000 # BX ADD Offset the limit by 8000H.
RP DEC RP DEC Make more room.
BX 0 [RP] MOV Push the modified limit on the return stack.
BX AX SUB Subtract limit from index, also offset by 8000H.
RP DEC RP DEC Make room.
AX 0 [RP] MOV Push the index scan range on the return stack.
NEXT All done.
END-CODE

CODE  (?DO)
( limit index --)
Same as (DO) except that if index is the same as limit, the entire loop is skipped.
AX POP Index.
BX POP Limit.
AX AX CMP Compare index and limit.
PDO JNE If not equal, execute the loop.
0 [IP] IP MOV If equal, jump over the do loop.
NEXT END-CODE

With the modified index, modified limit, and the exit address on the return stack, the task for end-of-loop routines are much easier. Believe it or not, this new loop structure is claimed to run faster than the old, traditional loop.

CODE  (LOOP) ( -- ) Branch back to the executable word after DO if the index does not cross the 8000H boundary. If it does, exit the loop after clearing the return stack.
1 # AX MOV Increment by one. LABEL PLOOP Increment top of return stack,
AX 0 [RP] ADD the scanning index.
BRAN1 JNO If overflow condition is not set, jump to the in-line address compiled after (LOOP)
and repeat the loop.
6 # RP ADD Pop all three numbers off the return stack. Clean up the return stack to the state before the do-loop.
IP INC IP INC Point IP to the next executable word. Exit the loop.
NEXT END-CODE

CODE  (+LOOP) ( inc -- ) Increment the scanning index by the value on the data stack and decide whether or not to loop.
Since the scanning index on top of the return stack is not the index as we understood, the functions of I and J are also different.

CODE I ( -- index )  Return the current loop index.
0 [RP] AX MOV  Get the scanning index on top of the return stack.
2 [RP] AX ADD  Add the modified limit to the scanning index. The result is the actual current index.
1PUSH  Push it on data stack.
END-CODE

CODE J ( -- index )  Return the loop index of the next outer loop in nested do-loops.
6 [RP] AX MOV  Get the outer index.
8 [RP] AX ADD  Add the outer limit.
1PUSH  Push the computed index on stack.
END-CODE

The New Leave

Forth-83 Standard requires that when LEAVE is executed inside a loop, the loop be exited immediately. It was agreed that the old LEAVE is not desirable in allowing execution to continue to the next LOOP before exiting the loop. Unwelcome guests should not be permitted to remain when the party is over. Since the exit address of the command after LOOP is compiled after (DO) and tucked on the return stack, LEAVE can be executed using this piece of information:

CODE (LEAVE) ( -- )  Immediately exit a DO-LOOP.
LABEL PLEAVE
4 # RP ADD  Pop the index and limit off the return stack.
0 [RP] IP MOV  Copy the exit address to IP, ready to exit the loop.
RP INC RP INC  Clear the return stack.
NEXT END-CODE

CODE (?LEAVE).( f -- )  Exit the loop immediately if the flag on stack is true. If not, continue the looping.
AX POP  Get the flag.
AX AX OR  Test the flag for zero.
PLEAVE JNE  True. Leave the loop.
NEXT  False. Continue.
END-CODE

LEAVE is not very useful all by itself because it will defeat the purpose of a do-loop. In most cases, it is used after a testing condition like IF. ?LEAVE combines the functions of IF and LEAVE, and is a much more useful command.
Chapter 5. The Forth Nucleus

The source code discussed here is in the file KERNEL86.BLK, Screens 16 to 37.

In the last chapter on Virtual Forth Computer, what we discussed was the 'hardware' of this conceptual computer, such as the registers, the memory and its organization, buffers, and stacks. The inner interpreters are similar to the CPU in this computer, which cause the machine to perform the most primitive operations like jumping from one Forth Command to the next. There is also a 'software' part of the Virtual Forth Computer, i. e., the primitive command set or the elementary operations from which programs can be constructed to solve complex, real life programming problems. This primitive command set, the counter part of the microcodes or random logic machine instruction set in a real, conventional computer, is what we mean by the Forth Nucleus. In a real Forth computer, this instruction set will probably be microcoded or committed to random logic in the Forth CPU. Before that becomes a reality, the Forth Nucleus will have to be implemented on a real CPU using its native machine codes.

F83 is available in three versions: one for 8080, one for 8086/8088 and one for the more recent 68000. It's a pain to discuss the Forth nucleus in 8080 machine code, because we have to pretend that the 8 bit 8080 is a 16 bit machine. There is so much noise in the 8080 codes that you can hardly hear the beautiful music played in Forth. 8086 is far from being a dream machine. Being a 16 bit machine with more than enough registers in CPU, the Forth nucleus put on it looks much nicer and the code is considerably shorter. For most of the commands in the Forth nucleus, the 8086 code is less than 1 line in length and the functions are fairly obvious. In fact, most code is simple enough that I really don't have to go through them line by line, as I did for the inner interpreters. I will only go through the code by functional groups, making some occasional comments on special features in the F83 implementation.

I encourage you to read the code in the nucleus carefully because they are good examples of assembly programming in. There are lots of techniques and styles we can learn from these code. When you want to write code commands to take advantage of the speed and to tackle some hardware facilities, the best way is to pick up a code command in the nucleus of similar functions and modify it to suit your need. Once you are at home with the manipulation of stacks and the CPU registers in Forth assembly style, you will be able to build your own castles.

5.1. 8086 Assembly Language in Forth

Assembly code in Forth is quite different from the normal assembly code in conventional assembler.
The most eye-catching difference is that the Forth assembly codes are written in reverse Polish notation, i.e., operands preceding the operator. The reason is simple. In Forth, the assembler is not a gigantic program which assembles mnemonic codes line by line. The assembly functions are scattered in many small pieces of Forth commands which are given assembly mnemonic names. When a Forth command like MOV is executed, it compiles a machine code to the dictionary where we are building the parameter field of a code command. When MOV is executed, it needs information like source register, destination register, and address mode. These information, or the operands, are provided on the data stack prior to the invocation of MOV. MOV takes the operand information from the data stack, does some computation to derive the correct machine code, and compiles this code to the top of the dictionary. All the other assembly commands do similar things, using data from the stack and compiling specific codes to the dictionary.

There is a major difference between the Forth colon compiler and the Forth assembler, even though they both build new commands on the dictionary. When compiling colon commands, the Forth computer is in the compiling mode, under which commands parsed out from the input stream are not executed, but have their addresses added to the dictionary. During assembly, the Forth computer is in the interpretive mode, under which all the assembly commands are executed. The net result produced by the execution of an assembly command is that a machine code is added to the dictionary. In other words, we can claim that it is the Forth text interpreter who does the assembly of machine codes. The full Forth system, with all its resources, are supporting the process to assemble machine codes. In a way, the assembly process is so much more complicated than compiling colon commands that it indeed needs the support of the whole Forth system. The complexity of the assembler is best seen in the actual codes of the Forth 8086 assembler, which will be the subject in a separated chapter. At this moment, we just have to learn how to read the Forth assembly code in the nucleus.

5.2. Code Definitions in Forth Nucleus

In the F83 Nucleus, all the code commands are written in the following general format:

```
CODE <name> < operands and assembly mnemonics > <end> END-CODE
```

A code command is enclosed between two commands CODE and END-CODE. Immediately following CODE is the name given to the command. After the name, there is a sequence of commands which are either assembly mnemonics or operands used by the mnemonics. The assembly mnemonics are Forth commands which assemble machine codes into the parameter field of the code command under construction. The command before END-CODE must be a special command which returns control to the routine which calls the command in runtime. Anywhere inside or outside of the code command, comments are placed between ( or (S and ), which are ignored by the Forth interpreter which does the assembly.
The assembly commands have mostly the same names as those mnemonics used in the regular 8086 assembler provided by Intel. However, they are not just names of machine instructions, they are actually Forth commands which assemble machine instructions into the dictionary when they are interpreted or executed. Many of these mnemonic commands require operands, which are supplied before the mnemonic commands. If two operands are needed, the format is:

<source operand> <destination operand> <mnemonics>

A partial list of the mnemonic commands is:

MOV  PUSH  POP  JMP  JE  JNE  JCXZ
ADD  SUB  MUL  DIV  AND  OR  XOR
MOVS  PUSHF  REPZ  SAHF  WAIT  LODS  XLAT

The following registers are defined in F83 for 8086:

AL  CL  DL  BL  AH  CH  DH  BH
AX  CX  DX  BX  SP  BP  SI  DI
ES  CS  SS  DS

Forth registers RP, IP, and W are equivalent to BP, SI, and BX, respectively.

Several registers are often used for indirect addressing. The indirect addressing operands are the following:

[RP]  [IP]  [W]  [SI]  [DI]  [BP]  [BX]

An offset number must precede the indirect addressing operand. Numeric values needed as operands must be used with a numeric operator following immediately:

#  #)  S#)

where # is preceded by an immediate constant, #) is preceded by an address, and S#) is preceded by an address for intersegment jump.

Three most frequently used code endings are NEXT, 1PUSH, and 2PUSH. They are assembly macros which return control to the next command in the execution sequence. 1PUSH pushes the AX register on the stack before jumping into NEXT, and 2PUSH pushes first the DX register and then jumps to 1PUSH. Sometimes a JMP is used as a code ending. The routine jumped to must eventually fall into NEXT so that the execution can be continued.

5.3. Examples of Code Definitions

The following are a few simple examples of the code commands. They are fully commented here for the purpose of demonstrating the Forth assembly syntax. Since 8086 has most of the functions required by Forth in machine instructions, the code commands in the F83 nucleus are fairly simple and obvious. I will not try to make dumb comments any more.
CODE @ (addr -- n) Fetch a 16 bit value from addr.
BX POP Pop addr into BX register.
0 [BX] PUSH Push the contents of addr, indexed by BX with 0 offset, onto the data stack.
NEXT Jump to next and return.
END-CODE End of code definition.

CODE !( n addr -- ) Store a 16 bit value at addr.
BX POP Pop addr to BX register.
0 [BX] POP Pop n into memory at addr.
NEXT END-CODE

CODE C@ (addr -- char) Fetch an 8 bit value from addr.
BX POP Pop addr into BX register.
AX AX SUB Clear the 16 bit AX register.
0 [BX] AL MOV Copy one byte at addr to AL.
1PUSH Push the byte value on stack and return.
END-CODE

CODE C! (char addr --) Store an 8 bit value at addr.
BX POP
AX POP Pop char into AX.
AL 0 [BX] MOV Store byte into addr.
NEXT END-CODE

Other code commands in the nucleus are fairly straightforward and are also adequately commented in the shadow screens. They are grouped together and shown here for references. You are encouraged to read the detailed code and comments in the source listing.

Memory Commands

@ ! C@ C! CMOVE CMOVE> FILL ERASE BLANK MOVE HERE PAD

Stack Commands

SP@ SP! RP@ RP! DROP DUP SWAP OVER TUCK NIP
ROT ROT FLIP ?DUP R> >R R@ PICK ROLL

Logic Commands

AND OR XOR NOT TRUE FALSE CSET CRESET CTOGGLE ON OFF

Arithmetic Commands

+ - ABS +! 2* 2/ U2/ 8* 1+ 2+ 1- 2- UM* U*D UM/MOD *D M/MOD MU/MOD * /MOD
/ MOD */MOD */

Comparison Commands

0= 0< 0> 0<> = <> ?NEGATE U< U> <
> MIN MAX BETWEEN WITHIN

Double Integer Commands
\texttt{2\textbackslash 0 \ \textbackslash 2! \ \texttt{ZDROP} \ \texttt{2DUP} \ \texttt{ZSWAP} \ \texttt{ZOVER} \ \texttt{3DUP} \ \texttt{4DUP} \ \texttt{2ROT} \ \texttt{D+}}

\texttt{DNEGATE \ \texttt{S\textgreater D} \ \texttt{DABS} \ \texttt{D2\textdiv} \ \texttt{D=} \ \texttt{D0=} \ \texttt{DU<} \ \texttt{D<} \ \texttt{D>} \ \texttt{DMIN} \ \texttt{DMAX}}

\textbf{String Commands}

\texttt{COUNT} \ \texttt{LENGTH} \ \texttt{-TRAILING} \ \texttt{UPPER} \ \texttt{COMP} \ \texttt{CAPS\textemdash\texttt{COMP}} \ \texttt{COMPARE}
Chapter 6. Terminal Input and Output

The source code discussed in this chapter is in file KERNEL86.BLK, screens 41 to 49.

Forth is an interpretive language which intimately interacts with you through a CRT terminal. Terminal input and output control is a very important part of the Forth system, allowing you to enter commands and data into the computer and display the results or messages on the CRT. Many Forth implementations have input/output commands coded in the host machine codes which access the terminal directly. These Forth systems are often stand-alone systems which do not need support from a traditional operating system. F83 was designed to run under the popular CP/M or MS-DOS system, so that it can be transported between different host computers. The terminal input/output commands in F83 thus utilize the CP/M or DOS BIOS routines to receive information from the keyboard and send information to the CRT display.

6.1. The BIOS I/O Calls to the Operating System

The fundamental interface between Forth terminal I/O commands and the CP/M or DOS is the Forth command BDOS:

```forth
CODE BDOS
  ( entry function -- return-value )
  LOAD function code into C register and entry parameter into D register. Call the
  BIOS. Return result are then pushed on the data stack.
  CX POP Load function code into C register.
  DX POP Load entry parameter into D register.
  33 INT Call BIOS by a software interrupt. This is the MS-DOS interrupt vector. For CP/M,
  it is 224 INT.
  AH AH SUB Clear the high byte in the AX register.
  1PUSH Return with the result on stack.
END-CODE
```

BDOS is not only used for terminal I/O, it can also be used for most of the disk I/O calls, making Forth I/O commands very neat and simple.

```forth
: (KEY?) ( -- f ) Return a true flag if the user presses a key. Otherwise, return a false flag.
  0 11 BDOS
  Function 11 is the direct console I/O call. Entry 0 specifies a console status
  command. If no character is ready, 0 is returned. If a character is entered,
  FFH is returned.
  0<> Reversed the BDOS flag.
; : (KEY) ( -- char ) Wait until a key is pressed and return the ASCII code.
  BEGIN
  Pause
  (KEY?) Is a key pressed?
  UNTIL Yes, then exit the loop. Otherwise, wait another round.
  0 8 BDOS Entry parameter 0 specifies a console input function and returns an ASCII code
  on the stack.
; : (CONSOLE) ( char -- ) Send the character on stack to the terminal for display.
  PAUSE Let other task have a run on the CPU.
  6 BDOS Call BDOS to send out the character.
```
DROP BDOS always returns a number on the stack. It has to be dropped.
1 #OUT +! Increment user variable #OUT, keeping track of the output character count.
;
: (PRINT) ( char -- ) Send a character to the printer.
PAUSE Always pause before an I/O operation because I/O operations are generally slow.
CPU can then be freed to serve other tasks or other users.
5 BDOS Function 5 is the BDOS call for output to listing device.
DROP Clear the stack.
1 #OUT +! Increment #OUT.
;
: (EMIT) ( char -- ) Send the character to both the terminal and the printer.
PRINTING @ Is the printing flag set?
IF Yes. Sent character to the printer.
DUP (PRINT) Print character.
-1 #OUT +! Back up #OUT so it will not be incremented twice
THEN
(CONSOLE) Output to the terminal.
;
(KEY?), (KEY), and (EMIT) are the actual commands vectored to by KEY?, KEY, and EMIT.
EMIT can be vectored to (PRINT) or (EMIT) like the regular CP/M system to activate the printer
with the console.

6.2. Terminal Output Commands

The following output commands are all simple derivatives of EMIT and they do not need extensive
comments:

: CRLF ( -- ) Send a carriage return and a line feed to the console.
3 EMIT Carriage return.
10 EMIT Line feed.
#OUT OFF Clear the output character count.
1 #LINE +! Increment the line count.
;
: TYPE ( addr len -- ) Display a string on the console.
0 ?DO Repeat len times, but skip if len is zero.
DUP C@ EMIT Send one character.
1+ Increment character address.
LOOP Clear stack.
DROP
;
: SPACE ( -- ) Send a space to console.
BL EMIT
;
: SPACES ( n -- ) Send n spaces to console.
0 MAX Eliminate negative counts.
0 ?DO Repeat n times.
SPACE LOOP
;
: BACKSPACES ( n -- ) Send n backspaces to console.
0 ?DO BS EMIT LOOP
;

6.3. Interpreting Control Characters

Forth is capable of using most of the ASCII characters for command names. Only a few ASCII
codes are reserved for system functions. Many figForth system reserve the NUL (ASCII 0), CR
(ASCII 13), and SP (ASCII 32) as delimiters for Forth commands. The DEL (ASCII 127) is
used to nullify the previously entered character, which is important in correcting typing errors.
Other non-printable characters or control character can be used freely to name commands.
Because it is difficult to document the non-printable characters, embedding them in names is
discouraged unless you want a very secured environment.

F83, on the other hand, provides a mechanism for you to implement special functions for control
characters. When a function is defined for a particular control character, the function will be
executed immediately when that character is entered on the keyboard. A jump table is maintained
for all the 32 control characters. A few of them are used for special purposes which are defined as
follows:

: BS-IN (n char -- n-1) Back up the input character buffer by dropping the character off the stack and
decrementing n by 1. If n is zero, sound the bell instead.
DROP Discard the character.
DUP IF Is n=0?
1- BS Yes. Decrement n and backspace.
ELSE BELL n=0. Sound the bell.
THEN EMIT Send either BS or BELL to console.
;
: (DEL-IN) (n char -- n-1) Backup the input and erase the previous character. If n=0, sound the bell.
DROP DUP IF
1- BS Backspace.
SPACE BS Send a space and backspace again. Erase the previous character.
ELSE BELL
THEN EMIT ;

: BACK-UP (n char -- 0) Erase the current line and set the character count to zero.
DROP Discard the character on stack.
DUP BACKSPACES Backup to the beginning of the current line.
DUP SPACES Erase all the characters on this line.
BACKSPACES Backup
0 Clear character count.
;
: RES-IN (char -- ) Reset the Forth system to a clean start again.
FORTH Set default vocabulary.
TRUE Force system abort.
ABORT* Reset* Abort with a message.
;
: P-IN (char -- ) Toggle the printer on or off.
DROP PRINTING @ Get the flag in PRINTING.
NOT Complement it to turn the printer on or off.
PRINTING ! Store it back.
;
: CR-IN (m addr n char
-- m addr m ) Finish input and remember the number of characters in SPAN.
DROP SPAN ! Store n in SPAN.
OVER Duplicate m.
BL EMIT Send out a space.
;
: (CHAR) (addr n char
-- addr n+1) Process a normal character by appending it to the input buffer.
3DUP EMIT Send character to console.
+ Addr+n, the memory address for the current character.
! Store the character into the input buffer at addr+n.
Increment n by 1 for the next character.

DEFER CHAR  CHAR will be vectored to (CHAR).
DEFER DEL-IN  DEL-IN will be vectored to (DEL-IN).
VARIABLE CC  CC will be used to point to the current control character table.
CREATE CC-FORTH  The control character table which can handle each control character as a special case. It is actually an execution array which is indexed into by EXPECT to do the right thing when it receives a control character.

Enter compilation mode to compile 32 execution address for the 32 control characters.

6.4. More Sophisticated Input Commands

KEY is the most elementary command to accept keyboard input. It simply gets a character and puts its ASCII code up on the data stack, not a very intelligent command. Once we have the control character table, we can build a very intelligent input command which can respond to many control characters to do a wide range of different things in response to our keyboard strokes. This command is EXPECT:

```plaintext
: EXPECT ( addr len -- ) Get a string from the terminal and place it in the buffer at addr specified. Perform a limited amount of line editing. Save the number of characters input in the variable SPAN. Process control characters as specified by the control character table pointed to by CC.
DUP SPAN ! Save len in SPAN.
SWAP 0 Stack is now: len addr 0 --
BEGIN Start the input loop.
2 PICK Copy len to top of stack.
OVER - { len addr count #left -- }
WHILE If all characters were received, exit the loop. If #left is not 0, continue on.
KEY Get one more character.
DUP BL < Is it less than 32, i.e., a control character?
IF Yes. A control character.
DUP 2* Offset to the CC table.
CC @ + The table entry address.
PERFORM Execute the CC table entry.
ELSE Not a control character.
DUP 127 = Is it a DEL?
IF DEL-IN Yes. Do delete the prior character.
ELSE CHAR No. A regular character.
THEN
THEN
REPEAT End of string input loop.
2DROP DROP Clear the stack.
;
: TIB ( -- addr ) Get the address of the terminal input buffer.
'TIB @ TIB is vectored through 'TIB.
;
: QUERY ( -- ) Get an input stream of text from the terminal and store it in the terminal input buffer. Prepare the system to interpret this input text.
TIB 80 EXPECT Receive up to 80 characters into the terminal input buffer.
.SPAN @ Get the actual length of the input stream, which may be less than 80.
#TIB ! Store it in #TIB so that the text interpreter will know when the text is exhausted.
>IN OFF Clear the character pointer to start from the beginning of the terminal input buffer.
;
QUERY is the Forth input command at the highest level. It waits on you to type a line of text on the keyboard. The line is terminated either by receiving 80 characters from the keyboard or by receiving a carriage return key. The line of text is stored in the terminal input buffer. All the pertinent parameters are set so that the text interpreter can take over and interpret or execute the commands given in the input line.

6.5. String Commands

Screens 41 to 43 are a set of commands to operate on strings in memory. A string in Forth is a sequence of ASCII characters preceded by a byte count. A string may have zero to 255 characters. It is generally identified by the address of the count byte. However, most string commands require the address of the first character in the string as argument, not the address of the count byte. String commands use the following generalized syntax:

<source addr> <dest addr> <length> <string command>

Destination address is optional in cases of single string operations.

Most of the string commands are standard Forth-83 commands and their commands are simple and straightforward. I will only list here their functions and stack parameters:

Table 6.1. String Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Stack Effects</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>( addr -- addr+1 len )</td>
<td>Convert the string address to address-length representation.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>( addr -- addr+2 len )</td>
<td>Return address-length for long strings whose character count is 16-bits.</td>
</tr>
<tr>
<td>FILL</td>
<td>( addr len char -- )</td>
<td>Initialize a string to char.</td>
</tr>
<tr>
<td>ERASE</td>
<td>( addr len -- )</td>
<td>Initialize a string to NUL.</td>
</tr>
<tr>
<td>BLANK</td>
<td>( addr len -- )</td>
<td>Initialize a string to blanks.</td>
</tr>
<tr>
<td>MOVE</td>
<td>( sour dest len -- )</td>
<td>Move a string without overlapping.</td>
</tr>
<tr>
<td>UPC</td>
<td>( char -- char' )</td>
<td>Convert a character to upper case.</td>
</tr>
<tr>
<td>UPPER</td>
<td>( addr len -- )</td>
<td>Convert a string to upper case.</td>
</tr>
<tr>
<td>-TRAILING</td>
<td>( addr len -- addr len' )</td>
<td>Delete trailing blanks from a string by changing its length.</td>
</tr>
<tr>
<td>COMP</td>
<td>( sour dest len -- n )</td>
<td>Compare source string with destination string. Return -1 if source&lt;destination. Return 1 if source&gt;destination. Return 0 if strings are the same.</td>
</tr>
<tr>
<td>CAPS-COMP</td>
<td>( sour dest len -- n )</td>
<td>Compare two strings regardless of character cases.</td>
</tr>
<tr>
<td>COMPARE</td>
<td>( sour dest len -- n )</td>
<td>Compare two strings. If CAPS is true, convert to upper case before comparing.</td>
</tr>
</tbody>
</table>
Figure 6.1   Representation of strings
Chapter 7. The Virtual Memory

The source code discussed in this chapter is in KERNEL86.BLK, Screens 50 to 56.

7.1. Mass Storage and Virtual Memory

Mass storage is a very important and integral part of a computer although we often think of it as a peripheral or an appendage. The computer uses the RAM memory for most of its normal operations, executing programs stored in RAM and operating on data stored in RAM. However, programs and data must be saved to more permanent and less expensive media before the power to the computer is turned off, or to transfer programs or data from one computer to another. Without mass storage, a computer is just as useful as a video game, operating entirely from the ROM memory with very limited amount of RAM. Most programming languages, by default or by neglect, do not include facilities to deal with the mass storage as part of the language. Thus we have to have a huge beast, an operating system, underneath the language to supply the functions necessary to use the mass storage conveniently and effectively.

Charles Moore perceived the need to use mass storage efficiently, to make Forth not only as a programming language, but also as a total environment in which you can describe and solve your programming problems. At the time he put together Forth, core memory was much more expensive than now and its use had to be optimized at all costs. His design used the mass storage, whether tape or disk, as a direct extension of the core memory. The user can address the mass storage in the same way he addresses the core memory, without worrying the detailed processes in storing data to disk or retrieving data from disk. This is the concept of 'virtual memory'.

The way how virtual memory operates is as follows. The mass storage, tape or disk, are divided into consecutive blocks as the basic storage units, each block consisting of 1024 bytes. The blocks are numbered from 0 to the capacity of the device, and are addressed by the block number. In the core or the RAM memory in the computer, an area called disk buffer is reserved as temporary storage for blocks of data from disk. One disk buffer is also of 1024 bytes and one or more disk buffers can be reserved. When a block of data is needed, it is read from the disk and stored in one of the disk buffers. Data in this disk buffer can then be used or modified, as needed by the program. When all disk buffers are filled and the system needs to read another block, the system will select the least used disk buffer to receive the new block of data. If the data in the selected disk buffer was modified by the program and was marked as 'updated', the contents of this buffer will be written back to the disk before the new block is read in. This way, the data on disk are assured of their integrity and constantly updated as required, while a few disk buffers can fulfill...
the need to gain access to the entire disk without large overhead.

If you know how to read and write a sector of the disk, it is not a big job to implement the virtual memory system in Forth. Many Forth systems include such functions. These Forth system have no need for an operating system because the functions of an operating system, handling terminal I/O and managing mass storage, are all provided by the Forth system. In this sense, Forth is its own operating system. F83 on the other hand was designed to run under the CP/M or MS-DOS system and uses the file system in CP/M for mass storage. The advantage is that the resulting Forth system can be easily transported from one computer to another, under the umbrella of CP/M or DOS, and that data can be dealt with within a more manageable file system. The disadvantage is that one cannot address disk at the sector level and the performance is degraded.

7.2. Disk Buffers

A set of pointers and constants are needed to construct the virtual memory system in allocating the disk buffers and defining their characteristics:

0 CONSTANT FIRST  The starting address of the disk buffers. The actual address is patched during Forth initialization.
0 CONSTANT LIMIT  The address above the top disk buffer. Also patched at initialization.
4 CONSTANT #BUFFERS  Four disk buffers are allocated in this example.
1024 CONSTANT B/BUF  1024 bytes per disk buffer.
128 CONSTANT B/REC  128 bytes per record in CP/M and DOS.
8 CONSTANT REC/BLK  8 records per block of 1024 bytes.
42 CONSTANT B/FCB  44 bytes in a file control block.
VARIABLE DISK-ERROR  Storing error code after a disk operation.
#BUFFERS 1+ 8 * 2+  The size of a buffer-pointer array. Each disk buffer uses 8 bytes in this array to store buffer information:
0-1  Block number
2-3  Pointer to file
4-5  Buffer address
6-7  Update flag

This disk buffer pointer array is reserved just below the first disk buffer or FIRST. Whenever a block is referenced, its pointer is moved to the head of this array, so that the most recently used buffer is always checked first. This allows the references to multiple disk buffers to be very fast. Disk buffers are 1024 bytes long. No trailing zeros are needed to stop the text interpreter, as in figForth, because the text interpreter in F83 will process only 1024 characters in a buffer. The following commands are defined to get pointers to address into this array:

: >BUFFERS ( -- addr )  Return the address of the first buffer pointer.
FIRST >>SIZE  Starting address of first buffer.
-  Total bytes in the pointer array.
;  First buffer pointer entry.

: >END ( -- addr )  Return the address of the last cell in the buffer pointer array.
.FIRST 2-  One cell below FIRST.
;
: BUFFER# ( n -- addr )  Return the address of the nth buffer pointer.
8*  Offset of the nth buffer pointer.
Figure 7.1  The File Control Block
7.3. The File Control Block (FCB)

CP/M-DOS programs accesses the disk files through BDOS calls. However, a program must maintain a special memory array which contains all the information about the file it is using. This memory array is called FCB, File Control Block, usually 36 bytes in length. The first byte in FCB stores the disk drive code. The next 11 bytes store the file name and extension. Bytes 33, 34, and 35 store the current random record number in read/write operation. F83 reserves 8 more bytes at the end of FCB for some special purposes. A user variable named FILE is used to store the address of the FCB currently used by the Forth system and indicates the current file. All other Forth commands doing disk I/O refer to the file pointed to by this user variable FILE.

To fully understand the structure of CP/M-DOS files and how they are utilized by programs is beyond the scope of this book. You have to go back to the CP/M manuals where these topics are treated in details. What I shall do here is to go through the F83 disk I/O commands and explain there functions. We only have to know a small portion of the CP/M to get a working knowledge of the CP/M-DOS files as required by F83 system.

CREATE FCB1 B/FCB ALLOT Create a default file control block in the Nucleus of F83. 41 bytes are reserved.
: CLR-FCB ( fcb -- ) Initialize the current FCB, given the address of the current File Control Block.
DUP B/FCB ERASE Clear the entire array to zero.
1+ Address of the first byte of the file name.
11 BLANK Initialize the name and extension to blanks, as required by CP/M.
;
: SET-DMA ( addr -- ) Set direct memory transfer address.
26 BDOS DROP A standard BDOS call.
;
: RECORD# ( fcb -- addr ) Return the address of the 3 byte pointer to the current random record.
33 + Offset to the random record pointer.
;
: MAXREC# ( fcb -- addr ) Return the address of the field storing maximum record number in the current FCB.
38 + ;

VARIABLE DISK-ERROR A variable storing the record number out-of-range flag.

: IN-RANGE ( fcb -- ) Make sure that the current random record is within range. Abort if it is not.
DUP MAXREC# @ Maximum record in the file.
OVER RECORD# @ Current record number.
U< Do an unsigned comparison.
DUP DISK-ERROR ! Store the flag in DISK-ERROR# for diagnostics.
IF 0 Error process.
BUFFER# ON Set buffer flag.
7.4. Read and Write Disk Files

The following commands are the fundamental interface to the disk drive through the CP/M-DOS BDOS. They specify the disk drive, the memory address, the sector to be read or written, and do the reading or writing.

```forth
: REC-READ ( fcb -- ) Read one record from the current file. The record number is stored in the field of random record.
    DUP IN-RANGE Check the random record number.
    33 BDOS Call the read random function.
    ?DISK-ERROR Store the returned error code in DISK-ERROR.
    ;

: REC-WRITE ( -- ) Write one random record.
    DUP IN-RANGE Check the random record number.
    34 BDOS Write the record from memory.
    ?DISK-ERROR Store error code.
    ;

One CP/M record is 128 bytes long. One Forth block is 1024 bytes long. To read or write one Forth block, we have to do eight consecutive reads or writes. Another thing we have to take care of is that there are four disk buffers allocated in the F83 system. The buffer to be used for disk I/O has to be specified by a pointer to the appropriate entry in the disk buffer pointer array in front of the buffer area.
```

```forth
DEFER READ-BLOCK Vectored to FILE-READ.
DEFER WRITE-BLOCK Vectored to FILE-WRITE.

: SET-IO ( buffer-pointer-entry -- buffer rec/blk 0 )
   Set up common parameters for file reads or writes.
    DUP 2@ Get the block number, the first cell in a buffer pointer entry.
    REC/BLK * The record number.
    OVER RECORD# ! Use it as the random record number. Put it in the FCB.
    SWAP 4 + @ Get the address of the disk buffer in the third cell or the buffer pointer entry.
    REC/BLK 0 These two parameters are the index and limit for read/write do-loops in FILE-READ and FILE-WRITE.
    ;

: FILE-READ ( buffer-pointer-entry -- )
   Read 1024 bytes from current file to the disk buffer specified on stack.
   SET-IO Set random record number and leave buffer address and loop parameters on the stack.
   DO Repeat 8 times.
      2DUP SET-DMA Address for one record.
      DUP REC-READ Read one record.
      1 SWAP RECORD# +! Increment the random record number for the next read
      B/REC + Address of next buffer area.
      LOOP
      2DROP Discard the addr on stack.
    ;

: FILE-WRITE ( buffer-pointer-entry -- )
   Write a block to file.
   SET-IO Get buffer address and loop parameters.
   DO Repeat 8 times.
      2DUP SET-DMA Record address.
      . DUP REC-WRITE Write one record.
      . 1 SWAP RECORD# +! Next random record.
      . B/REC + Address of next buffer area.
```

66
7.5. Disk Buffer Management

The above set of commands allow us to access files on disk. As mentioned earlier, F83 maintains 4 disk buffers in its memory. How are these buffers used? Who decides which buffer is given to which block? When does the block on disk get updated? These are problems we have to face in using a virtual memory system projected into a file system. The following commands are designed to deal with these problems. We might call them the 'Virtual Memory Management' in F83 system.

Let's review what we know about the virtual memory in F83. There are 4 disk buffers, each 1024 byte long. There is a buffer pointer array with 4 entries, each entry being 8 bytes long. Each entry has four cells containing the block number, the pointer to a file, the buffer address, and the update flag. This array has all the management information on the disk buffers, while the buffers contain the actual data from/to disk file.

The buffer pointer array is a prioritized structure, in which the first entry points to the most recently used buffer and the last entry points to the least recently used buffer. When a file block is requested, this array is searched. If the disk block is in one of the buffers, its pointer entry is moved to the head of the array. If the disk block is not in the buffers, then the buffer pointed to by the last pointer entry is assigned to the new disk block. However, if the contents of this buffers was modified and the pointer entry was marked as updated, this buffer will be written back to the disk file before the new disk block is read into this buffer. Thus the disk file is maintained to reflect the current state of any update and modifications, while disk read/write is kept to a minimum. The disk I/O activities are totally transparent to you, as long as you use the commands BLOCK or BUFFER to access his file.
Figure 7.2   Disk buffer management

: LATEST? ( n fcb -- fcb n | addr f )
Check if block n is the first entry in the buffer pointer array. If it is, return the buffer address and a false flag, and exit from the calling word ABSENT?. If
not, return the block number n with the file control block address.

DISK-ERROR OFF  First reset the error flag.
SWAP OFFSET @ +  Add the offset block number to obtain the true block number in the file.
2DUP Leave a copy for return.
1 BUFFER# The first entry in the pointer array.
2@ Get the FCB address and the block number of the buffer pointed to by the first entry.
D= IF If block n is pointed to by the first entry,
2DROP Drop n and fcb. They are not needed.
1 BUFFER# Get the address of the entry again.
4 + @ Get the address of the disk buffer.
FALSE Push a false flag on top.
R> DROP Discard the top address on the return stack and terminate the calling word ABSENT?.
The disk buffer was found and there's no point to search through the pointer array.
THEN Block n is not the first entry.
; Return with the block number intact.

The most recently referred block is also the block most likely to be referred to the next time.
LATEST? thus will cut down much buffer searching overhead and improve significantly the performance of the disk buffer management system.

The update cell in the buffer pointer entry is very important in the virtual memory management
system. When this cell is set to true (-1), nothing will happen for the moment. However, when this buffer space is assigned to a new disk buffer and before the data in the new block is brought in from the disk, the contents of this buffer will be written back to disk to where it came from. This way, any change we made in the disk buffer will eventually be written back to disk. On the other hand, if the update cell is set to 0 or 1, presumably that the contents of the disk buffer is the same as those on the disk, there is no need of writing the data in the disk buffer back to the disk. Therefore, new disk block can be brought into this buffer immediately without flushing the old block back to the disk. The disk accessing can thus be reduced while the integrity of data on disk is assured.

MISSING ( -- ) Discard the least recently used disk buffer. If this buffer was marked as modified, it is written back to disk. The first three buffer pointer entries are then shifted down by one entry. The first entry is made available to the new block to be brought in.

END 2- @ The address of the update cell in the last buffer pointer entry, which is the least used one.

0? IF

WRITE-BLOCK

END 2- OFF

THEN

BUFFERS

OFF

WRITE-BLOCK

END 2- OFF

THEN

BUFFERS

4 + ! Make it the buffer address of the 0th entry.

1 >BUFFERS 6 + ! Mark the 0th entry unmodified.

BUFFERS >UPDATE @ Get the update field.

0> Is it 1?

1 BUFFER# DUP

READ-BLOCK

6 + OFF

THEN ;

BUFFER ( n -- addr ) Do (BUFFER) on the current file.

FILE @ Fcb of the current file.

(BUFFER) Assign a buffer to the disk block. Write the old block in the buffer to disk if it was updated.

;

(BLOCK) ( n fcb -- addr ) Return the address of a buffer which contains data from block n. If block n is not already in one of the buffers, it is read in from the disk.

(BUFFER) Assign a buffer to the requested block.

UPDATE @ Get the update field.

0> Is it 1?

1 BUFFER# DUP

READ-BLOCK

6 + OFF

THEN ;

VARIABLE FILE Pointing to the file control block of the current file.

VARIABLE IN-FILE Pointing to the file control block of the second opened file or the in-file.

BLOCK ( n -- addr ) Read a block from the current file if it is not in the buffer. Return the buffer address.

FILE @ Fcb of the current file.

(BLOCK) Read it.
F83 allows you to open and use two files concurrently. The first file is opened with the command OPEN and is referred to as the current file. The second file is open by the command FROM and is called the in-file. The current file is always used as the output file and the in-file is always used as the input file. This way you can copy blocks from one file to another. When OPEN is executed, the invoked file is set to be both the current file and also the in-file so that you can read and write to the same file. The file control block address of the current file is stored in FILE, and that of the in-file is in IN-FILE.

: IN-BLOCK ( n -- addr ) Read a block from the in-file which is the second file opened to the system.
IN-FILE @ Get the fcb of the in-file.
(BLOCK) Read it.
;

BLOCK is the most important command to communicate with the disk. It is the virtual memory manager. If we need any block of data from disk, just give BLOCK the block number and it will make sure that you have the data in one of the disk buffers. From the address returned by BLOCK, you can access the disk block data using the regular memory accessing commands. If you remember to set the update cell when you change the data in the disk buffer, BLOCK will see to it that the modified data will be written back to disk. This is done explicitly using the UPDATE command. You can command BLOCK to ignore any change you made in a disk buffer by the command DISCARD.

In cases that you want to write raw data on to a fresh disk or you do not want to read in the disk block, BUFFER is the command to use because it does not do the disk read. You can use BLOCK for the same purpose, but then you will do a useless disk read operation. When you are writing a large file on to the disk, BUFFER can save you quite some time because only write operations are performed.

7.6. Saving Disk Buffers to Disk Files

BLOCK and BUFFER will write to disk only when disk buffers are full and new disk blocks are requested. If the computer is turned off, the buffers will be lost because their contents do not have a chance of being written back. The following commands force the system to write all the updated buffers back to disk. They are highly recommended, especially when you are doing editing work.

: SAVE-BUFFERS ( -- ) Write back all the updated buffers to disk and then mark them all as unmodified.
1 BUFFER# Address of the 1st pointer entry.
#BUFFERS 0 DO Scan all the pointer entries.
DUP @ Get the block number.
1+ IF Block number cells were initialized to -1 as empty buffer. Make sure the buffer is not empty.
DUP 6 + Address of the update cell.
@ IF Is the buffer updated?
DUP WRITE-BLOCK Yes. Write it back to disk.
DUP 6 + OFF
THEN
8 +
THEN
LOOP
DROP;

: EMPTY-BUFFERS ( -- ) First wipe out the data in the buffers. Initialize the buffer pointers to point to the right addresses in memory and reset all the update cells.

FIRST LIMIT Boundary of disk buffers.
OVER - ERASE Erase the entire disk buffer area.
>BUFFERS Buffer pointer array.
#BUFFERS 1+ 8 * Bytes in the pointer array.
ERASE Clear the pointer array.
FIRST 1st buffer address.
1 BUFFER# Address of the 1st pointer entry.
#BUFFERS 0 DO Go through all pointer entries.
DUP ON Initialize the block number cell.
4 + 2DUP ! Initialize the buffer address cell.
SWAP B/BUF + Address of the next buffer.
SWAP 4 + Address of the next pointer entry.
LOOP
2DROP ; Clear the stack.

: FLUSH ( -- ) Save and empty all the buffers.
SAVE-BUFFERS
0 BLOCK DROP Cheat the CP/M system to defeat its extra buffering in BIOS. By accessing a dummy block, you can be sure that the old one is flushed out of the pipeline and written to disk.

EMPTY-BUFFERS ;

Whenever you change disk, be sure to FLUSH out all the buffers to the old disk. During program developments, if you have any concern about losing data or crashing the system, FLUSH the buffers first. If you are absolutely sure that the data in the buffers are corrupted, use EMPTY-BUFFERS to clear the buffers. If you want to throw away only one buffer, use DISCARD immediately after you access this buffer by BLOCK, making it the most recently used buffer.
Chapter 8. Dictionary and Vocabulary

The source code discussed in this chapter is in the file KERNEL86.BLK, Screens 67-68, and 76.

If you had a figForth system, you might have noticed that it took a while to compile a screen of text. If you were to load a sizable system or application program, it might seem to be a long time before the computer came back and put an 'ok' on the screen. The reason is that the dictionary in figForth is basically a single, linearly linked list of commands. It takes some time for the text interpreter to travel through this list to find a command. The worst cases are the numbers. If the text interpreter cannot find the command in the context vocabulary, it will search again in the current vocabulary, which in most cases is the same as the context vocabulary with the entire root FORTH vocabulary tagged at the end.

F83 improves this situation by breaking the dictionary into four separately linked lists. To locate a command, only a quarter of the dictionary needs to be searched. This strategy visibly enhances the performance of the text interpreter. In this section, I hope that I can explain how this dictionary structure is implemented.

8.1. Threading of the Dictionary

First, there are several important system variables which perform the housekeeping chores in managing the vocabulary and the searching of dictionary:

VARIABLE DP Pointer to the top of the dictionary. Returned by HERE.
VARIABLE CURRENT Pointer to the current vocabulary to which new definitions are linked.
8 CONSTANT #VOC The number of vocabularies to be searched, as specified by the array in CONTEXT.
VARIABLE CONTEXT The context vocabulary pointer.
#VOCs 2* ALLOT Space to hold 8 transient vocabulary pointers. The array specifies the search order for the text interpreter.
VARIABLE VOC-LINK Pointer to the most recently defined vocabulary. Vocabularies are thus linked in the order of their creation.

The 8 numbers stored in the transient array are the parameter field addresses of up to eight different vocabularies. The text interpreter searches up to eight vocabularies and stops at the first encounter of the name it looks for.

Next, let us see how the vocabularies are defined and how to select the context and current vocabularies.

: VOCABULARY ( -- ) Define a new vocabulary.
CREATE Take the following string as the name of the new vocabulary.
#THREADS 0 DO Compile four 0's in the parameter field.
0 , They are the four threads
LOOP in the dictionary for the new vocabulary.
HERE The next cell is for the vocabulary link, VOC-LINK.
VOC-LINK @ Old vocabulary link is placed in this cell.
VOC-LINK ! The new vocabulary is the last in the vocabulary link list. Its link address must
be stored in VOC-LINK.
DOES> End of the compilation of a new vocabulary entry in dictionary. Next is the
vocabulary interpreter:
CONTEXT ! Store the parameter field address of this vocabulary in the first cell of the
CONTEXT array so that this vocabulary will be searched first by the text
interpreter.
;
: DEFINITIONS { -- ) Link subsequent definitions to the context vocabulary.
CONTEXT @ Get the address of the context vocabulary.
CURRENT ! Store it in CURRENT. New definitions will be linked to the vocabulary pointed
; to by CURRENT.
;
The interesting things are how new commands are linked to the current vocabulary and to the
threads in the dictionary. We may think that the vocabularies are the logical groupings of
commands in the dictionary and the threads are the physical groupings of commands in the
dictionary. New commands are created by CREATE, which invokes "CREATE to build the name
fields and link fields:

: CREATE ( -- ) Create an header for a new definition. The header consists of a view field, a
link field, and a name field.
COUNT Character count in the name.
HERE EVEN 4 + Address of the name field.
PLACE Move the name string into the name field.
,VIEW Lay down the view field in which the top 4 bits contain a file number and the
lower 12 bits contain a block number in the file.
HERE 0 , Store the name field address in LAST.
HERE LAST ! Save a cell for the link field to be filled later.
HERE Warning @ If the warning flag is set, search the dictionary to see if the name is unique.
IF FIND If it is an existing name,
THEN HERE COUNT TYPE print the name, \"isn't unique\" with an error message.
THEN HERE LAST ! Get the name field address of the last definition of this thread.
HERE 2- Store this link field address in the head of the linked chain.
ROT ! Store the link field address of the last definition in the current vocabulary.
SWAP ! Store the link field address of the last definition in the current vocabulary.
HERE Name field address saved on stack.
DUP @ Character length of name.
HERE 0 MIN 1+ Width of the name field.
ALLOT ALIGN Name field allocated.
128 SWAP CSET Set the MSB of the length byte, the first byte in name field
delimiter.
128 HERE 1- CSET Set the MSB of the last byte in name field as another delimiter.
COMPILE { Compile the variable interpreter in the code field.
DOCREATE , Thus complete the header.
The header in this F83 Forth is not the same as the other more popular Forth systems. A view field is added to help you locating a command in one of the CP/M files containing Forth source screens. The link field is placed before the name field so that the string comparisons can be performed more quickly without traversing through the name field. The linking of dictionary entries involves only the link fields. Name fields are no longer involved in the linkage.

### 8.2. Hashing and Searching the Dictionary
Two important commands in "CREATE above was not fully explained: HASH and DEFINED. These are the key commands used by the text interpreter to search specific commands in the dictionary. HASH is a code command. DEFINED, however, is a high level colon command which eventually calls a code command (FIND) to do the actual searching. Let us look at HASH and (FIND):

CODE HASH

( string-addr vocabulary-pfa -- thread-addr )
Given a string address and a pointer to a vocabulary, return the address of the thread in the parameter field of the vocabulary.

CODE (FIND)

( here lfa -- cfa true, if found; here false, if not found.)
Given the address of a string and the link field address of a word in dictionary, search the dictionary and return an address and a flag on the stack. Flag=1 for an immediate word; flag=-1 for a regular word; and flag=0 if the word is not found. If not found, the string address remains on the stack.
(FIND) searches through one thread, with a given link field address of a dictionary entry. To pick up one thread among four for searching and to do the searching, a high level command FIND has to be used.
4 CONSTANT #THREAD

: FIND
  { string-addr -- cfa true, if found; string-addr false, if not found}
  DUP C@ IF
  If the string is not a null string, do the dictionary searching. Otherwise, do
  the end of line processing.
  PRIOR OFF
  PRIOR is a user variable storing the last vocabulary searched. Clear PRIOR to
  begin searching.
  FALSE
  This is a dummy flag for the next do-loop.
  #VOCS 0 DO
  #VOCS=8, the number of vocabularies to be searched.
  DROP
  Drop the flag on stack.
  CONTEXT I 2* + @
  Get the vocabulary address in the CONTEXT array.
  DUP IF
  If the vocabulary address is zero, skip it because no vocabulary was specified
  for this CONTEXT entry.
  DUP PRIOR @
  Get the contents of PRIOR, the last vocabulary searched.
  OVER PRIOR != IF
  If the PRIOR vocabulary is the same as the present vocabulary, here is no need
  of repeating the searching.
  DROP FALSE
  Drop the vocabulary and replacing with a false flag. Loop back.
  ELSE
  Now search the new vocabulary.
  OVER SWAP
  Save a copy of the string address.
  HASH
  Hash the string and return the address of the head of a thread in the present
  vocabulary.
  @
  Pick up the thread, the link field address of the last entry in this thread in
  the dictionary.
  (FIND)
  Search the dictionary.
  DUP ? LEAVE
  If tos is a true flag, a word is found in the dictionary. Leave the loop immediately.
  If tos is false, repeat the loop and search the next vocabulary.
  THEN
  THEN
  LOOP
  ELSE
  Null string processing.
  DROP
  Discard the string address.
  END? ON
  Turn on the end-of-line flag.
  ['] NOOP 1
  Push the NOOP address on the stack with a true flag so that the end-of-line process
  will happen immediately.
  THEN

FIND thus scans the CONTEXT array, where up to 8 vocabularies can be specified and are to be
searched in the order of the array. When a vocabulary is to be searched, HASH selects one of the
4 threads, which are the link field addresses of the last entries in each of the threads stored in the
parameter field of the vocabulary, and hands the proper link field address to (FIND) to scan the
thread for a name matching the given string. When a vocabulary was searched, its address was
preserved in PRIOR to avoid searching the same vocabulary repeatedly. This allows the same
vocabulary to be specified in the CONTEXT array more than once without being searched more
than once. FIND can also skip nulls in the CONTEXT array. Nulls and multiple vocabulary
entries in CONTEXT are conveniences in manipulating vocabulary searching order, which will be
discussed in a moment.

: DEFINED
  { -- addr flag }
  Parse out the next word in the input stream and search the dictionary. If a matching
  entry is found, return its cfa and an l or -l. If not found, return HERE and a
  false flag.
  BL WORD
  Parse the next word, delimited by blank characters, and copy the word to HERE,
  the word buffer.
  CAPS @ IF
  If the contents of CAPS is true, the word will be converted to upper case characters.
  DUP COUNT UPPER
  Upper the cases.
  THEN
  FIND
  Now do the searching.
  ;

If an immediate command is found by FIND, the return flag is 1. If the found command is a regular,
non-immediate, command, -1 is returned. It is important for the colon compiler to know whether a command is immediate or not. The colon compiler normally compiles the code field addresses of regular commands, but executes the immediate command to take care of special compiling conditions or to build structures in a colon command.

In F83, because of the more complicated CONTEXT structure, it requires a few more commands to handle the vocabularies and to use them effectively. When a vocabulary is invoked, its parameter field address is stored into the first cell in the CONTEXT array. Next time a search is initiated, this vocabulary will be the first vocabulary to be searched. The command ONLY is used to initialized the CONTEXT array and places the address of a very small searching control vocabulary in the first and the last cell of the CONTEXT array. The commands in this control vocabulary allow us to select appropriate working vocabularies like FORTH, etc. The command ALSO copies the first CONTEXT entry to the second entry and moves the second and subsequent entry up by one cell, adding one entry to the searching order. This set of command can be used to specify any searching strategy within the size of the CONTEXT array.
Figure 8.3  Vocabularies and the dictionary structure.
Chapter 9. Number Input and Output

The source code discussed in this chapter is in KERNEL86.BLK file, Screens 58 to 61.

The Forth interpreter can only recognize two types of commands: commands or Forth commands compiled into the dictionary, and numbers. A large portion of the Forth system is devoted to processing numbers, including inputting numbers from console or disk, doing arithmetic and logic operations on them, and outputting them to console or other devices in a required format. In the nucleus layer, we've seen lots of arithmetic and logic operators. In this chapter, we will discuss how numbers are converted from the external representation in ASCII strings to the internal representation in the binary form, and vice versa.

9.1. Representation of Numeric Data

A very interesting aspect of Forth in its external representation of numbers is that numbers can be presented in many different bases. Not only decimal, octal, hexadecimal, and binary, but also in any reasonable base from 2 to 70, limited by the number of ASCII characters available to represent digits. The reason is that in Forth the primitive number input and output commands are directly accessible to you, giving you tools that you can use at will to define and modify rules in doing number input and output.

Internally, all numbers are represented in 16 bit binary form and processed in 16 bit units. In the case that more bits are required to represent large integer numbers, two 16 bit numbers are used together as a 32 bit double precision integer. For data requiring less than 16 bits, they are generally right justified in the 16 bit field and the high order unused bits are cleared to zeros.

F83 uses many different data types. Their ranges are show in the following table:

<table>
<thead>
<tr>
<th>Date Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>True flag</td>
<td>-1 or 32767</td>
</tr>
<tr>
<td>False flag</td>
<td>0</td>
</tr>
<tr>
<td>ASCII codes</td>
<td>0..127</td>
</tr>
<tr>
<td>Byte</td>
<td>0..255</td>
</tr>
<tr>
<td>Integer</td>
<td>-32768..32767</td>
</tr>
<tr>
<td>Unsigned integer</td>
<td>0..65535</td>
</tr>
<tr>
<td>Address</td>
<td>0..65535</td>
</tr>
<tr>
<td>Double integer</td>
<td>-2,147,483,648..2,147,483,647</td>
</tr>
<tr>
<td>Unsigned double integer</td>
<td>0..4,294,967,295</td>
</tr>
</tbody>
</table>

Forth is not a typed language. We can talk about data types and their external representations, but
once they are inside the Forth computer, they are all represented in the uniform 16 bit format. Forth doesn't care what type a number was when it is input into Forth. Thus you can do arithmetic on the flags and ASCII codes like any other number. You have to know what you are doing. You must use the right operator to process the data you entered. This is the price you have to pay for the convenience in using the data stack.

F83 maintains three user variables specifically for the purposes of number input/output:

- **VARIABLE BASE**: The current base for number input and number output conversions. A 10 stored in BASE causes input number strings to be treated as decimal numbers. A 16 in BASE makes the conversions done in hexadecimal.
- **VARIABLE DPL**: The decimal point location. It stores the location of the decimal point in an ASCII number string, from the right end of the string. In other words, the number of digits after the decimal point.
- **VARIABLE HLD**: The number of digits stored in the number output buffer for output.

### 9.2. Input Number Conversion

The text interpreter parses a word out of the input stream and places the parsed word in the word buffer, just above the last entry in the dictionary. It first searches the dictionary to see if the word is a pre-defined Forth command. If it fails to match the parsed word to a command, the parsed word is left in the word buffer for the number conversion routine to convert it to a number. The following set of F83 commands support the number conversion process.

- **LABEL FAIL**: A common return routine used when failed to convert the string to a number due to a number of reasons.
- **AX AX SUB**: Push a false flag on the stack
- **1PUSH and return to the NEXT routine.**

**CODE DIGIT ( char base -- n f )**

- **RETURN**: Return a flag indicating whether or not the character is a valid digit in the current base. If so, return the converted value with a true flag. Otherwise, return the character with a false flag.
- **DX POP**: Pop base into DX.
- **AX POP**: Pop character into AX.
- **AX PUSH**: Push character back to stack just in case of a conversion failure.
- **FAIL JB**: If char is below 0, it is not a valid digit. Jump to FAIL.
- **9 # AL CMP**: Is char > 9?
- **> IF**: No, a regular digit. Skip to DIGI1.
- **17 # AL CMP**: Is char between 9 and A?
- **FAIL JB**: Yes. Invalid digit. Jump to FAIL.
- **7 # AL SUB**: Eliminate the gap between 9 and A. A must be the next digit following 9.
- **THEN DL AL CMP**: AL has the converted value. Is it in the range of BASE?
- **FAIL JAE**: If the value is equal or above the base value, it is not a valid digit. Jump to FAIL.
- **AL DL MOV**: Copy value to DL for DPUSH.
- **AX POP**: Discard char from the stack.
- **TRUE # AX MOV**: Put true flag in AX.
- **2PUSH**: Push value and flag on stack and return.

END-CODE
Figure 9.1   Input and output number conversions

The sequence of digits is from 0 to 9, and from A up if the base value is greater than 10. Theoretically the sequence can go up to tilde (~ ASCII 126). Then anything you type would be
converted to a number.

: DOUBLE? ( -- f ) Return a true flag if a period is encountered in the number string.
    DPL @ Get the contents of DPL. If no period is in the number string, DPL is -1 as initialized.
    1+ 0 if no period.
    0<> Logic NOT.
    ;

: CONVERT ( ud1 addr1 -- ud2 addr2 ) Starting with the unsigned double integer ud1 on stack and the number string at addr1, convert the string to a number and add to ud1 according to the current base. Leave the resulting double integer and the address of the unconvertable digit addr2 on stack.
    BEGIN This is an indefinite loop.
    1+ Get the next character in the string.
    DUP >R C@ Get one digit and save a copy of its address on return stack.
    BASE @ DIGIT Convert one digit.
    WHILE Exit the loop if the digit is invalid.
    SWAP BASE @ UM* Left shift the upper half of the double integer by one digit.
    DROP Keep only the lower half of the product.
    ROT BASE @ UM* Left shift the lower half of the double integer by UM*. Result is a double integer sitting on top of the value of converted digit and the left-shifted upper half of ud1.
    D+ This is tricky, but the result is ud1*base+value.
    DOUBLE? Have we seen a period?
    IF 1 DPL +! THEN Yes, we get one more digit after the period. Increment DPL.
    R> Recall the character address.
    REPEAT
    DROP Discard the invalid digit left by DIGIT.
    R> Address of the invalid digit.
    ;

: (NUMBER?) ( addr -- d flag ) Given a string at addr with at least one digit, convert it to a double integer.
    0 0 The initial value of the double integer serving as accumulator.
    ROT Get addr to top of stack.
    DUP 1+ C@ Get the first digit.
    ASCII - = Compare it to ASCII - sign.
    DUP >R Save the negative flag on return stack.
    - Otherwise, start conversion at the current address.
    -1 DPL ! Initialize DPL.
    BEGIN
    CONVERT Convert the number string.
    DUP C@ Get the invalid digit.
    ASCII , ASCII / BETWEEN them is a valid punctuation mark, equivalent to a period.
    WHILE 0 DPL ! A punctuation mark is encountered. Reset DPL.
    REPEAT
    -ROT Rotate the invalid character address below the double integer.
    R> Get the negative flag.
    IF DNEGATE THEN If the number is preceded by a - sign, negate the double integer.
    ROT C@ BL = Compare the last invalid digit with blank and leave the result on stack as a flag.
    ;

F83 accepts numbers with an optional preceding - sign for negative numbers. Within the number string, four punctuation marks, ',', '-', '.', and '/' are allowed. When any of these punctuation marks appear in the string, DPL is reset to zero so that CONVERT can keep track of the number of digits following the punctuation mark, and the converting process continues on until an invalid digit other than these punctuation marks is encountered.

: NUMBER? ( addr -- d f ) Convert the number string at addr to a double integer. The number string may be preceded by a - sign, but must be terminated by a blank. The location of the last punctuation mark is saved in DPL. A true flag is left on the stack if successful.
    FALSE Put up a default flag on stack.
    OVER COUNT BOUNDS Set up loop limits to scan the supposed number string.
    ?DO Scan the string for valid digit.
    I C@ BASE @ DIGIT Is this a valid digit?
NIP I don't care its value now.
IF DROP TRUE LEAVE Leave the loop with a true flag if a valid digit is found in the string.
THEN
LOOP The purpose of this test is to filter out a mistyped word in which case it is
just a waste of time to do the number conversion.
IF (NUMBER?) Do the conversion if the string is potentially a number.
ELSE No valid digit in the string.
DROP Discard its address.
0 0 Put a null double integer on stack.
FALSE Top it with a false flag.
THEN ;

: (NUMBER) ( addr -- d ) Convert a number string to a double integer. The string may have optional leading
- sign and embedded punctuation. It must be terminated by a blank.
NUMBER? Conversion.
NOT If not a number or not terminated by a blank,
?MISSING print an error message and abort.
;
DEFER NUMBER Vectored to (NUMBER).

With this set of input conversion tool, we can type in numbers like:

| 415-424-3001 | 12/25/1983 | 123.45 | -0.4567 | 987,654,321 | 534,234.00 |

If we are in hexadecimal base the following numbers are also valid:

| A1 F9 BAD-FAD FEED/BEAD | -1B2A5D.0 |

However, after conversion, they are all internally represented by double integers. The embedded punctuation marks have no effect on the conversion except the contents of DPL.

9.3. Output Number Conversion

The primitive Forth output conversion routine converts a double integer to an ASCII string suitable
for outputting to a console or to a printer. You can explicitly format the string and insert special
characters into the string to design formats you desire. Let's look at these small tool commands
first and then see how they are strung together to build number output commands often used in
routine Forth programming.

: HOLD { char -- } Insert the character char into the output string. HLD contains a character pointer
to the output text buffer where the number output string is being constructed. The number character string is built backwards from the least significant digit
to the most significant digit. To insert a character into this string HLD has
to be decremented.
-1 HLD +!
HLD @ Get the character pointer.
C! Insert char to where HLD points.
;
: <# { -- } Initialize the number conversion process.
PAD PAD returns the location of the text buffer used for output.
HLD ! Point HLD to PAD so that the number string can be built in the PAD buffer.
;
: #> { d -- addr len } Terminate the output number conversion and leave the address and length of the
number string on stack suitable for TYPE to print out.
2DROP The double integer on stack is no longer needed.
HLD @ The address of the number string.
PAD The end of the string.
OVER - The length of string.
;
: SIGN { n -- } If n is negative insert a minus sign into the number string.
0< IF If n is negative,
ASCII - HOLD

THEN ;

: # ( d1 -- d2 ) Convert one digit and add the digit to the number string. The conversion is to divide d1 by base. The quotient d2 is left on stack and the remainder is converted to ASCII code and add to the output buffer.

BASE @ MU/MOD Divide d1 by base. The remainder and the double integer quotient are left on stack.

ROT Get the remainder to top.

9 OVER < If the remainder is greater than 9,
IF 7 + THEN add 7 to make A.
ASCII 0 + HOLD Convert to ASCII code and HOLD it in the output buffer.
;

: #S ( d -- 0 0 ) Convert a double integer until finished.
BEGIN
# Convert one digit.
2DUP OR Is the quotient 0?
0= UNTIL If it is zero, exit the loop. Otherwise, continue converting.
;

With these tools, we can format numbers for output in any format we want. However, it is always nice to look at how the F83 designers built some of the standard number output commands.

: (U.) ( u -- addr len ) Convert an unsigned single integer to a number string.
0 Make the unsigned integer a double integer.
<# Initialize the conversion.
#S Convert all digits.
> Convert.
;

: U. ( u -- ) Output an unsigned single integer with one trailing space.
(U.) Convert.
TYPE SPACE Print.
;

: U.R ( u len -- ) Output an unsigned integer in a field of len columns.
>R Save the column width.
(U.) Convert.
R> OVER - SPACES Pad with leading blanks.
TYPE Now print the number right justified.
;

: (.) ( n -- addr len ) Convert a signed single integer to a number string.
DUP ABS Get the absolute value of n.
0 Make it a double integer.
<# #S Convert all digits.
ROT SIGN Add a minus sign if n is negative.
> Finish the output string.
;

: . ( n -- ) Output a signed integer with a trailing space.
(.) Convert.
TYPE SPACE Type.
;

: .R ( n len -- ) Output a signed integer right justified in len columns.
>R (. ) Convert n first.
R> OVER - SPACES Pad with leading blanks.
TYPE Now print the number right justified.
;

### 9.4. Double Integer Output

: (UD.) ( ud -- addr len ) Convert an unsigned double integer to a number string.
<# #S #> ;
: UD. ( ud -- ) Output an unsigned double integer with a trailing space.
(UD.)
TYPE SPACE ;

: UD.R ( ud len -- ) Output an unsigned double integer right justified in len columns.
>RD (UD.)
R> OVER -
SPACES TYPE ;

: (D.) ( d -- addr len ) Convert a signed double integer to a number string.
TUCK
DAB
<# #S
ROT
SIGN
#>
;

: D. ( d -- ) Output a signed double integer with a trailing space.
(D.)
TYPE SPACE ;

: D.R ( d len -- ) Output a signed double number right justified in len columns.
>R (D.) R> OVER -
SPACES TYPE ;
Chapter 10.   Word Parsing

The source code discussed in this chapter is in the file KERNEL86.BLK, Screens 62 to 64.

10.1.   Text Processing

In communicating with you through a console, the computer must be able to accept a line of commands and find out what is your intention. The computer then can carry out the commands and do some useful work. In most conventional operating systems, the task which accepts commands from console and interprets the contents of the commands is called a command line interpreter (CLI). The user has to observe a set of rules in entering commands, because the computer uses this set of rules to determine what has to be done, given these commands. These rules are the syntax rules, or more generally, the grammar of the command line interpreter. When the command line interpreter becomes more powerful and has more functions built into it, its syntax becomes more complicated and the syntax rules multiply very quickly.

Forth uses a very simple and straightforward syntax rule in interpreting command lines. The command line consists of a sequence of words, separated by blanks or spaces. The words represent either commands pre-compiled in the Forth dictionary or numbers. Thus the Forth command line interpreter or text interpreter can be extremely simple, comparing to CLI's in other languages or operating systems. The interpreter just has to parse out words using blanks as delimiters, searches the dictionary to locate the executable code of the commands, and executes the code. If a word is not a command in the dictionary, the interpreter will try to convert it into a number and push the number on the stack. If the word is neither a command nor a number, it is beyond the capability of the computer to do anything about it, and the interpreter will send an error message to you protesting your mistake in a very mild manner.

The tool that provides the interpreter with the ability to parse out words from a command line, or an input stream of characters, is the Forth command WORD. Before we get into the details of WORD, a few other supporting commands have to be clarified.

10.2.   Input Stream and Input Buffers

First, what is an input stream? Where does the interpreter get the command lines? Forth interpreter can accept commands from two different sources: a console terminal or a disk. Two special areas in the computer memory are dedicated to store commands coming from these sources: a terminal input buffer or TIB for commands entered through the console, and one or more disk
buffers for commands coming from the disk. The terminal input buffer is managed by a number of variables and commands:

VARIABLE 'TIB
Contains the starting address of the terminal input buffer.

: TIB ( -- addr )
Return the address of the terminal input buffer.
'TIB @ ;

VARIABLE #TIB
Maximum number of characters that can be held in the terminal input buffer.

VARIABLE >IN
Pointer to the character currently being processed. It is an offset from the starting address of the input buffer, which is either the terminal input buffer or a disk buffer.

The disk buffers are managed by the virtual memory management in Forth. The details of this virtual memory system are discussed in a separate chapter. Here we are only concerned with the one disk buffer which is assigned to the interpreter so that the interpreter will get its commands from this buffer. The disk block number is store in a user variable:

VARIABLE BLK
Block number of source on disk to be interpreted.

The convention adopted by most Forth systems, including F83, is that if BLK contains a zero, the terminal input buffer is used for interpretation; otherwise, the disk block specified by BLK is used.

10.3. Low Level Parsing Commands

DEFER SOURCE
Vectored to (SOURCE). Return the starting address of the buffer used to hold current input stream.

: (SOURCE)
( -- addr len )
Return the string to be processed by the text interpreter. Addr is the beginning address of the input buffer and len is the length of the input buffer.
BLK @
Get the block number from BLK.
?DUP IF
If the block number is not zero,
BLOCK
fetch the block of commands from disk and return with the address of the disk buffer.
ELSE
If the block number is zero,
TIB
get the address of the terminal input buffer,
#TIB @
and the length of it.
THEN ;

Here are the hard stuff. Two code commands that scan the input stream to locate special characters in the stream.

LABEL DONE
A common returning point when the input stream is exhausted.

CX PUSH
Push the contents of CX on stack and return. CX register has the remaining length of the stream.

NEXT

CODE SKIP
( addr len char -- addr1 len1 )
Given the address and length of a string, and a character to look for, scan through the string while we continue to find the character. Leave the address of the mismatch and the length of the remaining string.
AX POP
Move char to AX register.
CX POP
Move len to CX register.
DONE JCXZ
If length of string is zero, jump to DONE and return.
DI POP
Move addr to DI register.
DX ES MOV
Set ES=DS for string manipulations.
REPZ BYTE SCAS
Repeatedly scan the string until we find a character different from that in AX.
0<> IF
CX now has the count of characters in the remaining string. If CX is not zero,
CODE SCAN

{ addr len char -- addr1 len1 }

Given the address and length of a string, run through the string until we find the character. Leave the address of the match and the length of the remaining string.

AX POP  CX POP
DONE JCXZ Same as SKIP.
DI POP
DS DX MOV DX ES MOV
CX BX MOV Set up looping parameters.
REP BYTE SCAS Repeat if character mismatches. Scan the string.
0= IF If the string is exhausted,
CX INC Backspace.
DI DEC
THEN
DI PUSH Restore string registers.
CX PUSH
NEXT END-CODE

SKIP is used to skip over the leading spaces in front of a word, because words can be separated by a number of spaces allowing source commands to be free-formatted. SCAN, on the other hand, will stop at the first match. Separating these two functions into two commands gives F83 much more versatility in handle strings than older versions of Forth like figForth.

: /STRING

{ addr len n -- addr1 len1 }

Index into the string by n characters. Return addr+n and len-n.

OVER MIN Change n to the smaller of n and len.
ROT OVER + Addr+n.
-ROT - Len-n.
;

: PLACE

{ from-addr len to-addr -- }

Move the characters at from-addr to to-addr. The final string has a preceding length byte of len.

2DUP C! Store the length byte.
1+ To-addr+1, address of the first character.
SWAP MOVE Copy the string.
;
10.4. High Level Parsing Commands

The real word parsing actions are embodied in the following two commands, which scan the input stream and parse out words with specified delimiting character.

: PARSE-WORD
  ( char -- addr len )
  Scan the input stream until char is encountered. Skip over leading chars. Update >IN pointer. Leave the address and length of the parsed word.
  >R
  Save char on return stack.
  >IN @ /STRING
  Get the current character pointer in >IN and modify addr and length accordingly.
  R> SKIP
  Skip over leading chars in the input stream starting at >IN.
  >R
  Save length of the remaining string.
  OVER SWAP R> SCAN
  Scan for the next occurrence of char.
  OVER - ROT
  Addr and length of parsed string.
  R>
  Retrieve the length of string.
  DUP 0<> + - >IN +!
  Update >IN to one character after the parsed word. However, if the parsed string is a null string, do not move >IN.

: PARSE
  ( char -- addr len )
  Do the same as PARSE-WORD without skipping the leading char.
  >R
  SOURCE >IN @ /STRING
  OVER SWAP R> SCAN
  OVER - DUP
  R> 0<> - >IN +!
  SCAN instead of SKIP.
  Len.
  Addr and length of parsed string.
  Update >IN to end of string.
Finally, we get to the most important command WORD, which parses the next word in the input buffer and copies the word to the word buffer for the text interpreter to do searching or number conversion. WORD will skip over leading delimiters so that words in the input stream can be spaced out to conform to various formatting conventions.

```
: WORD ( -- addr ) Leave on stack the address of the word buffer, which is on top of the dictionary.
     In F83 'WORD is the same as HERE. They might differ as indicated in 83-Standard.
     HERE ;
```

A blank character appended to the end of the parsed word in the word buffer is very important to the F83 system. It serves many important functions. One of them is for the number conversion routine to recognize the correct end of a number string. Another function is to help the text interpreter to detect the end of an input stream so that the text interpreter can prepare itself to process the next input stream or command line. For those familiar with the figForth system, there the end of an input stream is artificially terminated by one or more ASCII NUL characters. During console inputting, when a carriage return is received from the keyboard, the input routine appends a NUL at the end of the input stream. When using source texts in disk blocks, each disk buffer has two trailing NUL as the tail of the buffer. These artificial NULs force the interpreter...
loop to be terminated in a non-obvious and hard to document fashion. F83 tries to treat the end of line condition explicitly.

When WORD reaches the end of the input stream, the length of the parsed word will be zero. A string without character is then moved into the word buffer. The count byte is zero with a blank character appended to it. This null command, two bytes or one cell long, has a hex value of 2000. In the dictionary, there is a command of this name, whose hex value in the name field is A080. Masking off the MSB in these two bytes (the delimiters of the name field), the real name has a hex value of 2000, exactly the same as the parsed null command. The function of this null command is to turn on the end-of-buffer flag. Seeing that this flag is set, the text interpreter knows it has reached the end of the input stream and terminates the loop, readying itself for the next line of input.
Chapter 11. Text Interpreter

The source code of the text interpreter is in file KERNEL86.BLK, Screens 65 to 69.

11.1. The Operating System of Forth

The text interpreter is the heart of a Forth system. As a matter of fact, the text interpreter is 'the' operating system of Forth, if there is one in Forth. The text interpreter accepts input stream from console and extracts commands from the input stream. It looks up the commands in a dictionary and causes the system to perform the functions designed into these commands. After it successfully carries out the commands, it will come back to the console and ask for another line of commands. If all the commands designed into the dictionary have names similar to English commands commonly used, the Forth text interpreter makes a computer rather intelligent and easy to use, using a computer industry cliche, user friendly.

11.2. Entering the Text Interpreter

The functions of the text interpreter is best traced from the very beginning in the booting up of the Forth system to the point when a command line or input stream is accepted and processed. Instead of explaining all the low level commands first and build up layers of high level commands to reach the top level of the text interpreter, let’s try this top-down approach: explaining the functions of the high level commands and then detailing the functions of the modules invoked in the high level command. The most logical command to start is ABORT, which is the starting point of the Forth system and also the point of return whenever an error condition is encountered.

```forth
DEFER ABORT Vectored to (ABORT). Re-initialize all the Forth registers and start the text interpreter afresh.
: (ABORT) ( -- ) Unconditional abort routine.
SP0 @ Get the initial data stack pointer from the user variable SP0.
SP! Stuff that pointer into the data stack pointer register of the virtual Forth computer.
QUIT Jump to QUIT routine which is the point of return for normal forced termination of execution.
;
: WARM ( -- ) Perform a warm start.
TRUE Force an abort.
ABORT* Warm Start* Abort with a message.
;
: COLD ( -- ) High level cold start.
BOOT Execute a user defined bootup definition.
QUIT Jump to QUIT, a normal re-start point.
;
DEFER BOOT Vectored to a user selected initializing routine. The default boot routine is ABORT.
: QUIT ( -- ) The main Forth loop. Get more input from the console terminal and interpret it.
```
SP0 @ 'TIB! Initialize the terminal input buffer to address just above the data stack.
BLK OFF Store a zero in BLK. Force the interpreter to process input from the console terminal.

[COMPILE] { Store a zero in the variable STATE, forcing the system into the interpretive mode.
BEGIN Enter the main Forth loop.
RP0 @ RP! Initialize the return stack pointer.
STATE @ NOT Indicate status of the system. A defer word normally vectored to CR, doing a carriage return.
STATUS Query prompt the user to enter a line of commands on the console and copy this command line to the terminal input buffer.
RUN Process the command line.

STATE @ NOT If STATE is zero, the system is in the interpretive mode.
IF ." ok" THEN Then print the ok message.
AGAIN The Forth loop is an infinite loop. After one command line is processed, it goes back to ask for another line. It goes on this way forever.

; : RUN ( -- ) An enhanced INTERPRET. It allows for multiline compilation, enabling you to enter a colon definition that spans over several lines.
STATE @ IF If STATE is not zero, the system must be in the compiling dictionary.
STATE @ NOT After compiling one line of source codes, test STATE again.
IF INTERPRET THEN If the system left the compiling mode, then interpret the rest of the line. Otherwise, exit.
ELSE The state is zero,
INTERPRET interpret the command line.
THEN ;

11.3. INTERPRET

INTERPRET is a beautiful piece of code, a classic example of the simplicity and powerfulness of Forth language in describing complicated computational processes using high level commands. It is worthy of our time to read the code and do our best to gain the fullest understanding of it. The definition of INTERPRET reads:

: INTERPRET ( -- ) The Forth interpreter loop. It parses out a word from the input stream. If the word is defined execute it, otherwise convert it to a number and push it on the stack.
BEGIN Begin the interpret loop.
?STACK Check for stack underflow or overflow.
DEFINED Get the next word from the input stream and return its cfa and a flag.
IF EXECUTE If the word is defined, execute it using the cfa left on stack.
ELSE NUMBER Otherwise, convert it to a number.
DOUBLE? Is it a double precision integer?
NOT IF DROP THEN No. Only a single precision number. Drop the upper half of the double number, preserving only the lower half single integer.
THEN FALSE Put up a false flag for DONE?.
DONE? Is it the end of line?
UNTIL If we reach end of line here, exit the loop. Otherwise, loop back to interpret the next word.

; DEFINED is a very big command. It first parses a word out of the input stream and places it in the word buffer on the top of the dictionary. It then searches through the dictionary for a command with the same name. If a command is found, its code field address is placed on the data stack followed by a true flag. A valid code field address is then turned over to EXECUTE. EXECUTE executes this command by invoking the appropriate inner interpreter, which we had discussed in the chapter on inner interpreters. DEFINED is discussed in the chapter on vocabulary.
Now, if DEFINED failed to find a command with a matching name, control is passed to NUMBER, which converts the parsed word to a double precision number on the data stack. If a period was embedded in the number string, which causes DPL to differ from -1, the command DOUBLE? returns a true flag and the double number remains on the stack. Otherwise, the higher half of the double number is dropped from the stack and only a single precision number is left on the stack.

At the beginning of the loop, the data stack is checked for overflow or underflow by ?STACK. If the stack is ok, control falls into DEFINED to process the next word in the input stream. If a stack error condition is encountered, the system is forced into ABORT to start all over again. If an error condition is encountered during the number conversion process, an abort is also forced. These are the two conditions for abnormal exit from the INTERPRET loop.

11.4. DONE? and X

At the end of the INTERPRET loop, DONE? is executed to test the end-of-buffer condition. If it reached the end of the input buffer, the loop would be terminated and the control falls into the outer Forth loop in QUIT. Otherwise, the interpreter will loop back to parse and execute the next word in the input buffer.

A flow chart of this chain of activities might be helpful in visualizing the sequence of events when the Forth system is cranking in full steam, as shown in Fig. 12.1.

A number of loose ends have to be patched before we finish this chapter.

: ?STACK ( -- ) Check for data stack underflow or overflow. Abort if any of the error conditions occurred.
  SP@ Get the current data stack pointer.
  SP0 @ SWAP U< If the stack underflowed,
    ABORT" Stack Underflow"
    ABORT.
  SP@ PAD U< If the stack grows too close to the top of the dictionary,
    ABORT" Stack Overflow"
    Abort also.
  ; Otherwise, return normally.

: DONE? ( n -- f ) Return a true flag if the inputstream is exhausted or the STATE doesn't match with the current state.
  STATE @ <> Is the state flag left on stack he same as that in STATE?
  END? @ OR Or the end of line? Leave the or'ed flag on stack.
  END? OFF Turn off the end-of-buffer flag to let the interpreter get a new line of command and start over.
  ;

In F83 systems before Version 2.0, the end-of-buffer condition is detected and the END? flag is set by a command with a null string as its name. This null command was defined using a pseudo name of X and later patched to null. When the input stream is exhausted, the last word parsed out by WORD is this null command and it tells the text interpreter to stop processing the input buffer.
This technique had been used in most Forth systems including figForth. In F83 Version 2.0 and later, the end-of-buffer condition is detected and the END? flag is set in the word FIND; therefore, this mysterious null command is eliminated and the text interpreter is in much better shape. The discussion on the null command X is included here for completeness and for users with older versions of F83.

: X ( -- ) The null word to flag end-of-buffer and to terminate the interpreter loop.
END? ON Turn on the end-of-buffer flag in the user variable END?.
;
HEX A080 LAST @ !
IMMEDIATE DECIMAL

The real name of X in the Forth dictionary is a null string, with a character count of 0 and a blank character. This null string is returned by WORD to the word buffer when the end of the input stream is reached. The contents of the name field of this null command is A080 in hex, with the MSB's in both bytes set as name field delimiters. As we reach the end of the input stream, this null command is returned by WORD and executed. It turns on the END? flag and terminates the interpret loop. Explicitly terminating the interpret loop at the end of input stream makes the definition of INTERPRET comprehensible. Another advantage is that the end-of-buffer condition does not have to be artificially synthesized by appending a NUL character at the end of the input line form the console or at the end of every disk buffer (as done in figForth), which can be easily corrupted and causes the Forth system to behave erratically.
Chapter 12. Compiler

The source code discussed here is in the file KERNEL86.BLK, Screens 70, and 76 to 78.

12.1. The Colon Definitions

Colon commands are the most prevailing type of commands in Forth. A colon command has a variable length parameter field where a list of execution addresses is stored. Functionally, a colon command is the equivalence of the sequence of commands whose execution addresses are stored in its parameter field. When the colon command is invoked, this sequence of commands are executed by the address interpreter. Comparing to other high level languages, a Forth colon command is similar to a procedure or a subroutine, which contains a sequence of procedures or subroutine calls:

```
Forth Colon Definition      FORTRAN Subroutine
: Z                      SUBROUTINE Z
A CALL A
B CALL B
C CALL C
D CALL D
; RETURN
```

where A, B, C, and D are other pre-defined commands in Forth or subroutines in FORTRAN.

Colon commands allow us to build higher level functions from existing modules. The building process can continue on until the final colon command becomes the solution to our programming problem.

What, then, is the advantage of colon commands over the procedures or subroutines in other languages, since they serve very much the same purposes? The answer is that although a Forth colon command is the same as a procedure or a subroutine in functionality, it serves the functions more efficiently and it can be debugged more easily. The result is a solution or a program of much higher quality at lower cost. We can summarize the advantages of Forth colon commands in two words: efficiency and modularity.

Efficiency

Efficiency in computer programming has three aspects: memory utilization, execution speed, and programming productivity. Forth colon commands excel in all these aspects as compared to other high level languages. Each reference to another pre-compiled command in Forth costs two bytes in memory, the execution address of the referred command. The calling of a command and
returning to the caller in Forth is also very fast due to the efficiency in the inner interpreters, especially in hosts of good architecture design. In executing high level language programs, by far the largest overhead is doing subroutine calls and returns, with a host of parameters to be passed between the caller and the callee. Since Forth uses the data stack to pass all the parameters between commands, the overhead in parameter passing is cut to the minimum. Ease in testing and debugging greatly improve the productivity of programmers using Forth as software development tool.

**Modularity**

Forth commands are true modules because they are memory resident and individually executable routines. Once a command is defined and compiled into the dictionary, it is immediately available for execution and for compiling into other commands. In other languages, procedures and subroutines are modules only in the abstract sense. They have to be compiled and linked to a mainline program before they can be invoked to do any useful work, within the context of the mainline program. In the example above, the commands A, B, C, D, and Z are all executable modules in Forth. In FORTRAN, none of the subroutines A, B, C, D, and Z are executable. They are only modules on paper.

Why is true modularity so important? It greatly simplifies the testing and debugging of a program of large size, because individual modules can be thoroughly tested before being integrated into modules at a higher level of construction. In debugging a conventional program, the most valuable tool is the break point facility, allowing you to stop the program at selected break points to examine the progress of operations. In Forth, each command can be tested at the interpreter level with a natural break point at its end, eliminating the need of a debugger.

Due to the small overhead in nesting commands, Forth encourages the breaking of large modules into many small modules which can be tested thoroughly and separately. This modularization gives us a chance to prove the correctness of a large program by proving the correctness of each component and the correctness of their interconnections.

**12.2. Colon and Semicolon**

So much for propaganda. Let's now look at how the colon compiler itself is defined.

```
: : ( -- ) Define a colon definition. The new definition is hidden until it is completed.
The runtime code for : adds a nesting level.
!CSP Store the current stack pointer in a variable CSP for error checking at the end
of a definition. Normal compilation should not affect the depth of the data stack.
I stack depth is changed, it is a potential error condition.
CURRENT @ CONTEXT ! Select the current vocabulary as the context vocabulary to restore the environment
of compilation.
```
CREATE Create a header in the dictionary using the name following .
HIDE Smudge the name field of the new header so it is hidden from dictionary searches.
] Enter the colon definition compiler to start constructing the list of execution addresses in the parameter field.
;USES Insert the following code routine address into the code field of the new definition, making it a colon definition.
NEST , Compile the address of the address interpreter NEST here so that it can be put into the code field of new colon definitions.

; ( -- ) Terminate a colon definition. It compiles the runtime code of UNNEST to remove a nesting level and changes STATE to terminate compilation.
?CSP Check the current stack pointer with the contents of CSP. If they are not the same, abort.
COMPILE UNNEST Compile UNNEST at the end of the new colon definition to force execution to return to the caller.
REVEAL Unsmudge the name field of the new colon definition, making it available for dictionary searches.
[COMPILE] { Compile [ here to terminate the compilation of the new colon definition. It must be declared immediate.
; IMMEDIATE ; must be executed in the compiling state. It must be declared immediate.

The compilation process is very similar to the interpreting process in Forth. Instead of executing the word parsed out of the input stream, the execution address of the command is added to the top of the dictionary, where we are building the parameter field of a new colon command. If the word is a number, instead of leaving the number on the data stack, it is compiled into the new command as a literal so that when the new command is eventually executed, the same number can be retrieved and put back on the stack. The compiler is embodied in the command ], which is the twin brother of INTERPRET, because they share lots of common tools and structure. In figForth and many older Forth systems, the compiling functions are actually rolled into INTERPRET as one single piece of command. The good doctors in the Forth Standard Team decided that it is unsightly that the Siamese twin should share their umbilical cord forever, and cut them loose. The compiling functions are then welded into ]. F83 has no choice but to follow the doctor's order.

12.3. The Compiler Loop

; ( -- ) The compiling loop. It sets the compiling flag in STATE, and parses the next word out of the input stream. It is either executed or compiled depending on whether it is immediate or not. If it is a number, it is compiled into the dictionary as either a single or a double integer literal. Continue until the input stream is exhausted.
STATE ON Set the flag in STATE and enter the compiling mode.
BEGIN Loop to scan the input stream.
?STACK Check for stack over- or underflow.
DEFINED Parse out the next word and search the dictionary.
DUP IF If it is found in the dictionary,
0> IF and if it is an immediate word,
EXECUTE then execute it.
ELSE If it is not an immediate word, compile its cfa into the dictionary.
THEN ELSE It is not a word in the dictionary.
DROP Discard the flag left by DEFINED.
NUMBER Convert the word to a number.
DOUBLE? IF If a punctuation is detected in the string,
[COMPILE] DLITERAL compile the double integer literal.
ELSE No punctuation in the string.
DROP [COMPILE] LITERAL and compile the single integer literal.
THEN THEN TRUE The compiling flag.
DONE? If the input stream is exhausted, leave a true flag to exit the compiling loop.
UNTIL Otherwise, loop back to compile the next word.

Figure 12.1 The interpreter and the compiler

\begin{verbatim}
: ( -- ) Stop compiling and start interpreting.
STATE OFF Turn off the compiling flag in STATE, forcing the Forth system into the
  interpreting mode.
; IMMEDIATE It is declared immediate so that its effect can be revealed even during
  compilation.
\end{verbatim}
12.4. Low Level Supporting Commands

We have presented the compiler at the highest level. There are a long list of supporting commands behind these compiler commands to realize all the functions required in the compilation processes. Let's try to give recognition to all these unsung heroes.

VARIABLE DP
The user variable where the address of the first free memory above the dictionary is stored. It helps the compiler to keep track of its memory.

: HERE { -- addr } Return the address above the dictionary, the free memory available for the compiler.
    DP @ ;

: 'WORD { -- addr } Return the address of the word buffer, same as HERE.
    HERE ;

: ALLOT { n -- } Allocate more space on the dictionary by moving the DP pointer.
    DP +! ;

: , { n -- } Copy the top stack item to the top of the dictionary. This is the compiler in its most primitive form.
    HERE ! Compile n to dictionary.
    2 ALLOT

: C, { byte -- } Compile one byte to the dictionary.
    HERE C! Compile one byte.
    1 ALLOT Move DP.

: COMPILE { -- } Compile the next word in a colon definition to the dictionary when this definition is executed. It can only be used inside a colon definition.
    R> The address of the next word is on the top of the return stack. Retrieve it.
    DUP 2+ >R Increment the top of return stack so that the next word will not be executed.
    @ Get the execution address of the next word.
    , Compile it to the dictionary.

: IMMEDIATE { -- } Mark the most recently defined word to make it an immediate word. An immediate word will not be compiled by the ] compiler but will be executed.
    64 The precedence bit in the first byte of the name field.
    LAST @ Get the name field address of the most recently defined word from the variable LAST.
    CTOGGLE Set the precedence bit in the name field, marking the word immediate.

12.5. Immediate Commands

Two good examples of immediate commands are LITERAL and DLITERAL in ]. They are used to compile literal numbers in a colon command. They are needed because the address interpreter treats the data stored in a colon command as execution addresses. If we need to put a number on the stack between two addresses, we cannot simply compile the number in-line, because then the number will be interpreted as an address. In-line literal numbers in a colon command must be preceded by a special runtime command (LIT), which will push the following literal to the stack when executed. To compile a number into a colon command, LITERAL is executed immediately to compile first (LIT) and then the number, building the correct literal structure in the colon command.
Words in a colon definition are normally compiled. Immediate commands are not compiled but executed immediately. To compile an immediate command like other commands in a colon command, the immediate command must be preceded by [COMPILE]. To compile a command only when the command is executed, the command is preceded by COMPILE. It is rather confusing for a new comer to Forth. But, you have to remember, building compiler is not an easy task for everybody. These commands encompass activities in the designing of a compiler to build commands which will have the correct behavior at runtime. You will have to go to a graduate school of computer sciences to hear these topics discussed only peripherally. To understand these concepts, you have to read more code in the Forth compiler and let it gradually sink in. Or, try to write a few compiler routines yourself and see how they function.

I can offer you one more hint: immediate commands are the equivalents of the compiler directives or assembly directives in the conventional programming languages. They may or may not generate executable code in the program, but they control the process of compilation or assembly and they are executed during the compilation or assembly, but not when the final codes are executed. We will discuss more of these immediate commands in the following chapter.

[COMPILE] cannot wait to let the ] compiler to find the address it needs, because ] might have to execute the next command. [COMPILE] is executed inside the compiler loop and it has to find the address of the next command immediately to compile it. Therefore, [COMPILE] uses a special dictionary searching command ' (tick) to do the dictionary search:

```
: ' ( -- cfa ) Return the execution address of the next word. If the word cannot be found, abort.
    DEFINED Parse the next word and search the dictionary for it.
    0= If the search failed,
      ?MISSING Abort with an appropriate message.
    ;
    ?MISSING ( f -- ) Tell the user the word does not exist and abort.
      IF 'WORD COUNT TYPE Type the word failed to match.
      TRUE ABORT* ?"
      THEN Abort with a very mild message.
      ELSE Return if the flag is false.
    
```
Chapter 13. Structures in Colon Definitions

The source code discussed in this chapter is in KERNEL86.BLK, Screens 70-71, and 74-75.

13.1. Compiler Directives

We have discussed in great detail the contents and the functions of the colon command compiler which compiles colon commands, and the address interpreter which executes colon command as a list of execution addresses. If that is all, the usefulness of colon commands is severely limited, as they will not be able to cope with the wide variety of situations a programmer must solve using his computer. Very few problems can be solved by linearly strung procedures or commands. We need the capability of altering the execution sequence on the fly, depending upon the results obtained in runtime. We need the capability to compile and use different types of data and data structures, which are used to encode input/output information and to hold intermediate information during processing. Compiler directives are used to allow you to specify explicitly alternate or repetitive execution sequence and compile special data structures inside a colon command. Compiler directives are also called immediate commands because they have to be executed immediately during compilation so that special structures can be built inside a colon command. Immediate commands can be distinguished from normal commands by the fact that a bit, the precedence bit, in the first byte of the name field is set.

The compiler loop \( \) can compile normal, non-immediate commands and single or double integer literals. However, it incorporates an extremely powerful hook to take care of any special compiling conditions in the form of immediate commands. Whenever we have a situation that the compiler \( \) is not able to handle, we will design an immediate command to do whatever is necessary to take care of the situation and then let the compiler \( \) continue its normal compilation.

A few examples were shown in the chapter on the colon compiler. In fact, literals are handled this way. When the compiler fails to locate a command in the dictionary, it converts the word into a number and asks LITERAL or DLITERAL, two immediate commands, to compile the numbers into the dictionary in the form of two data types, single integer literal or double integer literal. This way, numbers can be compiled into colon commands, in-line with the execution addresses which are the default data type in colon commands.

There are other data types and different methods of interpreting them within the context of a colon command. F83 is very rich in these special commands, for the convenience of you the user. Let’s look at them closely.
13.2. Compiling Numeric Data Structures

Two data types were taken care of: the single integer literal and the double integer literal. The immediate commands which compile them are LITERAL and DLITERAL. The runtime commands which interprets them, pushing the number on the data stack, is (LIT).

Two immediate commands are provided to compile ASCII codes. They also use (LIT) to interpret the compiled character literals:

```forth
: ASCII ( -- char ) Compile the next character in the input stream as an ASCII character literal.
  BL WORD Parse out the next character.
  1+ C@ Get the ASCII code of this character from the word buffer.
  STATE @ Are we in the compiling state?
  IF [COMPILE] LITERAL Yes. Compile the character as a single integer literal. However, technically it is a character literal.
  THEN If interpreting, just leave the character on stack.
  ; IMMEDIATE

: CONTROL ( -- char ) Compile the next character in the input stream as a control character literal. The character must be upper case.
  BL WORD Get the next character.
  1+ C@ Get its ASCII code.
  ASCII @ Offset between the control character and the upper case character.
  - Control ASCII code.
  STATE @ If compiling,
  IF [COMPILE] LITERAL Compile the control code as a literal.
  THEN Leave the character on stack if interpreting.
  ; IMMEDIATE
```

We can always lookup the ASCII table and use the character codes directly in colon commands. ASCII and CONTROL, however, make very clear documentation to the intention of the programmer. Using these commands to invoke ASCII codes explicitly is highly recommended.

Ever heard of address literals? Well, there are really such things. Its usefulness has been demonstrated in many applications in which we want to locate a command in the dictionary in runtime. An example is to find the address of a colon command so that we can jump into the middle of it. The reason of doing so is not obvious and certainly is not orthodox Forth practice. Anyway, if you need the address of another command inside a colon command, the command ['] is the one to use.

```forth
: ['] ( -- ) Compile the address of the next word as a literal. At runtime, return that address to the stack.
  ' ( tick ) Find the execution address of the next word in the input stream.
  [COMPILE] LITERAL Compile the address as a literal.
  ; IMMEDIATE
```
13.3. Compiling String Literals

String literals are very useful data type. They can be used to compile messages in a colon command. At runtime, the message will be typed out on the console, creating a friendly environment for the end users.

```
: (") ( -- addr len ) Return the address and the length of an in-line string.
  R> Address of the in-line string compiled immediately after (").
  COUNT Get the addr and len of the string.
  2DUP + The address of the executable code after the string.
  EVEN Align to cell boundary.
  >R Replace it on the return stack to continue the execution process.

: (.") ( -- ) Type out the in-line string and continue executing the word after the string.
  R> Address of the in-line string.
  COUNT Addr and len.
  2DUP + EVEN >R Replace the address of the next word to be executed.
  TYPE Output the string to console.

: ;;" ( -- ) Compile the following string to the dictionary.
  ASCII " Use " as the delimiter of the string.
  PARSE Parse the string out.
  TUCK 'WORD PLACE Copy the string into the word buffer, just the right place to compile this string.
  1+ ALLOT ALIGN All we have to do is to move the DP pointer to include the string in the dictionary.
```

Figure 13.1 Numeric data structures.
Figure 13.2   The string literals.

: ." (--)
  Compile the following string to be typed out later.

  COMPIL(."
  Compile the runtime code (."") before the string so that the string will be
interpreted correctly.

  ; IMMEDIATE
  This is a compiler directive. Declare it to be immediate.

: " (--)
  Compile the string. At runtime, return its address and length.

  COMPIL(")
  Compile the runtime routine (").

  ; IMMEDIATE
  Must be immediate.
An important command also using string literals is the command ABORT". It forces the Forth system to return to the text interpreter with a clean state to start over again. It can also print out a message explaining why it has to take such a drastic measure to help you figure out what happened in the computer at run time.

```
: (ABORT") ( f -- )    The runtime routine compiled by ABORT".
R@ COUNT              Get the addr and len of the following string literal.
ROT                   Move the flag to the top of stack.
?ERROR                Turn over to ?ERROR to process the error condition.
R> COUNT + EVEN >R    Move the top of return stack to the word after the string, to resume execution

;                         as the error condition was not true.

: ABORT" ( f -- )       If the flag is true, issue an error message and quit.
COMPILER (ABORT")       Compile runtime routine.
."                        Compile the message.
;                         Immediate

DEFER ?ERROR             Vectored to (?ERROR).

: (?ERROR) ( addr len f  -- ) If the flag is true, execute WHERE to store useful debugging data, type a message, and quit.
IF                        If the flag is true, prepare to quit.
>R >R                     Save the string parameters.
SP0 @ SP!                 Initialize the data stack.
PRINTING OFF              Turn off the printer.
BLK @ IF                  If BLK is not zero, we are processing data from a disk block.
>IN @ BLK @ WHERE         Save the character pointer to the input buffer and the block number and call WHERE to provide debugging aids.
THEN
R> R>                     Restore the string parameters.
SPACE TYPE SPACE          Print the abort message.
QUIT                      Restart the text interpreter.
ELSE                      No error condition.
2DROP                     Clear the data stack.
THEN
DEFER WHERE               WHERE is vectored to an editor routine (WHERE) to display the block of source with the cursor pointing to the word that causes the abort.

DEEFER WHERE
```

There are other data structures that can be compiled into the colon commands. However, many of them can be taken care of by variables and arrays derived from variables. Other recurring structures may be handled by the CREATE-DOES> technique.

### 13.4. Compiling Control Structures

Forth is a structured language. A structured language has provisions for you to do two things: successive refinement to decompose a problem into smaller parts hierarchically, and building modules with control structures. Control structures, or simply structures, are segments of a program or groups of program statements which have only one entry and one exit. The one-entry-one-exit property of control structures allows the structures to be stacked linearly to form larger segments which can be built into other structures at a higher level. Execution can take alternate paths or repeat a portion of the path only within a structure. Very complicated high level structures can be built on simple structures, enabling programmers to deal with real life problems efficiently.
In a previous chapter, I emphasized that Forth is a truly modular language because the commands in Forth are true modules, which can be independently executed and compiled, quite different from modules in other languages which can function only within the context of a mainline program. Forth commands are also structures, with one entry and one exit. There are some exceptions when error conditions are encountered. In these cases, execution is forced to abort to the text interpreter. Forth commands, as structures, can be stacked linearly together to form higher level structures, which are basically the colon commands. Besides linearly stacked structures, Forth provides a special set of commands which allows you to build other more sophisticated control structures inside colon commands so that alternate paths can be chosen and segments can be repeated in runtime. These structure building commands are all immediate commands, because they have to perform extra work to build the desired structures correctly while the compiler is running.
The set of structure building commands in F83 are listed here according to the syntax of their usages:

IF <true clause> THEN
IF <true clause> ELSE <false clause> THEN
BEGIN <repeat clause> UNTIL
BEGIN <repeat clause> AGAIN
BEGIN <repeat clause 1> WHILE <repeat clause 2> REPEAT
DO <repeat clause> LOOP
DO <repeat clause> +LOOP
?DO <repeat clause> LOOP
?DO <repeat clause> +LOOP

Figure 13.3. The control structures
Inside the do-loops, the optional commands LEAVE and ?LEAVE can be used to force the termination of the loop.

13.5. Address Calculation for Control Structures

In the chapter on the kernel commands, we have already discussed the low level commands which change the execution sequence in runtime. What the structure building commands have to do is to compile these runtime routines into the colon command with additional branching addresses so that the execution sequence in runtime can be changed according to pre-defined rules. Thus a group of supporting commands are needed to calculate the branching addresses during compilation.

Error checking is a valuable service to you to make sure that you have laid down the control structures correctly. Structure commands not properly paired are frequent causes of system crashes, because execution can be steered to an unknown address.

13.6. Control Structure Compiler Directives

Here come the real heroes that compile the control structures in colon commands:
These structure commands look very simple and indeed they are. All they have to do is to pick and compile the right runtime routine and resolve the branching addresses. The runtime routines
know what to do with the branching addresses and change the execution sequence if necessary. These branching addresses can be considered as special address literals, different from the normal execution addresses compiled by the J compiler.

As it is evident in the definitions of these control structure commands, these commands must used in pairs, and they can be considered as the delimiters for structures in the colon command, clearly indicating the entry points and the exit points of the structures. IF must be followed by THEN. DO must be paired with either LOOP or +LOOP. BEGIN must be paired with UNTIL, AGAIN, or REPEAT. Structures can be nested but can not be overlapped. If the structures are overlapping, the system will behave erratically if not crashed.

The error checking in compiling the structures in F83 is not as extensive as that in the figForth model, in which different types of structures are assigned different error checking numbers instead of a true-false flag. figForth prohibits the compiling of improperly nested structures. Nevertheless, F83 is better than those earlier Forth system without any error checking on the control structures. If you want speed in compilation, you can strip out the error checking in F83 by using >MARK in place of ?>MARK, etc., and change all the 2DUP to DUP. Then you are entirely on your own.
Chapter 14. The MS-DOS Files

The source code managing files in the F83 system is scattered in Screens 51 and 57 in KERNEL86.BLK and also in Screens 7 to 12 in EXTEND86.BLK. Some of them were discussed in the chapter on the virtual memory.

14.1. CP/M-DOS File Primitive Commands

The DOS file management system consists of a set of commands in the DOS vocabulary that access the BDOS functions of the CP/M-DOS operating system, such as creating, opening, and deleting files. There is also a command that parses a string and creates a file control block (FCB). A very useful command SAVE is also provided to save the contents of memory as an executable DOS file. A number of commands were also defined in the basic F83 system which are used to access the default file defined by the FCB1 control block.

VOCABULARY DOS
All the DOS words are put in this vocabulary. For CP/M systems, its name is CP/M, of course.

DOS DEFINITIONS
Make DOS the current vocabulary so that all subsequent words will be added to this vocabulary.

CREATE FCB1 B/FCB ALLOT
Allocate space for the first FCB block of the current file.

CREATE FCB2 B/FCB ALLOT
Allocate space for the second FCB block of the in-file.

: CLR-FCB ( fcb -- ) Initialize the specified FCB.
DUP B/FCB ERASE
Clear the FCB to nulls.
1+ 11 BLANK
Initialize the file name and extension to blanks.

The following commands are simply BDOS functions with Forth names. Descriptive names make the Forth programs or commands more readable.

: RESET ( -- ) 0 13 BDOS DROP ;

: CLOSE ( fcb -- ) Close the given file and report errors.
16 BDOS
Call BDOS to close the file.
DOS-ERR?
If there is error, report it.
ABORT" Close error"

: SEARCH0 ( fcb -- n )
17 BDOS ;

: SEARCH ( fcb -- n )
18 BDOS ;

: DELETE ( fcb -- n )
19 BDOS ;
14.2. The File Control Block

The file control block FCB is a table containing essential information so that the DOS system can manage the file in association with the blocks. The next two commands build FCB blocks which is almost all that is needed to create files and gain access to them using the above commands.

: !FCB { FCB-addr -- } Use the following string as the file name string and create an FCB for it. If CAPS is false, allow lower case file names.
   BL WORD Parse out the next string and place it in the word buffer.
   COUNT Get the string length from the word buffer address left by WORD.
   CAPS @ IF If CAPS is true,
   2DUP UPPER convert the string to upper case.
   THEN Otherwise, allow lower case string.
   ROT Get the FCB address to top of stack for (!FCB).
   {!FCB} Now, get {!FCB} to fill the FCB with the name string in the word buffer.

: SELECT { drive -- } Make the given drive the default drive.
14.3. High Level File Commands

The following commands are defined in the basic F83 system as shown in KERNEL86.BLK file, screen 57. However, their functions make them a natural part of this chapter on the MS-DOS files. One of the problems in reading Forth source code is that the order in loading the Forth source codes does not necessarily bear any relationship with the logical order of commands. In this book, I hope that grouping commands together according to their functionalities will help you to perceive more clearly the logical structures in the F83 system.

: FILE-SIZE ( fcb -- n ) Return the size of the current file in number of records.
35 BDOS DROP
RECORD# @ Get the file size.

: DOS-ERR? ( -- f ) Return a true flag if the previous DOS operation is in error.
255 = BDOS returns 255 if an error occurred.

: OPEN-FILE ( -- ) Open the current file and store the size of this file in MAXREC#.
IN-FILE @ 15 BDOS Open the in-file.
DOS-ERR? IF Is there an error?
" Open error"
DISK-ABORT THEN If so, abort.
DUP FILE-SIZE Otherwise, size the file.
1- SWAP Number of the last record in file.
MAXREC# ! Save it.

92 CONSTANT DOS-FCB The zero page address where DOS puts a parsed FCB.

: CREATE-FILE ( n -- ) Create a new file and allocate n blocks to this file.
FCB2 DUP !FILES Set the file pointers in both the current file and in-file to point to FCB2.
DUP !FCB Build a FCB at FCB2 and make it the current file. The file name is taken from the input stream.
MAKE-FILE Call BDOS to make the file.
MORE Allocate the require blocks.

;
14.4. Save Core Image to a File

A very special usage of the file commands is to save the entire core image in a file which can be called for execution from DOS. This will save lots of compiling time to load in many blocks of utilities. It is also a good way to build an application program without giving the user all the Forth source code, a good way to protect your software product.

```
DEFER HEADER
' NOOP IS HEADER HEADER is used in the DOS system.

: SAVE { addr len -- } Use the name following as the file name and create an executable DOS file. Memory from addr to addr+len is saved into this file. The current file is not disturbed.
  FCB2 DUP !FCB Build a new FCB at FCB2, using the name following SAVE.
  DUP DELETE DROP If this file already exists, delete it.
  DUP MAKE-FILE Create a new file.
  HEADER Build an executable header.
  -ROT BOUNDS DO Scan the given range of memory.
  I SET-DMA Specify memory address for DMA transfer.
  DUP WRITE Write one record of 128 bytes.
  128 +LOOP Increment the index of length for next record.
  CLOSE Close the file.

: SAVE-SYSTEM { -- } The high level command to save the code image to a file. You do not have to remember the dictionary addresses.
  256 Starting memory address of the Forth dictionary.
  HERE End of dictionary.
  SAVE Make the executable file.
```

14.5. Directory Accessing

F83 can access the DOS directory on a disk directly without having to leave the Forth environment. They are convenience that makes you feel at home and eliminate the necessity of learning the DOS system and fight against it.

```
: .NAME { n -- } Print the name of the nth entry in the DOS directory.
  #OUT @ Get the current output character count in #OUT.
  C/L > If it exceeds the line length,
  IF CR THEN send a CR to start a new line.
  32 * PAD + 1+ The address of the nth entry, already copied to the PAD buffer.
  8 2DUP TYPE SPACE Print the file name.
  + 3 TYPE 3 SPACES Print the extension.

: DIR { -- } Print the DOS directory.
  [ DOS ] Switch context to CP/M vocabulary.
  " ????????.??" Put a file name template in PAD.
  FCB2 (FCB) Create a new FCB with the ? marks in its name and extension fields.
  CR PAD SET-DMA Fetch the directory information to PAD.
  SEARCH0 Search for the first directory entry that matches the ? mark name. Any valid file name would do. The stack item returned is the entry number of the file in PAD, just right for NAME.
  BEGIN Scan the entire directory.
  .NAME Print the file name and extension.
  SEARCH Search the next matching file name, i.e., the next file name
  DUP DOS-ERR? End of the directory?
  UNTIL If any error flag is returned, we have reached the end of the directory. Exit now. Otherwise, loop back to print the next file name.
  DROP Drop off the invalid entry number.
```
FILE ( addr -- ) Given the address of an FCB, print the name of this file.
COUNT ?DUP IF If the drive number is not zero,
ASCII @ + EMIT then print the drive prompt.
THEN .TRAILING TYPE Print the file name without the trailing blanks.
+ The address of the extension field.
"." " Print a period sign between name and extension.
3 TYPE SPACE Print the extension.
;
FILE? ( -- ) Print the name of the current file.
FILE @ Get the FCB of current file.
FILE Print its name.
;

F83 allows you to have two files opened at the same time: a current file and an in-file. The in-file is used for input and the current file is used for output. The command SWITCH can be used to switch these two files so you can input from the previous current file and output to the previous in-file.

SWITCH ( -- ) Exchange the current file and the in-file.
FILE @ IN-FILE @ Two fcbs.
FILE ! IN-FILE ! Exchange the fcbs addresses.
;
!FILES ( fcb -- ) Set both the current file and the in-file to the given fcb.
DUP FILE ! Set current file.
IN-FILE ! Set in-file.
;

14.6. System Level File Commands

The utility commands defined above allow the F83 system to manage DOS files and the associated facility. As a user, you will probably have no need for them unless you have to dig down into the system level. To use the file management system, you need only a few commands at the top Forth level to create files and to gain access to their contents. This section describes these commands and their functions.

FILE: ( -- fcbl Use the following string as the file name and create a new file. The address of the FCB is returned on the stack.
>IN @ Save the input character pointer because we will use the next name more than once.
CREATR Create a Forth word using the following name. When this word is referenced, the file of the same name in DOS will be opened and made the current file.
>IN ! Restore the input character pointer to the front of the file name.
HERE DUP The parameter field address of the file definition.
B/FCB ALLOT Put the FCB in the parameter field.
!FCB Now stuff the FCB with the new file name.
DOES> Now comes the execution part of the file definition.
>FILES Initialize both the current and the in-file.
;
?DEFINE ( -- fcbl Define the next word as a file if it is not already define. Leave its FCB address on stack.
>IN @ Save the input character pointer.
DEFINED Search the dictionary for the next word, which is supposedly a file name.
IF NIP >BODY If the file definition is in the dictionary, discard the character pointer because we will not need it. The cfa returned by DEFINEE is then changed to pfa which is the FCB of the defined file.
ELSE No. The file was not defined.
DROP Throw away the word buffer address.
>IN !  Restore the character pointer to the front of the file name.
FILE:  Define a new file with a new file definition in the dictionary.
THEN ;

FORTH DEFINITIONS

All the file management words were put into the DOS vocabulary, which are not accessible from Forth. The two most used words concerning files are to be defined in the FORTH vocabulary so that they can be accessed conveniently.

: OPEN ( -- )  Open the following file and make it the current file.
[ DOS ] OPEN has to refer to words in the DOS vocabulary.
?DEFINED Find the file in dictionary. If failed, create a new file.
!FILES Make this file the current file.
OPEN-FILE Open it.
;

: DEFINE ( -- )  Define the following word as a new file without opening it.
?DEFINE DROP ;

: FROM ( -- )  Make the next word in the input stream the FROM file. It will be created if not already being defined.
?DEFINE Open a file.
IN-FILE ! Make it the in-file.
OPEN-FILE And then open it.
;

DEFER LOAD Interpret a screen.

In the previous Forth systems, including F83 Version 1, LOAD always interprets a screen from the current file. To allow more natural and more convenient access to multiple files, F83 Version 2 modified the LOAD command so it will load a screen from the in-file, which is set up as the input file. Most of the other file commands access the current file as default. To make sure that other file commands can still access the current file, LOAD only loads one screen from the in-file and then restores the current file.

: (LOAD) ( n -- ) Interpret one screen from the in-file.
FILE @ >R Save the current file fcb.
BLK @ >R Save the currently processed screen number on the data stack.
>IN @ >R Save the word parsing pointer also.
>IN OFF Start at the beginning of the screen.
BLK ! Store n into BLK to process screen n.
IN-FILE @ FILE ! Make the in-file the current file for interpreting.
RUN Interpret the screen.
R> >IN ! Restore the parsing pointer.
R> BLK ! Restore the previous screen number.
R> !FILES After loading from the FROM file, restore the current file.
;
' (LOAD) IS LOAD Vector LOAD to execute (LOAD).

1 CONSTANT INITIAL In all the F83 source files, screen 1 is always a load screen which loads in the code in the file. INITIAL is defined to load this screen.

: OK ( -- ) Load applications in the current file.
INITIAL LOAD ;

: A: ( -- ) Select drive A as the default drive.
0 SELECT ;

: B: ( -- ) Select drive B as the default drive.
1 SELECT ;
Chapter 15.  Text Editor

The source code of the editors are in the file UTILITY.BLK, Screens 12 to 27.

An editor is the most often used utility in an operating system to support programming activity. The friendliness of an operating system depends heavily on the editor it provides to you. Since the source code in Forth is organized around blocks of 1024 bytes and the editor has to deal only with blocks of fixed size, the editor is simpler than the editors in other systems, which have to be able to handle large text files, usually of variable length.

The line editor in F83 system is compatible with the editor described in the popular book "Starting Forth". For details on the various commands in this editor, see the book by Leo Brodie. There are a few extensions, most notably the command NEW which allows you to enter multiple lines of text.

A screen editor is also provided in F83 so that you always have a full text screen showing on his terminal. The screen editor will be discussed in Chapter 16.

15.1. String Utility

The string manipulation primitives include string comparison and searching. The string search command is used in the line editor to find the desired string. The only unusual feature about this string package is the presence of a variable called CAPS, which determines whether or not to ignore the case of the source and target strings. If case is ignored then A-Z=a-z. The default is to ignore case.

Many string primitives are defined in the kernel, like string compare, lower-to-upper case conversion, etc. Many of them are defined in machine instructions to increase execution speed. Here their high level commands are shown for completeness and for reference. You should consult the sections in the kernel for the code commands actually used in the editor, in the file KERNEL86.BLK, Screens 41-43.

VARIABLE CAPS  If true, lower case characters are to be converted to upper case.

: UPC ( char -- char' ) Convert a character to upper case.
DUP Copy char for comparison.
ASCII a ASCII z Is it between a and z?
BETWEEN IF BL - THEN If so, convert to upper case by subtracting 32.
;
: ?CHAR ( char -- char' ) Convert a character to upper case if CAPS flag is set.
CAPS @ Is CAPS true?
IF UPC THEN If so, convert; otherwise skip.
;
: COMPARE ( addr1 addr2 count -- n ) Compare two strings at addr1 and addr2 of equal length. Case may be significant depending on CAPS. 0 is returned if the strings are equal. 1 is returned if string at addr1 is greater than that at addr2. -1 is returned if the string at addr1 is less than that at addr2.
>ROT Save the count.
0 The initial value of n to be returned.
-ROT Put it under addr1.
R> Retrieve the count.
0 ?DO Scan the strings.
OVER I C@ Get a character from string 1.
?UPCHAR Convert it to upper case if needed.
OVER I C@ Get the corresponding character from string 2.
?UPCHAR Convert.
-DUP Are the characters the same?
IF No. The characters are not equal.
R> Retrieve the comparison result.
0< IF If it is less than zero, return -1 because string 2 is larger.
ELSE 1 THEN If the comparison is positive, return 1 to indicate that string 1 is greater than string 2.
-ROT Put the result below addr1, replacing the initial n.
LEAVE Quit the do-loop immediately.
ELSE DROP Characters are equal. Discard the result of comparison.
THEN
LOOP 2DROP Discard the addresses.
;
: INSERT ( sa sl ba bl -- ) Insert a string at sa into the buffer at ba. The length of string inserted is the smaller of sl and bl.
ROT OVER MIN >R Save the smaller of sl and bl on the return stack.
R@ - bl-sl or 0 if sl>bl.
OVER DUP Buffer address ba.
R@ + Address of the remainder of the buffer.
ROT CMOVE> Shift the string in the buffer forward, making room for the string to be inserted.
R> Restore the count of insert string.
CMOVE Copy string to buffer.
;
: REPLACE ( sa sl ba bl -- ) Copy a string from sa to ba. Characters copied is the smaller of sl and bl.
ROT MIN Smaller of sl and bl.
CMOVE Copy from sl to ba.
;
: DELETE ( ba bl sl -- ) Delete sl characters from the start of buffer, ba. Fill the end of buffer with blanks.
OVER MIN >R Save the smaller of bl and sl.
R@ - The remainder of the buffer.
DUP 0> Any character to be move forward in the buffer?
IF Yes. Copy the remainder of buffer forward to the start of buffer.
2DUP Duplicate ba and remainder of buffer.
SWAP DUP Buffer address ba.
R@ + Address of remainder of buffer.
-ROT Swap CMOVE Copy the remainder of buffer to the beginning of the buffer.
THEN
+ Address of remainder of buffer.
R> BLANK Fill the remainder of buffer with blanks.
;
VARIABLE FOUND A local variable to be used by SEARCH as a flag.
: SEARCH ( sa sl ba bl -- n f ) Search for the string at sa inside the string at ba. If found, return the offset of the found string as n and a true flag. If not found, f is false and n is meaningless.
FOUND OFF Initialize FOUND to be false.
OVER >R Save buffer address ba.
ROT TUCK - bl-sl.
1+ 0 ?DO Scan the buffer for the string. Stack is now ( sa ba sl -- )
3DUP COMPARE Is the string found at this position?
0= IF If found,
FOUND ON
LEAVE
THEN
SWAP 1+ SWAP
LOOP
DROP NIP
R> -
FOUND @
;

These string commands are the basic commands used in implementing the string editing commands which are needed in both the line editor and the screen editor in this F83 system.

15.2. Terminal Dependent Deferred Commands

Several commands which will be used to control screen display in the screen editor are defined here as deferred commands so that commands defined in the line editor can be used also in the screen editor by re-vectoring these deferred commands.

DEFER AT (col row --) Position the cursor at the given location specified by the stack numbers.
DOES> A vectored word.
-ROT 2DUP #LINE ! Store col in #LINE.
#OUT ! Store row in #OUT.
ROT PERFORM Execute the word vectored to by AT.
;
AT Execute AT vectors to itself.
DEFER BLOT (col --) Delete the rest of the current line.
DEFER -LINE (-- ) Delete the current line and scroll the rest of screen up by one line.

: DARK (--) Clear the screen and home the cursor.
DOES> Dark is a deferred word and can be re-vectored.
PERFORM First execute the routine whose execution address was put into the parameter field.
#LINE OFF Reset line count.
#OUT OFF Reset character count.
;
DARK Vector DARK to itself.

VOCABULARY EDITOR Make it the current vocabulary so that following definitions will be included in it.
EDITOR ALSO DEFINITIONS

DEFER .SCREEN (-- ) Display the entire screen.

: (AT) (col row --) Do a carriage return in line editor mode.
2DROP CR
;
: (BLOT) (col --) Fill the rest of line with spaces.
C/L SWAP - Characters in the rest of this line.
SPACES Output spaces.
;
: (DARK) (--) Clear the screen with line feeds.
24 0 DO CR LOOP
;
' (AT) IS AT Initialize AT,
' (BLOT) IS BLOT and the rest of the deferred words to support the dumbest possible terminal.
' (DARK) IS DARK
' NOOP IS -LINE
' CR IS .SCREEN

15.3. The Cursor Commands
The cursor in the line editor is a pointer pointing to the character position where the next editing actions will occur. The cursor position is stored in a variable R#, as the offset from the beginning of the screen buffer to the address of the current character. All cursor commands use or modify this variable. Often the cursor is represented by a caret '^' on CRT display.

VARIABLE R# Defined in the nucleus.

: TOP ( -- ) Go to the top of the screen.
R# OFF Initialize R# to zero.
;

: C ( n -- ) Move the cursor by n characters, right or left.
R# @ Current cursor position.
C/SCR 1- 1023, a 10 bit mask.
AND Ensure the cursor is within the screen.
R# ! Replace it.
;

: T ( n -- ) Go to the beginning of line n.
TOP Reset R#.
C/L * Beginning of the nth line.
C Set the cursor. Always within screen.
;

: CURSOR ( -- n ) Return the current cursor position.
R# @ ;

: LINE# ( -- n ) Return the current line number.
TOP Reset R#.
CURSOR C/L / Divide the cursor position by characters per line.
;

: COL# ( -- n ) Return the current column number.
CURSOR C/L MOD The modulo of cursor position.
;

: +T ( n -- ) Increment the current line by n.
LINE# + Get the new line number.
T Select it as the current line.
;

: 'START ( -- addr ) The buffer address of the start of the screen.
SCR @ The current screen number.
BLOCK The address of the buffer.
;

: 'CURSOR ( -- addr ) The actual address of the current character in the buffer pointed to by the cursor. 'START Beginning of the screen. Add cursor offset to get the address in the buffer.

: 'LINE ( -- addr ) The address of the beginning of the current line. 'CURSOR Address of the cursor. Subtract the column number to get back to the beginning.
COL# -

: #AFTER ( -- n ) Return the number of characters after the cursor on the current line.
C/L Characters per line.
COL# - Characters after cursor.
;

: #REMAINING ( -- n ) Return the number of characters after the cursor on screen B/BUF Characters per screen. Characters after cursor on screen.
CURSOR - ;

: #END ( -- n ) Number of characters between the beginning of current line to the end of screen. Characters from cursor to end of screen. Characters from start of line to cursor on current line.
#REMAINING COL# + ;

124
PAD returns the address of a text buffer used by Forth system for string output and temporary storage. The Starting Forth editor requires two additional text buffers: an insert buffer and a find buffer. The insert buffer stores a text string which will be inserted into the screen during editing, and the find buffer stores a string to be used in string search. These two buffers are assigned immediately above the PAD buffer and they all float some distance above the top of the dictionary. Since they are using the free memory space between the data stack and the dictionary, they do not require fixed allocation in the RAM memory.

Some of other editing utility commands are also defined here.

15.4. Editing Buffers

VARIABLE CHANGED A variable indicating that the current screen has been edited so that date stamp can be applied automatically.

: MODIFIED ( -- ) Mark the screen as updated and also set the CHANGED flag.
CHANGED ON Set the CHANGED flag.
UPDATE Set the UPDATE flag.
;

ASCII ^ CONSTANT EOS EOS is the character to denote the end of a string on input. It allows multiple editing commands on one line.

: ?TEXT ( addr -- addr+1 n )
Accept a string to addr. For a null string, do not disturb the string already at addr.

> R Save addr on return stack.
EOS PARSE Scan the input stream for ^ or end of line.
DUP Get the character count of the input string.
IF If character count is not 0, do the string copying.
R@ Retrieve addr.
C/L 1+ BLANK Clear one line at addr.
HERE COUNT The string is in the word buffer.
R@ PLACE Copy the string to addr.
ELSE 2DROP Clean the stack after PARSE.
THEN
R> Get the addr back.
COUNT Replace it by addr+1 and count.
;

10 CONSTANT ID-LEN Length of the id stamp buffer.
CREATE ID ID-LEN ALLOT Id stamp buffer containing the user name and date stamp.
ID ID-LEN BLANK Initialize the id stamp buffer.
84 CONSTANT C/PAD All text buffers are to be 84 characters long.

: 'INSERT ( -- addr ) Return the address of insert buffer.
PAD C/PAD + 84 bytes above PAD.
;
Figure 15.1  The editing buffers.

: 'FIND ( -- addr ) Return the address of find buffer.
    'INSERT C/PAD + 84 bytes above the insert buffer.

: 'VIDEO ( -- addr ) Return the screen editor buffer.
    'FIND C/PAD + 4 bytes above the find buffer.

: .FRAMED ( addr -- ) Print a string at addr framed with single quotes.
    ." " Print preceeding quote.
    COUNT TYPE Print the string.
    ." " Print following quote.

: .BUFS ( -- ) Display the contents of the insert and find buffers.
    CR ." I " Header for insert string.
    'INSERT .FRAMED Print insert buffer.
    CR ." F " Header of find string.
    'FIND .FRAMED Print find buffer.

: ?MISSING ( nf -- n, or abort ) If flag is false, print the find buffer and abort. Otherwise, return only n.
    0= IF If flag is false, do the following.
    DROP Discard n.
'FIND .FRAMED Print contents of find buffer.
." not found " This is used when a string cannot be found in the screen.
QUIT Give up and return to the text interpreter.
THEN ;
: KEEP ( -- ) Copy the current line into the insert buffer.
'LINE Address of current line.
C/L 'INSERT PLACE Copy the line to insert buffer.
;
: K ( -- ) Exchange the contents of the insert and find buffers.
'FIND PAD C/PAD CMOVE Copy find buffer into PAD buffer.
'INSERT 'FIND C/PAD CMOVE PAD 'INSERT C/PAD CMOVE Copy old find string to insert buffer.
;
: 'F+ ( n1 -- n2 ) Add the length of the found string to n1.
'FIND Address of the find buffer.
C@ Length of find string.
+ ;
: W ( -- ) Abbreviation of SAVE-BUFFERS.
SAVE-BUFFERS ;
: 'C#A ( -- addr count ) Return the address of the cursor and the characters after cursor on the current line.
'CURSOR Address of the cursor.
#AFTER Characters after cursor.
MODIFIED Update flags.
;
:(I) ( -- len 'insert len 'cursor #after) Get input string into the insert buffer and leave addresses and lengths necessary to do the insertion.
'INSERT ?TEXT Get input text and copy it into the insert buffer.
TUCK Tuck a copy of len under address of insert buffer.
'C#A Push cursor address and character count on stack.
;
15.5. Line Editing Commands

Line editing commands modify the contents of the current line in the current screen. Many of these commands expect a string immediately following the command in the same input line. The string may be a null string, i.e., the command is followed immediately by a carriage return. In this case, the contents of the appropriate buffer are used in place of input string. The text string is shown as <text>, which is a sequence of ASCII characters. Blanks or spaces can be included in the string. The string is terminated either by a carriage return or by the carat character '^

: I ( -- ) I <text> inserts text string on the current line at the cursor.
(I) Get the insert string.
INSERT Insert it on the current line.
C Move the cursor to the end of the inserted string.
;
: O ( -- ) O <text> overwrites text string on the current line at the cursor.
(I) Get the insert string.
REPLACE Write over the current line with the insert string.
C Move cursor to end of inserted string.
;
: P ( -- ) P <text> replaces the current line with <text> and blank fill the rest of the line.
'INSERT ?TEXT Get insert string.
DROP Discard character count.
'LINE C/L CMOVE Copy the entire line.
15.6. String Editor Commands

The main task of the string editor is to locate a string inside the current screen and place the cursor at the end of the found string. This allows you to modify strings in a screen quickly, and to make local modifications as you debugs source code. The string pattern to be searched is put in the find buffer.

: FIND? ( -- n f ) Get the find string from the input stream and search for a matching string in the screen starting at the current cursor position. Return the character offset.
of the found string as n and a true flag if the string is found. Return a false flag if not found, and n is meaningless in this case.

'*FIND
?TEXT
'CURSOR #REMAINING
SEARCH
;
: F ( -- ) F <text> finds the text and leaves the cursor just past it.
FIND?
?MISSING
'F+ C
;
: E ( -- ) E <text> erases the string just found.
'FIND C8
The character count of the found string.
'DUP NEGATE C
Move the cursor to the beginning of the found string.
'C#A ROT DELETE
Delete the found string.
;
: D ( -- ) D <text> finds and deletes a string.
F
Find.
E
Erase.
;
: R ( -- ) R <text> replaces the text string just found with the string in the insert buffer.
E
Erase the found string.
I
Insert the insert string.
;
: S ( n -- ) n S <text> searches for the text through all screens from the current up to screen n. Each time a match is found, n remains on the stack until screen n is reached.
1 ?ENOUGH
IF
'F+ C EXIT THEN
DROP
FALSE
OVER SCR @ DO
Scan a range of screens.
N
Next screen.
TOP
Beginning of next screen.
'FIND COUNT
Find string.
'CURSOR #REMAINING
Buffer address and count in the next screen buffer.
SEARCH
Search the text string.
IF
'F+ C
Move cursor to end of found string.
DROP TRUE
Replace false flag with true.
LEAVE
Exit the do loop.
ELSE DROP
THEN
KEY? ABORT" Break!
If any key is received on the keyboard, quit the searching.
LOOP
Otherwise, continuing the search to the next screen.
?MISSING
n should be on the stack.
;
: (TILL) ( -- ) Search in the current line for the text string.
'*FIND ?TEXT
'CURSOR (TILL)
Search current line from the cursor for the find string.
?MISSING
If string can't be found, abort.
;
: TILL ( -- ) TILL <text> deletes all text on the current line from cursor to the end of the find string.
'C#A (TILL)
Search the find string.
'F+ DELETE
Delete from cursor to end of found string.
;
: JUST ( -- ) Justify. Delete up to but not including the text string.
'C&A (TILL)
Search the text string.
DELETE
Delete all characters between the cursor and the starting character of the found string.
;
: KT ( -- ) Keep-Till. Copy all the characters between the cursor and the end of the text string into the insert buffer.
'CURSOR (TILL)
Find the text string.
15.7. Screen Editor

The line editor works on the source screen one line at a time. It serves well all the editing functions. The only problem is that the text screen displayed on the terminal scrolls up with the entering of new lines at the bottom of terminal screen. After some editing, compiling and testing, the text screen starts to disappear over the top of the terminal and you must type L command to re-display the text again. As the terminals getting smarter and smarter, it is nice if we can use some of the extra functions in the terminal to keep the text screen at the top of the terminal all the time. This is basically what a screen editor does. Any time the text screen is modified, the modification will be written on the displayed text screen immediately.

If all the terminals were built the same way, it would be a simple exercise to write a screen editor. However, terminals are built with different screen control commands. The screen editor must be tailored to the specific terminal to use its specific cursor and display control commands. The screen editor in F83 uses all the commands developed in the line editor for editing functions. The terminal-specific functions are concentrated in four commands: AT, DARK, BLOT, and -LINE, which manage a continuous display of the edited text screen. The display is updated automatically as each command line is executed.
15.8. The Screen Display Commands

The display on the terminal is assumed to have a 24 by 80 character format. The screen editor displays the screen number on the top line or line 0 on the screen display. Lines 1 to 16 are used to display 16 lines of text in the current screen. Line 17 displays the current line. Lines 18 to 23 are used as a scrolling window for command input and character output. The text screen stays at the top of display and always shows the updated text of the current screen under editing.
: CHANGED? ( line# -- f )
    Return a true flag if the line has changed since last display. It is sensitive
    to case changes.
    C/L * Character offset to the line.
    DUP 'START + Address of the line in the buffer.
    SWAP 'VIDEO + Address of same line in the video buffer.
    C/L COMP Compare the lines in text screen and video buffer.
    \ Return true if two line are different.

: .ALL ( -- ) Redisplay all lines which have changed, the screen number, the cursor line, and
    scroll the command region.
    DISK-ERROR @ 0= If no disk error, display the screen.
        IF
        "DX 0 AT .SCR Display the screen number.
        #OUT @ BLOT Erase the rest of the line.
        [ FORTH ] Switch context to FORTH because
        ?STAMP Stamp the screen if not done.
        I CHANGED? Has this line been changed?
        IF I REDISPLAY If changed, redisplay the new line.
        THEN
        LOOP 'START 'VIDEO B/BUF
            CMOVE
        0 18 AT .LINE Display the current line under the displayed screen.
        0 19 AT -LINE Delete line 18 on display and scroll up the rest of screen display.
        0 23 AT Position the display cursor at the bottom of display, assuming 24 line display.
        #OUT OFF Clear output character count.
        THEN

: EDIT-AT ( -- ) Move the display cursor to show the position of the editor cursor for editing
    functions.
    CURSOR C/L /MOD Convert cursor offset to screen coordinates.
    SWAP DX + Column number.
    SWAP DY + Row number.
    AT Move the display cursor.

: NEW ( n -- ) Move the display cursor to the beginning of line n and accept text for following
    lines until a null line (a line begins with a carriage return) is entered, i.e.,
    2 CR's gets you out of NEW.
    L/SCR SWAP DO Scan from line n down.
        [ FORTH ] I Loop index.
        [ EDITOR ] T Position editor's cursor.
        EDIT-AT Position the display cursor.
        \IN OFF Reset the input character offset.
        QUERY Wait for a line of input text.
        IF If not a null line,
        P Put the text in the current line.
        ELSE For a null line,
            REDISPLAY Put back the old line.
            LEAVE Quit here.
        THEN
        .SCREEN Refresh the display.
        LOOP
        .SCREEN

: GET-ID ( -- ) Check the ID stamp field. If it is empty, prompt for user's id and date.
    ID ID-LENGTH Get the ID string address and length.
    -TRAILING NIP 0= Is the length 0?
        IF
        CR ." Enter you ID: "
        ID-LEN 0 DO ASCII . EMIT
        LOOP
        ID-LEN BACKSPACES Backspace over the dots.
        ID ID-LEN EXPECT Input the id stamp to the id stamp buffer.
        THEN

;
: STAMP ( -- ) Put the stamp at the end of line 0 in the current screen.
  ID  Id buffer address.
  'START C/L + End of line 0.
  ID-LEN 1- Backup the length of stamp text.
  ID-LEN 1- Copy the stamp to screen.

: ?STAMP ( -- ) Update the ID if the screen has changed and clear the change flag.
  CHANGED @ IF Changed?
  STAMP Stamp the screen.
  CHANGED OFF Reset changed flag.

; 2VARIABLE AUTO Addresses of CR and STATUS to patch vectors for CRT and TTY, respectively.
VARIABLE EDITING? Set during editing.
VARIABLE CHANGED Set if the edited screen has been changed.

: INSTALL ( -- ) Initialize the screen editor.
  EDITING? @ NOT IF If not in the editing mode, initialize screen editor.
  [']' .SCREEN Address of the screen refresher.
  AUTO @ ! Vector it through AUTO.
  EDITING? OFF Turn off editing flag.
  CHANGED OFF Turn off changed flag.
  THEN DISK-ERROR OFF Turn off the disk error flag always.

; 15.9. The Screen Editor Commands

FORTH DEFINITIONS The following screen editing commands should be made accessible from the common
FORTH vocabulary.

: DONE ( -- ) Normal exit from the screen editor. Update the id stamp, tell you if the screen
  was modified, flush the screen to disk file, and remove automatic screen refresh.
  [ EDITOR ]
  EDITING? @ IF If still in the editing mode,
  PREVIOUS
  EDITING? OFF turn off editing flag,
  CR SCR ? type the screen number,
  >UPDATE @ 0< NOT IF and look at the update field.
  '" Un" THEN Not updated. Print prefix.
  '" modified" Complete the message.
  ?STAMP Update the ID stamp.
  W Save contents to disk file.
  THEN
  DISK-ERROR OFF Turn of disk error flag to normal scrolling mode.
  AUTO 2@ ! Revector CR

; : ED ( -- ) Re-enter the screen editor. Clear and initialize the display and begin automatic
  screen refresh.
  [ EDITOR ] GET-ID
  INSTALL Initialize the screen editor.
  VIDEO B/BUFFER Clear the video buffer.
  DARK Home the cursor and clear CRT.
  .ALL Print the screen on CRT.

; : EDIT ( n -- ) Set n as the current editing screen and call ED to do screen editing.
  1 ?ENOUGH Abort if the stack has less than one item.
  SCR ! Store n in SCR, making screen n the current editing screen.
  [ EDITOR ] TOP
  ED Do screen editing.

; : (WHERE) ( pos scr -- ) When an error occurred during compilation, the position and screen number where
  error occurred are saved on the data stack. (WHERE) uses these two numbers to
  invoke the screen editor and show the screen to be edited.
  DISK-ERROR @ 0= Make sure the error is not caused by disk reading.

133
15.10. Configuring The Terminal

Each terminal manufacturer has its own way of controlling the display on the terminal screen. To use the full power of a screen editor, the screen editor must know how to position the display cursor at any position on the display screen and a few other things like initialize the display, erase one line and delete one line. In F83 system, terminal configuration commands are provided for a number of popular terminals. If your terminal is not in this list, you will have to define the proper commands. You can follow the pattern as provided. It is not a very difficult task.

An example is shown here. You should consult the F83 listing for other terminals.

```
: SMART ( -- ) Initialize vectored routines for the IBM PC.
'[ CR LF ] Execution address of carriage return.
'[ CR > BODY Vector address in the deferred word CR.
AUTO 2! Store them in the AUTO area.
'[ ALL IS SCREEN Vector SCREEN to ALL.

CODE IBM-AT ( col row -- ) Position cursor at the specified location on CRT screen.
AX POP DX POP AL DH MOV Copy col and row into DX.
BH BH XOR Clear BH register.
2 # AH MOV Cursor positioning code.
16 INT Call BIOS.
NEXT
END-CODE

CODE IBM-DARK ( -- ) Clear screen and home the cursor.
2 # AX MOV Home code.
16 INT Call BIOS.
NEXT
END-CODE

CODE IBM-BLOT ( col -- ) Clear from cursor to end of line.
80 SWAP - Remaining characters on the line.
SPACES Output that many spaces.

CODE IBM-LINE ( -- ) Delete the current line and scroll the rest of the screen.
BP PUSH Save BP register.
BH BH XOR Clear BH.
3 # AH MOV 16 INT Call BIOS for cursor position.
DH CH MOV Line number moved to CH.
CL CL XOR Column number cleared to zero.
24 256 * 79 + Bottom line and right-most column copied to DX register.
7 # BH MOV Code stored in AX.
16 INT Call BIOS.
BP POP Restore BP.
NEXT
END-CODE

: IBM ( -- ) Initialize the screen editor for IBM PC.
SMART Vector CR and SCREEN.
'[ IBM-AT IS AT Vector the rest of terminal specific commands.
'[ IBM-DARK IS DARK
'[ IBM-LINE IS -LINE
'[ IBM-BLOT IS BLOT
```
Chapter 16. Viewing Source Screens

The source code about viewing is scattered in UTILITY.BLK, screen 7, EXTEND86.BLK screen 12, and KERNEL.BLK Screens 66 and 76.

Forth with its hosts of commands can be looked upon from two opposing points of view. For the Forth advocates, it is called modularity because commands can be individually executed or compiled to build higher level commands. For others it is called fragmentation, because functions are scattered in hundreds of small bits and pieces. To decipher a colon command, you have to know the exact functions of every command in this command. It is not an easy job to find them because they seldom are grouped in one place.

F83 has more than 1000 commands in it. How can you find any particular command in this mess?

The designers of F83 provide us with a viewing facility which allows us to locate the source code of all the commands in the dictionary and display the block of texts that contains the source of the command we look for. To accommodate this facility, the format of Forth commands in the dictionary is changed from the traditional form of name field, link field, code field and parameter field to that as shown in Fig. 16.1.

16.1. The View Field

The view field is used to store information about where the source code of the command is located. It is two bytes long and divided into two sub fields: the lower 12 bits contain the block number of the source code and the upper four bits contain the file number of a DOS file containing the block. Let's first examine some of the low level commands associated with the view field:

: VIEW# ( -- addr ) Return the view field in the current FCB.
FILE @ Address of the current FCB.
40 + Offset by 40 bytes to the view field.
;

: .VIEW ( -- ) Compile the view field in a new definition.
VIEW# @ Get the view file number of the current file.
4096 * Shift it to the upper 4 bit file number subfield.
BLK @ + Add the block number of the screen under compilation.
;

: *CREATE ( -- ) Create the header of a new definition. This definition was discussed in the chapter on the vocabulary structure.

: >VIEW ( cfa -- vfa ) Go from code field to the view field.
>LINK Go to link field first.
2- View field is two bytes ahead of the link field.
;
Figure 16.1   The view field and the view files.

16.2.   The View Files

The view field in a command has a view file number in the view field. From this number the viewing command will have to find the file and open it for viewing. The view file number for different source files are assigned and pertinent information are stored in an array called VIEW-FILES. In the file control block (FCB) in the definition of each source file, bytes 40 and 41 are used to store the view file number. This provides the file number to store into the view field when the command was first compiled into the dictionary.

CREATE VIEW-FILES An array where the view files are arranged in sequential order so that the viewing command can find and open the source file of a word definition.

32 ALLOT There is enough space to define 16 files for viewing.

: VIEWS ( n -- ) Assign the view file number on the stack to a file and store the fcb address in
the corresponding entry in the VIEW-FILES array.

[DOS]

?DEFINE Search the dictionary for the word name following VIEWS. It must be a file name
and the fcb address is returned.

2DUP 40 + ! Store number n into the view# field in the FCB of the file
BODY>
SWAP 2* The address offset into the VIEW-FILES array.
VIEW-FILES + ! Store the fcb address into the VIEW-FILES array.
;

Now, view files can be assigned view file numbers to fill the VIEW-FILES array:

1 VIEWS KERNEL86.BLK
2 VIEWS EXTEND86.BLK
3 VIEWS CPU8086.BLK
4 VIEWS UTILITY.BLK

These are the source files in the F83 system. All the commands loaded from these files can be viewed. If you have some application programs in another file, you will have to assign a view file number to it using the VIEWS command as above. After that, you can load in your application and can view the commands loaded in from your file.

16.3. The Viewing Command

Assuming that all the source code files are on the default disk drive, it is very easy to display a screen which contains the command you want to examine. The command is as following:

VIEW <name>

where <name> is the name of the command you want to review. The command VIEW will locate the word <name> in the dictionary and find out the file and the screen number of the source code of this command. It will then open that file and read the screen from the file and display the screen on the CRT terminal.

: @VIEW ( cfa -- scr# file# ) From the code field address of a definition, find its view
field and return the screen number and the file number stored in the view field.

>VIEW Get the view field address.
@ DUP 4095 AND Mast off the top 4 bits in the view field and leave only the screen number.
DUP 0= If the screen number is 0,
ABORT" entered at terminal" abort because the word is not loaded from a file.
SWAP 4096 / 15 AND Extract the view file number from the top 4 bits.
;

: VIEW ( -- ) Allow the user to see the source screen of the following word. If the VIEW# is
zero, then the current file is used. Otherwise, the associated field is opened and viewed.

@VIEW Find the cfa of the following word.
?DUP IF If the file field is zero, use the current file. If not zero,
2* VIEW-FILES + @ find the cfa of the view file in array VIEW-FILES.
EXECUTE Make the file our current file.
OPEN-FILE Open it for reading.
ELSE The definition is in the current file: "FILE? may be in the current file: "
DUP >FILE? And also the screen number.
" screen " DUP > Print the file name,
" screen " . File name first.
EXECUTE Make the file our current file.
OPEN-FILE Open it for reading.
" screen " . And also the screen number.
EXECUTE Make the file our current file.
OPEN-FILE Open it for reading.
" screen " . And also the screen number.
View file number 0 is reserved for the current file if it has not been assigned a view file number. Up to 15 files can be stored in the VIEW-FILES array for viewing. If you have to use the second drive to store some of the files, the drive number in their FCB must be assigned accordingly.
Chapter 17.  WORDS

The source code in this chapter is in file UTILITY.BLK, Screens 3 and 5.

The command WORDS (VLIST in older Forth systems) displays all the commands defined in a vocabulary.  It is very useful in inspecting the dictionary contents, and also in determining the progress of compilation when a large application is being loaded.  The implementation of this command in F83 system is slightly more complicated because the vocabulary is hashed into four threads.

17.1.  Output Formatting Commands

To display a sequence of variable length names on the CRT terminal or printed on a printer, it is necessary to wrap a whole word around the end of a line instead of breaking a word.  A few commands allow us to detect the end of line condition and insert a carriage return before printing the last word.

VARIABLE Lmargin The column number of the left margin.
0 Lmargin !
VARIABLE Rmargin The column number of the right margin.
70 Rmargin !

: ?line ( n -- ) If the current line does not have space for n more characters, move to the next line.
#OUT @ Current character count.
+ Rmargin @ > IF Add n to it.  If it is greater than the right margin,
CR Move to the next line.
Lmargin SPACES Align to the left margin.
THEN ;

: ?cr ( -- ) Move to next line if we had passed the left margin already.
0 ?line
;

17.2.  WORDS

In the parameter field of the command of a vocabulary command like FORTH, EDITOR, or DOS, etc., there are four addresses pointing to the ends of four thread in the dictionary.  They are the link field addresses of the four newest commands defined in the vocabulary.  To print out the entire list of names in the vocabulary, we have to trace through these four threads and print out command names in the descending order according to the addresses.  These four addresses are first moved to the top of the dictionary.  The largest address is first used to print the name which was defined last.  This address is replaced by the next address in its thread, and the process is repeated until all names are printed.

: largest ( addr n -- addr' val ) Given an address and a number on the stack, return the
OVER 0 SWAP  
Add 0 and addr on the stack.

ROT 0 DO  
Scan through the array.

2DUP @ U<  
Compare contents of an array entry with the current largest value.

IF  
If the entry is greater,

-ROT 2DROP  
discard the old address and its value,

DUP @ OVER  
and replace them with the new address and its value.

THEN

2+  
Next address to be scanned.

LOOP

DROP  
Discard the address used for scanning.

;

: WORDS  ( -- )  
List the words in a vocabulary. It can be interrupted by pressing any key.

CR L_MARGIN @ SPACES  
Align to the left margin.

CONTEXT @  
The context vocabulary.

HERE #THREADS 2*  
Copy the array of thread ends to the word buffer.

CMOVE BEGIN  
Begin the printing loop.

HERE #THREADS  
Pick the largest address in the thread array.

LARGEST DUP WHILE  
If it is not zero, print a name. Otherwise, the vocabulary is exhausted.

DUP L>NAME  
From the link field, move to the name field.

DUP C@ 31 AND  
The length of the name.

LINE  
Make sure there is enough room at the end of a line.

.ID  
Print one name.

SPACE SPACE  
Add 2 spaces.

@ SWAP !  
Replace the largest address in the thread array by the address of the next word in the thread.

.KEY? IF EXIT THEN  
Exit if a key is pressed.

REPEAT  
Continue until all names are printed.

2DROP  
Discard the address and a value, which is zero.

;
Chapter 18. Disk File Utility

The source code in this chapter is in UTILITY.BLK, Screens 4, 7 and 8.

18.1. Displaying Screens In a File

The line editor allows us to examine and modify screens, one at a time. For large application which requires many screens, it is necessary to have some commands which allow us to scan the contents of a file so that we will know which screen to examine in detail. These commands are also useful in generating hardcopy of source code on a printer.

```lisp
: .SCR { -- ) Print the current screen number and the file name.
  . " Scr # " SCR ? Print the screen number,
  8 SPACES FILE? and the file name.
  ;

: LIST { n -- ) List the specified screen in the 16 by 64 character format. Pressing a key stops
  the printing. LIST also makes n the current screen.
  1 ?ENOUGH Make sure n is on the stack.
  CR DUP SCR ! Make n the current screen.
  .SCR Print screen number and file name.
  L/SCR 0 DO Scan 16 lines.
  CR I 3 .R SPACE Print the line number first.
  DUP BLOCK Get the screen from the file and return the buffer address.
  I C/L * + Line address in the buffer.
  -TRAILING >TYPE Print one line of text.
  KEY? ?LEAVE Quit if a key is pressed.
  LOOP
  DROP CR House keeping.
  ;
```

To print multiple screens on paper, it is nice to arrange three screens to a page. By convention, the first screen number is a multiple of three so that a group of three screens forms a unit in arranging your source code. TRIAD is the command to print screens in this style.

```lisp
: TRIAD { n -- ) Print three screens on a page. The nth screen must be printed. The top screen
  has a screen number modulo 3.
  12 EMIT Form feed.
  3 / 3 * Modulo 3 boundary.
  3 BOUNDS DO Print only 3 screens.
  I LIST One screen at a time.
  LOOP
  ;
```

The top line in a screen is usually a comment line. Besides the documenting function, it also
allows you to scan a range of screens and identify the contents of screens easily. F83 also puts an
ID stamp at the end of the top line. The command INDEX prints the top lines of a range of
screens. It is very handy as a substitute of a screen directory.

```lisp
: .LINE0 { n -- ) Print line 0 of nth screen.
  DUP 3 MOD If n is evenly divisible by 3,
  0= IF CR THEN send out an extra line feed.
  CR DUP 3 .R SPACE Print the screen number.
  BLOCK Get the screen from file.
  ;
```

142
**18.2. Disk Buffers**

Screens or blocks of 1024 bytes are the basic units for Forth program source code. The size of block is convenient to develop modularized programs because the block of text fits comfortably inside a standard CRT screen, while allow enough room to do loading and testing of the source code. Since the screens can be compiled independently, it is the common practice to use a load screen which loads other screens in the order that is required by the application. Therefore, screens do not have to be arranged in a specific order. However, there are times when we would like to move texts screens around and physically arrange them in some order, especially if the texts are to be printed and communicated to other people. Disk copying utility in F83 enables you to copy single or multiple screens within a file and also from one file to another.

To fully appreciate the disk copying utility in F83, it is necessary to understand fully the disk buffer structure used in the F83 system to handle the traffic from and to the disk files. Let us briefly review the F83 disk buffers.

In the high RAM memory, a number of disk buffers are assigned by the system at boot up time. Each buffer is 1024 bytes in size to hold a block of text or data, corresponding to the data stored in a corresponding block in a disk file. The number of disk buffers is specified in a constant #BUFFERS. Immediately before the first disk buffer, there is an array storing the essential information about the disk buffers. Each disk buffers has a corresponding entry in this array. An entry consists of four cells for the block number, a pointer to the parent file, the address of the disk buffer, and an update flag. Whenever a block is accessed, its array entry is moved to the beginning of the array, indicating it is the most recently accessed buffer. The buffer with an array entry at the end of the array is the least accessed buffer and will be re-assigned to receive other disk block when necessary.

In doing multiple block copying, it is generally desirable to read/write all the disk buffers together and in sequence to minimize the disk head movements and the start/stop of the disk drive. F83
disk copy utility tries to optimize the disk accessing, as it normally uses four disk buffers.

More detailed information on the disk I/O is covered in the chapters dealing with the nucleus and DOS of the F83 system.

18.3. Single Block Copying

To copy one block of text or data to another block in the same disk file, the most efficient way is not physically copying the block, but to bring the source block into one disk buffer and reassign that buffer to the destination block with the update flag set. The data in the buffer will be flushed into the destination block when the buffer is needed for other disk I/O transaction.

```
: ESTABLISH ( n -- ) Set the block number of the most recently referenced block to n, thus assign the buffer to the nth block.
FILE @ SWAP Get the current file fcb.
1 BUFFER# Get the address of the first entry in the buffer pointer array.
2! Store n into the first cell in that entry, forcing the most recently referenced block to become the nth block. The current file number is store into the second cell.

: (COPY) ( from to -- ) The primitive word to copy one block to another.
OFFSET @ Get the block offset number of the current file.
+ The actual block number of destination.
SWAP Get the source block number to top of stack.
IN-BLOCK Get the source block into the most recently referenced buffer.
DROP The buffer address is not needed.
ESTABLISH Claim this buffer for the destination block.
UPDATE Update the buffer so that it will be written back to the destination block.

: COPY ( from to -- ) To be careful, copy a block and explicitly flush it to disk.
FLUSH Empty the disk buffers to disk files. Important when accessing multiple files.
(COPY) Do the copying.
FLUSH Flush the destination block.
```

18.4. Multiple Block Copying

When we copy a range of consecutive blocks from one place to another, complication arises if the destination range overlaps with the source range. To avoid conflicts in the overlapped copying, the direction of copying must be carefully selected.

```
VARIABLE HOPPED The number of screens to skip when copying.
VARIABLE U/D The direction of copying to avoid overlap.
DEFER CONVEY-COPY A deferred word. It will be used to copy within one file and copy between files.
'(COPY) IS CONVEY-COPY For the moment, define it to copy blocks in the same file.

: HOP ( n -- ) Specify the number of screens to skip in copying.
HOPPED ! ;

: .TO ( n1 n2 -- n1 n2 ) Print a message while copying.
CR OVER . Print n1.
" to " . Print n2.
DUP .

: (CONVEY) ( blk n -- ) Move a range of screens in the direction specified by U/D.
```
blk+~n )  Copy n blocks. If n=0, exit immediately.
0 ?DO  Stop if user hit a key.
KEY? ?LEAVE  Source block number.
DUP DUP  Destination block number.
HOPPED @ +  Print something to indicate that the computer is working hard.
.TO  Copy one block a time.
U/D @ +  The next source block.
LOOP  Flush the blocks still in the disk buffers.

: CONVEY  Move a set of screens. First determine the direction of copying to prevent overlap. Move the blocks as a set whose size is determined by the number of available disk buffers.
FLUSH  Clear the buffers.
HOPPED @  Get the screen number to skip.
0< IF  If copying from high block to low block,
1+ OVER -  Number of blocks to copy.
1  Copy in the forward direction.
ELSE  Copying from low block to high block.
DUP 1+ ROT -  Number of blocks to copy.
-1  Copy in the backward direction.
THEN  Store the direction code into U/D.
#BUFFERS /MOD  Groups of blocks to be copied together.
>R (CONVEY) R>  Copy the remainder blocks which do not fill the pipeline.
0 ?DO  Pipe the rest of block through all the buffers.
#BUFFERS (CONVEY)  Copy #BUFFERS blocks all at once.
LOOP
DROP  Leftover block number.

If you know the destination block number and do not want to use HOP to specify the number of blocks to be skipped, you can use the following commands to do the copying:

<first> <last> TO <destination> CONVEY

where 'first', 'last', and 'destination' are block numbers indicating the range of source blocks and the first destination block number.

: TO  Calculate the blocks to be hopped and store the number in HOPPED.
SWAP  Get 'first' to top of stack.
BL WORD  Read the next number.
NUMBER DROP  Convert the number to an integer.
OVER -  The hopping distance.
HOP  Store it in HOPPED.
SWAP  Restore the 'first last' order on data stack.

18.5. Multiple File Block Copying

F83 Version 2.0 and above was modified so that when screens are copied, it is always read from the in-file and written to the current file. Therefore, it is not necessary to define special commands to copy screens from one file to another. You must remember that when you OPEN a file, you assign the file to both the current file and the in-file. If you open another file using the FROM command, this file becomes the in-file which is always to be read. In the early versions of F83 the role of in-file was not clearly defined, and many block copying commands had to be redefined in the FILES vocabulary. The following discussion applies only to the F83 Versions 1.x.
One of the advantages in using the DOS operating system to host a Forth system is that we can organize the disk storage into named files which help us using the disk storage more conveniently by grouping related screens into separated files. However, it is often necessary to transfer common utility from one file to another. The multiple file screen copying utility in F83 system makes it easy to copy either single screen or a range of screens from one file to another by new versions of COPY and CONVEY. These commands are redefined in the FILES vocabulary so that they can be accessed independent of the existing COPY and CONVEY commands in the FORTH vocabulary.

ONLY FORTH ALSO FILES
DEFINITIONS
Make FORTH and ONLY the resident vocabularies and FILES the context and current vocabulary.

: COPY ( from to -- ) Copy a screen from the FROM file to the current file. The FROM file must be declared by the FROM xxx commands, where xxx is the FROM file name.
SWAP Get the source block number to top of stack.
EXCHANGE Exchange the FROM file with the current file.
BLOCK Read the source block from the FROM file.
SWAP Get the destination block number to top of stack
EXCHANGE Restore the current file.
BLOCK Get the destination block from the current file.
B/BUF CMOVE Copy the contents of the source screen into destination screen buffer.
UPDATE Update the destination block so it will be copied back to the current file.
;

: CONVEY ( first last -- ) Copy a range of screens from the FROM file to the current file. The screen range is the current file is offset from that in the FROM file by the number in HOPPED.
['] CONVEY-COPY >BODY Get the parameter field of the word CONVEY-COPY.
@ The execution address of (COPY).
>R Save it on the return stack.
['] COPY Execution address of the COPY in the FILES vocabulary, which does copy between files.
IS CONVEY-COPY Vector CONVEY-COPY to the COPY we just defined above.
CONVEY Now, CONVEY will copy a range of screens from FROM file into the current file because CONVEY-COPY is vectored to the new COPY command.
R> ['] CONVEY-COPY >BODY ! Restore CONVEY-COPY to the old single file COPY.
;

This is an example in using a deferred word to do different thing by vectoring it to different commands. The commands using the deferred word do not have to be change at all.
Chapter 19. Memory Dump

Source code discussed here is in the UTILITY.BLK file, Screens 28 to 30 and KERNEL86.BLK screen 87.

Source code and text data can be displayed or printed using commands like LIST, SHOW, and TYPE at the primitive level. Non-ASCII data like object code and numeric data cannot be displayed conveniently. The dumping utility provided in F83 allows you to review the binary data in a conveniently formatted form. Large area of memory and large numeric data set can be either displayed on terminal or listed on printer.

19.1. The Dumb DUMP

A simple and primitive dump command is included in the kernel of F83 system. It helps you to debug the system before it is fully checked out.

```plaintext
: DUMP ( addr len -- ) Dump a range of memory from addr in bytes.
  0 DO Set up the loop.
  CR Start a new line.
  DUP 6 .R SPACE Print the address first.
  16 0 DO Dump 16 bytes.
  DUP C@ Get one byte.
  3 .R Print one byte.
  1+ Increment address.
  LOOP 16 +LOOP Loop for more lines.
  DROP ;
```

19.2. The Smart DUMP

More sophisticated dumping routines present data in both numeric and ASCII forms because in many cases the ASCII data are intermixed with binary data. It is convenient to have both types of display side by side. It is also nice if one can scan the memory forward and backward. The more elaborate dumping utility in F83 has many features not available in other systems.

```plaintext
: .2 ( n -- ) Display a 2 digit number followed by a space.
  0 Make a double number of n.
  `<# # # > Convert two digits.
  TYPE SPACE Type two digits with a trailing space.
; ;
: D.2 ( addr len -- ) Display a line of 2 digit numbers.
  BOUNDS
  OVER + SWAP Convert addr len to the limit-index format.
  ?DO Skip if len=0.
  1 C@ .2 Print one number.
  LOOP ;
: EMIT. ( char -- ) Emit one character if it is printable. Otherwise display a period.
  127 AND Mask off MSB.
  DUP Save a copy of char.
```
BL 126 BETWEEN  Is it between 32 and 126, the printable range?
NOT IF DROP ASCII .  If not printable, replace char with '.'.
THEN
EMIT  Send either char or '.'.
;
: DLN ( addr -- )  Dump 16 bytes of data starting at addr.  Display address first, then 2 sets of 8 bytes, followed by the ASCII equivalent.
   CR
   DUP 4 \R 2 SPACES  Display the address.
   8 2DUP D.2 SPACE  Display 8 bytes.
OVER + 8 D 2 SPACE  Second set of 8 bytes.
16 BOUNDS DO  Scan 16 bytes
I C@ EMIT  Print ASCII characters.
LOOP ;
: ?.N ( n1 n2 -- n1 )  If n1=n2, display a downwards pointer, otherwise display the number.
   2DUP =  n1=n2?
   IF  "/" DROP  Equal.  Display pointer and drop n2.
   ELSE 2 .R THEN
SPACE ;
: ?.A ( n1 n2 -- n1 )  If n1=n2, display a 'v' symbol. Otherwise display one character.
   2DUP =  n1=n2?
   IF  "V" DROP
   ELSE 1 .R THEN
SPACE ;
: .HEAD ( addr len -- addr1 len1 )  Display the header field of a dump, making it easy to index into the data portion of the dump.
   SWAP  Get addr to top of stack.
   DUP -16 AND  Mask off the least significant 4 bits in the address.
   SWAP  Second copy of address.
   15 AND  Preserve only the lower 4 bits.
   CR 6 SPACES  Skip the address field.
   8 0 DO I ?.N LOOP  Print numeric field markers.
   SPACE
   16 8 DO I ?.N LOOP  Second set of field markers.
   SPACE
   16 0 DO I ?.A LOOP  ASCII field markers.
   ROT +  Leave addr1 and len1 on stack, enabling full line display.
;
: DUMP ( addr len -- )  Dump a range of memory specified on stack.  The dump is always in HEX, but the current base is preserved.
   BASE @ -ROT  Save base under addr.
   HEX  Use hexadecimal conversion.
   .HEAD  Print the display header.
   BOUNDS DO  Scan the memory range.
   I DLN  Display a line.
   KEY? ?LEAVE  Quit if any key is pressed.
   16 +LOOP  16 bytes per line.
   BASE !  Restore the original base.
;
: DU ( addr -- addr+64 )  Dump 64 bytes at the specified address and increment the addr so that next block of memory can be display next
   DUP 64 DUMP
   64 +  Increment addr.
;
: DL ( line# -- )  Dump the specified line in the current screen to verify non-printable characters.
   C/L
   SCR LOCK +  Address of the specified line.
   C/L DUMP  Dump one line.
Chapter 20. Decompiler

The source code of the decompiler is in UTILITY.BLK, Screens 31 to 42.

A decompiler is a program which can translate an object program in machine executable form back to the source program a human being can read. This is normally impossible because traditional compilers produce more object code than source code, showing a 'code expansion'. However, decompilation is rather easy in Forth because there is an one-to-one correspondence between the source code and object code in Forth, as a word or a command in Forth is compiled to an execution address in the object. Exception to this one-to-one relationship occurs in the control structures and some other special compiler directives. A Forth decompiler must be able to deal with these exceptions.

The final command doing the decompilation is SEE, which is used in the following fashion:

```
SEE <name>
```

where <name> is the name of a Forth command. The decompiled source code will be displayed on the terminal as a sequence of Forth commands similar to the original source code.

20.1. Positional Case Defining Command

This is the simplest among the CASE control structures effecting an n-way branching. In the parameter field of the case command, there are a sequence of execution addresses. One address in the list is selected by the number on the stack and executed. In this version of case, additional range checking is also implemented for safety.

```
: OUT ( n pfa -- )  Display an error message if the index is out of range for a case word whose parameter field address pfa is on the stack.
  CR ." Subscript out of range on "
  DUP BODY>
  >NAME
  .ID
  .." Max is " ? ." tried ".
  QUIT
  ;

: MAP ( n pfa -- addr )  Given the pfa of a case word and the index n for case selection, return the execution address selected. Abort if the index is out of range.
  2DUP @
  U< IF
  2+ SWAP 2* +
  ELSE OUT
  THEN
  ;
```

The case defining command is CASE:. It is used in the same way as a regular colon defining command. The name of the new case command follows CASE:, and then a list of regular Forth
commands followed by ;. A range number should be on the stack before CASE: is encountered to specify the number of branches in the case command.

\[ \text{CASE: <name> <list of FORTH words> ;} \]

When the new case word <name> is executed, it uses the top item on the stack as an index to select one of the FORTH commands in the list and executes it.

\[ \text{CASE: \{ n -- \}} \]

A positional case statement. The range n is used for error checking. At runtime, the nth word is executed, depending on the value on stack when executed.

\[ \text{CONSTANT Compile the range n as a constant.} \]

\[ \text{HIDE Smudge the name field as : would do.} \]

\[ \text{CASE} \]

Now, use the colon compiler to compile the cases. Compilation will be terminated by the ; command.

\[ \text{DOES> \{ index -- \}} \]

At runtime, use the index to find the execution address among the compiled cases and execute it.

\[ \text{MAP Return the address pointing to one of the cases compiled.} \]

\[ \text{PERFORM Execute it.} \]

Because of the multitude of special compiler directives used in the F83 system, we need a big case statement to handle all the exceptions. This CASE: defining command, though very simple by borrowing facilities in the colon compiler, is extremely powerful to take care of a wide range of n-way branching structures. The limitation is that all the cases must be defined as single commands. This is not a problem because it is a good practice to modularize the cases into single testable commands before putting them into a big case structure.

20.2. **Associative Defining Command**

An associative commands also has a list of values in its parameter field. At runtime a value on the top of the data stack is compared with the list of values in the associative command. If a match is found, the index of the matched value in the parameter field is returned. This is the inverse of an array.

\[ \text{ASSOCIATIVE: \{ n -- \}} \]

Store the maximum range of the associative array as a constant. The values will be compiled explicitly by the , (comma) command.

\[ \text{CONSTANT Compile n as a constant.} \]

\[ \text{DOES> \{ value -- index \}} \]

Search value in the parameter field and return the index if found.

\[ \text{DUP } @ \]

Get the range n.

\[ \text{ROT} \]

Get another copy of n.

\[ \text{DUP } @ \]

Get another copy of n.

\[ \text{0 DO} \]

Scan the list in parameter field.

\[ \text{2+ Next number in the list.} \]

\[ \text{2DUP } @ = \]

Match?

\[ \text{IF Yes.} \]

\[ \text{2DROP DROP Clear the stack.} \]

\[ \text{I 0 0} \]

Put on the index and flags.

\[ \text{LEAVE Quit the loop.} \]

\[ \text{THEN} \]

\[ \text{LOOP 2DROP Return only the index. If no match, return n+1.} \]

Associative and case commands are using to build tables to drive the decompiler.
20.3. Decoding Different Classes Of Commands

There are several types or classes of commands which execute differently and thus require different actions to decode them. The decompiler does not have to do much other than printing the names of the commands and taking care of the additional information compiled into the object code with the command.

DEFER (SEE) A deferred word vectored to decompile deferred words.
HIDDEN DEFINITIONS Hide all the supporting words in the HIDDEN vocabulary.

: .WORD ( ip -- ip+2 ) Display the name of a colon word and increase the ip by 2.
  DUP @
  >NAME .ID
  2+
  ;

: .INLINE ( ip -- ip+4 ) Display an inline literal and its value.
  .WORD
  DUP @
  2+ Increment ip again.
  .

: .BRANCH ( ip -- ip+4 ) Display a word that has an inline branch address.
  .WORD
  DUP @ OVER - .
  2+ Increment ip again.
  ;

: .QUOTE ( ip -- ip+4 ) Handle the special case of COMPILE xxx .
  .WORD
  .WORD
  ;

: .STRING ( ip -- ip' ) Display a word with inline string argument.
  .WORD
  COUNT 2DUP TYPE
  SPACE
  + Add the string length to ip to skip over the inline string.
  EVEN Align the cell boundary.
  ;

: DOES? ( ip -- ip' f ) Increment simulated ip and return a true flag if DODOES is called as the first instruction in the parameter field.
  DUP 3 +
  SWAP @
  DOES-OP =
  ;
Figure 20.1  Decoding different types of commands.

: .(;CODE) ( ip -- ip' )  Decompile a DOES> word.
  .WORD   Print name.
  DOES?   Is it a DOES> word?
  IF ." DOES> "   Yes. Print DOES>.
  ELSE DROP FALSE  Otherwise, replace ip with a 0.
  THEN
  .UNNEST ( ip -- 0 )  End of a colon definition.
  ." ; "  Print ; .
  DROP 0  Replace ip with 0.
  .FINISH ( ip -- 0 )  Display current word and quit.
  .WORD  Replace ip with 0, indicating end of decompilation.

<table>
<thead>
<tr>
<th>Association Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(LIT)</td>
</tr>
<tr>
<td>?BRANCH</td>
</tr>
<tr>
<td>BRANCH</td>
</tr>
<tr>
<td>(LOOP)</td>
</tr>
<tr>
<td>(+LOOP)</td>
</tr>
<tr>
<td>(DO)</td>
</tr>
<tr>
<td>Compile</td>
</tr>
<tr>
<td>(&quot; )</td>
</tr>
<tr>
<td>(ABORT&quot;)</td>
</tr>
<tr>
<td>(:CODE)</td>
</tr>
<tr>
<td>UNNEST</td>
</tr>
<tr>
<td>( )</td>
</tr>
<tr>
<td>(?DO)</td>
</tr>
<tr>
<td>(.USES)</td>
</tr>
<tr>
<td>All others</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>.INLINE</td>
</tr>
<tr>
<td>.BRANCH</td>
</tr>
<tr>
<td>.BRANCH</td>
</tr>
<tr>
<td>.BRANCH</td>
</tr>
<tr>
<td>.BRANCH</td>
</tr>
<tr>
<td>.QUOTE</td>
</tr>
<tr>
<td>.STRING</td>
</tr>
<tr>
<td>.STRING</td>
</tr>
<tr>
<td>.(CODE)</td>
</tr>
<tr>
<td>.(UNNEST)</td>
</tr>
<tr>
<td>.STRING</td>
</tr>
<tr>
<td>.STRING</td>
</tr>
<tr>
<td>.BRANCH</td>
</tr>
<tr>
<td>.FINISH</td>
</tr>
<tr>
<td>.WORD</td>
</tr>
</tbody>
</table>
20.4. Sorting and Execution Tables

The associative commands EXECUTION-CLASS collects all the special cases that must be decompiled differently from normal Forth commands like DUP, +, etc. At runtime if the address pointed to by IP matches the address of a command in this table, the corresponding index will be returned. This index will be used to select an execution address in the following case table and the appropriate decompilation function will be invoked. These two tables make up the basic mechanism of this table driven decompiler.

14 ASSOCIATIVE: 14 classes of special compiler words are to be processed.
EXECUTION-CLASS
' (LIT) , Each execution address must be compiled explicitly using , .
' ?BRANCH , ' BRANCH , ' (LOOP) , ' (+LOOP) ,
' (DO) , ' COMPILE , ' .(*) , ' (ABORT") ,
' (;CODE) , ' UNNEST , ' (*) , , (?DO) ,
' (;USES) ,

15 CASE: EXECUTION-CLASS A giant case statement handles the special case decompilation. Each entry corresponds to an entry in the EXECUTION-CLASS associative table. In case of no match, .WORD will be executed, assuming a normal Forth word.
INLINE .BRANCH .BRANCH .BRANCH .BRANCH .BRANCH
.QUOTE .STRING .STRING .(;CODE) .UNNEST .STRING
.BRANCH .FINISH .WORD
;
CASE: must end with a ; , because it uses the colon compiler to do the compiling.

20.5. Decompiling Different Command Classes

When the decompiler is given a command to decompile, it has to determine first which type this command is. If the command is simple, like constant or variable, all the decompiler has to do is to tell you its name. Decompilation is only needed for the more complicated colon commands. Therefore, we need another case and associative table pair to handle different types of commands.

: .PFA ( cfa -- ) Given the code field address of a colon word, decompile the list of execution addresses in its parameter field.
>BODY Transform cfa into pfa.
BEGIN Scan the parameter field.
DUP @ Get an execution address.
EXECUTION-CLASS Identify which class the word belongs.
.EXECUTION-CLASS Decompile it.
DUP Dup the ip or the flag.
0= KEY? OR If it is 0 or a key was pressed, terminate the loop.
UNTIL Otherwise continue decompiling.
DROP Last flag.
;
: .IMMEDIATE ( cfa -- ) Indicate whether the current word is immediate or not.
>NAME Get to the name field.
C@ The count byte at the beginning of the name field.
64 AND Is the precedent bit set?
IF Yes.
" IMMEDIATE" Print that it is immediate.
THEN
;
: .CONSTANT ( cfa -- ) Decompile a constant and print its value.
DUP >BODY ? Print its value first.
." CONSTANT " Print the type.
>NAME .ID And the name.
;
: .:VARIABLE ( cfa -- ) Decompile a variable. Print its location and value.
DUP >BODY . Print its location.
." VARIABLE " Type.
DUP >NAME .ID Name.
." Value = " >BODY ? Value.
;
: .:. ( cfa -- ) Decompile a colon definition.
." " Print the almighty colon.
DUP >NAME .ID Name.
2 SPACES Decompile the parameter field.
DUP .PFA Decompile the interpreter.
;
: .DOES> ( cfa -- ) Decompile a word defined by a CREATE-DOES> defining word.
DUP >NAME .ID Name.
." DOES> " Type.
BODY> Address of the high level runtime code or the interpreter.
.PFA Decompile the interpreter.
;
: .USER-VARIABLE ( cfa -- ) Decompile a user variable. Print its offset from the base of user area and its current value.
DUP >BODY ? Offset.
." USER VARIABLE " Type.
DUP >NAME .ID Name.
." Value = " >IS . Value.
;
: .DEFER ( cfa -- ) Tell the user that this is a deferred word and decompile its current definition.
." DEFERRED " Type.
DUP >NAME .ID Name.
." IS " Deferred.
>IS @ Decompile the vectored word.
;
: .USER-DEFER ( cfa -- ) Tell the user that it is a user deferred word and decompile its current definition.
." USER DEFERRED " Type.
DUP >NAME .ID Name.
." IS " Deferred.
>IS @ Decompile the current definition.
;
: .OTHER ( cfa -- ) Decompile words whose type is not known.
DUP >NAME .ID Print the name first.
DUP @ Contents of code field.
OVER >BODY = Is it pfa?
IF DROP ." is code" Print type.
EXIT Quit because we have no disassembler.
THEN
DUP DOES? IF Is it a 'does' word?
DUP DOES> EXIT Decompile it as a DOES> word.
THEN
2DROP ." is unknown" Tell the truth also.
;
20.6. Command Classification

Different classes of commands are characterized by the inner interpreters which execute the commands. Commands of the same class share the same inner interpreter, whose address is stored in the code field of these commands. Inner interpreters are code routines in the Virtual Forth
Computer and generally they do not have names and cannot be referred to directly. However, we can find the address of an inner interpreter by looking at the code field of any command in the corresponding class.

6 ASSOCIATIVE:
DEFINITION-CLASS

Categorize different classes of words that the decompiler will handle. For each class defined by the same defining word, the code field is identical. Thus standard classes can be recognized.

' QUIT @ ,
' 0 @ ,
' SCR @ ,
' BASE @ ,
' KEY @ ,
' EMIT @ ,

Colon word.
Constant.
Variable.
User variable.
Deferred word.
User deferred word.

7 CASE: .DEFINITION-CLASS

Define a table of routines to handle decompilation of each class of definition.

Colon word decompiler.

20.7. The Decompiler SEE

: ((SEE)) ( cfa -- ) Given an arbitrary code field address, decompile it based upon its definition class. Upon completion, indicate whether or not the word is immediate.

CR DUP DUP @
DEFINITION-CLASS
.DEFINITION-CLASS
.IMMEDIATE

Get the contents of the code field.
Determine the type of definition.
Decompile it.
If it is an immediate word.

' ((SEE)) IS (SEE) (SEE) is a deferred word so that .DEFER and .USER-DEFER can make use of it before it is actually defined. Now patch it in.

FORTH DEFINITIONS

All the above supporting word are defined in the HIDDEN vocabulary. Now switch context back to FORTH and declare it the current vocabulary so that the decompiler word SEE will be available to the user in the FORTH vocabulary.

: SEE ( -- ) SEE <name> decompiles the word whose name follows SEE.

' (SEE) Get the code field address of the word <name>.

Decompile it

<table>
<thead>
<tr>
<th>Association Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEST</td>
</tr>
<tr>
<td>DOCONSTANT</td>
</tr>
<tr>
<td>DOCREATE</td>
</tr>
<tr>
<td>DOUSER-VARIABLE</td>
</tr>
<tr>
<td>DODEFER</td>
</tr>
<tr>
<td>DOUSER-DEFER</td>
</tr>
<tr>
<td>All others</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>.CONSTANT</td>
</tr>
<tr>
<td>.VARIABLE</td>
</tr>
<tr>
<td>.USER-VARIABLE</td>
</tr>
<tr>
<td>.DEFER</td>
</tr>
<tr>
<td>.USER-DEFER</td>
</tr>
<tr>
<td>.OTHER</td>
</tr>
</tbody>
</table>
Figure 20.2    Decompiling different types of commands.
Chapter 21. Printing Utility

The code discussed in this chapter is in UTILITY.BLK, Screens 43 to 48.

The printing utility in F83 is designed for an EPSON printer. Using the compressed character size in the EPSON printer, 6 screens can be squeezed on a single 8.5” by 11” sheet of paper. You can print 6 consecutive screens to a page, or 3 screens of source code with 3 corresponding shadow screens to a page, which is nice to show source and comments side by side. To use these high density printing formats, it requires that your printer can print 128 characters per line. If your printer cannot handle 128 characters per line, the old faithful TRIAD should be used to print 3 screens to a page.

21.1. Variables and Setup

: EPSON ( -- ) Set up the EPSON MX-80 printer to print 132 columns per line.
CONTROL O EMIT Send control O to printer, initialize compressed mode.
;
DEFFER INIT-PR Printer initialization. Default is EPSON printer.
  ' NOOP IS INIT-PR
DEFFER FOOTING Print message at the bottom of a page.
66 CONSTANT L/PAGE Lines per page.
0 CONSTANT LOGO The screen number of the logo screen where copyright notice can be stored and displayed.
VARIABLE #PAGE Current page number during printing.

: PAGE ( -- ) Do a form-feed and start a new page. It also increments the page number and resets line and column numbers.
DOES> Vectored word.
PERFORM Do the form-feed in place of NOOP.
1 #PAGE +! Increment page number.
#LINE OFF Reset line number.
#OUT OFF Reset column number.
;
PAGE Initialize itself.

: FORM-FEED ( -- ) EPSON form feed control character.
CONTROL L EMIT
;
: (PAGE) ( -- ) Print enough linefeeds to get to the next page.
L/PAGE 66 lines.
#LINE @ Current line number.
OVER MIN ?DO Use the lesser of the two.
CR Output that many linefeeds.
LOOP ;

' (PAGE) IS PAGE

: (SEMIT) ( char -- ) Send a character to either printer or the console, but not both.
PRINTING @ IF If printing flag is true,
(PRINT) Send to printer.
ELSE (CONSOLE) Otherwise, send to console.
THEN ;

HIDDEN DEFINITIONS HIDDEN is a vocabulary collecting words for internal usage to avoid cluttering up FORTH vocabulary with all kinds of junk words.

CREATE SCR#S 14 ALLOT An array to hold a screen count and up to six screen numbers to be printed.

: PR-START ( -- ) Initialize all printing functions.
PRINTING ON
*LINE OFF
['] (SEMIT) IS EMIT
SCR#S OFF
1 #PAGE !
INIT-PR
;
: PR-STOP ( -- ) Stop the printer as the character output device.
['] (EMIT) IS EMIT
PRINTING OFF
;

Vectoring the output command EMIT allows us to change the function of EMIT dynamically.
The power of vectored execution is quite vividly demonstrated here. The output character string

21.2. Print Two Screens Side By Side

: TEXT? ( scr# -- f ) Given a screen number, return true if the first character in the screen is printable
and the screen is not blank.
BLOCK DUP C@ Get the first character in screen.
BL ASCII ~ BETWEEN Is this character printable?
IF Yes.
B/BUF -TRAILING Count of non-blank characters.
NIP Drop the buffer address.
0<> Return true if not a blank screen.
ELSE First character nonprintable.
FALSE Push the false flag.
THEN
;
: PR ( scr# -- ) Add a screen to the SCR#S array and also increment the screen count at the beginning
of SCR#S.
DUP CAPACITY >= Is scr# out of range?
IF DROP LOGO THEN Yes. Substitute with logo screen.
1 SCR#S +! Increment the screen count.
SCR#S DUP @ Fetch the screen count.
2* + ! Store scr into the appropriate cell in the SCR#S array.
;
: 2PR ( scr1 scr2 line# -- ) Print the specified line from two screens given on the stack. First the line
in scr1, followed by the line in scr2.
CR DUP 2 .R SPACE Print the line number.
C/L * >R Save the character number.
PAD 129 BLANK Clear the PAD buffer.
SWAP BLOCK Buffer address of scr1.
R8 + Address of first character in the specified line.
PAD C/L CMOVE Copy one line from scr1 to PAD.
BLOCK R8 + Address of the first character of the line in scr2.
PAD C/L + 1+ Second half of PAD buffer with an additional space.
C/L CMOVE Copy the line from scr2.
PAD 129 -TRAILING Print the entire 129 characters.
TYPE
;
: 2SCR ( scr1 scr2 -- ) Print 2 screens across on a page. Call 2PR on a line by line basis.
CR CR 4 SPACE Space between screens.
OVER 4 .R Print header of scr1.
61 SPACES DUP 4 .R Header of scr2.
16 0 DO Scan down 16 lines.
I 2PR Print.
LOOP

158
2DROP Discard the screen numbers.

21.3. Print 6 Screens on a Page

To print 6 screens on one page, one has to manage the screens and also put headings and footings on the page, making it look good and convenient to read.

: P-HEADING ( -- ) Print a heading for each new page.
  CR CR 5 SPACES Top blank.
  " page#" #PAGE ? Print page number.
  8 SPACES
  FILE? CR Print the file name.
  ;

: P-FOOTING ( -- ) Print the footing for each page and also do form feed.
  CR CR 58 SPACES Some space.
  " Forth 83 Model" Footing.
  PAGE Form feed.
  ;

' P-FOOTING IS FOOTING
Figure 21.1  Two printing formats.

: PR-PAGE ( -- )  Print a page worth of screens, 6 to a page without shadows.
P-HEADING  Print the heading.
SCR#S OFF  Reset the screen count.
SCR#S 2+  Address of first screen number to be printed.
3 0 DO  3 screens per column.
DUP @  Screen number for 1st column.
OVER 6+ @  Screen number for 2nd column.
2SCR  Print two screens side by side.
2 +  Next cell in SCR#S array.
LOOP  DROP  Discard the SCR#S pointer.
FOOTING

: PR-S-PAGE ( -- ) Print a page worth of screens with shadows Source screen on the left and shadow screens on the right.

P-HEADING
SCR#S OFF
SCR#S 2+ 3 0 DO
OVER 2+ 0
DUP @
2SCR
4 +
LOOP DROP
FOOTING

: PR-FLUSH ( -- f ) Fill the SCR#S array with LOGO screen if a page is partially filled. Return true flag if there is more to print.

SCR#S @ Screen number.
DUP IF Yes, more screens to print.
BEGIN
SCR#S @ Screen number again.
5 < If screen number is less than 5, the SCR#S must be filled.
WHILE 0 PR Fill the array with 0's.
REPEAT
LOGO PR Put the LOGO screen as the last.
THEN
0<> Return the flag.

21.4. SHOW

There are two versions of SHOW defined in the F83 system to print screen files. One version prints consecutive screens and the other prints 3 screens of source with their respective shadows. The first version is defined in the FORTH vocabulary and the second one in the SHADOW vocabulary so that they are both accessible to you.

1 20 SHADOW SHOW Print source with shadows.
1 20 FORTH SHOW Print source without shadow.

FORTH DEFINITIONS Define the SHOW without shadow screens in the FORTH vocabulary.

: SHOW ( first last -- ) Print 6 consecutive screen s on a page. Blank screens are not printed.
[ HIDDEN ] PR-START Call PR-START in the HIDDEN vocabulary to turn on the printer.
1+ SWAP DO Run through the range of screens.
I TEXT? Is this screen printable?
IF I PR THEN Yes. Include it in the SCR#S array to be printed.
SCR#S @ Get the number of screens in SCR#S array.
6 = Full?
IF PR-PAGE THEN Yes. Print one page.
LOOP
PR-FLUSH Fill the last page.
IF PR-PAGE THEN Print it if necessary.
PR-STOP Turn off the printer.

SHADOW DEFINITIONS Now get the SHADOW vocabulary to define the second version of SHOW with shadow screens.

: SHOW ( first last -- ) Print 3 source screens with their shadow screens.
[ HIDDEN ALSO ] Push HIDDEN into the resident vocabulary array so that other vocabulary can be invoked while HIDDEN is still available for searching.
PR-START Turn on printer.
1+ SWAP DO A valid source screen?
I TEXT? Yes.
IF I PR Put it in SCR#S array.
[ SHADOW ] We need some words in the SHADOW vocabulary.
>SHADOW
PR
THEN
SCR#S @ 6 =
IF PR-S-PAGE THEN
LOOP
PR-FLUSH
IF PR-S-PAGE THEN
PR-STOP
;

ONLY FORTH ALSO
DEFINITIONS
: LISTING ( -- )
  Print the entire current file with shadow screens.
  0  First source screen.
  CAPACITY  Last screen in current file.
  2/ 1-  Last source screen in file.
  [ SHADOW ]  We want the SHOW with shadow, which is in SHADOW vocabulary.
  SHOW  Print the entire file in the shadow screen format.
;

Source screens printed with their corresponding shadow screens side by side serve very well as program reference and documentation.
Part IV.  8086 Specific Utilities

Chapter 22.  Debugger

The source code of the debugger is in CPU8086.BLK, Screen 18 to 20, and in UTILITY.BLK, Screen 49 to 51.

The debugger in F83 is designed to let you single step through the execution of a high level command.  To invoke the debugger, type

```
DEBUG xxx
```

where xxx is the name of the colon command you want to trace.  DEBUG patches the NEXT routine with a special routine DEBNEXT to display the contents of the data stack at every step when DEBNEXT is encountered.  The real single step action occurs only when the command xxx is executed.  When xxx is executed, you will get a single step trace showing the commands within xxx that is about to be executed, and the contents of the data stack.  At each step, you can use the commands C (continue), F (Forth) , and Q (quit) to control the stepping process.

F allows you the execute any Forth command to poke around, until you type

```
RESUME
```

to continue with the debugging.  Q stops the debugging process and restores xxx to its original condition.

22.1.  Low Level Supporting Commands

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUG</td>
<td>The vocabulary holding all the debugger supporting words.</td>
</tr>
<tr>
<td>BUG ALSO DEFINITIONS</td>
<td>Declare BUG as the current vocabulary to add new words to it.</td>
</tr>
<tr>
<td>VARIABLE 'DEBUG</td>
<td>A variable holding the code field address of the word to be traced.</td>
</tr>
<tr>
<td>VARIABLE &lt;IP</td>
<td>Lower limit of tracing range for IP, the interpretive pointer.</td>
</tr>
<tr>
<td>VARIABLE IP&gt;</td>
<td>Upper limit of IP for tracing.</td>
</tr>
<tr>
<td>VARIABLE CNT</td>
<td>Count of tracing through debug NEXT.</td>
</tr>
<tr>
<td>ASSEMBLER HEX</td>
<td>Invoke the assembler to use the LABEL word in ASSEMBLER vocabulary.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEST</td>
<td>A machine code subroutine restoring NEXT back to its original form.</td>
</tr>
<tr>
<td>AX LODS</td>
<td>Load IP into AX and increment IP by 2.</td>
</tr>
<tr>
<td>AX W MOV</td>
<td>Copy IP into W register.</td>
</tr>
<tr>
<td>0 [W] JMP</td>
<td>Indirect jump through W register.</td>
</tr>
<tr>
<td>DNEXT</td>
<td>A copy of NEXT that gets executed during debugging in place of the regular NEXT.</td>
</tr>
</tbody>
</table>
| DEBNEXT    | The debugger’s version of NEXT. If IP is between <IP and IP>, the contents of the execution variable 'DEBUG are executed. The word pointed to by 'DEBUG must
<IP #) IP CMP U> IF
IP> #) IP CMP
U< IP
CNT #) AL MOV
AL INC
AL CNT #) MOV
2 # AL CMP
0= IP
AL AL SUB
AL CNT #) MOV
FNEXT #) CALL
IP PUSH
'DEBUG #) W MOV
0 [W] JMP
THEN
THEN
DNEXT #) JMP
None of the above cases are true. Execute the regular NEXT.

CODE PNEXT ( -- )
Patch the regular NEXT in FORTH to jump to DEBNEXT. This puts us in the debug mode and allows for tracing single steps.

CODE UNBUG ( -- )
Restore FORTH's NEXT to its original condition and disable tracing.

22.2. High Level Trace Commands

BUG ALSO DEFINITIONS
Put the following words in the BUG vocabulary. Only the very last word TRACE needs to be in the FORTH vocabulary for ease of accessing.

: L.ID ( nfa len -- )
Print the name of a word left justified in a field of at least ten characters.

VARIABLE SLOW
When true, step continuously. When false, single step.

: (DEBUG) ( low-addr hi-addr -- )
Prepare to trace the words between the specified range for IP pointer.

: 'UNNEST ( pfa -- pfa' )
From the starting address of a parameter field, find the end of this field indicated by UNNEST.

: TRACE ( ip -- )
Display the contents of the data stack and the name of the next word about to execute in the routine being debugged. It then waits for a key unless SLOW is true. If the key is C, F, or Q, special action is taken; otherwise, a single step
is performed.

>R              Save ip.
.S              Display the data stack.
>R>             Restore ip.
.CR @ >NAME     Get the name field of the word pointed to by ip.
.10 L.ID        Print its name.
.SLOW @ NOT     If SLOW is false,
.KEY? OR        or a key is received, do the following:
>               IF
.SLOW OFF       Reset SLOW flag.
.RES OFF        Reset RESume flag also.
." -->"         Print a prompting message.
.KEY            Wait for a key here.
.UPC            Change the key code to upper case always.
.ASCII C OVER = IF If the key is C,
.SLOW @ NOT     complement SLOW.
.SLOW ! THEN    If this key is F,
.ASCII F OVER = IF If this key is F,
.DROP           throw away the key code,
.BEGIN          and entry a Forth interpreter loop.
.QUERY RUN      Interpret any Forth command.
.RES @          Continue if RES is false.
.UNTIL          THEN
.ASCII Q OVER = IF If the key is Q,
DROP THEN       abort debugger.
.PNEXT          Patch NEXT again to continue tracing the next word.
;
' TRACE 'DEBUG ! Vector 'DEBUG to execute TRACE. This is the function of DEBNEXT.

FORTH DEFINITIONS Put the final debugger commands in the regular FORTH vocabulary so that the user can invoke it conveniently. Other debugging words are hidden in the BUG vocabulary.

: DEBUG ( -- ) Patch NEXT to DEBNEXT and set the range of IP to be debugged.
  2- Convert cfa to pfa.
  DUP [ BUG ] 'UNNEST (DEBUG) Find the IP range.
  Set IP range and patch NEXT.
  ;

: RESUME ( -- 0 ) Turn on RES to enable tracing to continue.
  [ BUG ] RES ON Set RES flag.
  0 Leave a dummy stack item to replace the key code dropped.
  PNEXT Patch NEXT to continue debugging.
  ;

ONLY FORTH ALSO DEFINITIONS Re-initialize the vocabulary searching order.
Chapter 23. Multitasker

The source code of the multitasker is in CPU8086.BLK, Screen 22-23, and in UTILITY.BLK, Screen 52-54.

23.1. Multitasking

Multitasking is the technique to use one computer to do several things at the same time. Most of the microcomputers run rather inefficiently in the single user, interactive mode, because the computer wastes most of its time in waiting you to type in commands. This waiting time can be utilized to perform useful work, like printing a long file, keeping a timer clock, watching over the heater or the air conditioner and other instruments, etc. If properly scheduled, all these activities can be handled by a single computer, allocating each task sufficient time to do its work and still satisfies your programming needs.

Most mainframe computers and minicomputer operating systems have the multitasking function and can support many users and tasks to run at the same time. They use rather complicated hardware and software to schedule and run the tasks, and to manage a host of peripheral devices like disk drives, tape drives, printers, plotters, etc. Scheduling and resource allocation are big headaches in these operating systems, contributing a fair share to the complexity in the operating system.

People generally conceive Forth as a toy language only suitable for single user microcomputers. This is probably due to the limited capability presented in the public domain figForth model, which has become the most widely distributed Forth dialect. However, in many more expensive commercial Forth implementations, especially those developed by Charles Moore himself and later marketed by Forth, Inc. under the trade name polyForth, multitasking was a standard feature using the very simple but effective round-robin scheduling technique.

F83 also includes the elementary commands to implement multitasking. The basic system design takes the task switching into consideration, so that tasks can be easily added to the system when needed. Task switching is very fast because of the brevity of code involved. Here we will go through the entire system to describe the multitasker in great details.

23.2. User Variables and the User Area

Special commands managing multitasking or multiuser Forth system are collected in a special
vocabulary USER. Some of them have the same names but different functions as other regular Forth commands. They have to be used with care. They allocate space for user variables in the user area which is a unique memory area for every task in the system.

VARIABLE #USER Count of the number of user variables allocated in a user area.
VOCABULARY USER Declare the user vocabulary.
USER DEFINITIONS Put all subsequent words in the USER vocabulary.

: ALLOT ( n -- ) Allocate space in the user area for a task.
#USER +! Move the user area pointer forward for n bytes.
;

: CREATE ( -- ) A special header builder for user variables.
CREATE Build a regular header.
#USER @ , Compile the current user area pointer in the parameter field, to be used as the offset of the user variable from origin of the user area.
;USES Get the code field address of the new user variable.
DOUSER-VARIABLE , Patch the code field using DOUSER-VARIABLE as the runtime interpreter for user variables.

: VARIABLE ( -- ) New defining word for user variables.
CREATE This is the newly defined CREATE.
2 ALLOT Allocate two bytes in the user area, not on top of dictionary.
;

: DEFER ( -- ) New defining word for deferred words in the user area.
VARIABLE Create a new user variable.
;USES Patch code field with DOUSER-DEFER, address of the runtime routine DOUSER-DEFER.

When tasks are switched, the environment of the task currently under execution must be preserved before the task is put to sleep, so that when the task is woke up the next time around it will be able to pick the execution sequence where was left off and continue the task until finished or put to sleep again. What defines the environment of a task is a set of parameters stored in a set of 'user variables' and an area where the data stack and the return stack used by the task are allocated. Each task must have its own copy of these user variables and stacks. As the essential information are stored independently for each task, task switching becomes very easy because only a minimal amount of information has to be preserved explicitly during task switching.

Following is the list of user variables needed in every task:

Table 23.1. User Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOS</td>
<td>Top of data stack.</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Entry point to be jumped to when the task is activated.</td>
</tr>
<tr>
<td>LINK</td>
<td>Point to next task in the round-robin circle.</td>
</tr>
<tr>
<td>SP0</td>
<td>Origin of the data stack.</td>
</tr>
<tr>
<td>RP0</td>
<td>Origin of the return stack.</td>
</tr>
<tr>
<td>DP</td>
<td>Top of dictionary.</td>
</tr>
<tr>
<td>#OUT</td>
<td>Number of characters emitted.</td>
</tr>
<tr>
<td>#LINE</td>
<td>Number of lines typed.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Block offset from block 0 in the current file.</td>
</tr>
<tr>
<td>BASE</td>
<td>Numeric base for I/O conversion.</td>
</tr>
<tr>
<td>HLD</td>
<td>Point to the last character converted in the PAD buffer.</td>
</tr>
<tr>
<td>FILE</td>
<td>Point to FCB (file control block) of currently opened file.</td>
</tr>
<tr>
<td>IN-FILE</td>
<td>Point to the PCB of the input file.</td>
</tr>
<tr>
<td>PRINTING</td>
<td>A flag. True if printer is active.</td>
</tr>
<tr>
<td>EMIT</td>
<td>Send a character to the output device currently active.</td>
</tr>
</tbody>
</table>
One should note that these user variables have their names in the main dictionary while the parameters are stored in the user area. Only one copy of the names needs to be defined. Every user or task will have its own copy of the user variables. The functions of these user variables will become apparent in the following sections.

23.3. Pause and Restart

PAUSE and RESTART are the two most crucial commands in the task switching process. (PAUSE) stops the current task and passes control of the CPU to the next task in the round-robin loop. It saves all necessary information in the user area so that the task can continue the next time it gains the control of the CPU.

CODE (PAUSE) Stop executing the current task and pass control to the next task.
IP PUSH Save the Interpretive pointer on the data stack.
RP PUSH Save the return stack pointer.
UP #) BX MOV Fetch the user area pointer into BX register.
SP 0 [BX] MOV Save the data stack pointer in user variable TOS.
BX INC BX INC Increment BX and point it at the user variable LINK.
BX INC BX INC Calculate the address of the user area from the offset in LINK.
0 [BX] BX ADD Now point to the ENTRY user variable of the next task.
BX JMP Execute the next task.
END-CODE

CODE RESTART The reverse of (PAUSE). Retrieve the stored information and start executing the task left asleep during the last pass.
-4 # AX MOV Store -4 in the AX register for later use.
BX POP Pop the LINK address of the next task to be waked up.
AX BX ADD Point BX to the TOS user variable.
BX UP #) MOV Discard two items from the stack. They were pushed on the stack by the interrupt routine waking up this task.
0 [BX] AX POP Restore the data stack pointer.
RP POP Restore the return stack pointer.
IP POP Restore the interpretive pointer.
NEXT IP contains the pointer pointing to the next word to be executed in this task.
NEXT will continue the execution left off last time.
END-CODE

CODE PAUSE A dummy word allowing multitasker to be switch on and off.
NEXT In the single user mode, PAUSE returns immediately without doing anything.
END-CODE However, its code field will be patched so that the word. (PAUSE) will be invoked in the multitasking mode of operation.

A few more commands are defined to manipulate data stored in the user area to control the tasks:

HEX 80 CONSTANT INT# 8086 software interrupt number, used to wake up a task.
Figure 23.1 The round-robin task scheduler.

: LOCAL
   ( base addr -- addr1 )
   Get the address of a user variable in another task's user area.
   UP @
   The origin of current user area.
   -
   Offset of current user variable from its origin.
   +
   Add offset to the origin of the other user area (base), returning the address
   of the same user variable in the other task.
   ;

: @LINK
   ( -- addr )
   Return a pointer to the entry point in the next task.
   LINK
   Address of LINK in this task.
   DUP @ +
   Get the ENTRY in the next task.
   2+ LINK field in the next task.
   ;

: !LINK
   ( addr -- )
   Set the LINK field of the current task, given the origin of the user area of the
   next task.
   LINK -
   The relative distance from LINK to the other task.
   2+ Relative distance from this LINK to the other LINK.
   LINK !
   Store it in the current LINK.
   ;

: SLEEP
   ( addr -- )
   Make the addressed task pause indefinitely.
   E990
   90 is NOP and E9 is JMP in 8086 machine instruction. JMP (link) passes CPU to
   the next task whose address is in the LINK field right after the ENTRY field.
   SWAP
   Get the task address to top.
   ENTRY LOCAL
   Get the address of the ENTRY field in the target task.
   !
   Store the sleep code in its ENTRY field and force that task to pass control
   immediately to its next task.
   ;

: WAKE
   ( addr -- )
   Wake up a task so that it will execute in its next turn.
   80CD
   Machine instruction INT 80H which wakes up this task by a software interrupt with
   vector number 80H.
   SWAP
   Get the task's user area address.
   ENTRY LOCAL !
   Store that wake code in the ENTRY field of the target task and make it an active
member in the round-robin loop.

;  

: STOP { -- )  Make the current task pause indefinitely.
UP @  Get the address of the current task.
SLEEP  Put it to sleep.
PAUSE  Stop right now.

;

23.4. The Multitasker

The Forth multitasker is implemented using a round-robin scheduler and dispatcher technique. All tasks are linked into a loop. A task must use the PAUSE command explicitly to relinquish CPU control to the next task. If a task does not need the CPU service, it will pass the CPU control directly to the next task. When a task needs CPU, it will put a wakeup code in the ENTRY field in its user area. Next time when the control of CPU is passed to it, it will be able to grab the CPU and restart or continue on the execution left off last time. Each task therefore must include PAUSEs at suitable intervals to let other tasks have a piece of the action. Otherwise a task can hold onto the CPU indefinitely and no other task can use the CPU. The cooperative nature of this scheme thus requires that each task be designed to pause regularly. The advantage is that each task can stop and restart at known points and the commands to do the multitasking are simple and fast.

CODE MULTI ( -- )  Install the multitasker's schedule/ dispatcher loop by patching the appropriate interrupt vector and enabling PAUSE.
' (PAUSE) @ # BX MOV  Copy the contents of code field of (PAUSE) to BX register. It is the starting address of the (PAUSE) code routine.
BX ' PAUSE #) MOV  Patch the code field of PAUSE with code of (PAUSE) so that the task will pause at PAUSE.
' RESTART @ # BX MOV  Get the RESTART code address.
DS AX MOV
AX PUSH  Save DS register on data stack.
AX AX SUB  Clear AX register.
AX DS MOV  Clear DS register.
CS AX MOV  Copy code segment into AX.
AX INT# 4 * 2+ #) MOV  Store the code segment in the second cell of the interrupt vector for this task.
BX INT# 4 * #) MOV  Store the RESTART address in the first cell of the interrupt vector. Hereafter, INT 80H will activate this task
AX POP
AX DS MOV
NEXT
END-CODE

;  

: SINGLE ( -- )  Remove the multitasker's scheduler/dispatcher loop.
[*] PAUSE >BODY  Get the parameter field address of PAUSE, which points to a simple NEXT routine.
[!] PAUSE !  Restore the code field of PAUSE so that PAUSE will return immediately rather than go through (PAUSE) for the multitasker.

;

23.5. Task Definition

A task must be first defined as a command in the dictionary. At the same time, user area and space for two stacks must also be allocated for this task. The new task must then be installed in the round-robin loop. When functions must be performed by this task, a command to be executed by this task must be passed to it and the task must be waken. The tools to create and activate new
tasks are discussed in this section.

```lisp
: TASK: ( size -- )  Create a new named task and initialize its user variables.
CREATE   Build an header for the task.
TOS      Address of the user area for the current task.
HERE     The user area of the new task.
#USER @  Size of the user area.
@LINK    New user area.
UP @ -ROT Save the current user area pointer at the bottom of the data stack.
HERE UP ! Put new user area address in the user area pointer UP.
!LINK    Store address of current user area in the LINK field of the new task.
DUP HERE + Address of the space after the new user area and dictionary space.
DUP RP0 ! Initialize the return stack pointer of the new task.
100 - SP0 ! Initialize the data stack pointer of the new task.
SWAP UP ! Restore the user area pointer to the current user area.
HERE ENTRY LOCAL !LINK Store the address of the new task in the LINK field of the current task.
HERE #USER @ + Starting address of the dictionary for the new task.
HERE DP LOCAL ! Initialize the dictionary pointer of the new task.
HERE SLEEP Put the new task to sleep first.
ALLOT   Allocate space for the new task according to the size parameter given on the stack.

: SET-TASK  ( ip task -- )  Assign an existing task to execute the code pointed to by ip on stack.
DUP SP0 LOCAL @ Get the top of data stack of the task to be used.
2- Decrement stack pointer preparing a push.
ROT OVER ! Push ip on data stack.
2- Prepare another push.
OVER RP0 LOCAL @ Get the new tasks return stack pointer.
OVER Push it on the data stack.
SWAP TOS LOCAL ! Store the data stack pointer in the TOS field of the new task. These actions simulate (PAUSE) in a way that when the new task is waken the code pointed to by ip will be executed.

: ACTIVATE ( task -- )  Assign the invoked task to execute the following code, and wake the task making it ready to execute these code.
R> Address of the next code to be executed.
OVER SET-TASK Assign this code to the task.
WAKE wake the task.

23.6.  Background Tasks

A background task is some functions the computer does in the background, while still allowing you to use the computer interactively doing programming or execute other programs. F83 allows you to create many of such tasks to run concurrently, using the task defining and control commands. Here are some examples illustrating the command sequence to create background tasks and activate them.

: BACKGROUND: ( -- )  Create a new task with 400 bytes of dictionary space. Initialize this task to execute the following code.
400 TASK: Define a new task with the name following.
HERE IP for code to be compiled so that it can be executed by the new task.
@LINK 2- Address of the new task.
!CSP Compiler error checking initialization.
| Turn on the compiler to compile following code to be executed by the new task.

: BACKGROUND: SPOOLER ( -- ) Create a new task named spooler which will print the current file.
1 First screen to be listed.
CAPACITY Last screen in the current file.
SHOW List the entire file.
```
Executing SPOOLER will print out the entire current file on the printer while you can still use the terminal for normal activities. Low level input/output commands in SHOW contain enough PAUSE commands to alternate the spooling task with the terminal task to allow you seemingly full control over the computer while it is also printing a long file. SPOOLER can also be assigned a different function by the following command:

```
:SPOOL-THIS { -- }  Assign new functions to an existing task.
SPOOLER ACTIVATE Force the task SPOOLER to execute the following code:
3 15
{ SHADOW } SHOW Invoke the special SHOW command in the SHADOW vocabulary to print shadow screens along the source screens.
STOP Terminate the task.
```

Another example is to use the computer to keep a counter which keeps the number of circles the multitasker running around the round-robin loop:

```
VARIABLE COUNTS
BACKGROUND: COUNTER Define a new background task.
BEGIN Beginning of an infinite loop.
PAUSE Allow other tasks to run once.
1 COUNTS +! Increment the counter.
AGAIN ;
```

The task COUNTER executes an infinite loop, so STOP is not required at the end of the command. However, note that you must use PAUSE in the loop, or no other task will be executed. PAUSE is built in all the input/output commands, so that tasks which do I/O like SPOOLER do not have to use PAUSE explicitly.
Chapter 24.  8086 Assembler

The source code of the assembler is in CPU8086.BLK, Screen 3 to 17, and KERNEL86.BLK, screen 80.

The assembler in Forth allows you to define commands which will be executed at the raw machine speed and make use of all the resource in the host computer hardware system. It is often used to optimize a system or application program by recoding the critical or most often executed commands to improve the performance of the program. A high level command can be substituted by a machine code command if the interface to the system, most notably the stack effect is kept the same.

The assembler is invoked by the defining command CODE which creates a header in the dictionary and makes the code field pointing to the parameter field. Machine instructions can then be assembled into the parameter field by a set of machine code commands with mnemonic names similar to those used in regular assembler of the host processor. These machine code commands are executed by the text interpreter and the net effect is to add new machine instructions to the parameter field so that when the new CODE command is executed, these machine instructions are executed in sequence. A CODE command must be terminated by the inner interpreter NEXT or its derivative to return control to the calling command. The code command must be terminated with a command like END-CODE or C:, which makes the new command available for execution or compilation.

24.1. Assembly Tools

A set of tool commands are needed to assemble a machine code command in its most primitive form. If you know the host processor well enough, you can hand code a routine and generate a Forth CODE command without using the assembler.

```
VARIABLE AVOC
: CODE ( -- )  The defining word that starts the assembling process to build a machine code word.
CREATE
HIDE
HERE DUP 2- ! Store the pfa into code field. This is required for indirectly threaded code.
CONTEXT @ AVOC ! Save the old context vocabulary.
ASSEMBLER
;
: LABEL ( -- )  Mark the start of a subroutine. Return its address when the label is invoked.
CREATE
ASSEMBLER
;
232 CONSTANT DOES-OP  Op-code for CALL instruction used in DOES>.
3 CONSTANT DOES-SIZE  A CALL instruction consumes 3 bytes.
```
Return a true flag with the IP moved over the CALL instruction. 

DOES? ( ip -- ip+3 f )  
DUP DOES-SIZE +  
SWAP C@  
DOES-OP =  
;

ASSEMBLER DEFINITIONS  
All the assembler tool words are to be collected in this vocabulary.

: END-CODE ( -- )  
Terminate a code definition, make it available and also restore context vocabulary. 

AVOC @ CONTEXT !  
Restore the old context vocabulary. 

REVEAL  
Unsmudge the header, making the word available to the text interpreter. 

: C; ( -- )  
Synonym for END-CODE.

The following set of deferred commands are used in assembling machine instructions or constructing structures in the code commands. They are defined as deferred commands so that they can be shared by the assembler and the metacompiler. 

DEFER C, ( byte -- )  
Assemble a machine code byte to the dictionary.

FORTH ' C, ASSEMBLER IS C,  
Vector it to the FORTH C, .

DEFER , ( n -- )  
Assemble a cell to the dictionary.

FORTH ' , ASSEMBLER IS ,  
Vector to the FORTH , (comma).

C, and , are the primitive Forth assembler. Using them alone, you can generate code commands without using an assembler. However, the machine instructions and operands have to be hand coded and stored into the dictionary by , and C, .

DEFER HERE ( -- addr )  
Return a pointer to the top of dictionary, the address of the next code to be assembled.

FORTH ' HERE ASSEMBLER IS HERE

DEFER ?>MARK  
Set up forward branch with error checking.

DEFER ?>RESOLVE  
Resolve a forward branch with error checking.

DEFER ?<MARK  
Set up a backward branch with error checking.

DEFER ?<RESOLVE  
Resolve a backward branch with error checking.

24.2. 8086 Register Definitions

Registers are used as operands, which are inserted into the register field in a machine instruction. They are thus defined as constants. The register constants are defined in the following format to facilitate the building of machine instructions which uses the corresponding register, as shown in Fig. 24.1. The lower byte may become a byte instruction or the second instruction byte to specify addressing mode. The upper byte with the mode field is used only during assembly.
175

Figure 24.1  Register addressing mode constant.

Using the powerful REGS, we can define all the registers with all possible addressing modes as distinct Forth commands. They return the appropriate register constants to be used by machine code assembler commands to construct machine instructions.

10 0 REGS
AL CL DL BL AH CH DH BH
Define operands addressing only bytes in the registers.

10 1 REGS
AX CX DX BX SP BP SI DI
Define operands addressing word registers.

10 2 REGS
[BX+SI] [BX+DI] [BP+SI]
[BP+DI] [SI] [DI] [BP] [BX]
Define indexed addressing operands.

4 2 REGS
[SI+BX] [DI+BX] [SI+BP]
Duplicated definitions.
Four registers are of special interests to the Forth system because they are used as registers in the Virtual Forth Computer. They are assigned generic names more appropriate in the manipulation of the Forth machine:

BP CONSTANT RP
[BP] CONSTANT [RP] Indirect addressing mode.
SI CONSTANT IP
[S1] CONSTANT [IP] The interpretive pointer.
BX CONSTANT W

The data stack pointer uses the SP register which already has the correct name; therefore, it does not need a new name.

24.3. Addressing Mode Operators

: MD ( mode -- ) Define words which will test various addressing modes.
   CREATE Make header.
   1000 * , Compile a template of addressing mode.
   DOES> ( mode-field -- f )
   .@ Get the template
   .SWAP 7000 AND Mask over the mode field
   .= 0<> Return true if the mode matches
   .;

0 MD R8? ( operand -- f ) Is it in byte register mode?
1 MD R16? ( operand -- f ) Is it in word register mode?
2 MD MEM? ( operand -- f ) Is it in memory addressing mode?
3 MD SEG? ( operand -- f ) Is it in segment addressing mode?
4 MD #? ( operand -- f ) Is it in immediate addressing mode?

: REG? ( operand -- f ) Test for either byte or word addressing mode.
   7000 AND Mask the mode field.
   2000 < Byte or cell.
   < 0<> Return the flag.
   .;

: BIG? ( n -- f ) Test the size of address offset. Return true if n>255.
   ABS Absolute offset.
   -200 AND Examine upper byte.
   0<> True if not zero.
   .;

: RLOW ( n1 -- n2 ) Retain only the low r/m field.
   7 AND
   .;

: RMID ( n1 -- n2 ) Retain only the middle register field.
   70 AND
   .;

VARIABLE SIZE A flag. True for 16 bit and false for 8 bit number.

: BYTE ( -- ) Reset SIZE to indicate byte operations.
   SIZE OFF
   .;

: OP, ( n opcode -- ) OR the operand and the opcode and assemble the machine code.
   OR C
   .;

: W, ( opcode operand -- ) Assemble opcode with the W field set if operand indicates a word register.
177

R16? 1 AND
OP,
Set W field according to word mode.
;
SIZE, { opcode -- }
SIZE @ 1 AND
Assemble the opcode with W field determined by SIZE.
OP,
Set W field using SIZE.
;
; ,/C, { n f -- }
IF, Assemble a cell if f is true. Otherwise assemble a byte.
ELSE C,
f is false. Assemble a byte.
THEN ;
: RR, { operand1 operand2 -- }
Assemble a register to register instruction.
RMID
Operand1 to r/m field.
SWAP RLOW
Operand2 to reg field.
300
Register to register operand.
OP,
Assemble the reg-reg second instruction byte for addressing.
;
VARIABLE LOGICAL
True while assembling logical instructions.
: B/L? { n -- f }
BIG? of LOGICAL.
BIG? LOGICAL @ OR
BIG? = 1 OR
;
: MEM, { disp mr rmid -- }
Assemble a memory reference instruction. It takes a displacement, and
memory/ register, and a register a arguments and encode them into an
instruction.
OVER #) = Is it in immediate indirect mode?
IF
RMID
Assign immediate indirect instruction.
DROP
No need of mr now.
ELSE
Not immediate indirect.
RMID OVER RLOW OR
OR together the registers.
-ROT
Save it.
[BP] = mr=[BP]?
OVER 0= AND
AND disp=0?
IF
SWAP
Get the register field to top.
100 OP,
Byte displacement mode instruction. Mode 1.
C,
With the byte displacement.
ELSE
SWAP
Examine disp.
OVER BIG?
More than 8 bits?
IF
200 OP,
Mode 2 instruction.
C,
With cell displacement.
ELSE
OVER 0=
Is disp=0?
IF
C,
Assemble byte instruction.
DROP
No displacement.
ELSE
100 OP,
All tests failed to reach here.
C,
Assemble mode 1 instruction anyway.
THEN
THEN
THEN
THEN
;
: WMEM, { disp mem reg op -- }
Assemble a word memory reference instruction.
OVER W,
Pack the word referencing bit into reg and assemble opcode.
MEM,
Use MEM, to assemble the right mode instruction.
;
: R/M { mr reg -- }
Assemble either a register to register or register to memory instruction.
OVER REG?
Is it in a register mode?
IF RR,
Yes. Assemble a register to register instruction.
ELSE MEM,
Else assemble a memory referencing instruction.
THEN ;
: WR/SM ( rm reg op -- )  Assemble either a register mode instruction with size field, or a memory
mode instruction with size from SIZE.
    2 PICK  Get the mode.
    DUP REG?  Is it register mode?
    IF  Yes.
        W,  Squeeze in the word bit.
        RR,  Assemble a register-register instruction.
    ELSE  Not register mode.
        DROP  Discard mode.
        SIZE,  Use SIZE for word bit.
        MEM,  Assemble memory instruction.
    THEN
        SIZE ON  Set default size to 16 bits.
    ;

VARIABLE INTER  True if doing inter-segment jump, call, or return.

: FAR ( -- )  Set INTER true.
    FAR ON ;

: ?FAR ( n1 -- n2 )  If INTER is true, set the far bit in the instruction.
    INTER @  If INTER is true,
    IF 10 OR THEN  set bit 3 in the instruction.
    INTER OFF  Reset far flag.
    ;
Figure 24.2  8086 instruction types.
24.4. Defining Commands To Generate Opcodes

8086 is a rather complicated microprocessor. It was designed to be 8080 downward compatible so that it must be able to execute all the 8080 instructions. With lots of 16 bit machine instructions and operators for the extra registers and different mode of operation, the instruction set becomes very big. Consequently, the assembler also becomes complicated in order to take care of these diverse types of instructions. There are 14 identifiable classes of instructions in 8086. A defining command is used to generate opcodes for each class of instructions.

: 1MI (opcode --) Define one byte constant instructions.
CREATE C,
Create header and compile the opcode.
DOES> C
Fetch opcode from the parameter field of the assembler word.
C,
Assemble it.
;
HEX
37 1MI AAA 3F 1MI AAS 98 1MI CBW F8 1MI CLC
FC 1MI CLD FA 1MI CLI F5 1MI CMC 99 1MI CWD
27 1MI DAA 2F 1MI DAS F4 1MI HLT CE 1MI INTO
CF 1MI IRET 9F 1MI LAHF F0 1MI LOCK 90 1MI NOP
9D 1MI POPF 9C 1MI PUSHF F2 1MI REP F2 1MI REP2
9E 1MI SAHF FD 1MI STD FB 1MI STI
9B 1MI WAIT D7 1MI XLAT

OCTAL
: 2MI (opcode --) Define ASCII instructions.
CREATE C,
Header and parameter field.
DOES> C
Assemble the opcode.
C,
Assemble the ASCII mode byte.
;
HEX D5 2MI AAD D4 2MI AAM
OCTAL

: 3MI (opcode --) Define branch instructions with one byte offset.
CREATE C,
DOES> C
Assemble opcode.
HERE - 1-
Offset from current address.
C,
Assemble the offset.
;
HEX 77 3MI JA 73 3MI JAE 72 3MI JB 76 3MI JBE
E3 3MI JCXZ 74 3MI JECX 7F 3MI JG 7D 3MI JGE
7C 3MI JL 7B 3MI JLE 75 3MI JNE 71 3MI JNO
79 3MI JNS 70 3MI JOD 7A 3MI JPE 78 3MI JPO
78 3MI JS 82 3MI LOOP E1 3MI LOOPZ E0 3MI LOOPNZ
OCTAL

: 4MI (opcode --) Define LDS, LEA, LES instructions.
CREATE C,
DOES> C
Assemble opcode.
MEM,
Memory reference.
C
;
HEX C5 4MI LDS 8D 4MI LEA C4 4MI LES
OCTAL

: 5MI (opcode --) Define string instructions.
CREATE C,
STORE opcode.
DOES> C
Assemble opcode with size bit.
SIZE ON
Enable word addressing.
;
HEX   A6 5MI CMPS   A4 5MI MOV$   AE 5MI SCAS   OCTAL

: 6MI ( opcode -- ) Define string instructions where byte/word mode is determined at assembly time.
CREATE C,
STORE opcodes.
DOES> ( mr -- )
C@ Opcode.
SWAP W,
Use mr to decide the word bit and assemble accordingly.
;
HEX   AD 6MI LODS   AA 6MI STOS   OCTAL

: 7MI ( opcode -- ) Define multiply and divide instructions.
CREATE C,
STORE opcode.
DOES> ( r/m -- )
C@ The opcode will be put in the reg field of the second byte.
366 The real first byte opcode.
WR/SM,
Assemble the whole mess.
;
HEX   30 7MI DIV   38 7MI IDIV   28 7MI IMUL   20 7MI MUL
.O1 7MI NOT   OCTAL

: 8MI ( opcode -- ) Define input/output instructions.
CREATE C,
Opcode.
DOES> ( port -- )
C@ Opcode.
OVER R16?
1 AND OR
1 OR Set the port# a 16 bit number?
OR the word bit to opcode.
OVER # =
IF Yes, a port# is given.
C,
Assemble opcode.
C,
Assemble port number.
ELSE
Implied port.
10 OR
Set the implied port bit in opcode.
C,
Assemble one byte i/o instruction.
THEN
;
HEX   E4 8MI IN   E6 8MI OUT   OCTAL

: 9MI ( opcode -- ) Define increment/decrement instructions.
CREATE C,
STORE opcode.
DOES> ( reg -- )
C@ Get opcode first.
OVER R16?
Mode 1 operation?
IF Yes.
100 OR
Opcode for one byte inc/dcr instruction.
SWAP RLOW
Retain only r/m field.
OP,
Assemble one byte instruction.
ELSE
Other modes.
376
First byte opcode.
WR/SM,
Use stored opcode as second byte instruction.
THEN
;
HEX   08 9MI DEC   00 9MI INC   OCTAL

: 10MI ( opcode -- ) Define Shift/rotate instructions.
CREATE C,
STORE opcode.
DOES> ( reg -- , or reg CL -- )
C@ Stored opcode.
OVER CL =
Top register is CL?
IF Multiple bit shift.
NIP
Discard CL because it is implied.
322
Number of bits shifted in CL.
ELSE
Single bit shift.
320
THEN
WR/SM,
Assemble the two-byte instruction.
;
HEX   10 10MI RCL   18 10MI RCR   00 10MI ROL   8 10MI ROR
.38 10MI SAR   20 10MI SHL   28 10MI SHR   OCTAL
: 11MI ( opcode1 opcode2 -- ) Define call/jump instructions.
CREATE Header.
C, Indirect call/jmp opcode.
C, Direct call/jmp opcode.
DOES> ( addr -- ) Immediate address?
OVER #) =
IF Yes.
NIP Discard #) mode operator.
C@ Get the opcode.
INTER @ IF If it is intersegment addressing,
1 AND and a Jump?
IF 352 Yes. Jump opcode.
ELSE 232 THEN No. Call opcode.
C, Compile jmp/call opcode.
SWAP , , Compile offset and segment.
ELSE Not intersegment addressing.
SWAP Target address addr.
HERE - 2- Displacement.
SWAP ( disp opcode -- )
2DUP 1 AND Is it JMP?
SWAP BIG? NOT AND And disp<256?
IF If so, assemble short jump.
2 OP, Short jump opcode.
ELSE Long jump or call.
C, Opcode.
1- Offset for three-byte instruction.
, Long displacement.
THEN THEN
ELSE Not immediate addressing.
DUP S#) = Is it intrasegment addressing?
IF DROP #) THEN Yes. Restore the immediate address code.
377 C, Assemble opcode.
1+ C@ Get the initial r/m mode code.
?FAR Add the intersegment far bit if necessary.
R/M Append it to the opcode.
THEN
;
HEX 10 EB 11MI CALL 20 E9 11MI JMP OCTAL

: 12MI ( reg-op seg-op r/m-op -- ) Define push and pop instructions.
CREATE Header.
C, C, C, Store three different opcodes for push or pop.
DOES> ( reg -- )
OVER REG? Register mode?
COUNT >R Fetch and store opcode1.
C@ LOGICAL ! Save opcode2 in LOGICAL.
REG? Is operand2 a register?
DUP RLOW OP, Assemble it.
ELSE
1+ Point to the second opcode.
OVER SEG? Segment register mode?
IF Yes.
C@ Register mode opcode.
SWAP RLOW OP, Assemble it.
ELSE
1+ Point to the second opcode.
OVER REG? Register mode?
IF Yes.
C@ Get segment opcode.
RLOW Save only r/m field.
SWAP R MID Put in the reg field.
OP, Assemble.
ELSE COUNT Get second opcode and point to the third opcode.
SWAP C0 Get the third opcode.
C, MEM, Assemble the third opcode as the first byte of instruction.
THEN THEN
;
HEX 8F 07 58 12MI POP FF 36 50 12MI PUSH OCTAL

: 13MI ( op1, op2 -- ) Define arithmetic and logic instructions.
CREATE Header.
C, C, Store opcodes.
DOES> ( operand1 operand2 -- )
COUNT >R Fetch and store opcode1.
C@ LOGICAL ! Save opcode2 in LOGICAL.
DUP REG? Is operand2 a register?
IF Yes. 
OVER REG? Is operand1 also a register?
IF Yes. A reg-reg math/logic operation. 
R> Get opcode1. 
OVER W, Assemble opcode1 with w field. 
SWAP RR, Assemble addressing byte. 
ELSE Operand1 is not a register. 
OVER DUP MEM? Memory reference?
SWAP #) = OR Or memory indirect? 
IF Yes. 
R> 2 OR Assemble opcode1 with direction field. 
WMEM, Memory reference. 
ELSE Not memory referencing. 
NIP Discard operand1. 
DUP RLOW 0= Is operand2 the accumulator? 
IF Yes. 
R> 4 OR One byte math instruction. Fill the math field (bit 2). 
OVER W, Fill in the w field. 
R16? ,/C, Assemble the byte or word immediate value. 
ELSE Operand2 is not the accumulator. 
OVER B/L? Big and long? 
OVER R16? Operand2 a 16 bit register? 
2DUP AND True for 16 bit logic instruction. 
-ROT Save the flag. 
1 AND W field. 
SWAP 16 bit-logic flag. 
NOT 2 AND Sign extension field. 
OR Combine s and w fields. 
200 OP, Assemble first byte opcode. 
SWAP RLOW r/m field. 
300 OR Mode 3. 
R> OP, Second byte mode instruction. 
,/C, Third byte or word value. 
THEN 
THEN 
ELSE Operand2 is not a register. 
ROT DUP REG? Is operand1 a register? 
IF R> WMEM, Assemble memory referencing math instruction. 
ELSE Not memory referencing. Must be immediate value. 
DROP It is not a register. Discard it because it is a memory code. 
2 PICK Pick the displacement. 
B/L? Larger than 255? 
DUP NOT 2 AND Fill in the s field. 
200 OR SIZE, Assemble first instruction with w field. 
-ROT Save the BIG? flag. 
R> MEM, Assemble the mode byte. 
SIZE @ Must be BIG and word size. 
AND ./C, Assemble the immediate value. 
SIZE ON Reinitialize SIZE to 16 bit. 
THEN 
THEN 
HEX 
0 10 13MI ADC 0 00 13MI ADD 2 20 13MI AND 
0 38 13MI CMP 2 08 13MI OR 0 18 13MI SBB 
0 28 13MI SUB 2 30 13MI XOR OCTAL 
; 14MI (--) Returns. 
CREATE C, Compile the opcode. 
DOES> C# Get opcode. 
DUP ?FAR Add the intersegment bit if necessary. 
C, Assembler opcode. 
1 AND 0= If it has immediate offset, 
IF , THEN Assemble the address offset. 
; 
HEX C3 14MI RET C2 14MI +RET OCTAL 

24.5. Special Opcodes

A small number of instructions do not belong to any of the above types. They are defined
individually as colon commands which have to do special assembly work to assemble their respective machine instructions.

**HEX**

: ESC  ( source opcode -- )  
Escapse to external device.

: RLOW  ( -- )  
Retain only the low register field.

: 0DB OP,  ( -- )  
Assemble the ESC opcode.

: R/M,  ( -- )  
With the r/m code.

: INT  ( n -- )  
Assemble an interrupt instruction.

: OCD C,  ( -- )  
INT, interrupt instruction.

: C,  ( n -- )  
Interrupt vector number.

: SEG  ( seg -- )  
Assemble a segment instruction.

: RMID  ( seg -- )  
Mask over the segment field.

: 26 OP,  ( -- )  
Opcode for segment instruction.

: XCHG  ( mr1 mr2 -- )  
Assemble register exchange instruction.

: DUP REG?  ( -- )  
m2 a register?

: IF DUP AX =  ( -- )  
And the AX register?

: IF  ( -- )  
m2=AX.

: DROP  ( -- )  
AX is implied.

: RLOW 90 OP,  ( -- )  
Assemble opcode 90 with m1.

: ELSE  ( -- )  
m2 is not AX.

: OVER AX =  ( -- )  
Is Ax AX?

: IF  ( -- )  
m1=AX.

: NIP  ( -- )  
No need of m1 anymore.

: RLOW 90 OP,  ( -- )  
Assemble XCHG with m2.

: ELSE  ( -- )  
Neither is AX.

: 86 WR/SM,  ( -- )  
Assemble XCHG with a mode byte.

: THEN  ( -- )  

: ELSE  ( -- )  
m2 is not a register.

: ROT 86 WR/SM,  ( -- )  
Assemble XCHG with mode byte.

: CS:  ( -- )  
Code segment override.

: DS:  ( -- )  
Data segment override.

: ES:  ( -- )  
Extra segment override.

: SS:  ( -- )  
Stack segment override.

: MOV  ( source dest -- )  
Assemble a MOV instruction, the most complicated instruction in 8086.

: DUP SEG?  ( -- )  
Is dest a segment register?

: IF 8E C,  ( -- )  
Assemble segment MOV,

: R/M,  ( -- )  
and the mode byte with source.

: ELSE DIP REG?  ( -- )  
Is dest a register?

: IF  ( -- )  
Dest is a register.

: OVER # =  ( -- )  
Source is from memory?

: OVER RLOW 0=  ( -- )  
And dest is AX?

: IF AD SWAP W,  ( -- )  
Yes. Assemble mem to AX MOV,

: DROP  ( -- )  
discard dest, and

: ,  ( -- )  
assemble memory address.

: ELSE OVER SEG?  ( -- )  
Is source a segment register?

: IF  ( -- )  
Yes.

: SWAP 0C C,  ( -- )  
Assemble segment to r/m MOV,

: RR,  ( -- )  
with the mode byte.

: ELSE  ( -- )  
Source and dest are not segment register.

: OVER # =  ( -- )  
Immediate source?

: IF NIP  ( -- )  
Yes. Discard # code.

: DUP  ( -- )  

: R16?  ( -- )  
Is dest 16 bit?

: SWAP  ( -- )  
reg field of dest.

: OVER 8 AND OR  ( -- )  
Combine reg field and w field.

: 00 OP,  ( -- )  
Assemble immediate to reg MOV,

: /C,  ( -- )  
with the immediate value.

: ELSE  ( -- )  
Not immediate source.
Assemble segment to r/m MOV, with a mode byte.

THEN
THEN
THEN
ELSE Dest is not a register. Treat it as memory reference.

ROT DUP SEG?
IF 8C C, Yes. Assemble segment to memory MOV, with memory reference mode byte.
MEM, ELSE DUP # = Immediate source?
IF DROP Yes. Discard immediate code.
C6 SIZE, Assemble immediate to reg/mem MOV, 0 MEM, with a mode byte having 0 reg field, SIZE @ ,/C, and the immediate value.
ELSE OVER #) = Is dest a memory reference?
OVER RLOW 0= And the source is AX?
IF A2 SWAP W, Assemble AX to memory MOV.
DROP Memory code #).
ELSE Non of above. Must be register to re/m MOV.
88 OVER W, Assemble reg-r/m MOV instruction, R/M, with the mode byte.
THEN THEN THEN
THEN SIZE ON Default size is 16 bit words.
;

: TEST
( source dest -- )
Assemble TEST instruction which AND source with dest and set the status register.

DUP REG?
IF OVER REG? And is source also a register?
IF 204 OVER W, Assemble opcode.
SWAP RR, Assemble reg to reg mode byte.
ELSE Source is not a register.
OVER DUP MEM? Is source a memory reference
IF 204 WMEM, Assemble memory to register code and mode byte.
ELSE Immediate data.
NIP DUP RLOW O= Is the source AL register?
IF 250 Yes. Code for AL reg and immediate data mode.
SWAP W, C, Assemble code and the immediate byte.
ELSE Memory-immediate data mode.
366 OVER W, Assemble code 366 with the word field.
DUP RLOW 300 OP, Assemble immediate byte value.
R16? ,/C, If 16 bit data, assemble high byte.
THEN THEN
THEN THEN ELSE Destination is not a register.
ROT UP REG? Is source a register?
IF 204 WMEM, Yes. Assemble reg-mem TEST instruction.
ELSE Immediate value.
DROP 366 SIZE, Immediate data and reg/mem mode.
0 MEM, Memory reference.
SIZE @ ,/C, If 2 bytes operand, assemble the second byte.
SIZE ON Activate 16 word mode.
THEN THEN

24.6. Structures in Code Commands

Structures similar to those in the regular colon commands can also be assembled in code commands. However, the structures in code commands are constructed using the branching and looping machine instructions. The test condition for branching is not a flag on top of the data stack but the
condition flags kept in the CPU status register.

Forward and backward branching are constructed using some commands similar to the MARK and RESOLVE in the colon compiler.

: A?>MARK ( -- f addr ) Set up a forward branch in code definition.
TRUE Leave a flag on stack for error checking.
HERE Address to branch from.
0 C, A dummy byte later to be resolved to a branching offset.
;
: A?>RESOLVE ( f addr -- ) Resolve a forward branching.
HERE OVER 1+ - Calculate the branching offset.
SWAP C! Store it after the branch instruction.
?CONDITION Abort if the flag is not true.
;
: A?<MARK ( -- f addr ) Set up a backward branch in code definition.
TRUE Set the flag.
HERE Leave current address on stack.
;
: A?<RESOLVE ( f addr -- ) Resolve a backward branch.
HERE 1+ - Backward branch offset.
C, Assemble the offset. Complete the branching instruction.
?CONDITION Abort if flag is not true.
;

The branching instructions are vectored through the commands >MARK, >RESOLVE, <MARK, and <RESOLVE. The execution routines vectored by these commands can now be resolved by pointing them to the above structure commands just defined in the assembler.

" A?>MARK ASSEMBLER IS ?>MARK
" A?>RESOLVE ASSEMBLER IS ?>RESOLVE
" A?<MARK ASSEMBLER IS ?<MARK
" A?<RESOLVE ASSEMBLER IS ?<RESOLVE

Conditionals in the assembler are machine codes to be assembled by the structure commands like IF, UNTIL, and WHILE. The conditionals are defined as constants to be assembled:

HEX
75 CONSTANT 0= 74 CONSTANT 0<> 79 CONSTANT 0< 78 CONSTANT 0>= 7D CONSTANT < 7C CONSTANT >= 7F CONSTANT <= 7E CONSTANT > 73 CONSTANT U< 72 CONSTANT U>= 77 CONSTANT U<= 76 CONSTANT U> 71 CONSTANT OV

DECIMAL
: IF ( opcode -- f addr ) Assemble a conditional branch instruction to start a forward branch.
C, Assemble conditional opcode.
?>MARK Set up forward branch.
;
: THEN ( f addr -- ) Close a conditional branch.
?>RESOLVE
;
: ELSE ( f1 addr1 -- f2 addr2 ) Resolve forward branch for IF and set up another forward branch to THEN.
0EB Unconditional branch opcode.
IF 2SWAP THEN Assemble it here.
Resolve forward branch from IF.
BEGIN ( -- f addr )  Set up a backward branch.
?/<MARK

UNTIL ( f addr opcode -- )  Resolve the backward branch to BEGIN.
0EB  Unconditional branch.
UNTIL  Let UNTIL do the resolving and assembling.
;
C,<<RESOLVE  Assemble the conditional opcode.
;  Resolve the branch offset.

AGAIN ( f addr opcode -- )  Resolve the backward branch with an unconditional branch instruction.

WHILE ( -- f addr )  Forward branch.
IF
;

REPEAT ( f1 addr1 f2 addr2 -- )  Branch back unconditionally.
2SWAP  Get the BEGIN location.
AGAIN  Assemble unconditional branch to BEGIN.
THEN  Resolve WHILE clause.
;

DO ( n -- addr )  Set up an assembler do-loop.
# CX MOV  Assemble an instruction setting up the loop counter.
HERE  Leave address for branch instructions.
;

NEXT ( -- )  The inner interpreter.
>NEXT #__ JMP  Assemble an indirect jump.
;
DECIMAL
Chapter 25.  Metacompiler

The source code discussed in this chapter is in the file META86.BLK and KERNEL86.BLK, Screen 1 to 10.

Metacompilation is a special feature of Forth to generate a new Forth system by an existing Forth system. It is impossible in other operating systems and languages because of the complexity in the conventional operating systems and language compilers. The most you can do in those environment is to do a 'sysgen', which allows a user to delete unnecessary or unused features in the full system and build a simpler system tailored to your application. The simplicity and conciseness of a Forth system give you much more freedom in selecting and eliminating features to suit your application. Metacompilation enables you to create a new system precisely customized to your needs, and the new system can be targeted to a different computer even with different CPU's.

25.1.  Concept of Metacompilation

The theory behind Forth metacompilation is rather straightforward. Commands in the Forth dictionary can be compiled or assembled according to the specifications of the target machine. A new dictionary can be created for the target machine containing all the commands which are necessary for execution on the target machine. This new dictionary can then be transferred to the target machine and executed on the target machine. The new dictionary will, of course, contain the required nucleus to operate on the new host CPU with necessary interpreter, compiler, and applications. A special initialization routine must also be included to power up the new target system. The metacompilation is the process to generate the new dictionary for the target computer.

Currently F83 has been implemented on 8080, 8086/88, and 68000 microprocessors running under CP/M and MS-DOS operating systems. Metacompilation was used extensively to transport the F83 model to different CPU's and to different operating systems. The metacompiler used to create a F83 system is included in the F83 system so that you can use it to rebuild the system or to generate new systems suitable for your applications. The authors of F83 intended that you will modify their F83 systems to explore new uses of Forth and to develop commercial products for public utilization.

Metacompilation is considered to be the highest level of extensibility in Forth. The first level of extensibility is to use predefined defining commands like : and CODE to add new commands to the
existing system. The second level of extensibility is to create new defining commands which can be used to compile new classes of commands or data structures to the dictionary, and to interpret them according to your specifications. The third level of extensibility, metacomilation, is to regenerate the entire Forth system with whatever extensions you might attach to it. This activity has been the privilege of large corporations and large teams or systems programmers at the expenses of billions of dollars, to build computer operating systems. In Forth, this privilege is accorded to us ordinary souls in the form of a metacompiler.

The most fundamental issue in metacomilation is that the target Forth system occupies an addressing space entirely different from the regular memory space the host Forth system addresses. The virtual address space of the target Forth system must be mapped to the real memory in the host Forth system. The metacompiler must be able to build the new system in the virtual memory space and resolve all the addresses and linkage accordingly. For one thing, the commands in the new system cannot be executed, and the new dictionary cannot be searched like normal Forth dictionary. Searching must be done through one or more symbol tables, and compiler directives must be defined in special vocabulary to help building structures in the commands belonging to the target system. These are non-trivial tasks.

25.2. Vocabulary for Metacomilation

The main purpose of metacomilation is to build new commands in the new target system using source code in the existing Forth system. It is thus necessary that a Forth command should execute differently depending upon when and where it is invoked. Multiple commands of the same name is a bad practice in normal Forth programming, but it is absolutely necessary in metacomilation. It is accomplished by using many vocabularies to house different commands of the same names. Any of these commands can be invoked by selecting a specific vocabulary searching order.
Figure 25.1  The chicken-egg cycle of meta-Forth

ONLY FORTH ALSO  Start with the normal FORTH and ROOT vocabularies.
VOCABULARY META  Define META vocabulary which will contain all the words to effect metacompilation.
META ALSO  Many of them are re-definitions.
META DEFINITIONS  META vocabulary will be searched before FORTH vocabulary and ONLY vocabulary.
: [FORTH]  An immediate version of FORTH.
; IMMEDIATE  Declare [FORTH] as immediate so that it will be executed during compiling.
: [META]  An immediate version of META.
META ; IMMEDIATE

: SWITCH ( -- )  Exchange the saved values of CONTEXT and CURRENT with themselves. It should always
be used in pair to save and restore the CONTEXT and CURRENT vocabularies. Between
the pair one can change vocabulary and select new vocabulary definitions.
NOOP NOOP  Two cells to save the current CONTEXT and CURRENT vocabularies.
DOES>  ( -- )
DUP @  Contents of the first cell of NOOP
CONTEXT @  Context vocabulary.
SWAP CONTEXT !  Copy save context to CONTEXT.
OVER !  Save CONTEXT to NOOP cell.
2+  Address of second NOOP cell.
DUP @  Fetch saved current vocabulary.
CURRENT @  Current CURRENT vocabulary.
SWAP CURRENT !  Save it in second NOOP cell.
SWAP !  Restore CURRENT.
;
VOCABULARY TARGET  A vocabulary to hold the symbol table for all definitions in the target system.
VOCABULARY TRANSITION  A vocabulary holding special case compiling words like "." and [. 
VOCABULARY FORWARD  A vocabulary holding all forward references as deferred words.
VOCABULARY USER  A vocabulary holding the USER version of defining words.
ONLY DEFINITIONS  Add all the vocabulary names to the ONLY vocabulary so that they are always
accessible and that all words in every vocabulary are accessible.
FORTH ALSO META ALSO  Collect vocabulary names from both FORTH and META vocabularies.
A few useful commands are defined to re-order the vocabularies in the searching sequence to locate specific commands in a specific vocabulary.

: IN-TARGET
  ONLY TARGET DEFINITIONS
;  
: IN-TRANSITION
  ONLY FORWARD ALSO
  TARGET DEFINITIONS ALSO
  TRANSITION
;  
: IN-META
  ONLY FORTH ALSO
  META DEFINITIONS ALSO
;  
: IN-FORWARD
  FORWARD DEFINITIONS
;  

Figure 25.2   Supporting vocabularies for metacompilation

25.3.    Accessing Memory In The Target System
During metacompilation, the dictionary of the target system is in a virtual memory space to which new commands can be added but are not accessible for other purposes. These new commands can never be executed because the target system will only be useful in the target computer which may be a totally different machine from the host computer performing the metacompilation. The target virtual memory is mapped onto the host memory space by a constant offset and a variable dictionary pointer:

```
0 CONSTANT TARGET-ORIGIN  The offset address in the host memory where the target dictionary begins.
VARIABLE DP-T           The dictionary pointer for the target system during metacompilation.
```

All memory accessing commands in Forth must be redefined for meta-compilation to access the memory of the target system.

```
: THERE ( taddr -- addr )  Map a target address to a host address.
  TARGET-ORIGIN +       Add offset.
  
: C@-T ( taddr -- char )  Fetch a byte from given target address.
  THERE C@            

: @-T ( taddr -- n )  Fetch a word from given target address.
  THERE @            

: C!-T ( char taddr -- )  Store a byte at the target address.
  THERE C!          

: !-T ( n taddr -- )  Store a word at the target address.
  THERE !           

: HERE-T ( -- taddr )  Return target address of the next available dictionary byte.
  DP-T @            

: ALLOT-T ( n -- )  Allocate more space in the target dictionary.
  DP-T +!          

: C,T ( char -- )  Add a byte to the target dictionary.
  HERE-T C!-T  Compile one byte.
  1 ALLOT-T  Move target dictionary pointer.
  
: ,T ( n -- )  Add a word to the target dictionary.
  HERE-T !-T  Store one word.
  2 ALLOT-T  Move pointer.
  
: S,T ( addr n -- )  Add a string to the target dictionary.
  0 ?DO     Scan the length of string.
    DUP C@  Fetch one character.
    C,-T  Compile one byte.
    1+  Increment addr.
    LOOP
    DROP  Discard addr still on stack.

25.4.    Branching Constructs

Two sets of commands setting up and resolving branches are needed in the metacompiler: one for colon commands and one for code commands.
The following branching commands are to be used in the assembler to set up branches in code commands.

: M?>MARK ( f addr -- ) Set up a forward branch in code definition.
TRUE
HERE-T Leave current address.
0 C,-T Reserve one byte for branching offset.
;
: M?>RESOLVE ( f addr -- ) Resolve a forward branch in code definition.
HERE-T Current address.
OVER 1+ - Offset to the mark.
SWAP C1-T Store into the reserved space.
?CONDITION Error checking.
;
: M?<MARK ( -- f addr ) Set up a backward branch in code definition.
TRUE Push flag on the stack,
HERE-T with the dictionary address.
;
: M?<RESOLVE ( f addr -- ) Resolve a backward branch in code definition.
,,T Store the address to be branched to.
?CONDITION Error checking.
;
These assembler branching commands are to be used to build code commands in the target system. The regular Forth assembler can be used for target metacompilation if the branching commands are smart enough to assemble structures in the virtual memory space of the target system. They are made smart by patching the executing addresses of the above commands into the corresponding deferred commands in the regular assembler:

' C,-T
  ASSEMBLER IS C,
' ,,T
  ASSEMBLER IS ,
' HERE-T ASSEMBLER IS HERE
' M?>MARK ASSEMBLER IS ?>MARK
' M?><RESOLVE ASSEMBLER IS ?>RESOLVE
' M?<MARK ASSEMBLER IS ?<MARK
' M?<RESOLVE ASSEMBLER IS ?<RESOLVE
All the tools provided in the assembler are now available for the metacompiler to build the nucleus portion of the target system.

25.5. Forward Reference

Forth normally does not allow forward referencing to commands that is not yet defined in its dictionary. This is a good practice to ensure that any defined command is immediately available for execution, testing and compiling. However, it presents a problem in metacompilation because we have to have defining commands to compile new commands into the target system, while these defining commands can only be defined much later in the metacompiling process. The defining commands must be made available at the very beginning of compiling the target system. F83 allows forward referencing to commands not yet defined by creating deferred commands which will be resolved and made executable at a later stage when the tools are available. The deferred commands will be linked together in a list stored in the FORWARD vocabulary. At the end of metacompilation, the list of forward references will be examined and vectored to executable commands.

```
: MAKE-CODE ( pfa -- ) Take the code field address pointed to by pfa and compile it in the target system.
    @ Fetch cfa from pfa.
    ,T Compile to target dictionary.

: LINK-BACKWARDS ( pfa -- ) Extend the linked list of unresolved forward references.
    HERE-T Current dictionary address.
    OVER @ ,T Store the address pointed to by pfa into current dictionary.
    SWAP ! Store current dictionary address into pfa, thus extending the linked list.

: RESOLVED? ( pfa -- f ) Return a true flag if the word whose pfa is on the stack is already resolve.
    2+ Flag indicating that the word is resolved.
    C@ Get it on stack as the flag.

: FORWARD-CODE ( pfa -- ) If a forward reference is resolved, compile the code. Otherwise link it to the forward reference list.
    DUP RESOLVED? A resolved forward reference?
    IF MAKE-CODE If so, compile.
    ELSE LINK-FORWARD Else link.

: FORWARD: ( -- ) Define a forward reference word and initialize it to be unresolved.
    SWITCH Save the current vocabularies.
    FORWARD DEFINITIONS Make FORWARD the current and context vocabulary to build the forward reference word.
    CREATE Make the header.
    SWITCH Revert back to the original environment.
    0 , Dummy execution address.
    0 C Unresolved flag.
    DOES> ( -- ) When a forward reference word is executed, either compile it to dictionary or link to the list of unresolved references.

```

194
25.6. Compiling New Commands to Target System

VARIABLE WIDTH
The maximum length of the names in target definitions.
31 WIDTH !
It is initialized to allow 31 characters in names.
VARIABLE LAST-T
A variable pointing to the name field of the most recently defined word in target.
VARIABLE CONTEXT-T
Pointer to the array of context and resident vocabularies in the target.
VARIABLE CURRENT-T
Pointer to the vocabulary where new definitions are to be linked.

: HASH
( str-addr voc-addr -- thread )
From the name of a definition and the address of the current vocabulary, return the thread address to link the new definition.
SWAP 1+ C@ Get the first character in the name.
3 AND Only four threads are implemented.
2* + Return the address of the thread in the body of the vocabulary.
;

: HEADER ( -- )
Create a header in the target dictionary. It makes a header out of the next word in the input stream and fixes up all the appropriate pointers to link it into the target dictionary.
BL WORD
Get the name.
C@ 1+ WIDTH @ MIN The length of the name field.
7 DUP IF If length is not zero, make the header.
ALIGN
Align new header to word boundary.
BLK @ Current block number.
4096 + The view field with the block number and the file number, assumed to be 1.
,-T Compile the view field.
HERE CURRENT-T @ HASH Find the thread to link the new word.
DUP @-T ,-T Compile the link field.
HERE- T 2- The link field address of the new word in the target dictionary.
SWAP 1-T Update the top of linking thread in the current vocabulary.
HERE-T Save a copy of the name field address.
HERE ROT S,-T Move the name from host to target.
SWAP !-T Update the top of linking thread in the current vocabulary.
HERE-T Save a copy of the name field address.
HERE-T , Compile the execution address of the new target word to pfa of the symbol table entry.
1 C,
Make sure the code field fall on even word boundary.
DOES> ( -- ) When the entry in the symbol table is executed, the execution address of the target word will be compiled to the target dictionary.
MAKE-CODE Compile the contents of the pfa to target dictionary.
;

: RECREATE ( -- )
Same as TARGET-CREATE, but don’t advance the input stream pointer so that the name of word can be used again.
>IN @ Save the input stream pointer.
TARGET-CREATE Create headers in target and symbol table.
>IN ! Restore input stream pointer.
;

: CODE ( -- )
Set up to assemble a new code definition to the target dictionary. The target cfa is set to target pfa.
TARGET-CREATE Make the headers.
HERE-T 2+ Parameter field address of the new code word.
,-T Compile code field in the target code word.
ASSEMBLER !CSP Set trap for error checking.
;

: LABEL ( -- )
Remember the current target dictionary address and assign it a name so that a subroutine can be called from a code definition.
ASSEMBLER DEFINITIONS
HERE-T CONSTANT Assign a name to the target address.
;
ASSEMBLER DEFINITIONS Go back to the Forth assembler.
: END-CODE Redefine the code word terminator for the target compiler.
IN-META Specify metacompiling environment.
?CSP Do error checking by comparing stack depth at the beginning and end of a code definition.
;
META Return to metacompiler.
IN-META And reorder the vocabularies as needed by the metacompilation.

25.7. Transition Compiler Directives

Compiler directives, which build structures in the target commands or performing special actions other than compiling execution addresses, cannot be executed immediately within the target compilation environment. The compiler directives in the normal Forth cannot be used either, because structures and special conditions must be built in the target system. These metacompiler directives are all put in the TRANSITION vocabulary and are executed from there. When these compiler commands are encountered, nwq commands are compiled into the target dictionary. The corresponding commands in the TRANSITION vocabulary are executed so that the special condition in the target command can be dealt with immediately.

: 'T ( -- cfa) Look up the next word in the input stream only in the target vocabulary. Preserve original context.
CONTEXT @ Save context vocabulary.
TARGET DEFINED Look up next word in the TARGET vocabulary. ( context cfa f -- )
ROT CONTEXT ! Restore context.
0= ?MISSING Abort if the word cannot be found.
;
: [TARGET] ( -- ) Force the compilation of a TARGET word regardless of the current CONTEXT vocabulary.
'T Find the word in TARGET vocabulary.
; Compile its execution address.
;
: 'F ( -- cfa ) Look up the next word in the input stream only in the FORWARD vocabulary. Preserve current context.
CONTEXT @ Save context on stack.
FORWARD DEFINED Search FORWARD vocabulary for the next input word.
ROT CONTEXT ! Restore context.
0= ?MISSING Abort if word cannot be found in the FORWARD vocabulary.
;
: T: ( -- ) Define a new compiler directive word in the TRANSITION vocabulary. It is otherwise the same as : .
SWITCH Save the current CONTEXT and CURRENT vocabularies.
TRANSITION DEFINITIONS Make TRANSITION the current vocabulary.
CREATE Define the following word in the TRANSITION vocabulary.
SWITCH Restore original context.
] Compile the body of the new word.
DOES> This is how the new word defined by T: should be executed:
>R Push the parameter address of the new word on the return stack. The list of execution addresses compiled in the parameter field will be executed in sequence. Similar to what NEST might have done.
;
: T; ( -- ) Terminate a word defined by T:.
SWITCH Save context.
TRANSITION DEFINITIONS Change context to TRANSITION.
[COMPILE] ; Compile end of word definition.
SWITCH Restore context.
Following are the string commands for inline comments and documentation:

T: ( ( -- ) [COMPILE] ( -- ) Inherit ( from host to TRANSITION.
T;

T: (S ( -- ) [COMPILE] (S ( -- ) Inherit (S from host.
T;

T: \ ( -- ) [COMPILE] \ ( -- ) Inherit \ from host.
T;

Special commands are needed to compile string literals into the target dictionary:

: STRING,-T ( -- ) Scan the input stream for a " as the string delimiter and compile the string into target dictionary.
ASCII " PARSE Parse input text to ".
DOP S @ 1+ Length of string just parsed.
S,-T Move the string and compile into the target dictionary.
ALIGN Align to cell boundary.
;
FORWARD: <(.")> Runtime forward reference for the code compiled by ".
T: ." Compile the runtime code <(.")> and a string literal in the target dictionary.
[FORWARD] <(.")> Compile the forward reference.
STRING,-T Compile the string literal.
;
FORWARD: <(")> Runtime forward reference for the code compiled by ".
T: " Compile the unknown runtime code <(")> with a string literal.
[FORWARD] <(")> Compile the forward reference word <(")>.
STRING,-T Compile string literal from input stream.
;
FORWARD: <(ABORT")> Runtime forward reference for ABORT".
T: ABORT" Compile the unknown runtime code for abort, followed by a string.
[FORWARD] <(ABORT")> Compile the abort code.
STRING,-T With the string literal.
;
FORWARD: <(USES)> Forward reference for code routine compiled by ;USES.
FORTH VARIABLE STATE-T True if in the compiling state inside a colon definition. False if outside or in the interpreting state.
T: ;USES ( -- ) Compile a code field whose runtime routine already exists. It is similar to ;CODE otherwise.
[FORWARD] <(USES)> Compile the code field using the address of <(USES)>.
IN-META Force the context of metacompiler.
ASSEMBLER Invoke assembler to start assembling code routine.
!CSP Install error checking.
STATE-T OFF Assembler words are interpreted, not compiled.
T;

T: [COMPILE] ( -- ) Compile a TARGET word rather than execute its TRANSITION counterpart.
' EXECUTE Find the next word and return its execution address.
Execute the word in the symbol table vocabulary TARGET. The effect is to compile the word into the target dictionary.
;
FORWARD: <(IS)> Forward reference to the runtime routine of IS.
T: IS ( -- ) Compile the unknown address of <(IS)>.
[FORWARD] <(IS)> T;
: IS ( cfa -- )  This is the version of IS in the metacompiler which actually patches the forward reference.

'T Find the cfa of the next word to be patched.

>BODY @ The execution address pointing to execution routine.

>BODY 1-T Patch in with the cfa on stack. Thus resolve the forward reference.

T; ALIGN Align the dictionary pointer to word boundary.
T; This is not needed in 8086, which is a true byte machine.

T: EVEN ( n -- n' ) Make the number n even.
T; Noop in 8086 or 8080.

25.8.  Defining Words In Metacompiler

FORWARD: <VARIABLE> Forward reference for runtime routine of CREATE and VARIABLE.

: CREATE Create a target word using the runtime routine for VARIABLE and a host word to return the HERE address in target.

RECREATE Create target word and an entry in the symbol table vocabulary TARGET. The input pointer is not advanced.

[FORWARD] <VARIABLE> Compile code field in target.

HERE-T CONSTANT Define the pfa as a constant in the host.

; : VARIABLE Define a variable in the target.

CREATE Use the above CREATE.

0 ,-T Initialize the parameter field.

; FORWARD: <DEFER> Forward reference for the runtime routine of DEFER.

: DEFER Define a deferred word or an execution vector in the target.

TARGET-CREATE Create a target word and a symbol table entry.

[FORWARD] <DEFER> Compile code field.

0 ,-T Compile a dummy parameter to hold execution address.

; : DIGIT? ( char -- f ) Return true if the character is a digit in current base.

BASE @ DIGIT Convert the char using current BASE.

NIP Only the flag is needed. Discard the result of conversion.

; : PUNCT? ( char -- f ) Return true if the character is a valid punctuation character for numbers such as leading - or decimal point.

ASCII . OVER = SWAP A period?

ASCII - OVER = SWAP Or a minus sign?

ASCII / OVER = SWAP Or a / ?

DROP OR OR Return the flag.

; : NUMERIC? ( addr len -- f ) Return true if the string is a valid number in the current base. At least one valid digit should be present in the string.

DUP 1 = Only one character?

IP Yes.

DROP Discard len.

C8 DIGIT? Is it a valid digit?

EXIT No other action needed.

THEN 1 -ROT Initial flag.

0 ?DO Scan the length of the string.

DUP C8 Get one character.

DUP DIGIT? Is it a digit?

SWAP PUNCT? Or a punctuation?

OR ROT AND AND the test results to flag.

SWAP 1+ Increment addr.

LOOP

DROP Discard addr.

;
25.9. User Variables

User variables are collected in a table called user area for multitasking context switching. All the variables pertinent to the independent operation of a task have to be preserved for each user or task when it relinquishes control of CPU to other tasks, so that when it regains the control of CPU the task can continue from where was left off. The managing of the user area requires redefining many dictionary accessing commands. These redefinitions are collected in a separated vocabulary USER. The target compiler must have its own versions of these commands to compile user variables in the target system.

FORTH VARIABLE #USER-T
A variable in FORTH to count the number of user variables defined in the target system.

META ALSO
Revert to metacompiler.

USER DEFINITIONS
Following words are added to the USER vocabulary.

: ALLOT ( n -- )
Allocate space in the user area.

#USER-T +!
Add n to the user area counter.

; FORWARD: <USER-VARIABLE>
Forward reference for the runtime routine of USER-VARIABLE.

: VARIABLE ( -- )
Create a user variable in the user area.

SWITCH
Save context.

[FORWARD] <USER-VARIABLE>
Compile code field pointing to <USER-VARIABLE>.

#USER-T @
Current user area pointer.

DUP ,,-T
Compile the pointer in parameter field.

2 ALLOT
Move user area pointer.

TARGET-CREATE
Create target and symbol table entries.

[FORWARD] <USER-VARIABLE>
Compile code field pointing to <USER-DEFER>.

SWITCH
Restore context.

#USER-T @ ,,-T
Compile the user area pointer.

2 ALLOT
Move user area pointer.

; ONLY FORTH ALSO META ALSO DEFINITIONS
Restore the metacompiling environment.

25.10. Vocabulary

The defining command VOCABULARY creates new vocabularies when the target system is brought up and running. In the parameter field of the vocabulary command, four cells are used to store the addresses of the ends of four dictionary threads for the hashing algorithm. The last cell stores the VOC-LINK address, which points to the vocabulary defined immediately before the currently defined vocabulary. This way, all the vocabularies defined in the running system are linked together themselves. This vocabulary linkage is necessary when vocabularies have to be trimmed by FORGET.
FORTH VARIABLE VOC-LINK-T

A variable linking all defined vocabularies together.

FORWARD: <VOCABULARY>

The forward reference for the runtime routine of VOCABULARY.

: VOCABULARY ( -- )

Create a vocabulary in the target system.

RECREATE

Create headers.

{FORWARD} <VOCABULARY>

Compile code field.

HERE-T

Save the parameter field address of the defined vocabulary.

#THREADS 0 DO 0 ,-T LOOP

Initialize four threads in the vocabulary.

HERE-T VOC-LINK-T @ ,-T

Store previous vocabulary pfa after the thread field.

VOC-LINK-T !

Store pfa of this vocabulary into VOC-LINK-T and extend the vocabulary linkage.

CONSTANT

Define a constant in the host.

DOES>

( -- )

Fetch the starting address of the thread field in the vocabulary.

CONTEXT-T !

Store it in the context variable to make it the context vocabulary.

;

: IMMEDIATE ( -- )

If heads are compiled in target, set the precedent or immediate bit in the name field.

WIDTH @ IF

If heads are compiled,

64

Precedent bit.

LAST-T @ THERE

Address of the name field.

CTOGGLE

Flip the precedent bit.

THEN

;

25.11. Resolving Forward References

: FIND-UNRESOLVED ( -- cfa f )

Search for a word in the FORWARD vocabulary and return the status.

' T

Find the next word in the input stream in the FORWARD vocabulary.

DUP >BODY

Get the parameter field address.

RESOLVED?

Return the a true flag if the word is resolved.

;

: RESOLVE ( taddr cfa -- )

Run through the linked list of forward reference and resolve each of them with the given address.

>BODY

The parameter field address from cfa.

2DUP

TRUE OVER 2+ C!

Store the 'resolved' flag in the third byte of the forward reference word in FORWARD.

@ BEGIN

Address of the last member in the linked list of unresolved reference.

DUP WHILE

Run down the list.

&, T

If address is not 0, go do the resolving.

-ROT

Get the next unresolved reference.

SWAP !-T

Replace the old one.

REPEAT

Resolve the old reference.

2DROP

Clear stack.

;

: RESOLVES ( taddr -- )

The command used by user to resolve forward reference.

FIND-RESOLVED

Search the next word in the FORWARD vocabulary and determine if it is resolved.

IF

Yes. Resolved.

>NAME .ID

Print its name.

." Already resolved."

And a message.

DROP

No need of taddr.

ELSE

RESOLVE

Then resolve the references.

THEN

;

At the end of metacompilation, all the forward references must be resolved before the target system is saved. Otherwise, the target system will surely crash when it is executed. There is a long list
of reference to be resolved. Following is a short list of examples for illustration. You should consult the source listing for the complete list.

\[\begin{align*}
'\text{.}^*\text{.} &\text{ RESOLVES }<\text{.}^*\text{.}> \\
'\text{.}^*\text{.} &\text{ RESOLVES }<\text{.}^*\text{.}> \\
'\text{.}^*\text{.} &\text{ RESOLVES }<\text{.}^*\text{.}> \\
'\text{.}^*\text{.} &\text{ RESOLVES }<\text{.}^*\text{.}> \\
\text{[FORTH]} &\text{ ASSEMBLER DOCREATE META RESOLVES }<\text{VARIABLE}> \\
\text{[FORTH]} &\text{ ASSEMBLER DOUSER-DEFER META RESOLVES }<\text{USER-DEFER}> \\
etc., etc.
\end{align*}\]

Deferred commands and many system variables need also be initialized:

\[\begin{align*}
'\text{LOAD} &\text{ IS LOAD} \\
'\text{CRLF IS CR} \\
'\text{KEY?} &\text{ IS KEY?} \\
etc., etc.
\end{align*}\]

\[\begin{align*}
'\text{FORTH } &\text{BODY CURRENT }!\text{-T} \\
'\text{FORTH } &\text{BODY CONTEXT }!\text{-T} \\
\text{HERE-T } &\text{DP UP }$\text{-T} + !\text{-T} \\
etc., etc.
\end{align*}\]

25.12. Redefining Host Commands

Many important commands in the host or Forth vocabulary are redefined to be used in metacompilation. To use the host versions of them explicitly, they are redefined as host commands prefixed with an 'H' character before the regular name.

\[\begin{align*}
\text{H: } &\text{[COMPILE] : } \\
\text{H: } &'\text{T }\text{BODY }\theta \text{ ;} \\
\text{H: } &',,-\text{T }\text{ ;} \\
\text{H: } &\text{C, C,-T }\text{ ;} \\
\text{H: } &\text{HERE HERE-T }\text{ ;} \\
\text{H: } &\text{ALLOT ALLOT-T }\text{ ;} \\
\text{H: } &\text{DEFINITIONS DEFINITIONS} \\
\text{H: } &\text{CONTEXT-T }\theta \text{ CURRENT-T }! \text{ ;} \\
\text{H: } &\text{[[ ] ; Alias of ], which will be used by the target compiler.} \\
\text{H: } &\text{[ ] [COMPILE] [ } \\
\text{FORTH IMMEDIATE META } &\text{[[ ]] has to be stopped by the FORTH [, which takes an alias of [[.} \\
\text{FORWARD: DEFINITIONS Making both [ and DEFINITIONS forward references so that the target compiler can assign compiler functions to them.} \\
\text{FORWARD: [}
\end{align*}\]

25.13. Running The Metacompiler

Metacompilation is a complicated process and should be used only when you have to tailor the
Forth system to very specific application. Since F83 systems were generated using metacompilation and the authors were kind enough to provide us with the complete source of the metacompiler and the actual loading commands to generate the F83 system, we have an excellent guide and example to follow. It is worthwhile to review the loading sequence in generating the F83 system. When you do metacompiling of your own system, you probably should follow this sequence as close as possible, making minimal changes and modifications in the kernel and adding your applications on top of the kernel. After you have gone through this process several times and obtain working systems, then you can start re-work the kernel.

To fire up the metacompiler, you must first prepare a disk with F83.COM, META86.BLK, and KERNEL86.BLK files on it. Type

```
F83 META86.BLK
and 1 LOAD
```
to bring up the F83 system, which in turn will load the first screen in the META86.BLK, the load screen of the metacompiler. The loading command in screen 1:

```
3 21 THRU
```
loads the metacompiler, containing all the commands discussed in this chapter. After the metacompiler is loaded, we are ready to generate the kernel Forth or the minimal Forth operating system. The following command in screen 1 of META86.BLK open the KERNEL86.BLK file and compile the kernel Forth:

```
ONLY FORTH DEFINITIONS ALSO FROM KERNEL86.BLK 1 LOAD
```
Since it will need the KERNEL86.BLK file, this file must also be on the disk. If your disk is not big enough to hold all these files, you should delete the FROM ... line from the screen 1 in META86.BLK. After the metacompiler is loaded, you can change disk and type it in on the keyboard to load the kernel.

Screen 1 of the KERNEL86.BLK file is the loading screen of the kernel. The commands:

```
ONLY FORTH META ALSO FORTH
```
include the META vocabulary in the search order and we are ready to compile the kernel. However, we have to first allocate memory space to store the target kernel Forth system. This is done by:

```
256 DP-T ! Initialize the dictionary pointer and leave 256 bytes at the bottom of the dictionary for interrupt vectors.
HERE 12000 + The physical address of the target dictionary.
' TARGET-ORIGIN >BODY ! Store it in the constant, the address offset into the target dictionary.
IN-META Establish the metacompiling environment.
2 92 THRU Load the entire kernel FORTH system.
```
We should pay special attention to the last screens in the KERNEL86.BLK file, where all the forward references are resolved, all the deferred commands are vectored to proper executable commands, and all the system variables are initialized. To make a target system run properly, these things have to be done correctly.
After the kernel Forth is metacompiled, it must be saved on the disk as an executable object file. It is saved and given the name KERNEL.COM:

META 256 THERE The physical address where the target dictionary starts.
HERE-T The logical address of the end of the target dictionary, which is the length of the target dictionary in bytes.
ONLY FORTH ALSO DOS Switch to DOS vocabulary to access the SAVE command.
SAVE A:KERNEL.COM Copy the target dictionary into KERNEL.COM, which is executable.
FORTH

At this point, we have generated a minimal Forth system and put it in an object file KERNEL.COM. This is a usable Forth system containing the text interpreter and colon compiler. However, its function is limited and not quite usable as a system to do programming and development work. If you wanted to develop a Forth application, this kernel serves well as the foundation to support your application. You can load the application program on top of the kernel and it will become a product you can sell. As a product, F83 system has lots of bells and whistles to add to the kernel.

The sequence to add applications to the kernel Forth is as following.

BYE Exit F83 and return to the DOS environment.

Copy the KERNEL.COM file to a disk which contains a file with all the application programs. In this file, screen 1 must be a load screen which will load all the application programs. In the case of F83 system, this file is EXTEND86.BLK. Moreover, EXTEND86.BLK will load programs in UTILITY.BLK and CPU8086.BLK. Therefore, you will have to copy these two files to the disk. If your disk does not have enough room for all these files, you can delete the loading commands in screen 1 of the EXTEND86.BLK file and type them on the keyboard after switching disks.

To load the application on top of the kernel, type:

KERNEL EXTEND86.BLK
and 1 LOAD

The object file KERNEL will be loaded into the memory and the kernel Forth will be booted. It then loads the first screen in EXTEND86.BLK which loads in all the utility making up the entire F83 system. In this screen you will find the following loading commands:

3 LOAD Load basic utility words and the ONLY-ALSO vocabulary mechanism.
6 LOAD Load DOS file management words.
FROM CPU8086.BLK 1 LOAD Load the 8086 assembler, and some CPU specific words to support I/O, debugger, and multitasker.
FROM UTILITY.BLK 1 LOAD Load all the utility we discussed in Part III.

The F83 system is now complete and it is also saved on the disk in a file named F83.COM

SAVE A:F83.COM

This process is what was needed to build the F83 system. You have to follow it closely in building your own Forth system.
| SIGN 85 | SINGLE 170 | SIZE 176 | SIZE, 177 |
| SKIP 89 | SLEEP 34,169 | SLOW 164 | SMART 134 |
| SOURCE 89 | SP 30 | SPOOL 169 | SPACE 57 |
| SPACES 57 | SPLIT 128 | SPOOLER 34 | SPOOL-THIS 34 |
| STAMP 133 | 'START 124 | STOP 170 | STRING-T 197 |
| SWITCH 119,190 | T 22,124 | 'T 196 | T: 196 |
| T; 196 | TARGET 190 | TARGET-CREATE 195 | TARGET-ORIGIN 192 |
| TASK: 171 | TEST 185 | TEXT? 158 | THEN 113,186 |
| THERE 192 | THRU 26 | TIB 59,88 | 'TIB 88 |
| TILL 23,129 | TO 145 | TOP 124 | TOP 169 |
| TRACE 164 | TRANSITION 190 | TRIAD 142 | TYPE 57 |
| U 23, 128 | U. 86 | U.R 86 | UN 144 |
| UD. 87 | UDB 87 | UNBUG 164 | UNNEST 44 |
| 'UNNEST 164 | UNTIL 113,187 | UP 45 | UPC 60,121 |
| UPDATE 23,69 | UPPER 32,60 | USER 10,167,190 | VARIABLE 167,198,199 |
| VIEW> 137 | VIEW-FILES 137 | VIEW field 136 | VIEW# 65,136 |
| VOC-LINK 73 | VOC-LINK-T 200 | VIEWS 137 | VOCABULARY 73,199 |
| W, 176 | MAKE 34,169 | WHERE 24,109 | WHILE 113,187 |
| WIDTH 195 | WIPE 128 | WMEM, 177 | WORD 92 |
| 'WORD 92 | WORDS 9,141 | WR/SM 178 | WRITE 116 |
| WRITE-BLOCK 66 | X 23,97,128 | XCHG 184 |