CAM-PC: A High-Performance Cellular Automata Machine

USER'S GUIDE
(Revision 0.1)

Andrea Califano
Norman Margolus
Tommaso Toffoli

MIT Laboratory for Computer Science
Cambridge, MA

Contents

Introduction

CAM-PC is a high-performance cellular automata machine intended to serve as a laboratory for experimentation, a vehicle for communication of results, and a medium for real-time, interactive demonstrations.

The software described in this manual has been designed to provide access, at different levels, to all the resources of the machine. At the highest level, you have an interactive program which turns the keyboard into a dedicated *control panel* for CAM-PC. This program allows you to run predefined rules and experiments, and to create and modify the cell-state patterns that the rules act on.

Some control-panel functions give you access to a second level, i.e., the machine's *programming language*, which is an extension of Forth. In this language you can write new cellular-automaton rules and design experiments using these rules. The Forth programming environment includes an *interpreter/compiler/loader*, a text *editor*, and a built-in machine-language *assembler*. The software that makes up this programming environment is itself written in Forth, and you are given the tools to expand it and modify it, if you wish.

This "open box" approach extends to the hardware. Much hardware reconfiguring can be performed under software control—and most users will be satisfied with just plugging the machine into their computer and never touching it again. However, all the essential inputs and outputs of the machine's functional blocks are brought out to a *user connector*, so that for specialized applications one can complement or replace some of the machine's internal resources with external hardware. In particular, several CAM-PC modules can be interconnected so as to obtain a larger cellular automata machine.

This manual is intended to be used in conjunction with the *CAM-PC Hardware Manual*, published by AUTOMATRIX, INC., and a book of a more methodological nature, *Cellular Automata Machines—A new environment for modeling* (Tommaso Toffoli and Norman Margolus, MIT Press, 1987), which discusses many of the machine's intended applications. We shall avoid unnecessary duplication of material.

A number of annotated experiment files distributed in machine-readable form as part of the CAM software are to be treated as an integral part of this manual. A *glossary* at the end of the manual (Appendix G) contains extensive technical information on the data structures, functions, and commands that make up the software. Finally, the amply annotated source files for the CAM Forth system provide a good source for further understanding and self-instruction.

Included as part of the CAM *software package is a version of the F83 system (originally produced by Laxen, Perry, and many others as a "model" implementation of the Forth-83 standard). F83 is a public-domain system, and may be freely distributed and copied as long as the authors are given credit and no copyright notice is placed upon it.*

On the other hand, in order to maintain some degree of control on the future development of CAM *Forth, any applicable proprietary rights and copyrights for the* CAM*-specific portion of the software (files whose names begin with the prefix* CAM- *) remain with its authors (Norman Margolus and Tommaso Toffoli) and, inasmuch as that applies, to the MIT Laboratory for Computer Science.*

Additional software (in Forth or other languages), utilities, and applications developed by CAM users can be made available to the CAM community through the *CAM Users' Group.*[1] This group will also provide a bulletin board listing the availability of applications notes (some are already available, but are of too specialized a nature to be included in this manual), revisions and updates; and mentioning articles, books, and meetings concerning CAM.

[1]Charles Bennett has offered to help organize this group; he can be contacted at the IBM Thomas J. Watson Research Center in Yorktown Heights, NY.

Chapter 1

Getting ready

CAM-PC (briefly, 'CAM') is a special-purpose computer designed for the high-performance simulation of cellular automata. Computation is done by table lookup, with provisions for on-the-fly display processing and data analysis.

The hardware of CAM consists of a module that fits into a full-length slot of an IBM-PC, -XT, or -AT computer (briefly, a 'PC')—or compatible machine. The module itself consists of a single six-layer printed-circuit board and contains approximately sixty integrated circuits. Several modules can be installed in the same PC and interconnected into a single, larger cellular automata machine, as discussed in the *CAM-PC Hardware Manual* and in Chapter 11.

CAM directly produces output for an IBM-PC compatible color monitor (cf. Appendix A). Because the video signal has 256 lines instead of 240, you may have to adjust your monitor slightly to see data near the top and bottom edges of the display. The regular output from the PC can be sent to a separate monitor; however, a routing cable (provided with the module) allows you to use the same monitor for displaying, under software control, either the output from CAM or that from the PC. Alternative monitor configurations are discussed in Appendix A.

1.1 Installing the hardware

Programming switches are used to control which part of the PC memory-address space and which of the PC interrupt lines are used by CAM. In most cases the default position of these switches, as set at the factory, will allow your module to run on the PC without further configuration operations. However, memory or interrupt conflicts may arise if your PC is already populated with a number of cards; in this case you'll have to change one or two switch settings, as explained in the *CAM-PC Hardware Manual*, and one or two software parameters (see `CAM-BASE` and `CAM-IRQ#` in the glossary).

Warning: *To avoid permanent damage to* CAM *or the* PC, *make sure that the power switch is turned off before performing the following operations.*

Remove the PC cover, thus gaining access to the internal expansion slots. Remove the shiny metal bracket at the rear of any empty full-length slot, saving the screw. Insert the CAM module into this slot and firmly attach its bracket with the screw. Put the cover back on. Connect the monitor(s) as explained in Appendix A.

You are now ready to run the software.

1.2 The software package

CAM's software consists of a number of program, data, and documentation files. Different types of files are distingushed by the following extensions:[1]

`DOC` Text files containing miscellaneous documentation.

`EXP` CAM experiment files.

`PAT` CAM screen pattern files.

`TAB` Files containing precompiled lookup tables.

`DAT` Files containing numeric data, typically accumulated in the course of a CAM experiment.

`4TH` Source files for the base Forth system and dedicated extensions to Forth, which together comprise the CAM programming environment (here the term 'extension' has its usual meaning).

`EXE` Executable files, i.e., files executable directly at the DOS level. (Earlier versions of some of these files have the extension `COM`.)

The software comes on a number of distribution diskettes, containing the following material:

1. A program called `CAM.EXE`, which puts you in complete control of the CAM machine and its associated software utilities.

2. Sample experiments and demos that illustrate all major CAM features and some typical applications.

[1]In DOS jargon, the *extension* is an optional second part of a file name, separated from the first part by a period; thus, files with extension `4TH` have a name of the form `xxxx.4TH`.

3. The base Forth programming system (of which `CAM.EXE` is an extension) in executable form, under the name `F83.EXE` . This system, which can be used as a general-purpose programming environement quite independently of CAM, is accompanied by complete and annotated source code.[2]

4. The source code for the CAM extension of Forth.

Before attempting to do anything else, you should make two back-up copies of each distribution disk. Make the copies using the DOS command `DISKCOPY` *, as explained in the DOS manual. Put the original set of distribution disks and one back-up set in a safe place. Use the second copy as a master, should further duplication be necessary.*

Look for a file called `READ-ME.DOC` in one of these diskettes, and type it out on the terminal or list it on the printer. It will contain up-to-date information on the contents of the software package, and useful hints on how to handle it.

Finally, format one more disk, to be used a *work disk*, and copy to it the `CAM.EXE` program (which by itself is sufficient to run CAM) and the following experiment and pattern files, which will be used for practice in conjunction with this manual:

```
BARELIFE.EXP  IRGB.EXP  VARLIFE.EXP  BRAIN.EXP
SAMPLE.PAT    DISK.PAT  GUNS.PAT
```

If you have standard equipment, no installation procedure is needed to start using the CAM software. If you have a hard disk, eventually you will want to install all the CAM software there, as explained in Section B.3.

Some configuration options are available for driving different kinds of printers, supporting more than one CAM module, and personalizing other minor features. The available options are described in Sections 11.2, B.1, and B.2.4.

1.3 Start-up

The only program required to run CAM is `CAM.EXE` , which is contained on the work disk together with sample experiments and patterns.

1. Boot the PC using DOS version 2.1 or higher.

2. Insert the work disk in one of your drives.

[2]The Forth system provided with CAM is a modified version of F83, and has been placed in the public domain; it includes the Forth interpreter/compiler/loader, the screen editor, the assembler, and many utilities. Also included is a *metacompiler*, by means of which one can completely regenerate the system, and produce extended or customized versions of it.

3. Make that drive the default one (e.g., if the work disk is in drive `A:`, type `A:` followed by carriage return).
4. Type `CAM` followed by carriage return.[3]

The terminal should display the CAM logo and then print

```
Press 'm' for a menu
```

CAM has passed a cursory hardware test, and you are now in command of the machine. To return to DOS, type `F` followed by `BYE` (cf. end of Section 4.1).

If instead the PC beeps and you get the message '`CAM hardware not responding!`' there is some kind of hardware problem. The most likely problem is a memory or interrupt conflict; this may require moving a hardware jumper, as explained in the *CAM-PC Hardware Manual*, as well a software "jumper," as explained under `CAM-BASE` and `CAM-IRQ` in the glossary at the end of this manual (Appendix G). Of course, you should verify that you have at least 256K of memory in your PC, and that a CAM card has actually been installed in the PC! After this message, the `CAM.EXE` program will still run, but without attempting to talk to CAM; thus, program development is possible even when the CAM card is not physically present.

[3]Since the extension `COM` , typical of an executable program file, is supplied by DOS by default, you need not explicitly type `CAM.EXE` .

Part I

The control panel

Chapter 2

First steps

This chapter introduces you to the use of the PC terminal as a "control panel" for CAM. As an example we'll load and run the game of "life"—a well-known cellular automaton you may already be familiar with. Most of this material will be covered in a more systematic way in the following chapters.

For ready reference, Appendix H displays a complete summary of control-panel commands.

2.1 The terminal as a control panel

The CAM program takes control of all of the PC hardware and presents you with an integrated programming environment suitable for running CAM. Most of the time the terminal (i.e., the PC's keyboard and monitor) is turned into a dedicated *control panel*, where each key has a special function with immediate effects. Some keys allow you to temporarily leave the control-panel mode to access more generic programming resources, such as the Forth interpreter or the editor; in this context, the function of the terminal reverts to that of a (smart) typewriter.

As we've seen in the previous chapter, the CAM program comes up with the message

```
Press 'm' for a menu
```

if everything seems to be in working order. Just below this message you will notice a little "happy face;" this is a *prompt* telling you that the terminal is in the control-panel mode and is ready to accept a command. The prompt comes in one of two colors. A *black* face means that an experiment is actually running; a *white* face means that no experiment is in progress, or that the experiment you were running is temporarily stopped.

2.2 Menus

If you type `m` now, the screen will display the *master menu* (the "menu of menus"), which should look something like this

```
0m GENERAL
1m DISPLAY/CONTROL
2m EDITING,RUNNING
3m PLANE-OPS
4m DOTS,SHIFTS
5m ALTERNATE
```

If you type, say, `3m` you'll get menu 3, concerned with CAM-plane operations. Each menu lists a whole set of one-key commands, each accompanied by a short mnemonic name to help you remember its function. Once you learn which key invokes some desired function you can type it directly, without going through a menu. All these functions will be discussed in detail in the next few chapters.

2.3 Typing conventions

Each control-panel command consists of a single keystroke, possibly preceded by a numeric argument[1] and possibly followed by a text-string argument. We'll indicate the command keystroke by a box containing the corresponding character; for example, the *Menu* command will be denoted by $\boxed{\texttt{m}}$. The "carriage return" or "enter" key, typically the largest on the keyboard, is indicated by the symbol $\boxed{\hookleftarrow}$.

In general, the control panel and the Forth interpreter distinguish between upper- and lower-case letters—and the case of a letter depends, as usual, on whether you are pressing the SHIFT key and on whether you are in CAPSLOCK mode.[2]

The following typing conventions apply to the control panel (which is always in no-CAPS-LOCK mode) ;

1. A nonalphabetic symbol in a box, such as $\boxed{:}$ or $\boxed{;}$, means you have to hit the corresponding key, with a SHIFT when appropriate.

2. A lower-case letter in a box, say $\boxed{\text{a}}$, means that you must simply hit the `A` key.

[1]See Appendix H.0 for how to enter an argument in a base different from ten.

[2]In many PC models, software is not able to control the CAPSLOCK light (where it exists) in a consistent manner. This difficulty is solved by letting the *size* of the cursor on the terminal—*tall* (▌) or *short* (_)—tell you whether or not you are in CAPSLOCK mode.

3. An upper-case letter in a box, say $\boxed{\texttt{A}}$, means that you must hit the `A` key while holding the SHIFT key pressed.

4. A small-caps word in a box, such as $\boxed{\text{Esc}}$, means that you must hit the corresponding key on your keyboard (refer to the PC operating manual).

5. The function-key boxes $\boxed{\texttt{f1}}$ and $\boxed{\texttt{F1}}$ refer to the unshifted and shifted versions respectively of the `F1` key.

6. A word in typewriter font, such as `life` , denotes a text-string argument, to be typed character-by-character *and followed by* $\boxed{\hookleftarrow}$.

7. A letter in a box starting with the prefix `Ctrl-` (such as $\boxed{\texttt{Ctrl-A}}$) means that you must hit the corresponding key while holding the CONTROL key pressed.

8. A letter or function key in a box starting with the prefix `Alt-` (such as $\boxed{\texttt{Alt-A}}$) means that you must hit the corresponding key while holding the ALT key pressed.

9. Finally, $\boxed{\text{Break}}$ will denote "break"—a brute-force way of aborting execution of any command—which in the PC is actually achieved by pressing SCROLLLOCK while holding the CONTROL key pressed.

With the above conventions, a sequence such as $\boxed{\texttt{L}}$`life` indicates that you must type $\boxed{\texttt{L}}$ (using the SHIFT key, since this is an upper-case letter), and then the string `life` followed by $\boxed{\hookleftarrow}$. Note that, as soon as you enter the command $\boxed{\texttt{L}}$, the happy face disappears and is replaced by an appropriate prompt message—in this case

```
Load.new.file:
```

and the keyboard temporarily switches from control-panel mode to typewriter mode, so that `life` will be treated as a text string rather than as the sequence of commands $\boxed{\texttt{l}}\boxed{\texttt{i}}\boxed{\texttt{f}}\boxed{\texttt{e}}$; the keyboard reverts to the control-panel mode as soon as you enter $\boxed{\hookleftarrow}$. You can always escape from the typewriter mode by pressing $\boxed{\text{Esc}}$: this aborts the current command and immediately returns you to the control-panel mode.

2.4 A first experiment

At this point you are ready to run a simple CAM experiment.

- Type $\boxed{\texttt{L}}$. You'll see the message

```
Load.new.file:
```

- Answer with the name of the experiment file you want to load.[3] In this case, type `barelife`.

Before typing anything else, *wait until you again see the happy face prompt.* The red light of your disk drive will flash briefly, some comments will appear on the terminal, and the happy face will reappear: the program for the experiment BARELIFE has been successfully loaded.

Now we need to give some initial conditions to the experiment. Type `0`$\boxed{;}$ to fill CAM's plane 0 with random bits. The plane's contents is displayed on CAM's monitor; you should see a random pattern of green dots.

If you are sharing the color monitor between CAM and the PC, you must use the command $\boxed{a}$ ("alternate display") in order to see CAM's output rather than that coming from the PC (cf. Appendix A). The $\boxed{a}$ command is a *toggle*, i.e., it will move you back and forth between the two display modes. On the other hand, $\boxed{A}$ will force the display to the default, PC-output mode.

$\boxed{a}$	*Alternate display.* Toggle between PC and CAM display (if monitor is shared). See Appendix A for the use of an optional numerical argument.	(2.1)
$\boxed{A}$	*Restore* PC *display* (if monitor is shared between PC and CAM).	

The usefulness of a "restore default state" command such as $\boxed{A}$ is not obvious in the normal, interactive situation, since you can easily see from the display what mode you are in. However, such commands are essential in certain situations (remote control, "macros" consisting of canned sequences of keys, etc.) where the state of the display (or of some other CAM feature) may not be known a priori. In general, the shifted version of a control-panel command has a meaning closely related to the unshifted version, and is used to restore a default state, to ask for a text string as an argument, or for some other less-frequently used variant of the command.

While the unshifted and shifted versions of keys have a predefined meaning, the **Alt-** versions available for user-defined commands, as explained in Section 8.4.

[3]The $\boxed{L}$ command expects the name of a disk file, and assumes for this file the default extension EXP , so that you don't need to append the extension unless your experiment file has one other than EXP. Moreover, file names are passed on to DOS, which is "case-blind;" thus, in the present context `life` and `LIFE` are equivalent.

2.5 Running control

Now that you have loaded an experiment and set the initial conditions, you can control the evolution of the system by using one of the commands listed below.

From the `EDITING,RUNNING` menu

$\boxed{\text{s}}$	*Step.* Execute a single step of the evolution of the cellular automaton. If this command is preceded by a numeric argument n (in the range 1 through $2^{32}-1$), it will execute n steps at the current step rate (unless stopped earlier by the *Stop* command).
$\boxed{\text{S}}$	*Run.* Run steps continuously, at the current step rate. The result is real-time animation on the screen.
$\boxed{\text{SPACE}}$	*Stop.* Stop running, and display on the control panel the value of the step counter. Use $\boxed{\text{S}}$ to resume running.
$\boxed{<}$	*Slowest.* Go to the lowest step rate (approximately 1 frame/sec).
$\boxed{>}$	*Fastest.* Go to the highest step rate (approximately 60 frames/sec).
$\boxed{,}$	*Slower.* Halve the step rate. If you hit this key several times, the movie will turn into a slide show. (As a mnemonic, note that $\boxed{,}$ and $\boxed{.}$ are just the unshifted versions of $\boxed{<}$ and $\boxed{>}$.)
$\boxed{.}$	*Faster.* Double the step rate.

From the `DISPLAY-CONTROL` menu

$\boxed{\text{x}}$	*Expand.* Display a magnified view of the central portion of the CAM screen (use the arrow keys, discussed in Section 3.5, to move to the center of the screen the area you are interested in). Hit $\boxed{\text{x}}$ again to return to the normal view. The magnified view may be used even while the simulation is running, but it will slow it down. All control-panel commands can still be used while in expanded mode. See Section 3.9 for more details.

The step counter is initialized to the value $-n$ by the command $n\boxed{\text{s}}$ (where n denotes the numeric argument), and is incremented by 1 after each step. When the counter reaches 0 the simulation is stopped. If you type `100`$\boxed{\text{s}}$ and press the $\boxed{\text{SPACE}}$ bar before 100 steps have completed, the counter will display a negative value; if you resume by typing $\boxed{\text{S}}$ with no argument, the counter will continue incrementing up to 0 and then the simulation will stop. If you resume again with $\boxed{\text{S}}$, it will start going through positive values. The counter works modulo 2^{32}: if you type 0$\boxed{\texttt{s}}$, it will count 1, 2, 3, ... and will stop when it reaches 0 again—after 2^{32} steps. Thus, $n\boxed{\texttt{s}}$ is useful for a "count-down" of n steps, and 0$\boxed{\texttt{s}}$ for counting from 0 upwards.

Chapter 3

The plane editor

The bits of information that make up the state of the CAM universe are arranged in four arrays of size 256×256 called *bit-planes*. Normally, each of the bit-planes is wrapped around so that its left edges adjoins the left edge, and similarly for top and bottom edges (but cf. Section 11.1 and Chapter 10).

A number of control-panel commands are available for editing the contents of the CAM planes, in order, for instance, to construct a given initial state for the cellular automaton. We shall refer to this subset of control-panel commands as the *plane editor*. The plane editor is interactive, in the sense that whatever changes you make to the contents of CAM's planes is immediately displayed on the CAM monitor.[1]

The CAM plane editor is a quite different object from the Forth *screen editor* (Chapter 6), which is used for interactively editing Forth programs on the PC monitor. (In this context, 'screen' is a Forth technical term; it refers to a block of Forth text residing on disk, or to the same block as displayed in the screen editor's window.) In brief, the plane editor allows one to manipulate some of CAM's contents as *graphic* material, displayed on the CAM monitor, while the screen editor allows one to manipulate some of the PC's contents as *text* material, displayed on the PC monitor.

3.1 The state of the universe and its representation

The bits of information that make up the state of the CAM universe are arranged in four arrays of size 256×256 called *bit-planes*. Thus, each of the 256×256 sites or "cells" of the cellular automaton contains four bits, one from each bit-plane.

[1]Provided that an appropriate color map is being used. This will be explained in a moment.

Treating the state of a cell as a collection of four bits is convenient for programming purposes. For display purposes, it is better to represent each cell by a colored dot, or *pixel*, on the screen. The color, of course, will correspond to the cell's state according to a definite assignment: the table that decides which color has been assigned to each of the 16 possible cell states is called the *color map.* In CAM, the contents of the color map can be specified by the user to suit the requirements of each experiment.

We'll explain later how to build different color maps and how to use them for different purposes. For the moment we'll use a color map that establishes a *one-to-one correspondence* between the contents of a cell and the color of the corresponding pixel, so that the picture on the screen provides an unambiguous representation of the bit-planes' contents.[2] In this way, nothing is hidden from view: if you change a bit in one of the planes something will change on the screen.

Color map: `IRGB-MAP`

Plane				Monitor line				Pixel color	Color code
0	1	2	3	I	R	G	B		
0	0	0	0					black	0
0	0	0	1				●	blue	1
0	0	1	0			●		green	2
0	0	1	1			●	●	cyan	3
0	1	0	0		●			red	4
0	1	0	1		●		●	magenta	5
0	1	1	0		●	●		yellow	6
0	1	1	1		●	●	●	white	7
1	0	0	0	●				gray	8
1	0	0	1	●			●	bright blue	9
1	0	1	0	●		●		bright green	A
1	0	1	1	●		●	●	bright cyan	B
1	1	0	0	●	●			bright red	C
1	1	0	1	●	●		●	bright magenta	D
1	1	1	0	●	●	●		bright yellow	E
1	1	1	1	●	●	●	●	bright white	F

(3.1)

3.2 The *IRGB* color map

The color map we are going to use for editing practice is called *IRGB*. These four letters stand for the four input lines of the color monitor, namely *Intensity*, *Red*,

[2]A color map may assign the same color to two or more different cell values; in this case one cannot unambiguously reconstruct the contents of a cell from its color on the screen.

Green, and *Blue*. The last three signals separately drive the PC monitor's three color beams, while the first adds some extra white light by increasing the intensity of *all three* beams. This color map (defined by the Forth word `IRGB-MAP`) establishes a direct link between the contents of the four bit-planes and these four signals, as shown in Table (3.1). For instance, the blue beam will be turned on for a given pixel on the screen if in the corresponding cell the bit of plane 3 is "on" (i.e., has value 1). If more than one bit is on, the overall pixel color will be a superposition of the corresponding primary colors, plus intensity if present.

3.3 Editing practice

To start a tutorial editing session, where you will be able to immediately practice the commands of this chapter as they are introduced, it will be convenient to initialize CAM to an agreed upon state; in particular, to get a sample pattern into the planes and to load the color-map table with `IRGB-MAP` .

To this purpose, type $\boxed{\text{L}}$`irgb` [3] and wait for the happy-face prompt. This loads a "dummy experiment" that (a) specifies `IRGB-MAP` as the color map—so that all the editing you do on the bit-planes will be clearly visible on the CAM screen—and (b) specifies *identity* as the rule for the cellular automaton (this is a "do nothing" rule)—so that even if you inadvertently hit the *Step* or *Run* keys (cf. Section 2.5) the pattern on the screen will not change. Now type $\boxed{\text{G}}$`sample` to get a sample pattern contained in the file `SAMPLE.PAT` and copy it into CAM's bit planes; this pattern will appear on the screen (cf. Section 2.4 if you have any problems here).

As explained in Section 2.3, many control-panel commands may be preceded by a numeric argument. For most of the commands that make up the plane editor this argument is used to specify on which plane (or planes) the command itself should operate. Planes are numbered 0 through 3, and the available options for a "plane" argument are:

0–3 : Select only the specified bit-plane (0 through 3).

4 : Select plane pair 0–1 (i.e., both planes 0 and 1).

5 : Selects plane pair 2–3 (i.e., both planes 2 and 3).

No argument : The command is carried out on all four planes.

All keys which take a "plane" argument follow this convention (with only one exception, namely the $\boxed{\text{d}}$ key, which will be explained when we come to it).

[3]Don't forget to hit $\boxed{\hookleftarrow}$ at the end of the string argument.

For example, the command $\boxed{\texttt{z}}$ is defined as follows in Table (3.8) below:

$\boxed{\texttt{z}}$	*Zero.* Fill plane(s) with 0s ("clear" the planes).

Thus, typing $\boxed{\texttt{z}}$ will clear all four planes; 0$\boxed{\texttt{z}}$ will clear only plane 0; 4$\boxed{\texttt{z}}$ will clear planes 0 *and* 1; etc.

3.4 Color filters

It is possible from the control panel to temporarily modify the color map currently in use so that one or more of the inputs to the color monitor are cut off. This is occasionally useful while debugging an experiment: if the current color map is one that directly routes the contents of the four planes to the four monitor inputs (as does the *IRGB* color map), then cutting off one of the beams is equivalent to *hiding from view* the corresponding plane—without actually affecting its contents.

The relevant commands are

$\boxed{\texttt{f2}}$	*IRGB map.* Use `IRGB-MAP` as the color map, and restore any beams that are cut off. This "faithful" map is useful for debugging purposes.
$\boxed{\texttt{f4}}$	*Intensity.* Whatever color map is in use, toggle "intensity" beam on or off.
$\boxed{\texttt{f6}}$	*Red.* Similarly, toggle red beam on or off.
$\boxed{\texttt{f8}}$	*Green.* Toggle green beam on or off.
$\boxed{\texttt{f10}}$	*Blue.* Toggle blue beam on or off.

(3.2)

Note that $\boxed{\texttt{f2}}$ reactivates any beams that were turned off by $\boxed{\texttt{f4}}$–$\boxed{\texttt{f10}}$. Practice the above commands right away (if the pattern `SAMPLE.PAT` has been destroyed by the previous practice session, get it again from disk using the $\boxed{\texttt{G}}$ command); then type $\boxed{\texttt{f2}}$ to restore full view of the planes.

The following commands are also available

$\boxed{\texttt{F2}}$	*Standard map.* Use `STD-MAP` as the color map, and restore any beams that are cut off. This is the default or *standard* color map, discussed in Section 8.2.	
$\boxed{\texttt{F4}}$	*Restore intensity.* Restore intensity beam, if cut off by $\boxed{\texttt{f4}}$.	(3.3)
$\boxed{\texttt{F6}}$	*Restore red.* Restore red beam.	
$\boxed{\texttt{F8}}$	*Restore green.* Restore green beam.	
$\boxed{\texttt{F10}}$	*Restore blue.* Restore blue beam.	

For sake of reference, we shall list here the following

$\boxed{\texttt{Alt-F2}}$	(*Custom map*). As all `Alt-` commands, this control-panel command is initially undefined. However, a custom color map defined by the user for a given experiment should preferably be attached (cf. Sections 8.4, 9.6) to this key, for convenience in locating it.	(3.4)

3.5 Shift, hold

The following commands use the *numeric keypad* section of the keyboard (briefly, the "arrow keys," even though only four of these keys are actually marked by arrows). They are used to shift all or some of the four bit-planes in any of eight directions.

$\boxed{\uparrow}$	*Shift* plane(s) up.	
$\boxed{\downarrow}$	*Shift* plane(s) down.	
$\boxed{\leftarrow}$	*Shift* plane(s) left.	
$\boxed{\rightarrow}$	*Shift* plane(s) right.	(3.5)
$\boxed{\text{PgUp}}$	*Shift* plane(s) up-right.	
$\boxed{\text{PgDn}}$	*Shift* plane(s) down-right.	
$\boxed{\text{Home}}$	*Shift* plane(s) up-left.	
$\boxed{\text{End}}$	*Shift* plane(s) down-left.	

For instance, pressing the "right arrow" key $\boxed{\rightarrow}$ will make the screen pattern shift rightwards, wrapping around from the right edge to the left one. The pattern will keep shifting as long as you keep the key pressed. If you just hit the key momentarily, the pattern will shift by an amount determined by the shift-size register (see below), with initial default of eight pixels.

For these commands, the optional numeric argument specifies the amount of shift in pixel units (thus, it is not interpreted as a "plane" argument). Thus, if you type 4$\boxed{\rightarrow}$ the pattern will shift by *four* pixels, and every subsequent shift operation will also move the pattern by a multiple of four pixels—until you again give a shift command preceded by an argument. In fact, the eight *Shift* commands share a single *shift-size* register, which is initially set to 8 but is modified when you give an explicit argument to any of these commands. This argument can be any number in the range 0–511 (other numbers are reduced modulo 512). To go back to eight-pixel shifts, type 8$\boxed{\rightarrow}$.[4]

Sometimes it is desirable to shift only *some* of the planes and leave the others fixed in place. Each bit-plane can be held fixed or made free to shift. The following commands can be used to obtain any possible combination of "free-to-shift" and "fixed-in-place" bit-planes:

$\boxed{\mathtt{h}}$	*Hold.* Make plane(s) fixed.
$\boxed{\mathtt{H}}$	*Unhold.* Make plane(s) free to shift.

(3.6)

These commands take an optional "plane" argument. As explained in Section 3.3, this means that $\boxed{\mathtt{h}}$ by itself will hold all planes; 0$\boxed{\mathtt{h}}$ will hold only plane 0; 4$\boxed{\mathtt{h}}$ will hold plane pair 0–1; etc.

Examples:

- Type $\boxed{\rightarrow}$ to shift all planes eight cells to the right.
- Type $\boxed{\leftarrow}$ to put the planes back to their original position.
- Type 64$\boxed{\rightarrow}$ to move the planes right by 64 pixels.
- Now, simply type $\boxed{\leftarrow}$ to put the planes back to their original position.
- Type 8$\boxed{\uparrow}$ $\boxed{\downarrow}$ to restore the shift-size register to its "default" value of 8.
- Type 1$\boxed{\mathtt{h}}$ to hold plane 1 fixed. All subsequent shift operations will affect only planes 0, 2, and 3 (try some!).
- Type $\boxed{\mathtt{H}}$ to set all planes free (in this case, to release plane 1).
- Type $\boxed{\mathtt{h}}$ 1$\boxed{\mathtt{H}}$ to hold all planes except plane 1.
- Type $\boxed{\mathtt{H}}$ to return to the default state (all planes free).

[4]Since this will execute an eight-pixel shift, you may follow it with 8$\boxed{\leftarrow}$ if you want to reset the shift size to 8 but leave the pattern in its original position.

3.6 Rotate, reflect

Six commands are available to perform rotations or reflections on the bit-planes. They accept an optional "plane" argument.

$\boxed{\texttt{r}}$	*Rotate* plane a quarter-turn clockwise.
$\boxed{\texttt{R}}$	*Rotate* a quarter-turn counter-clockwise.
$\boxed{\backslash}$	*Reflect* about the principal diagonal (which runs from the upper-left corner to the bottom-right one).
$\boxed{/}$	*Reflect* about the secondary diagonal.
$\boxed{\_}$	*Reflect* about the horizontal midline.
$\boxed{\vert}$	*Reflect* about the vertical midline.

(3.7)

The best way to understand these commands is to see them at work.

Since they involve much data movement between CAM and the PC, the first four of these commands take a little time to execute; this time is proportional to the number of planes involved. It is wise to use a numeric argument in order to restrict a command to just the planes you are interested in.

3.7 Fill, complement

Commands are available for *filling* a plane with all `0` s or all `1` s, and for *complementing* a plane (i.e., turning all `0` s to `1` s and vice versa, realizing the logic-NOT function).

$\boxed{\texttt{z}}$	*Zero.* Fill plane(s) with `0` s.
$\boxed{\texttt{Z}}$	*Unzero.* Fill plane(s) with `1` s.
$\boxed{-}$	*Negate.* Complement plane(s).

(3.8)

As usual, a numeric argument is used to select individual planes or plane-pairs.

Examples:

- Type `3`$\boxed{-}$ to complement plane 3.
- Type $\boxed{\texttt{z}}$ to clear all planes.
- Type `5`$\boxed{\texttt{Z}}$ to fill planes 2 and 3 with `1` s.

3.8 Random

Random initial conditions are useful in many experiments. The command $\boxed{;}$ generates a random pattern where 1 s appear with a certain probability p on a background of 0 s (the value of p is specified by the commands $\boxed{:}$ or $\boxed{\%}$, as explained below).

With no argument, all planes are filled with the *same* random pattern. As usual, a numeric argument restricts the command to the specified plane or plane pair. Every time you use $\boxed{;}$ a *new* pattern is generated.

Without a numeric argument, the command $\boxed{:}$ resets the value p of the *probability register* to a default value of 1/2. When preceded by a numeric argument n in the range 0–65536, this command sets p to the value

$$p = \frac{n}{65536}.$$

Since the size of a plane is 256×256 (=65536), n corresponds to the *expected* number of 1 s: for example, once you've typed 100$\boxed{:}$, the command $\boxed{;}$ will produce patterns containing approximately one-hundred 1 s (and averaging precisely one-hundred over many patterns).

It is often convenient to think in terms of a percentage f rather than a number n of cells. The command $\boxed{\%}$ preceded by a numeric argument f in the range 0–100 sets p to $f/100$, so that, after typing 33$\boxed{\%}$, the command $\boxed{;}$ will produce approximately 33% of 1 s.

$\boxed{;}$	*Random.* Fill the selected plane(s) with a random pattern using the current value of p.
$\boxed{:}$	*Set random count.* The value of the argument (range 0–65536) specifies the expected *number* of 1 s. With no argument, p =1/2.
$\boxed{\%}$	*Set random percentage.* The value of the argument (range 0–100) specifies the expected *percentage* of 1 s. With no argument, p=1/2.
$\boxed{i}$ or $\boxed{I}$	*Initialize seed.* An argument in the range of 0–65536 is used to generate a 272-bit seed for the random number generator. If no argument is given, a default value is used for the seed.

(3.9)

The random bits are generated by a deterministic, "pseudo-random" algorithm. In certain experiments, to make significant comparisons it is necessary to use the *same* sequence of random patterns as was used for a previous run of the experiment. The command $\boxed{\texttt{i}}$ re-initializes the random-number generator with a "seed" specified by a numeric argument in the range from 0 to $2^{16}-1$: from the same seed you get the same sequence, while different values of the seed yield different sequences.

The generator is always initialized with the same default seed upon entering the CAM program. $\boxed{\texttt{i}}$ or $\boxed{\texttt{I}}$ without an argument re-initialize the generator to this default seed.

When p=1/2, a planeful of random bits is generated in a very short time; for other values of p the process is somewhat longer.

Examples:

- Type $\boxed{\texttt{;}}$ to get a random pattern with 50% of `1` s and 50% of `0` s, identical for the four bit-planes.
- Type `10`$\boxed{\texttt{\%}}$ $\boxed{\texttt{;}}$ to do the same with only 10% of `1` s.
- Type `34901`$\boxed{\texttt{i}}$ to initialize the random-number generator with the seed 34901. This seed can be used as a label to identify a repeatable sequence of random patterns.
- Type `100`$\boxed{\texttt{:}}$ `0`$\boxed{\texttt{;}}$ to have about one-hundred `1` s in bit-plane 0.
- Type $\boxed{\texttt{:}}$ (or $\boxed{\texttt{\%}}$, or `50`$\boxed{\texttt{\%}}$) to reset p to its default value of 1/2.
- Type `0`$\boxed{\texttt{;}}$ `1`$\boxed{\texttt{;}}$ `2`$\boxed{\texttt{;}}$ `3`$\boxed{\texttt{;}}$ to obtain four *different* 50% patterns on the four planes.

3.9 Magnify screen

The individual pixels on the screen are rather small, and it's hard to discern the fine details of a pattern—especially on monitors that have poor resolution or are poorly aligned. Moreover, dot operations (cf. Section 3.10) are more easily performed under magnification.

An "expanded screen" mode is provided in CAM. When this mode is on, the screen is expanded both horizontally and vertically by a factor of four, and each cell appears as a 3×3-pixel square surrounded by a 1-pixel-wide black frame. Also, in this mode a positioning grid becomes available, so that one can count cell positions even when many adjacent cells are empty.

The commands to control the above features are:

$\boxed{\text{x}}$	*Expand.* Toggle between normal and expanded mode.	
$\boxed{\text{X}}$	*Unexpand.* Turn off expanded mode.	(3.10)
$\boxed{\text{'}}$	*Grid.* Toggle the grid on and off.	
$\boxed{\text{"}}$	*Ungrid.* Turn off the grid.	

In the expanded mode, only 64×64 cells are visible on the screen, instead of the usual 256×256. To show a close-up of a portion of the screen one must move this portion close to the center by shifting the whole screen pattern in the appropriate direction. The expanded mode may be used even while the simulation is running (all control-panel commands remain available in this mode) but it will slow it down.

The grid is visible only in the expanded mode. It consists of dots placed at the intersections of the horizontal and vertical black lines that separate pixels. Note that the spacing between grid dots, either horizontally or vertically, is *two* cells (rather than one), so that each square of the grid encompasses a 2×2 block of cells. This makes counting rows and columns easier, and is also useful when one uses special rules or neighborhoods where at any particular step it makes a difference whether a cell lies on an even or an odd row (or column) of the array (cf. Section 9.4.1).

3.10 Dot graphics

The *dot* mode is used to set or reset single bits of the bit-planes. Only one plane (the "target" plane) is accessible at any given time.

The commands available for this mode of operation are:

$\boxed{\texttt{d}}$	*Dot.* Toggle dot mode on or off. If a plane number is given as an argument (range 0–3), that plane becomes the target, superseding the previous target. At start-up, plane 0 is the target.	(3.11)
$\boxed{\texttt{D}}$	*Undot.* Turn off dot mode.	
$\boxed{\texttt{o}}$	*Cursor to origin.* Move the dot cursor to its initial position at the center of the screen (coordinates ⟨128, 128⟩, whereas the upper-left corner has coordinates ⟨0, 0⟩).	
$\boxed{\texttt{O}}$	*Shift to origin.* Shift the screen and the cursor as a whole until the cursor comes to the center of the screen.	
$\boxed{\text{INS}}$	*Insert.* Complement bit at cursor.	

When in dot mode, a cursor is displayed on the screen, centered on the target cell; the arrow keys (cf. Section 3.5) are now used to move the cursor rather than shift the planes.

In dot mode, the selected plane is only affected when you press the $\boxed{\text{INS}}$ key: this complements the value of the target bit. If the $\boxed{\text{INS}}$ key is kept pressed and the cursor is moved using the arrow keys, the cursor will toggle all the bits it finds on its path; in this way one can draw continuous lines and complex shapes.

When you leave the dot mode, the current cursor position and target plane are remembered. Though the cursor is invisible, the $\boxed{\text{INS}}$, $\boxed{\texttt{o}}$, and $\boxed{\texttt{O}}$ keys remain active.

On certain keyboards, pressing $\boxed{\text{INS}}$ inhibits the autorepeat feature of one or more of the numeric-keypad keys. In this case, use $\boxed{\text{CAPSLOCK}}$ or $\boxed{\text{SCROLLLOCK}}$ as a replacement for $\boxed{\text{INS}}$.

The dot mode also supports an optional "mouse" (see Section B.4) as a pointing device: the cursor on the screen will track the mouse's movements.

When using the mouse, bits are only toggled while one of the mouse buttons (or the $\boxed{\text{INS}}$ key) is pressed. The right button is used for drawing horizontal and vertical lines: while the right button is held down, mouse motion will be interpreted as either horizontal or vertical, but never diagonal. Similarly, the left button is used for drawing diagonal lines. The middle button (if your mouse has three buttons, rather than just two) is a convenient duplicate of the $\boxed{\text{INS}}$ key.

When in dot mode, the screen shows the contents of the planes as usual. The planes may already contain complex patterns, and some care is needed to tell whether the bit to be toggled is a 0 that will become a 1 or vice versa. To help in this, the target cell itself will blink: during each blinking cycle, the pixel spends one-third of the time in its proper color (which reflects the *current* state of the cell's four bits as specified by the color map) and two-thirds of the time in the color it would have if it were toggled. In this way, one has a *preview* of the change that $\boxed{\text{INS}}$ would make.

The normal cursor is a crosshair. When the expanded mode is on at the same time as the dot mode, the crosshair disappears and the position of the cursor is represented only by the target cell's blinking.

If the cursor is outside the 64×64-cell central area when you enter the expanded mode, the cursor will be moved into this area.

Examples:

- Type $\boxed{\text{z}}$ to clear all planes; the empty planes will look *black* under the `IRGB` color map (cf. Section 3.1). Now type $\boxed{\text{d}}$, and the cursor will appear, with the target cell blinking *gray* at a 2/3 duty-cycle. In fact, toggling the target bit will put a 1 in plane 0 (the "gray" plane under `IRGB`).

- Type $\boxed{\text{INS}}$ to toggle the target bit; the bit becomes a 1 , and correspondingly its duty-cycle becomes 1/3 gray and 2/3 black). Type $\boxed{\text{d}}$ to exit the dot mode, leaving a single gray pixel on the screen.

- Type $\boxed{\text{INS}}$ to turn the bit off.

- Type $\boxed{\text{d}}$, and use the arrow keys to move the cursor around. Now, while holding the $\boxed{\text{INS}}$ key pressed, use the keypad keys to draw some lines.[5] Type $\boxed{\text{z}}$ to clear the screen.

- Type 2$\boxed{\text{d}}$ $\boxed{\text{INS}}$ $\boxed{\text{d}}$ to get a single green pixel, corresponding to a single 1 in plane 2.

- Type $\boxed{\text{z}}$ $\boxed{\text{d}}$ $\boxed{\text{x}}$ to clear the planes and have the blinking green cursor on the expanded screen. While holding the $\boxed{\text{INS}}$ key pressed use the arrow keys to draw some lines. Now type $\boxed{\text{d}}$ $\boxed{\text{x}}$ to turn off both the dot mode and the expanded mode. The screen will show the final result of your graphics editing.

[5] If while drawing a line you happen to cross another line the pixel at the intersection will be toggled off again: intersecting lines are XORed with one another.

3.11 Plane buffers

The contents of one or more planes can be temporarily saved to memory buffers in the PC, and retrieved from there. Each plane has an associated buffer. Here we shall discuss mere data movements between planes and buffers. Logic operations involving planes and buffers are available too; they are discussed in Section 3.13.

The operations described in this section work with a modified meaning when the *cage* feature is turned on (see Section 3.12).

$\boxed{\text{p}}$	*Put to buffer.* Save plane(s) to corresponding buffer(s).
$\boxed{\text{P}}$	*Put to disk.* Save plane(s) to a specified file (see Section 3.14).
$\boxed{\text{g}}$	*Get from buffer.* Retrieve plane(s) from corresponding buffer(s).
$\boxed{\text{G}}$	*Get from disk.* Retrieve plane(s) from a specified file (see Section 3.14).
$\boxed{*}$	*Exchange.* Exchange contents of plane(s) and corresponding buffer(s).

(3.12)

With a numeric argument, the command affects only the specified plane (or plane pair) and the corresponding buffer(s).

Note that the names of all plane operations imply as a syntactical object the image on the screen. Thus, to remember the data direction of a *Put* operation in the present context, verbalize "put the plane." When the cage is active (cf. next section), verbalize "put the cage."

Examples:

- Type $\boxed{\text{G}}$`sample` to load pattern file `SAMPLE.PAT` ; type $\boxed{\text{p}}$ to save all four CAM planes to the buffers. If you clear the screen typing $\boxed{\text{z}}$, you can restore the previous image by typing $\boxed{\text{g}}$.

- Type $\boxed{\text{z}}$ to clear the screen and then `0`$\boxed{\text{g}}$ to restore only the contents of plane 0.

- Type $\boxed{;}$ to put the same random pattern on all four planes. Now each time you type $\boxed{*}$ you exchange the contents of the planes and buffers. Try `0`$\boxed{*}$ a few times.

3.12 The cage

It is useful to be able to copy a small portion of a screen pattern and move it to some other place on the screen; to prepare a "rubber stamp" with a certain picture and drop copies of this picture in several places on the screen; to apply commands such as *Zero* and *Random* only to a limited portion of the screen; etc.

The *cage* is a collection of four mini-buffers (one per plane) that can be used for operations of this kind. The horizontal size and the vertical size of the cage can be independently specified (in multiples of eight pixels, and up to a maximum size of 64×64 pixels). When the cage is active, it is visible as an overlay in the middle of the screen, on top of the planes. While the cage is visible, all operations that take a plane argument apply to the cage: the cage takes on the role of the screen, and the part of the screen hidden under it takes on the role of the buffers. Thus $\boxed{\text{g}}$ gets an image into the cage from the middle of the planes, and $\boxed{\text{p}}$ puts the image in the cage into the middle of the planes; $\boxed{;}$ puts randomness into the cage, and rotations and flips will apply to the cage rather than to the whole screen.

In other words, one may visualize three "levels" of storage, namely

cage
planes
buffers

and the operations normally defined in terms of the two bottom levels shift their meaning "one level up" when the cage is active.

Since the cage always appears in the middle of the screen, to "position the cage" one actually has to move the planes relative to the cage, using the usual *Shift* and *Hold* commands (which retain their usual meaning). The cage is of course used in exactly the same way in normal or expanded mode—the maximum-sized cage completely fills the expanded view.

The commands to select the cage are:

$\boxed{\text{c}}$	*Cage.* Toggle cage mode on or off. A numeric argument consisting of one digit in the range 1–8 specifies the size of the cage (the argument is multiplied by 8); with a two-digit argument, the first digit denotes the horizontal size, the second digit the vertical size. Whenever an argument is given, the previous cage is discarded and replaced by a brand new one.
$\boxed{\text{C}}$	*Uncage.* Turn off cage mode.

(3.13)

When the cage is active its outline is shown by four blinking corners. If the cage is empty (all `0` s in each of the four mini-buffers) it looks "transparent" (you can see the CAM planes underneath), and a solid frame with blinking corners indicates its extent. (The cage's extent includes the frame itself; the frame is drawn by showing the plane's bits as complemented.) On the other hand, if as much as a single bit in the cage is set (a symptom of this is that the solid frame disappears) the cage's contents (whether `0` s or `1` s) will entirely hide from view the underlying portion of the planes.

Examples:

- Type $\boxed{\texttt{G}}$`sample` to get the pattern `SAMPLE.PAT` , and `46`$\boxed{\texttt{c}}$ to activate a cage of horizontal size 32 (=4×8) and vertical size 48 (=6×8). You'll see a hollow rectangular shape with flashing corners: a new cage is empty, and the pattern underneath will show through.

- Choose a particular feature in the plane pattern, and use the arrow keys to bring it underneath the cage.

- Type $\boxed{\texttt{Z}}$ to fill the cage; now that feature is hidden behind the cage.

- Use the arrow keys to move the planes until the feature reappears from behind the cage: the CAM planes were not affected by activating the cage or writing something to it.

- Type $\boxed{\texttt{c}}$ or $\boxed{\texttt{C}}$ to deactivate the cage. The cage disappears from view, but its contents are not lost; type $\boxed{\texttt{c}}$ again to get the cage back on the screen.

- Now type $\boxed{\texttt{p}}$: the rubber stamp is lowered on the planes (the normal meaning of $\boxed{\texttt{p}}$, namely "put plane to buffer," is modified to "put cage to plane") which receive its imprint. Turn off the cage by typing $\boxed{\texttt{c}}$: the cage disappears, but its imprint on the planes remains. In a similar way, by typing $\boxed{\texttt{g}}$ in cage mode you can "capture" into the cage a fragment of the planes, creating a new rubber stamp.

- Type $\boxed{\texttt{z}}$ to clear the cage—the pattern underneath will again show through. Shift the pattern by means of the arrow keys, and capture some interesting piece of it using $\boxed{\texttt{g}}$. Use $\boxed{\texttt{r}}$, $\boxed{\texttt{|}}$, $\boxed{\texttt{-}}$, etc. to rotate or flip the captured piece.[6] Use this piece as a stamp, making several copies of it in different positions of the bit-planes by "lowering" the stamp with $\boxed{\texttt{p}}$ and moving the planes around with the arrow keys.

---

[6] If the cage is not square, rotations or diagonal reflections acting on all four planes will affect its *format* as well as its contents (for example, a quarter-turn will turn a 3×2 cage into a 2×3 one); if you try to do that on a single plane you get an error message.

- Make a smaller cage by typing 2$\boxed{\texttt{c}}$ (this is the same as typing 22$\boxed{\texttt{c}}$); fill the cage with random data (identical for the four mini-buffers) by typing $\boxed{;}$. Now stamp copies of this random template here and there on the planes. Do the same using a numeric argument for $\boxed{\texttt{p}}$, so that only a certain plane will be affected by the rubber stamp.

3.13 Logic operations

The plane editor provides commands for performing logic operations between planes and buffers; these operations are performed in parallel on all the sites of the array. For example, 0$\boxed{\&}$ will replace the contents of plane 0 with the logic-AND (see below) of this plane and the corresponding buffer. All the logic operations described take two inputs and return one output: the two inputs are a plane and the corresponding buffer, and the output is put in the same plane.[7] As usual, these commands are carried out on all four planes unless a numeric argument restricts their application to a single plane or plane-pair: $\boxed{\&}$ will AND each of the four buffers onto the corresponding plane.

The commands that perform logic operations are

$\boxed{\&}$	*And.* Logic-AND buffer(s) to plane(s).
$\boxed{+}$	*Or.* Logic-OR buffer(s) to plane(s).
$\boxed{\$}$	*Xor.* Logic-XOR buffer(s) to plane(s).

(3.14)

As a reminder, AND, OR, and XOR are defined as follows:

$$\begin{array}{c} \text{AND} \\ 00 \mapsto 0 \\ 01 \mapsto 0 \\ 10 \mapsto 0 \\ 11 \mapsto 1 \end{array}, \quad \begin{array}{c} \text{OR} \\ 00 \mapsto 0 \\ 01 \mapsto 1 \\ 10 \mapsto 1 \\ 11 \mapsto 1 \end{array}, \quad \begin{array}{c} \text{XOR} \\ 00 \mapsto 0 \\ 01 \mapsto 1 \\ 10 \mapsto 1 \\ 11 \mapsto 0 \end{array}.$$

Examples:

- Type 0$\boxed{\texttt{G}}$`circle` to load the pattern `CIRCLE.PAT` in plane 0, and 0$\boxed{\texttt{p}}$ to save this pattern (a disk-shaped mask) in buffer 0. Now type 0$\boxed{;}$ to put randomness in plane 0. Finally, type 0$\boxed{\&}$ to AND the buffer with the plane's contents. The randomness that lies outside the circle will be "masked out."

[7]In addition to these *dyadic* logic operations, we have already encountered the *monadic* ("one-input") operation $\boxed{-}$ (logic-NOT), which takes a specified plane as input and puts the result in the same plane, and the 0-adic operations $\boxed{\texttt{z}}$ ("set to 0") and $\boxed{\texttt{Z}}$ ("set to 1"), which take no input and put the result in a specified plane.

3.14 Disk Read/Write

It is possible to store on a disk file or retrieve from a disk file a verbatim image of one or more CAM bit-planes. We shall call such files *pattern* files. The default extension for a pattern file is PAT . The two basic commands are

P	*Put to disk.* Save plane(s) to a specified file. Each plane contributes 8192 bytes to the file.
G	*Get from disk.* Retrieve plane(s) from a specified file.

(3.15)

After the command keystroke, the control panel prompts you to type the name of a file. If no extension is given, PAT is assumed.

With no numeric argument, all four planes are saved or retrieved. If the command is preceded by a numeric argument, only the specified planes are affected.

If one tries to read from a file that doesn't have enough data, the file is "wrapped around." For example, if one tries to read all four planes from a file that only contains two, the remaining two planes are read starting again from the beginning of the file.

If the file name contains either '*' or '?', no data is transferred. Instead, the control panel prints a directory listing of all disk files whose names match the given string under the DOS naming conventions; after this, the control panel prompts you again for a file name. Typing ↵ without a file name produces a listing of all the files with extension PAT . This is handy if you don't remember what pattern files are present, or the exact name of a file.

If you change your mind, you can (as always) exit the prompt loop and abort the command by hitting the Esc key.

Examples:

- Type G`sample` to load all four screens with the contents of the file SAMPLE.PAT .
- Type z to clear all planes. Type 2G`sample` to load only plane 2 from the file SAMPLE.PAT , leaving the other three planes clear.
- Type P`scrap.tmp` to save all four planes to file SCRAP.TMP .
- Type 4P`scrap.tmp` to save planes 0 and 1 to file SCRAP.TMP ; the system warns you that you are about to overwrite an existing file, and gives you a chance to abort the command (by answering with `n` to the question "`Overwrite existing file? (Y/N)`").
- Type G ↵ to have a directory listing of all the files with extension PAT . After that, type the desired file name, or Esc to return to the control panel.

3.15 Other forms of bit-plane editing

Bit-plane configurations suitable for a given experiment can of course be produced in a variety of ways other than through the CAM plane editor. One way is to generate the desired pattern in a PC memory buffer, using Forth for this purpose as an ordinary programming language, and then copy this buffer to a CAM plane (see BUF>PDAT in the glossary).

Another possibility, especially for free-hand drawing of more complex figures, is to use a separate graphics editor and convert its output to the format of a CAM PAT file.

For example, the enclosed program HALOPAT.EXE converts output files generated by "Doctor Halo II"—a graphics editor that accompanies the MOUSE SYSTEMS mouse (Section 3.10)—to PAT files. Type

```
HALOPAT source-file dest-file
```

The file names must not include the extensions: the default extension PIC is provided for the source file, and PAT for the destination. If the second argument is missing, the destination file will have the same name as the source.

The resolution of "Doctor Halo II" is 320×200 pixels. As CAM bit-planes are 256×256, only the leftmost portion of the Dr. Halo picture is converted, and the last 56 lines are left empty. To avoid losing the portion of the picture located under the Dr. Halo menus, move them to the upper-right position of the screen (as explained in the Dr. Halo manual).

Part II

The programming environment

Chapter 4

The programming environment

How does the control panel actually control the operation of CAM?

The CAM module is plugged into the PC *bus*—a set of address, data, and control lines which are managed by the microprocessor contained inside the PC. To make CAM perform a certain activity one has to instruct the microprocessor to conduct a certain dialog with CAM via this bus.

To make a long story short, the `CAM.EXE` program which you execute when you want to run CAM consists of a number of program modules that have been written by somebody who knows CAM and the PC intimately, and of a main "dispatcher" program which continuously monitors the keyboard. Whenever, say, the $\boxed{s}$ key is hit, the dispatcher summons the program module in charge of telling CAM how to perform a step; when you hit $\boxed{G}$, the dispatcher summons the module in charge of loading a configuration from disk. Some of the program modules activated in this way open up a dialogue with *you*, and allow you to expand in an unlimited way the repertoire of things that CAM can be told to do.

The most "uncontrolled" way for you to gain control is (a) to summon (via the control-panel $\boxed{F}$ command) the *Forth interpreter*, through which all of the system's resources (PC and CAM) become directly accessible; at that point you are in charge, with all the advantages but also all the burden that this entails. Except for occasional *impromptu* interventions, you'd rather be able to (b) compose the "score" (cf. Section 5.5) at your leisure and (c) give it to CAM to perform (with the appropriate tempo) whenever you wish. These needs are addressed respectively by the *screen editor* (summoned by the $\boxed{E}$ command) and the *loader* (summoned by $\boxed{L}$).

In the rest of this chapter we'll briefly describe the procedures for accessing the Forth interpreter, the loader, and the editor from the control panel. More detailed information on the use of these programming resources is given in the following chapters and in some of the appendices.

4.1 Accessing the Forth interpreter

From the control panel, the *Forth* command ($\boxed{\text{F}}$ or $\boxed{\text{f}}$ keys) allows you to enter into direct conversation with the Forth interpreter; the control-panel prompt—namely the "happy face"—disappears and the interpreter itself will prompt you with an `ok` (or an appropriate error message) after processing each line you enter at the keyboard (see Chapter 5). As usual, $\boxed{\text{ESC}}$ returns you to the control panel.[1]

4.2 Accessing DOS, and returning to DOS

The normal way to leave the `CAM` program and return to DOS is to go to the Forth interpreter and type `BYE`.

It is also possible to execute DOS commands without abandoning the `CAM` program. From the control panel, the $\boxed{\text{\^{}}}$ key will access the DOS command interpreter; typing `EXIT` will return control to the panel. DOS can also be accessed from Forth, as explained under `EXEC` , `COM` , and `DOS...` in the glossary.

4.3 Program files

Early implementations of Forth, where a full-blown disk operating system was not available or desirable, let Forth directly manage the contents of the disk. For this purpose, the entire disk was organized as a collection of sequentially-numbered *blocks* of 1024 bytes each.

The present F83 implementation uses DOS as an operating system, thus indirectly providing all of the DOS services. In particular, a Forth program can access an arbitrary number of named files. However, within each of these files the traditional block structure is retained.[2]

Blocks used for storing Forth code or other text material are called *screens* (for this reason, a block file may also be called a "screen file"). Each 1024-character screen is logically organized as 16 lines of 64 characters each. Since all lines have the same length, lines are not terminated by an end-of-line control character as in conventional text files.[3]

[1]This key is sensed only when the system is waiting for keyboard input; to forcibly interrupt execution of a Forth word, use $\boxed{\text{BREAK}}$ (cf. Section 2.3).

[2]Future implementations will likely drop the block structure for text files entirely.

[3]For this reason, attempting to view or list a Forth screen file using the ordinary DOS facilities such as `TYPE` , `COPY` to printer device, etc. yields unreadable results (the whole file looks like a single unbroken line). Forth's own listing facilities are discussed in Section 7.5.

The numbering of screens within a file starts from 0. However, screen 0 itself cannot be interpreted by the loader (though it can be edited, listed, used as virtual memory, etc.). This screen is typically used for extended comments (e.g., a brief description of the file's contents, or directions for its use).

For mnemonic convenience, all program files in the CAM Forth system have been given extension 4TH —except for CAM experiments, which have extension EXP .

4.4 The loader

Loading a screen or a sequence of screens has essentially the same effect as typing their contents from the keyboard at the Forth interpreter level. For anything that involves typing more than a dozen words of Forth it is more convenient and reliable to compose the text using the screen editor, save it in a screen file, and from there feed it to the interpreter by means of the loader.

The loading commands are

$\boxed{\text{l}}$	*Load.* Load the current file.
$\boxed{\text{L}}$	*Load new.* Load a specified file, and make it the current file.

(4.1)

The command $\boxed{\text{L}}$ is used for loading an experiment from disk; it works in a way analogous to $\boxed{\text{G}}$ (which is used for loading a screen pattern; cf. Section 3.14). $\boxed{\text{L}}$ prompts you to type the name of the desired experiment file, and supplies the default extension EXP to the file name you type.

The control-panel loader first prints all nonblank lines from screen 0 of the file, as a comment, and then loads all the screens from 1 on.

When a file is loaded with $\boxed{\text{L}}$ (or edited with $\boxed{\text{E}}$), its name is stored in the *current-file* register. The lower-case command $\boxed{\text{l}}$ has the same effect as $\boxed{\text{L}}$, but looks in that register for the file name,[4] so that you don't have to type it again if you subsequently want to access the same file. (Typically, you may want to make a minor change in the file and try the experiment again).

As with $\boxed{\text{G}}$, you can request a directory listing of all EXP files, or of all files matching a certain template. If you enter a *Load* command by mistake there are two ways of getting out of it. If the control panel is waiting for you to type the name of a file, just type $\boxed{\text{Esc}}$—the usual way to return to the control panel. If loading has already started, you can type $\boxed{\text{Break}}$ (cf. Section 2.3) to abort the loading.

[4]The control panel will complain if you type $\boxed{\text{l}}$ when no file has yet been made current: you must first explicitly supply a file name with $\boxed{\text{L}}$ or $\boxed{\text{E}}$.

If $\boxed{\text{L}}$ or $\boxed{\text{l}}$ is preceded by a numeric argument n, instead of the whole file only screen number n is loaded (but see `-->` , `THRU` , `+THRU` , and `INCLUDE` in the glossary), and the comments on screen 0 are not printed.

4.5 The editor

The commands to enter the screen editor are:

$\boxed{\text{e}}$	*Edit.* Edit the current file.
$\boxed{\text{E}}$	*Edit new.* Edit a specified file, and make it the current file.

(4.2)

As with the *Load* commands, a numeric argument specifies a screen number—in this case the screen to be edited. If you specify a screen number past the end of the file, you will begin editing at the last screen of the file. If $\boxed{\text{E}}$ is used without an argument, the editor starts at screen 1. As with $\boxed{\text{L}}$, you can get a directory listing by using wildcard characters in the filename, or by hitting $\boxed{\hookleftarrow}$ instead of typing a file name.

If the file name specified with $\boxed{\text{E}}$ doesn't already exist, the program will offer to create a file with that name, initially consisting of two blank screens.

An editing session is terminated by typing $\boxed{\text{Esc}}$ (any modified screens are automatically saved to disk either during the session or at the end of it). The editor remembers at what screen and cursor position you were when you exited it. If you subsequently re-enter the editor by typing $\boxed{\text{e}}$ with no argument, you return to where you left off.

You also enter the editor automatically if an error is encountered during loading—in this case the cursor will be positioned at the point just after where the error was detected, and an error message will appear below the editing window.[5]

The commands available from within the editor are explained in Chapter 6.

4.6 CAM programming

The editor and loader together allow the construction and use of your own CAM experiments. The editor is discussed in detail in Chapter 6. Chapter 7 discusses a number of utilities mostly concerned with program maintenance.

[5]The most likely error is for the loader to encounter an undefined word; in this case the error message is simply a question mark.

The remainder of this manual is devoted to explaining how to write your own CAM experiments. We begin with a Forth tutorial, in the next chapter.[6] Part III is devoted to the programming of CAM itself.

4.7 F83

Incidentally, among the pieces of software distributed with CAM you'll find the base F83 system—i.e., "Forth without CAM"—with some revisions and improvements over Perry and Laxen's original *F83 model.*

You enter this bare F83 system by typing `F83` at the DOS level (make sure that the file `F83.EXE` is on line); `BYE` returns you to DOS.

[6]This is a slightly extended version of the tutorial appearing in the *CAM Book*.

Chapter 5

A Forth tutorial

The main purpose of this tutorial is to give you an overall *reading* familiarity with Forth—enough to follow the CAM programming examples given in this book and in the accompanying software.

Relatively little knowledge of Forth is needed to compose full-fledged CAM experiments: the same few constructs appear over and over with minor variations, while many features of the Forth environment that are prominent in other programming contexts are not required at all.

However, these few constructs must be understood well. In many cases an intuitive presentation will be sufficient, but we shall not hesitate to give the appropriate amount of technical detail in those few cases where this is necessary to insure exact comprehension.

To practice this tutorial, you have two choices. You can work with the bare Forth system, by running the program F83.EXE from the DOS level; or you can run CAM.EXE as you have been doing so far, and access the Forth interpreter from the control panel by issuing the command $\boxed{\text{F}}$ (or $\boxed{\text{f}}$).

5.1 The command interpreter

You may visualize the Forth *command interpreter* as a competent but not-too-literate technician who sits in the machine room of your computer and has access to all the levers and dials. From the deck, you speak to him through an "intercom"—i.e., your terminal—issuing orders and receiving reports and acknowledgements (see Sections 4.1 and 4.2 for how to summon the command interpreter). For instance, if you say

```
BEEP
```

(type it at the terminal, followed by a carriage return) the terminal will respond with a "beep;" if you say

```
    0 100 DUMP
```

the contents of the first 100 memory locations (starting from location 0) will be dumped on the screen. After that, the interpreter will say

```
    ok
```

to tell you it's done and ready for a new command. (From now on, we'll take this carriage-return and `ok` business for granted.)

Saying `BYE` dismisses the interpreter; you are then automatically returned to the context (namely, the CAM control panel or the operating system) from which you had gained access to the interpreter itself.

The interpreter's "ears" are conditioned to break up the input character stream into *tokens*, using "blank space" (one or more consecutive spaces) and *only* blank space as the token separator.[1] Even though some Forth tokens consist of a single character (including comma, period, semicolon, etc., which in other computer languages are often used as "punctuation marks") one should be careful not to omit the blank space between them and adjacent tokens. The tokens are passed on one-by-one to the interpreter's "brain" and the spaces are discarded. Thus, the interpreter hears the above command as a sequence of three tokens— `0` , `100` , and `DUMP` —and would hear the same thing if you typed, say,

```
                    0 100             DUMP
```

In order to be understood by the interpreter, the two of you must share a *dictionary* of terms and some miscellaneous conventions, which together make up the Forth language. The dictionary's contents reflect the range of things that the interpreter currently knows about. As you take command, you'll find that the Forth interpreter has already gone through "standard training"—and possibly some additional, more specialized training (for example, how to run a CAM machine). This standard training, which is documented in any good Forth manual, is more extensive than that of many common computer languages; in this sense, one speaks of Forth as a programming *environment* rather than just a programming *language*.

5.2 The compiler

The entries that make up the Forth dictionary are called, as you might expect, *words*. To *program* in Forth, you successively add new words to the dictionary—defining each new word in terms of existing ones. In this way you extend the interpreter's knowledge—and at the same time your expressive range—with regard

[1]At the interactive terminal, carriage return acts, of course, as a token terminator. For "canned" text submitted to the interpreter from a disk file, cf. footnote 6.

to the set of activities you are interested in. At any stage of this construction you may say something that uses the new words, and check that its actual meaning (i.e., what the *interpreter* does in response to your words) is what *you* had in mind.

For example, to make up a new word for "beep three times" you type

```
: 3BEEP   BEEP BEEP BEEP ;
```

As soon as it sees the colon token (':'), the interpreter summons the aid of another technician, called the COLON *compiler*. This technician takes the token that immediately follows, in this case 3BEEP , and starts a new dictionary entry under that name. After that, it expects a phrase (a sequence of tokens) describing the action of the new word; this phrase, namely BEEP BEEP BEEP , is not executed at this time, but is compiled in the dictionary as the meaning of 3BEEP . The end of the phrase is marked by a semicolon token (';')—don't drop the space before it—which tells the COLON compiler to return control to the command interpreter.

The language understood by the COLON compiler is slightly different (and somewhat richer) than that understood by the command interpreter; some things only work with either the compiler or the interpreter, but not with both. Except when noted below, everything we will discuss works well either way.

If you now type

```
3BEEP
```

the interpreter will look up this word in the dictionary, execute it, and respond with the usual ok . If you type, say,

```
4BEEP
```

the interpreter will look it up but won't find it in the dictionary; it will then ask an assistant whether it might possibly be a number (see next section); and finally will print

```
4BEEP ?
```

to tell you it can't make sense of the token 4BEEP .

5.3 The dictionary

If you type FORTH WORDS , you'll get a listing of all the words currently contained in the main section of Forth's dictionary,[2] starting with the ones Forth has learned most recently. Thus, if you had just had the above conversation with the interpreter, the word 3BEEP would be on top of the list. The older word BEEP would appear somewhere down the list.

[2]In addition to this main section, called FORTH , the dictionary may contain some additional specialized sections, as will be explained in a moment.

The following entries

```
BEEP  HERE  +  '  CONSTANT  C@  ;  :  C!  0=
```

taken at random from the dictionary give you an idea of what typical Forth words look like. *Any* token can be entered in the dictionary as a word. In particular, characters that in other languages are used as punctuation marks, such as '`:`', can appear as part of a Forth word, or even make up a word all by themselves; the word '`C,`' is very different than the two-word sequence '`C ,`'.

A word already present in the dictionary may be redefined by you. For instance, if the speaker in your terminal is dead, as a temporary fix you may use a version of `BEEP` that prints on the screen, on a new line, the message '**`Believe it or not, this is a beep!`**'. This is done by redefining `BEEP` as follows

```
: BEEP    CR ." Believe it or not, this is a beep!" ;
```

Type the above line exactly as it appears (there must be at least one space after `BEEP` , after `CR` , after `."` , and before the `;`). The word `CR` will execute a "carriage return" (moving the terminal's "printing head" to the beginning of a fresh line); the construct `."` ⟨text⟩`"` will print ⟨text⟩ on the screen (this construct can only be used within a COLON definition). If you now define

```
: 4BEEP    BEEP BEEP BEEP BEEP ;
```

this word will be compiled using the *new* version of `BEEP` , and when executed will print

```
Believe it or not, this is a beep!
Believe it or not, this is a beep!
Believe it or not, this is a beep!
Believe it or not, this is a beep!
```

The old version of `BEEP` is not deleted from the dictionary, and the previously defined word `3BEEP` *will retain its original meaning*—which is tied to the *old* version of `BEEP`). If you say `3BEEP` now, the three beeps will still be routed to the (dead) speaker.

It's all right, and often useful, to redefine a word in terms of its previous namesake. For instance, if you discover that beeps in your terminal have such a long decay time that three consecutive beeps sound more like a long one, you may redefine `BEEP` as

```
: BEEP    BEEP 10 TICKS ;
```

(where `10 TICKS` means "Wait for ten ticks of an internal computer clock"), and a sequence of beeps will now sound "staccato" rather than "legato."

In some English dictionaries, words belonging to certain specialized areas of discourse are listed in separate sections (e.g., geographical names, measurement units, abbreviations). In Forth, these sections are called "vocabularies;" in addition to the main FORTH vocabulary there may be an ASSEMBLER vocabulary containing machine-language op-codes and other assembler-specific terms, an EDITOR vocabulary containing editor-specific terms, etc. As explained in Section 5.17, there are commands available for specifying which vocabularies should be searched at any particular moment, and in what order.

5.4 Numbers

If in an ordinary piece of English text you find a phrase such as 'the motion was passed with 371 votes in favor', you won't look up '371' in the dictionary—and for that matter you wouldn't find it there if you did. The meaning of a number derives from its make-up; this makes it possible for any fool to produce more numbers than any one would want to list, but at the same time makes it unnecessary to list the meaning of individual numbers in a dictionary.

The situation is analogous in Forth: numbers are parsed as they are encountered and their meaning is reconstructed from their make-up by another technician, called NUMBER , summoned by the interpreter as the occasion arises. This "meaning" is nothing but an internal representation in binary form. For instance, if the interpreter sees the token 100 while operating in DECIMAL mode, this token will be recognized as a number and will be internally converted to 0000000001100100 (Forth stores integers in 16-bit cells); however, if you have told the interpreter to operate in HEX mode,[3] the same token will be given the meaning 0000000100000000.

Some numbers that for historical or practical reasons deserve explicit mention, such as 'three', *are* listed (in a spelled-out form) in the English dictionary. In an analogous way, some common numbers such as 0 and 1 have been entered (in this form, i.e., 0 —not ZERO) as *words* in the Forth dictionary.[4] This leads to more efficient execution (their meaning has been established in advance, once and for all, and there is no need to ask for the help of the NUMBER technician) and more compact code.

By default, Forth handles numbers as signed integers with a 16-bit range (only 2^{16}=65,536 different values are available) so that the ordinary counting sequence -2,-1,0,1,2,3,... at a certain point "wraps around": ...32765,32766,32767,-32768,-32767,-32765,...,-2,-1,0 (cf. Section 5.15). We'll not try to convince you that what is most natural for a computer (or for a person that lives with computers) should be the most natural thing for you; indeed, while these features of Forth's

[3]I.e., base sixteen rather than base ten. In Forth, numbers can be read and printed in any base you choose.

[4]Words of type CONSTANT; cf. Section 5.8.

"numbers" are actually an asset in low-level programming tasks, serious numerical applications require much more powerful number-handling facilities. The Forth philosophy is that specialized applications should be served by specialized extensions to the language (cf. Section 5.16).

5.5 The stack

Forth manages to achieve remarkable expressive power and efficiency; yet a Forth *system* (i.e., the language as implemented on a computer) can be amazingly simple and compact. These advantages are bought at the cost of ruthless standardization, in particular in the way nearby words in a phrase communicate to one another the information that binds them together as a semantic whole.

For the purpose of this communication, all data (characters, Boolean variables, numbers, addresses, etc.) are packaged in a standard-size "carton," called a *cell*, having a capacity of 16 bits, and all data exchanges take place—in a preordained choreography, as we shall see in a moment—through a single clearing-house called the *stack*. This is just a pile of cells that grows or shrinks according to the traffic.

Imagine a stage with this stack in the middle. The Forth interpreter is the ballet conductor; as he reads off the words that make up your phrase, the corresponding actors show up in sequence, do their thing, and disappear. If an actor's part tells him to leave behind certain data for later actors, he'll walk to the stack and pile these data, cell by cell, on top of it; if his part expects data from previous actors, he will walk to the stack and pick them up.

Some data that an actor needs may end up, say, buried two cells deep into the stack. The actor won't go fumbling through the stack looking for them (the cartons don't carry a label!); rather, his score will explicitly tell him to lift just the top two cells, grab the cell that is now on top, and put the first two back down (incidentally, the score notation specifying this sequence of actions is `ROT` , discussed in Section 5.12).

With this scheme, there is no need for each piece of data to have an *absolute* address—a permanent mailbox with a distinguished name. Instead, all addressing is by position *relative* to the top of the stack. If a new, self-contained piece of choreography is inserted in the old score, at the moment of executing it one will find the stack already built up to a certain height; during execution of this piece one will see the stack grow more, shrink a bit, etc., and by the end of the inserted piece of choreography return to its original height. The rest of the score will then resume, finding its own data where they had been left.

5.6 Expressions

The stack discipline is well suited to the communication needs of a hierarchically built program. It allows one to use a particularly simple scoring notation—called *reverse Polish notation*—by which arithmetical and logic expressions of arbitrary depth can be written without making recourse to parentheses or other place markers.

When you type a number to the command interpreter, this number is packaged in one cell and put on top of the stack. The one-character word '`.`' ("dot") picks up the top cell of the stack and prints its contents *as a number* on the screen. Thus, if you type

```
356 .
```

(where the "dot" is part of what you type) the screen will respond with

```
356
```

(In a more conventional programming language, the equivalent of '`356 .`' would be something like '`print(356)`'.) Note that the stack went up one level with `356` , down one level with '`.`', and is now the same height as before.

The word '`+`' ("plus") gobbles up the top *two* cells of the stack, adds them together, and places the result—consisting of *one* cell—on top of the stack; thus, it leaves the stack one level lower than it found it. For example, the expression

```
2 3 +
```

will leave the result 5 on the stack (from where you can move it to the screen with '`.`'). If you want to see how much $1+2+4$ is, you type

```
1 2 + 4 + .
```

We can picture the evolution of the stack as follows

```
STACK   INPUT  OUTPUT
...       1
... 1     2
... 1 2   +
... 3     4
... 3 4   +
... 7     .       7
...
```

where each row displays (a) the current state of the stack (with the top element on the right), (b) the text to be interpreted, and (c) what is printed on the screen. The dots on the left indicate the part of the stack that we haven't touched; this indication will be dropped in the following stack examples.

Note that once 3 has been placed on the stack, it does not matter *how* it got there; from a functional viewpoint, the expression `1 2 +` is interchangeable with, say, `3` , or `1 1 + 1 +` , or anything that eventually bows out having left just a 3 on top of whatever else the stack contained before. Note also that the two different expressions

`1 1 + 1 + 1 +` and `1 1 1 1 + + +`

produce the same end result even though the second one temporarily builds up the stack to a higher level:

STACK	INPUT	STACK	INPUT
	1		1
1	1	1	1
1 1	+	1 1	1
2	1	1 1 1	1
2 1	+	1 1 1 1	+
3	1	1 1 2	+
3 1	+	1 3	+
4		4	

5.7 Editing and loading

Once you have said something to the Forth interpreter, you cannot take it back; if something goes wrong, you may not even remember exactly what you said.[5] While immediate interaction with the interpreter is very useful, there are times where you would like to carefully think out in advance a whole sequence of commands and definitions, review and edit it, and perhaps discuss it with somebody else before you give it to the interpreter. You want to be in a position to give *written* orders, and you might have a collection of different "orders of the day" to be handed to the interpreter according to the circumstances.

You can do all of this by first writing your text on a disk file, where it can be inspected and modified by means of the Forth *screen editor*. Then you can ask the interpreter to use this file (or a portion of it) as the input stream, instead of what comes from the keyboard; this process is called *loading*. When you load a file, everything (well, *almost* everything) works as if you were typing the file's contents from the keyboard—except that the interpreter now processes your tokens as they come, without waiting for a carriage return.[6]

[5]But see `SEE` in Section 7.1.

[6]A typical Forth source file is organized as a collection of text "screens;" the lines that make up a screen are stored one after the other in the file without any intervening separation marks. Thus, within a text screen, end-of-line does *not* act as a carriage return or as a token separator (cf. footnote 1 in this chapter); on the other hand, the physical end of the screen does.

The commands for accessing the editor and the loader directly from the control panel without explicitly going through the Forth interpreter are discussed in Sections 4.5 and 4.4; from within the Forth interpreter, use `EDIT` and `LOAD` (Sections 7.3 and 7.4).

When you compose your text with the editor for subsequent loading, you may choose to format it in a way that facilitates comprehension, and here and there add a comment to yourself.

The formatting scheme used in this book is the following

```
                                : 3BEEP
               BEEP BEEP BEEP ;
                                : BEEP
                          BEEP
                      10 TICKS ;
                                : 4BEEP
          BEEP BEEP BEEP BEEP ;
```

where dictionary entries are lined up on the right half of the page and the "bodies" of the definitions are segregated on the left half.

The Forth word `(` removes from the input stream everything that follows, up to and including the matching character ')'; thus, you may write a line as follows

```
BEEP BEEP ( two beeps ) BEEP BEEP ( two more )
```

and the interpreter will never hear what is "in parentheses."[7]

The word '\' ("backslash") treats as a comment the remainder of the line on which it appears

```
                                : BEEP   \ New version!
                          BEEP           \   plain beep
                      10 TICKS ;         \   insert delay
```

5.8 "Constants" and "variables"

In Section 5.2 we said that the interpreter "executes" the words you type. Actually, each word in the Forth dictionary carries a notice saying "I am to be executed by technician so-and-so, who knows how to handle me," and the interpreter will just pass the buck to this technician. For words that have been

[7]Observe that the following spacing is correct too

```
BEEP BEEP ( two beeps)BEEP BEEP ( two more)
```

even though there is no intervening space between ')' and `BEEP`, since the effect of the word '(' is precisely to throw away the string '`two beeps)`'. On the other hand, the following spacing

```
BEEP BEEP (two beeps) BEEP BEEP (two more)
```

won't do, since it will make the interpreter think that '`(two`' is a token to be processed.

entered in the dictionary by the COLON compiler, the competent technician is the COLON *interpreter*. In general, each type of word has its own compiler and a corresponding interpreter. The buck stops with words that have been compiled by the CODE "compiler;" this technician actually produces code written directly in machine language (i.e., your microprocessor's native language), and is more properly termed an *assembler*. At this point the hardware takes over.

All of this works much more simply than it sounds. Suppose you are writing a telescope-driving program that needs to know your town's latitude, say, 43°. It is good programming practice to give this number a *name*—say, `LATITUDE`—so that whenever this name appears in your program it will have the same effect as if you had typed '43'. To do this, in Forth you say

```
43 CONSTANT LATITUDE
```

The word `LATITUDE` will be entered in the dictionary as a constant, and when executed it will place the number 43 on the stack.

What happens is that as soon as it sees the word `CONSTANT` the command interpreter summons the aid of the CONSTANT compiler, (cf. Section 5.9) who gobbles up the next token—namely `LATITUDE`—and starts a new dictionary entry under that name. The entry will consist of two parts: the first (*code field*) contains a notice saying "I am to be executed by the CONSTANT interpreter;" the second (*data cell*) is reserved for the value of the constant. At this point the CONSTANT compiler takes the top cell of the stack—with the 43 you had just put there—and moves it to the data cell in the dictionary. At execution time, the CONSTANT interpreter will look at the data cell and place a copy of it on the stack.

With the above definition of `LATITUDE`, the command

```
LATITUDE 7 + .
```

will print `50` on the screen.

The term 'CONSTANT' is somewhat of a misnomer (though it is retained for historical reasons), since the contents of the data cell may be altered at will; in the present implementation of Forth, to change the value of `LATITUDE` to 45° you say

```
45 IS LATITUDE
```

The most relevant aspect of a Forth CONSTANT is that it returns the *value* of its data cell, rather than a *pointer* to it (cf. `VARIABLE`) below).

In CAM, "neighbor words" such as `NORTH`, `SOUTH`, etc. act like Forth CONSTANTs insofar as they return a value; however, this value will change very many times during the construction of a rule table.

Forth provides another mechanism for accessing a piece of data, namely by its *address*[8] rather than by its *value*. The VARIABLE compiler, used in a construct such as

```
VARIABLE TIME
```

is analogous to the CONSTANT compiler insofar as it constructs a dictionary entry, namely `TIME`, with a data cell in it. However,

- This data cell is not initialized to a particular value (and therefore the defining word `VARIABLE`, unlike `CONSTANT`, does *not* expect a value on the stack).
- When `TIME` is executed, the VARIABLE interpreter puts on the stack the *address* of the data cell, rather than its contents.

Thus, if we type

```
TIME
```

what will be placed on the stack is not the current value of `TIME` (which may change several times during execution of the program), but its address (which is always the same).

To get the *value* of `TIME` you use the word '`@`' (pronounced "fetch"), as in

```
TIME @  ( data )
```

(i.e., '`@`' expects an address on the stack, and replaces it with the data at that address), and to set it you use the word '`!`' ("store"), as in

```
( data ) TIME !
```

(i.e., '`!`' expects a piece of data *and* an address, and ships the data to that address). For example, to increment time by one unit you write

```
TIME @ 1 + TIME !
```

Supposing that the `TIME` data cell is at location 1000 and its initial contents is 5, the evolution of the relevant data is the following

TIME's contents	STACK	INPUT
5		TIME
5	1000	@
5	5	1
5	5 1	+
5	6	TIME
5	6 1000	!
6		

[8] In an ordinary computer, memory locations are sequentially numbered; the address of a piece of data is the number of its location.

(The word `+!` allows you to use the more efficient construct `1 TIME +!` , where the cell's value never appears on the stack. Similarly, `TIME OFF` will store all `0` s in the cell at the address supplied by `TIME` , and `TIME ON` will store all `1` s, corresponding to the logic values `FALSE` and `TRUE` —cf. Section 5.15.)

From the viewpoint of CAM's user, Forth VARIABLEs need seldom be used. In this book, the term 'variable' always has the usual meaning of 'a generic quantity to which we may assign an arbitrary value' rather the the more technical Forth meaning, for which we reserve the small-caps term VARIABLE.

5.9 Defining words

The words `:` ('colon'), `CONSTANT` , and `VARIABLE` belong to the class of *defining words*, whose action is to create new dictionary entries.

In Forth, a number of defining words of common utility are built-in; each one activates a miniature compiler (and, at execution time, a miniature interpreter) for a specific data structure. If an application is going to make regular use of other data structures, it is possible (and quite easy) to introduce new, appropriate defining words. Similar considerations apply to *control constructs*, discussed in the following sections.

In other words, while, for the casual user, programming will consist merely of extending the *lexicon* of the received Forth system, the more sophisticated user will find it both expedient and easy to extend Forth's very *syntax*. Practically every major programming effort can benefit from a moderate investment in "language development."

The developers of the CAM system have made extensive use of the above features of Forth.

5.10 Iteration

Once you've stored a program in the computer's memory, portions of it can be executed over and over, perhaps with some variations.

For instance, you can define

```
            : BEEP-STUCK
    BEGIN
   BEEP
    AGAIN ;
```

When this word is called, once the execution reaches `AGAIN` it jumps back to `BEGIN` , producing an endless series of beeps. Short of turning off the power, there is no way you can get out of this loop.[9]

The above BEGIN / AGAIN pair delimits a phrase somewhat like a pair of parentheses: the phrase in between gets iterated forever. Note that for readability we vertically aligned the two elements of the pair, flush on the *right* (this is recommended in reverse-Polish-notation style), and indented the phrase inside.

A more flexible pair is DO / LOOP ; the word

```
                 : 100BEEPS
        100 0 DO
          BEEP
            LOOP ;
```

will beep 100 times.[10] What happens in detail is that DO gobbles up the top two numbers on the stack, 100 (the loop LIMIT) and 0 (the loop INDEX), and saves them for later use. The execution proceeds until LOOP is encountered. At this point, INDEX is incremented by one and compared with LIMIT: if INDEX=LIMIT the loop is terminated; otherwise, execution jumps back to DO .[11]

Inside a DO loop, the word I returns on the stack the current value of the index. Thus, the word

```
                       : PRINT-ASCII
127 32 DO I EMIT LOOP ;
```

will produce all printable ASCII characters. (Character 0–31 are control characters; all others are printable except for the last character of the ASCII set, namely character 127, which on old-fashioned paper tape "prints" 7 holes on the last punched character, effectively DELETEing it.)

Pairs of "parentheses" such as BEGIN / AGAIN and DO / LOOP can be nested as ordinary parentheses, and a lot more bells and whistles are available. You can look up the details of these and other flow-control constructs in a Forth manual. Here we shall only mention that flow-control constructs can only appear inside a COLON definition: you cannot say

```
100 0 DO BEEP LOOP
```

at the command-interpreter level.

[9]Unless your computer has a working BREAK key—the equivalent of a "panic button."

[10]Of course if you had typed 999BEEPS you still would have gotten a word that beeps 100 times. A name is a name is a name...

[11]The loop index is a modulo-2^{16} counter. The minimum number of iterations, namely 1, is achieved when the loop is entered with INDEX just one less than *limit*; the maximum, when the loop is entered with INDEX *equal* to LIMIT: 0 0 LOOP will cycle 2^{16} times!

5.11 Stack comments

By now, we have encountered many words that expect arguments on the stack or leave results on the stack. Since breaches of stack discipline may send a computation berserk (if you leave an extra item on the stack everyone after you will get his data wrong), it will be convenient to have a notation to remind us of just how many items a word takes from the stack or leaves on it.

The following are examples of *stack comments*:

```
DO        ( n1 n2 -- )
DO        ( limit index --)
BEEP      ( -- )
LATITUDE  ( -- n)
TIME      ( -- addr)
+         ( n1 n2 -- n3)
+         ( m n -- m+n)
2         ( -- n)
2         ( -- 2)
```

The general convention is as follows. We put in parentheses a "dash" (customarily a double-dash) to indicate the word in question. Before the dash we write a list of what the word *expects* on the stack; after the dash we write a list of what the word *leaves* on it. If nothing appears *after* the dash, the dash itself is usually omitted.

What really counts is the *number* of items in each list—which corresponds to the number of cells taken from or given to the stack; but the items themselves may be elaborated upon a little, for extra clarity.

For example, `DO` takes two items and leaves none. A minimal notation is

```
DO        ( n1 n2 -- )
```

which just tells us that `DO` expects two items. A better mnemonic is provided by

```
DO        ( limit index --)
```

which reminds us that the first item is used as the *limit* and the second as (the initial value of) the *index* of the loop.

As a final grand example, let us load the following three definitions from a disk file

```
  100 CONSTANT HUNDRED  ( -- n)
10000 CONSTANT A-LOT-OF ( -- n)
             : BEEPS    ( n --)
      0 DO
 BEEP LOOP ;
```

and then type the following three commands

```
    3 BEEPS
    HUNDRED BEEPS
    A-LOT-OF BEEPS
```

We have seen above that DO wants two arguments. When we type 3 BEEPS , the first argument is left on the stack by the 3 we typed, while the second is placed on the stack by the 0 appearing within the definition of BEEPS : DO 's hunger is satisfied.

Note that if we define words giving a little thought to the "stack interface" (*who* should supply or consume *what* and *when*) and to choosing appropriate names, the flow of a Forth phrase can be given a natural-language flavor that is hard to achieve in other programming languages. Properly trained Forth words can talk to one another under the surface of the phrase, without bothering us with their chatter.

A generally obeyed convention in Forth is to make words "use up" their arguments rather than leave them on the stack. If an object on the stack is needed as an argument by a given word and also by another word that closely follows, a second copy of this object is made—using the word DUP introduced below—before the first copy is used up.

5.12 DUP, DROP, and all that

The Forth word '*' ("times") takes two numbers and returns their product; to compute the square of 3 you have to type 3 twice: 3 3 * . How about a word SQR that will take a *single* argument and multiply it by itself?

Forth provides a number of general-purpose words for manipulating the stack; one of these is DUP (pronounced "dupe"), which looks at the cell on top of the stack and puts a *duplicate* copy of it on top of it; e.g.,

```
STACK  INPUT
5      DUP
5 5
```

Thus, SQR can be simply

```
      : SQR ( n -- n*n)
DUP * ;
```

since here '*' will see two copies of the argument.

Related to DUP are DROP (which drops the top item from the stack), SWAP (which swaps the top two items), OVER (which makes a copy of the next-to-the-top stack item), ROT (which pulls the third item from underneath and puts in on top of the first two), and a few more. Words of this kind act somewhat like *pronouns* in English ('this', 'that', 'one another', etc.), in that they allow one to refer by *position* rather than by *name* to objects introduced in a different part of the sentence. As an exercise, verify that the function $y(m,n) = (m+n)(m-n)$ is computed by the following Forth expression

```
    ( m n )  OVER OVER + ROT ROT - *  ( y )
```

(where the comments tell you what's on the stack before and after).

5.13 Case selection

A flow-control construct that is extensively used in programming CAM is the CASE statement, which allows one to select for execution one of several alternative actions. Suppose we have three words called BEEP , HONK , and WHISTLE ; we can then make up a new word, called SOUND , which will take an integer argument from the stack, with value 0, 1, or 2, and respectively beep, honk, or whistle:

```
                           : SOUND ( n --)
        { BEEP HONK WHISTLE } ;
```

That is, 0 SOUND will execute BEEP , 1 SOUND will execute HONK , and so on. The selection list may consist of any number n of entries, which are to be thought of as consecutively numbered from 0 upwards, and is delimited by the two "brace" words, namely '{' and '}'. If you attempt to execute the case statement with an argument that is less than 0 or greater than $n-1$ you get an error message.

Suppose you want to make up a word that returns the number of days in a month. You'd probably try the following

```
                           : DAYS ( month -- days)
                       1 -
        { 31 28 31 30 31 30
          31 31 30 31 30 31 } ;
```

Assuming that months are numbered 1, 2, ..., 11, 12, you subtract 1 in order to have the numbering 0, 1, ..., 10, 11—better suited to the case statement—and then you look up the number of days. The reasoning is correct, but there is one minor catch: in CAM Forth the case statement only accepts individual dictionary words as items in the selection list; with a few exceptions (mentioned in Section 5.4) numbers are *not* in the dictionary. There is an easy fix to this problem: *before* defining DAYS , enter the desired numbers in the dictionary as *constants*

```
28 CONSTANT 28 ( -- 28)
30 CONSTANT 30 ( -- 30)
31 CONSTANT 31 ( -- 31)
```

From this moment the token `28` (for one) will be recognized as a *word*—one that leaves a 28 on the stack just as the *number* `28` used to do before. The case statement will now accept it as a list item.

5.14 Conditional statements

The phrase between a `BEGIN` and `AGAIN` pair is iterated forever, that between `DO` and `LOOP` is iterated a number of times as specified by the two arguments that `DO` finds on the stack. A phrase between the words `IF` and `THEN` is executed only if the argument found on the stack by `IF` has the logic value 'true'.[12]

The word

```
=    ( m n -- flag)
```

compares the two arguments *m* and *n* and returns a *logic flag* having the value 'true' if they are equal and 'false' if different. Thus, the following word will beep only when the top two stack items are equal

```
               : BEEP-IF-EQUAL ( m n --)
            = IF
       BEEP THEN ;
```

A richer construct is the `IF` / `ELSE` / `THEN` , used as follows

```
               : BEEP-OR-WHISTLE ( m n --)
            = IF
       BEEP ELSE
    WHISTLE THEN
  HONK HONK HONK ;
```

This word will beep if *m* and *n* are equal, and whistle if they are different; after that, it will honk three times.

There exists also two *conditional-iteration* statements. The first has the form ' `BEGIN` ⟨body⟩ `UNTIL` '; the ⟨body⟩ (which returns a logic flag) is iterated *until* the value of this flag becomes 'true'. The other has the form ' `BEGIN` ⟨setup⟩ `WHILE` ⟨body⟩ `REPEAT` '; in which, after the first ⟨setup⟩ (which is always executed), the sequence ⟨body⟩ ⟨setup⟩ is iterated *as long as* the value of the logic flag returned by ⟨setup⟩ remains 'true'.

[12]How logic values are encoded in a Forth cell doesn't matter at this point, and is discussed in the next section.

5.15 Logic expressions

In defining a CAM rule, sometimes it is convenient to treat the contents of a CAM cell as a logic quantity ('on' or 'off', 'true' or 'false') and sometimes as a number (0 or 1—or even 0, 1, 2, or 3 when one is dealing with two bit-planes at once). To understand precisely what is passed on the stack by one word to another in these cases, it is important to be aware of the coding conventions employed in CAM Forth concerning arithmetic and logic expressions.

This lengthy section is meant as a reference for cases where doubts might arise; you may quickly go over it (or skip it altogether) on first reading.

We have seen that the contents of a Forth cell consists of a 16-bit pattern. The same pattern can have different meanings, depending on agreed-upon conventions. For instance, it can be used to encode an integer from 0 to 65,535 ("unsigned number"), an integer from -32,768 to +32767 ("signed number"), two ASCII characters (8 bits each) or, quite commonly, a single ASCII character (using only the lower 8 bits), etc. The cell does not carry a label telling what kind of encoding was used: it is up to the programmer to arrange things so that any "user" of the pattern will know what conventions to use in interpreting it.

For example, suppose that the top cell of the stack contains the pattern 1000000000101010; the three words '`.`', `U.` , and `EMIT` will all print on the terminal the contents of this cell. However, '`.`' will treat it as a signed number, and print '`-32726`'; `U.` will treat it as un unsigned number, and print '`32810`'; and `EMIT` will treat it as a character, and print '`*`' (since the lower eight bits of the pattern, namely 00101010, make up the ASCII code for '`*`').

In many cases it is convenient to treat the cell pattern just as a collection of separate bits—each one representing an individual binary choice. The words

```
NOT   (   p -- r)
AND   ( p q -- r)
OR    ( p q -- r)
XOR   ( p q -- r)
```

are useful in this context, since they allow one to individually or jointly manipulate these bits. For instance, one can "turn off" the upper eight bits of a pattern by ANDing it with an appropriate *mask* pattern, namely 0000000011111111, in which the upper eight bits are "off" and the lower eight are "on."

As a reminder, the logic operations NOT, AND, OR, and XOR are defined as follows:

NOT	AND	OR	XOR
$0 \mapsto 1$	$00 \mapsto 0$	$00 \mapsto 0$	$00 \mapsto 0$
$1 \mapsto 0$,	$01 \mapsto 0$,	$01 \mapsto 1$,	$01 \mapsto 1$.
	$10 \mapsto 0$	$10 \mapsto 1$	$10 \mapsto 1$
	$11 \mapsto 1$	$11 \mapsto 1$	$11 \mapsto 0$

In particular, the logic operation NOT complements its one-bit argument, and thus the Forth word `NOT` complements *each* of the 16 bits of a cell. The other three logic operators act on corresponding bits of two input cells to produce a 16-bit result.

To drive a conditional statement along one or the other of two possible paths (see previous section) all one needs is a binary "flag"—with values 'true' and 'false'.[13] For this purpose, a one-bit cell would be sufficient; but Forth cells come in a standard size of 16 bits, and one must have an agreement on which 16-bit pattern(s) should mean 'true' and which 'false'.

The Forth-83 standard stipulates that words that *return* a logic flag (such as `=` and similar "compare" words) should never put on the stack anything but the patterns 1111111111111111 for 'true' or 0000000000000000 for 'false'; for convenience, these patterns have been entered in the dictionary, as CONSTANT words, under the names `TRUE` and `FALSE` .[14] On the other hand, words that *expect* a logic flag (such as `IF` and similar "conditional" words) will treat as 'false' the pattern 0000000000000000 and as 'true' *any other pattern*.

As long as one uses only the `TRUE` and `FALSE` patterns for 'true' and 'false', the bitwise logic operations `NOT` , `AND` , etc. can also be used to manipulate such logic flags. However, if one tries to take advantage in an indiscriminate fashion of the wider "catching range" of `IF` [15] some subtle problems may arise. We shall give just one example, as a warning to the reckless programmer.

Consider the word

```
                    : BEEP-IF-NOT-EQUAL ( m n --)
        = NOT IF
        BEEP THEN ;
```

(cf. previous section), which beeps only when m and n are *not* equal. This word would work the same if one replaced '`= NOT`' by just '`-`'; In fact, if m and n are equal their difference $m - n$ will be 0, and will be seen as 'false' by `IF` ; on the

[13]When more than two choices present themselves, it is usually more natural to use a case statement (cf. Section 5.13) rather than many nested `IF` statements.

[14]Note that, when printed as signed numbers, `TRUE` will yield `-1` and `FALSE` will yield `0` ; as an unsigned number, `TRUE` will yield `65535` (`FFFF` in hexadecimal).

[15]As when using arithmetic as a shortcut to logic.

other hand, if they are different $m - n$ will be a pattern containing at least one non-zero bit, and will be seen as 'true' by `IF` .

Well, if '-' works "the same" as '`= NOT`', won't '`- NOT`' work the same as '=' in `BEEP-IF-EQUAL` of the previous section? If $m = n$, their difference $m - n$ is 0 and its complement as given by `NOT` is the pattern of all `1` s (the `TRUE` pattern)—which of course is recognized as 'true' by `IF` , as we intended. If $m \neq n$, the difference pattern *will* contain some 1's but *may* also contain some 0's; thus the complementary pattern returned by `NOT` may contain some `1` s—in which case it will again be recognized by `IF` as 'true', which is *not* what we intended.

Since neighbor words such as `CENTER` , `NORTH` , etc. return `1` or `0` as a value, we shall use these values as respectively 'true' and 'false' whenever expedient. Logic operations involving such 1-bit flags work well, except for `NOT` (since it complements all 16 bits). The two word sequence '`1 XOR`' can be used to get the 1-bit complement. Alternatively, comparisons such as '`=`' '`>`' '`<`' and '`<>`' (not equal) can be used to convert 1-bit flags into standard logic flags.[16]

Finally, it will be useful to remember that words such as `>PLNO` and `>AUXO` , which take a stack item and write it as an entry of a CAM look-up table (Section 9.3), only use the least significant bit of the item: any garbage that may have accumulated in the remaining bits as a result of arithmetic/logic manipulation tricks will be ignored. ("Joint" versions of these words, such as `>PLNA` and `>AUXA` , use the lowest *two* bits.)

5.16 Extended precision

It is occasionally necessary to handle numerical quantities having a greater range or resolution than that provided by 16-bit integers. Forth's traditional approach has been to provide minimal support for *double numbers* (i.e., double-precision integers) and let the user define whatever data structures and operations are needed beyond this.

On the stack, double numbers occupy two consecutive Forth cells, with the most significant cell on top. The words `2DUP` , `2DROP` , `2@` , `2!` , etc., which are useful also in other contexts, can be used for manipulating double-numbers. The words `D+` , `D-` , `D=` , etc. are specifically intended for double-number arithmetic.

No floating-point package is provided with the current version of CAM Forth.[17] For the moment, it is recommended that other programs be used for numerical

[16]The words '`0=`' '`0>`' etc. also exist as abbreviations for '`0 =`' etc.

[17]A floating-point interface that takes advantage of the 8087 or 80287 coprocessor may be included in a later release.

tasks, using data files for exchanging experiment data between CAM Forth and these programs. (Data file formats are discussed in Section E.)

Occasionaly, quadruple precision integers ("quad" numbers) may come in handy for some data-analysis tasks; the file QUAD.4TH provides optional quad-number utilities.

5.17 Search order

We have seen that, in addition to the main section, called FORTH , the Forth dictionary may contain a number of more specialized sections (ASSEMBLER , EDITOR , etc.). *Search order* commands allow one to do the necessary vocabulary switching. At any moment, the CURRENT vocabulary is the one in which new definitions are inserted, while the CONTEXT vocabulary is the one where old definitions are looked up. (Of course, most of the time both CURRENT and CONTEXT pointers point to the FORTH vocabulary.) Just mentioning a vocabulary, such as EDITOR , makes it the CONTEXT vocabulary, superseding the former CONTEXT; the word DEFINITIONS makes CURRENT be whatever the CONTEXT is at that moment. Thus, after saying

```
EDITOR DEFINITIONS FORTH
```

FORTH is CONTEXT but EDITOR is CURRENT: the interpreter will keep looking words up in the FORTH vocabulary, but any new definitions will be added to the EDITOR vocabulary.

The vocabulary organization provides a way to use the same name with different meanings in different contexts, or to make a word unavailable in a certain context. In CAM Forth, for instance, certain "neighbor words" (such as S.EAST , &CENTER , etc.) are segregated in special vocabularies in order to make them available only when the corresponding "neighbor wires" are actually connected to the lookup table.

F83 enriches the concept of "context vocabulary" by providing a number of optional "second (third, fourth, etc.) choices" when a given token is not found in the primary context vocabulary. Although CAM experiments will not need to use this mechanism, it is an integral part of the way that neighborhood and menus are defined in the CAM software, and so we briefly discuss it here.

The additional vocabulary choices are arranged stack fashion—the *search-order* stack. In addition, there is a minimal ROOT vocabulary which normally is always available as a last-resort choice, and which contains the search-order words described below. The command ONLY empties the search stack and makes

ROOT the CONTEXT vocabulary. After saying ONLY , the search order will be

$$\text{CONTEXT:}\ \begin{array}{c} \texttt{ROOT} \\ \hline \texttt{ROOT} \end{array}$$

(where the black line represents the search-order stack, now empty). Naming a vocabulary makes that vocabulary the CONTEXT; if you say FORTH now, the search order becomes

$$\text{CONTEXT:}\ \begin{array}{c} \texttt{FORTH} \\ \hline \texttt{ROOT} \end{array}$$

ALSO pushes a copy of the CONTEXT vocabulary onto the stack; if you said ALSO now the search order would become

$$\text{CONTEXT:}\ \begin{array}{|c|} \multicolumn{1}{c}{\texttt{FORTH}} \\ \hline \texttt{FORTH} \\ \hline \multicolumn{1}{c}{\texttt{ROOT}} \end{array}$$

meaning, "Look it up in FORTH ; if that fails, try FORTH again; if that fails, you've got a last chance with ROOT ; if *that* fails, then give up!"

If you further say META ALSO ASSEMBLER , the search order (which is displayed if you type ORDER) will become

$$\text{CONTEXT:}\ \begin{array}{|l|} \multicolumn{1}{l}{\texttt{ASSEMBLER}} \\ \hline \texttt{META} \\ \texttt{FORTH} \\ \hline \multicolumn{1}{l}{\texttt{ROOT}} \end{array}$$

Finally, the command PREVIOUS pops the top of the search stack into CONTEXT, dropping the previous CONTEXT.

This may sound confusing, but is in practice quite straightforward if you don't try to take advantage of all the tricks that are in principle possible. Usually you put in the search stack *all* of the vocabularies to be searched in a certain situation, without worrying much about their order and without relying on CONTEXT (so that the latter can be switched rather freely, without going every time through the ALSO – PREVIOUS business). You explicitly make a vocabulary the CONTEXT—by just naming it—only (a) when it's a rather secret one that you want to occasionally pull out of your hat to give a special meaning to the word that follows, or (b) though it's already in the back-up stack, you want to make sure that it will be searched *first*.

The CONTEXT vocabulary and the rest of the search-order machinery only have to do with where a word will be looked up. As mentioned above, newly defined word are added to the *current* vocabulary—and there is only one CURRENT vocabulary at any moment. The word DEFINITIONS makes the CURRENT vocabulary be whatever CONTEXT is. Thus, if you say

```
    EDITOR DEFINITIONS FORTH
```

the CONTEXT switches to `EDITOR`, then `EDITOR` is made the current vocabulary (by `DEFINITIONS`), and finally the CONTEXT reverts to `FORTH`. Though the search order still starts with Forth, the next definitions will be added to the `EDITOR` vocabulary.

5.18 The Forth glossary

Probably the most useful piece of documentation to accompany a Forth system is a *glossary*, i.e., a plain English description, arranged in dictionary form, of the words that make-up the system itself.

The glossary provided at the end of this manual (Appendix G) consists of two sections. The first documents those CAM-specific words that are meant to be employed by the CAM users; the second section describes a number of general-purpose Forth words that are specific to the present implementation, as well as a few F83 words that deserve clarification.

Common-usage Forth words are documented in any of a number of Forth manuals (cf. next section). Excerpts from the *Forth-83 Standard*—which specifies standards for a number of fundamental words and constructs—are reproduced in Appendix F. This standard is closely obeyed by the F83 implementation.

5.19 Further readings

Starting FORTH by Leo Brodie (Prentice-Hall, 1981) is an excellent practical introduction to Forth, while *Thinking FORTH*, by the same author (Prentice-Hall, 1984), discusses the methodology that inspires this programming language. *Inside F83*, by C. H. Ting (Offete Enterprises, 1306 South B. St., San Mateo, CA 94402; 1985) is a thorough description of F83's internal structure. The book *Mastering Forth*, by A. Anderson and M. Tracy (Brady Communications Co., 1984), describes Micromotion Forth—a commercial package based on the F83 model (and thus having a close family relationship with the present implementation).

The complete specifications for the *Forth-83 Standard* (cf. Appendix F) are available from the Forth Interest Group (P.O. Box 8231, San Jose, CA 95155).

The periodical *FORTH Dimensions*, published by the Forth Interest Group, P.O. Box 8231, San Jose, CA, is a good source for news, applications, programming techniques, literature listings, and software and hardware developments; most of the above books can be purchased from the F.I.G. The *Journal of Forth Application and Research*, published by the Institute for Applied Forth Research, Inc., is a more academically oriented publication.

Chapter 6

The screen editor

As a preliminary to working on your own set of CAM experiments you should familiarize yourself with the Forth screen editor, since you'll be continually using it.

6.1 Editing commands

As you enter the editor,[1] the contents of the selected screen are displayed in an editing window, with the cursor somewhere on the screen. You can now move the cursor to any point on the screen; you can change, insert, delete, or move text; and then either move to another screen or exit the editor.

Above the editing window is a *status* line telling you the screen number, the file name, and how far you are into the file (screen 0 is 0%, last screen is 100%). The keyboard is no longer a control panel for CAM; rather, it has been turned into a smart typewriter. Typing ordinary printing characters places them on the screen at the current cursor position; the other keys are interpreted as special editing commands; for example, $\boxed{\text{RUBOUT}}$ will backspace and rub out the character it backspaced over.

The editor has two modes, namely *Insert* and *Noninsert* ("overwrite"); you can go from one mode to the other with the toggle command $\boxed{\text{INS}}$. The current mode is indicated at the end of the status line.[2] When you are in *Insert* mode, typing a character will push the rest of the line rightwards, and characters at the end of the line will be lost; similarly, typing $\boxed{\text{RUBOUT}}$ will pull the rest of the line leftwards, injecting blank characters at the end of the line.

While a screen is being edited, the changes are only made on a copy of it kept

[1]From the control panel, with $\boxed{\text{e}}$ or $\boxed{\text{E}}$ (Section 4.5), or from Forth, via EDIT and related words (Sections 7.3).

[2]See Section B.1 for selecting which mode should be the default one.

in a memory buffer. When one moves to another screen or leaves the editor, the modified screen is *saved* to disk—the previous contents of that screen in the disk file are overwritten. Up until the moment a screen is saved one can bring back the unmodified version from disk by hitting $\boxed{\texttt{Ctrl-X}}$ ("Cancel all changes").

It is important to always exit from the editor (using $\boxed{\text{Esc}}$*) before removing the diskette you are working on. This will ensure that all changes have been saved—and saved on the* right *diskette.*[3]

The following lists describe the editing commands. For the benefit of touch typists, most commands that use a special key also have an alternate "control" version that can be typed without moving the fingers far from the home position. We have striven to retain the *Emacs* mnemonics[4] for most control keys; for example, the function of the arrow keys $\boxed{\uparrow}$, $\boxed{\downarrow}$, $\boxed{\leftarrow}$, and $\boxed{\rightarrow}$ is duplicated by the control characters $\boxed{\texttt{Ctrl-P}}$, $\boxed{\texttt{Ctrl-N}}$, $\boxed{\texttt{Ctrl-B}}$, and $\boxed{\texttt{Ctrl-F}}$ (*Previous line*, *Next line*, *Back*, and *Forward*).

The intended purpose of the screen editor is to handle screens, i.e., blocks containing text (see Section 4.3). However, the editor can be useful on occasion for inspecting blocks containing non-text material (for instance, to check that something was actually written to a certain block of a data file by a program under test). In this case, the editor screen may show all sorts of odd symbols; however, to each of the 256 possible byte values there corresponds a *unique* representation—in the form of a single text or graphic character—on the editor's screen.

If such nonstandard use of the editor is desired, the *auto-extend/auto-shrink* feature implied by $\boxed{\text{PgDn}}$, $\boxed{\text{PgUp}}$ may be disabled by typing `AUTO-X OFF` (see glossary).

[3]If there are fresh modifications in the screen buffer and you swap diskettes before leaving the editor, the editor will save the screen buffer on the wrong diskette, possibly contaminating its directory and destroying valuable data.

[4]EMACS is a text editor, originally written by Richard Stallman at MIT, that is widely used in the scientific community.

Keys		Function performed
Esc		*Exit.* Save the modified screen to the file and exit the editor
Ins		*Insert.* Toggle *Insert* mode on or off
↑	Ctrl-P	*Previous line.* Move cursor one line up.
↓	Ctrl-N	*Next line.* Move cursor one line down.
←	Ctrl-B	*Back.* Move cursor one position left
→	Ctrl-F	*Forward.* Move cursor one position right.
End	Ctrl-E	*End.* Move cursor past the last nonblank character of the current line
	Ctrl-A	*Beginning* Move cursor to the beginning of the current line.
↩	Ctrl-M	*New line.* Move cursor to the beginning of the next line
Tab	Ctrl-I	*Tab.* Move cursor to the beginning of the next word.
Backtab		*Backtab.* Move cursor to the end of the previous word.
Home		*Home.* Move the cursor to the upper-left corner of the screen.
PgUp	Ctrl-U	*Up one screen.* Move to previous screen, saving current screen to file. If the current screen is the last of the file and is all blank, it is removed from the file before you move.
PgDn	Ctrl-V	*Down one screen.* Move to next screen, saving current screen to file. An attempt to move past the last screen of a file (indicated by 100% in the status line) automatically adds one blank screen to the file.
Del	Ctrl-D	*Delete.* Delete character under cursor and move the rest of the line one position to the left
Rubout	Ctrl-H	*Rub out.* Delete character before the cursor, and move cursor back one position. In insert mode, the rest of the line is moved left one position.
	Ctrl-Z	*Zap.* Erase from cursor to end of line. The kill buffer (see below) is not affected.

f1	Ctrl-L	*Left align*. The cursor is thought of as dividing the line into two "halves." The right half is shifted leftwards until a non-blank character appears under the cursor; the left half is not affected. Useful for aligning definitions or comments.
f2	Ctrl-R	*Right align*. The opposite of *Left align*. The right "half" of the current line is shifted rightwards until a nonblank character touches the right edge of the screen.
f3	Ctrl-J	*Join*. The opposite of *Open*. Join the left "half" of the current line (i.e., from the left margin to the cursor position) with the right half of the next line. The right half of the current line and the left half of the next line are lost. All following lines are pulled up.
f4	Ctrl-O	*Open*. Break the current line into two "halves" at the cursor position. The right half will make up a new line. All following lines are pushed down.
f5	Ctrl-K	*Kill line*. Remove the current line from the screen; all following lines are pulled up. Lines deleted by consecutive uses of *Kill* are accumulated in a buffer, from which they can be retrieved by *Yank*; the maximum capacity of this buffer is 16 lines (one screenful). The contents of the kill buffer remains available to be yanked until the start of a new series of (one or more) consecutive *Kill*s.
f6	Ctrl-X	*Cancel all changes*. Cancel any changes made on the current screen since last entering it; the screen is re-read from disk.
f7	Ctrl-W	*Wipe*. Fill the kill buffer with the current screen, without destroying the screen's contents.
f8	Ctrl-Y	*Yank*. Insert at the current line the contents of the kill buffer. The contents of the current line and all following ones are pushed down to make room. The kill buffer can be *Yank*ed as many times as desired, possibly to different screens, or even different files (by leaving the editor and then starting editing a new file).
f9	Ctrl-S	*Search for string*. Given a text string, start from the cursor position and search forward through consecutive blocks until the string is found. The string argument is typed in right after Ctrl-S; hitting ↩ instead makes this command reuse the previous string.
f10	Ctrl-C	*Copy file*. Copy current file to a new file. You'll be asked to type in the name of the new file; the default extension is the same as that of the current file. Once the copy is made, *the editor will automatically take you to the new file!*

For convenience, we display below the layout of the editing commands that are attached to the function keys

f1	*Left align*	f2	*Right align*
f3	*Join*	f4	*Open*
f5	*Kill*	f6	*Cancel all changes*
f7	*Wipe*	f8	*Yank*
f9	*Search*	f10	*Copy file*

Some users have found it convenient to photocopy this layout and tape it to their keyboard.

6.2 Date stamp

If you look at the first line of any of the present system's source screens, you'll notice that it contains a "stamp" such as

```
04Jan87NHM
```

indicating *when* the screen was last written or modified, and by *whom*. This stamp is automatically updated whenever one leaves the screen (by moving to another screen or by leaving the editor) after having made any changes to it. Of course, the system needs to know the date and the identity of the user—as explained in Section B.1. The default name used in date stamps will be `cam` until you change it.

The date stamp is added to a screen only if the first line begins with a backslash (in particular, if the screen begins with one of the "comment" words `\` or `\S`).

6.3 Shadow screens

The F83 Forth implementation provides support for *shadow screens*—a rudimentary mechanism for using the second half of a file as screen-by-screen documentation for the first half.

Suppose you have a Forth program file (Section 4.3) consisting of 10 screens, numbered 0–9. Add to this file ten more screens, numbered 10–19, with the understanding that screen 10 will contain comments or other text documentation for screen 0, screen 11 for screen 1, and so forth—that is, each screen in the second half of the file is treated as a "comment shadow" for the corresponding screen in the first half. When you want to *load* the file, you will still load only screens 1 through 9 (as explained in Section 4.3, screen 0 cannot be accessed by the loader) and ignore the second half of the file; but when you want to *examine* the file, you'll find on screen 13 extra documentation for screen 3.

Most of the source files for the CAM system are in the above format.

Two editor commands, `Ctrl-T` and `Ctrl-G`, are meant to provide support for this kind of file organization.

`Ctrl-T`	*Toggle shadow.* Toggle between a source screen and its shadow.
`Ctrl-G`	*Get shadow line.* Copy to the current line of your screen the corresponding line from the screens's "twin" (the shadow screen if this is a program screen, and vice versa). Provides a convenient way of copying source screen information into shadow screens, to be edited into documentation.

Note that these commands have no way of knowing whether your file actually contains shadow screens; they just work on that assumption, treat the file as divided into two equal halves, and match corresponding screens in the two halves.

Chapter 7

Miscellaneous utilities

It is possible to program CAM entirely from the control panel, without ever conversing directly with the Forth interpreter. However, a number of miscellaneous commands and utilities that pertain chiefly to system maintenance are not individually attached to control-panel keys, and are meant to be used interactively at the Forth-interpreter level. Some of these (seeing, listing, moving screens around, etc.) are going to be valuable even to the most casual user.

7.1 Viewing and seeing

If you have the source code on line (see Section B.3), you can look at the definition of any Forth word in the CAM system by using the `VIEW` command. For example, `VIEW DUP` will list the definition for the CODE word `DUP` . Once you have accessed the desired screen, you can use the following words to move to adjacent screens:

L	List the same screen again
N	List the next screen
P	List the previous screen
T	Toggle between the current screen and its shadow.

(7.1)

In addition to `VIEW` , which looks at the disk source, there is a decompiling utility, `SEE` , which looks at the compiled version of a word (in memory) and tries its best at reversing the compilation process. For example,

```
SEE LIST
```

will decompile the word `LIST` (try it!). This works well with COLON words, CONSTANTS, VARIABLES, etc. For DEFERred words (execution vectors), `SEE` knows how to call itself recursively to track down the definition. For CODE words such as `DUP` , `SEE` will tell you that they are in machine code, but won't try to disassemble it (cf. `VIEW` above).

If you are trying to trace the purpose of some Forth word and you are unable to `VIEW` some of the words called by it, it's probably because they are in some vocabulary that isn't part of your current search order (Section 5.17). Until you figure out the `ONLY` / `ALSO` business that this Forth implementation uses for managing vocabularies, all you need to know is that you can find out what vocabularies contain a given word such as `SHOW` (useful for listing Forth code to the printer) by typing

```
    VOC SHOW
```

Forth will respond by telling you that `SHOW` has definitions in both the `FORTH` and `SHADOW` vocabularies. To `VIEW` its definition in the `SHADOW` vocabulary, type

```
  SHADOW VIEW SHOW
```

To see a list of all vocabularies, type

```
    VOCS
```

To see a list of all words in a given vocabulary, say, `SHADOW` , type

```
    SHADOW WORDS
```

To see the current search order, type

```
    ORDER
```

Finally,

```
    'USED MY-WORD
```

will list all the words that use `MY-WORD` in their definition (cf. glossary).

7.2 The current file

We have seen that a file is automatically opened and made the *current* file when it is accessed from the control panel via $\boxed{E}$ or $\boxed{L}$ (note that `VIEW` does *not* affect the current file). From Forth, if you want to open a file called `FOO.BAR` you type

```
    USING FOO.BAR
```

(or perhaps `A:FOO.BAR` if you wish to specify the drive). If no extension is specified, `USING` will add the default extension `EXP` to the filename. As with $\boxed{E}$ and $\boxed{L}$ in the control panel, wildcard characters or no filename results in a directory listing. The Forth words `A:` , `B:` , and `C:` can be used to change the default drive selection.

To move one's attention to a new file, one simply opens that file; the explicit closing of a block file (cf. Section 4.3) is not required. *Avoid using* `OPEN` *(the usual*

F83 word for opening a file)—as it can produce unexpected results if followed by the execution of `NEW-EXPERIMENT` *or* `FORGET`.

If the file name given to `USING` does not already exist, the program offers to create it; the initial size of this file will be two blocks, corresponding to two screens of text.

The word `CAPACITY` returns the number of blocks that make up the current file.

To add 10 more blocks to the end of the file, simply type

```
10 MORE
```

To remove 2 blocks, you type

```
2 LESS
```

To insert 2 blocks right before the screen that was edited last (in order, say, to insert some new text), type

```
2 INS-SCR
```

7.3 Editing

To enter the screen editor from Forth and edit, say, screen 7 of the current file, type

```
7 EDIT
```

To re-enter to the editor from Forth,[1] just type `...` ("dot-dot-dot")—a Forth word meaning "resume editing."

There is an additional, "contents-addressed" way to enter the editor, namely, by using the word `FIX`.

If `XYZ` is a word that has been compiled into the dictionary by loading a screen file, the compiled version of the word will contain a pointer (called *view field*) to the file and screen it "came from" (i.e., to its source); typing

```
FIX XYZ
```

will enter the editor with the cursor positioned right after the word `XYZ`. For this to work, the relevant source file must be on-line on the expected drive (cf. Section B.3), or you'll get an `Open error` message.

For example, the word `FIX` itself was loaded into the Forth dictionary from the source file `PC.4TH`; if this file is accessible, typing

```
FIX FIX
```

[1]Note that Esc will return you to the Forth interpreter (rather than the control panel) if you entered the editor from there.

will set you up for editing the definition of `FIX` . In a similar way, if the word `DUMMY` was just loaded from the current file, typing `FIX DUMMY` will set you up for editing its definition in that file.[2]

7.4 Loading

From the control panel, the commands $\boxed{1}$ and $\boxed{L}$ allow one to *load*—i.e., submit to the Forth interpreter (cf. Section 5.7)—an entire file or, when the command is preceded ny a numerical argument, a single screen of that file. From Forth, to load, say, screen 7 of the current file, type

```
7 LOAD
```

To load screens 7–9, type

```
7 9 THRU
```

In particular, to load the entire file you have to type

```
1 CAPACITY 1- LOAD
```

which, admittedly, is a bit awkward. Note that (a) screen 0 of a file can never be loaded, and is usually reserved for comments; and (b) `CAPACITY` returns the size, in screens, of the current file, so that a 10-screen file will yield a `CAPACITY` of 10 but will have its screens numbered 0 through 9!

7.5 Listing

Files having the extension `DOC` are ordinary PC text files, and should be listed using the appropriate DOS utilities.

To inspect the contents of a Forth block file (Section 4.3), the simplest way is of course to use the screen editor (for blocks that contain arbitrary data rather than text, cf. end of Section 6.1).

From Forth, `7 LIST` will list screen 7 of the current file on the monitor without going through the screen editor. This will work well only for *text* screens.

The first line of a text screen usually contains a comment describing the contents of that screen. The phrase

⟨first screen⟩ ⟨last screen⟩ `INDEX`

produces an "index" consisting of the first lines of all screens of the current file within the specified range.

[2] `FIX` can find source code in one of the `F83` or `CAM` source files; otherwise it looks in the current file. If after loading a word you change the current file, `FIX` will look for that word in the "right place" but in the wrong file.

Anything that is sent to the monitor by the Forth interpreter can be additionally routed to the printer, using `PRINTING`. Thus, if you want a hardcopy of the index of screens 10–15, you type

```
        PRINTING ON   10 15 INDEX   PRINTING OFF
```

and the index will be sent both to the screen and to the printer.

The two "listing" words `SHOW` and `LISTING` send their output only to the printer, preceded by a control code that sets the printer to compressed mode (132 characters/line); check that the Forth system knows what kind of printer you are using (cf. Appendix B.2.4).

`SHOW` is used to *print* a listing of any consecutive portion of the current file, with the following syntax

⟨first screen⟩ ⟨last screen⟩ `SHOW`

(If you want to print the whole file, type `0 CAPACITY 1- SHOW`.)

On the other hand, `LISTING` (with no arguments) assumes that the current file is organized as "source" and "shadow" screens (cf. Section 7.3), and produces a complete listing of the file with source and shadow screens side-by-side. `SHOW` can also be used in this fashion, by saying, for example, `1 10 SHADOW SHOW`.

7.6 Moving screens around

The editor commands *Kill*, *Wipe*, and *Yank* are adequate for moving around a few words or lines, perhaps up to one screenful; the *Copy* command is used for making a copy of the entire current file to a newly created file. For moving several screens, `CONVEY` is useful: to move screens 3–10 of file `SOURCE.EXP` to the current file, starting at screen 12 (and thus overwriting screens 12–19 of the current file), you type

```
    FROM SOURCE.EXP 3 10 TO 12 CONVEY
```

`CONVEY` can also be used, of course, to move screens within the same file. For overlapping source and destination ranges, `CONVEY` does the right thing.

The clause `FROM SOURCE.EXP` makes `SOURCE.EXP` the *from* file ("from" and "current" are two distinct "slots" managed by F83 for dealing with whatever files are relevant at the moment); if `SOURCE.EXP` had already been declared as the *from* file, this clause may be dropped.

Note that, when you open a file, the *from* "slot" (as well as the *current* "slot") is assigned by default to this file; thus

```
    3 10 TO 12 CONVEY
```

(with no explicit `FROM` clause) will copy to screens 12–19 of the current file screens 3–10 of the *same* file, *provided that you have not reassigned the* from *"slot" since you opened the current file.*

`CONVEY` is not allowed to change the length of the destination file. If this file is too short, it must be lengthened first (using `MORE` , for instance) so that the screens to be conveyed may fit in.

7.7 Moving files around

In addition to the above utilities, you can always use the utilities provided by DOS for erasing, copying, and renaming files, etc. As mentioned in Section 4.2, the DOS command interpreter can be accessed from within CAM Forth.

Part III

Writing and running experiments

Chapter 8

The make-up of an experiment

The purpose of this chapter is to make you familiar with the overall structure of an experiment file and with what sort of things may appear in it. The available resources are discussed more systematically in the following chapters.

For a good understanding, you'll have to refer quite often both to the book *Cellular Automata Machines* mentioned in the introduction—briefly, the "*CAM book*"—and to the glossary (Appendix G) at the end of this manual. The former provides the conceptual background for much of the present material; references to it use a notation of the form '[*CAM Book* 7.2]' for "Section 7.2 of the *CAM book*." The latter provides concise technical definitions; references to it are implicit, that is, all Forth words (which are set in a typewriter font) mentioned in this book that are not of purely local interest will be found listed and explained in the glossary.

8.1 A minimal experiment

Assuming the work disk mentioned in Section 1.2 is on line, from CAM's control panel type $\boxed{\text{E}}$`barelife` (that's an upper-case E) to examine, through the screen editor's window, the source code for the BARELIFE experiment mentioned in Section 2.4 (cf. [*CAM Book* 3.1,5.3]). Figure 8.1 is a verbatim reproduction of what you see in the window; the entire source for this experiment consists ofjust that one screen.

Before discussing the contents of the screen, a few words about its layout—which is meant to enhance readability. We have chosen to set flush with the left edge of the screen those commands that are to be directly *executed* by the Forth interpreter, while words that are to be *compiled* into the dictionary are aligned vertically on the right. Thus, by "thumbing" the left edge of the screen you see what Forth is being asked to *do*; by thumbing the right edge, you see what it is being taught—and thereafter expected to *know*. These conventions are followed (somewhat loosely) throughout this manual.

```
Screen 1                       BARELIFE.EXP                      (100%)
\ BARELIFE (LIFE, with no frills)                              04Jan87tmt

NEW-EXPERIMENT

N/MOORE
                                                    : 8SUM ( -- n)
                           N.WEST NORTH N.EAST
                             WEST         EAST
                           S.WEST SOUTH S.EAST
                                  + + + + + + + ;
                                                    : LIFE
         8SUM { 0 0 CENTER 1 0 0 0 0 0 } >PLN0 ;

MAKE-TABLE LIFE
HALF 0 RND>PL
```

Figure 8.1: Source screen for the BARELIFE experiment

The first line of code, NEW-EXPERIMENT , resets CAM's hardware and software to a known, repeatable default state (see Section 9.12 for more details). In particular, (a) all the Forth words added to the dictionary by the previous experiment are wiped out, (b) the bit planes are cleared, (c) the look-up tables are filled with zeros, (d) the color map defaults to the "standard" map STD-MAP (Section 8.2), and (e) all the neighbor "probes" (as explained in [*CAM Book* 7.2], which you may want to re-read now; Section 9.2 of the present manual can be used as a quick reference to neighborhoods) are disconnected.[1] The screen is black, and would remain black if you issued *Step* commands at this point.

The second line, N/MOORE , is a *neighborhood assignment*, telling both software and hardware that you plan to use the Moore neighborhood[*CAM Book* 7.3.1]. The hardware will connect the relevant neighbor probes[*CAM Book* 7.2], while the software will make the corresponding neighbor names available as Forth words to be used in defining the rule. We recall that CAM consists of two "halves," namely CAM-A (planes 0 and 1) and CAM-B (planes 2 and 3). For the moment it will be

[1]More precisely, they are parked on the *user inputs* of the external connector[*CAM Book* 7.3.1].

convenient to imagine that only CAM-A is present; then, the neighbors that are connected and made available as Forth words by the N/MOORE assignment are

```
N.WEST  NORTH  N.EAST
  WEST CENTER   EAST        CENTER'
S.WEST  SOUTH  S.EAST
```

where the unprimed variables refer to bits in plane 0. The primed variable CENTER' refers to a bit in plane 1; our LIFE rule will have no use for it.

The word 8SUM defines a function—namely the sum of the values of the eight nearest neighbors—which will be used in defining the rule itself. (The pretty layout of these eight neighbor words in the Forth text is of course only of mnemonic value; we could have typed them one after the other. Their order is irrelevant, since we add them all up.)

The word LIFE provides an algorithmic description for the "life" rule—in terms of the behavior of a single, generic cell (all cells will obey the same rule). With reference to Figure 8.1, from the values of a number of neighbors (namely, the eight nearest neighbors, through the word 8SUM , and the center cell, through the word CENTER) the algorithm (embodied in the word LIFE) computes a 1-bit result, using a CASE construct (the one with curly braces; see Section 5.13); the word >PLN0 specifies that this result is meant as the "new-state" bit for plane 0.

The line before the last, MAKE-TABLE LIFE , tells the software to translate the above description into tabular form[*CAM Book* 7.6] and download the resulting string of 0 s and 1 s—one "new-state" bit entry for each possible neighbor configuration (see [*CAM Book* 7.2] and Section 9.3 for details)—to CAM's look-up tables. Only at this moment is CAM ready to run "life," as in Section 2.4.

Thus, it is not the Forth word LIFE as such that CAM will execute 65,536 times (256×256) per step. LIFE is merely a *table descriptor*[*CAM Book* 4.2,7.6], and is executed *by the* PC once for each entry of CAM's lookup tables, that is, 2^{12}=4096 times;[2] this is done once and for all while loading an experiment, before CAM even starts running. This "computational investment" may take a few seconds, but will be handsomely repaid every time CAM takes a step: by having precomputed results available in the lookup tables, the amount of "thinking" CAM has to do in real time in order to update a cell is reduced to a minumum.

When you start writing your own CAM experiments, if may happen that any initial configuration will vanish after one step. Before looking for bugs in the rule, check that you do have a MAKE-TABLE command; without it, CAM can't possibly know anything about your rule.

In conclusion, the minimal requirements for a complete, repeatable experiment setup are

[2]Even though we are explicitly using only 9 neighbors, the tables are generated for all possible values of the 12 available address lines (Section 9.1), some of which are, in this case, irrelevant.

- The `NEW-EXPERIMENT` command,
- a neighborhood assignment,
- a table descriptor for the desired rule, and
- a `MAKE-TABLE` command.

Of course, before running the experiment one will have to put in the bit-planes a suitable initial pattern. The last line of Figure 8.1, `HALF 0 RND>PL` , fills bit-plane 0 randomly with half 0s and half 1s.

Practically all of the experiments of Chapters 1–10 of the *CAM Book* are based on this minimal format; the variety of behavior illustrated there arises from a suitable choice of rules and initial conditions—rather that from the use of more advanced hardware or software CAM resources.

8.2 The standard color map

In running `BARELIFE` you'll have noted that live cells are shown green on a black background. In fact, the default, or *standard*, color map—called `STD-MAP` —uses *green* for any cell that contains a `1` in plane 0 only, *blue* for plane 1 only, and *red* for cells having a `1` in both planes[*CAM Book* 3.2]; in addition `1` s in planes 2 or 3 turn on the "intensity" beam, thus lightening a cell's color.

Color-map: `STD-MAP`

Plane		Monitor line				Pixel
0	1	I	R	G	B	color
0	0	*				black
0	1	*			●	blue
1	0	*		●		green
1	1	*	●			red

* 'On' when planes 2 or 3 are 'on'.

(8.1)

This map is convenient when most of the action is in planes 0 or 1, as in many simple (and not so simple) experiments. Later on (Section 9.6) we shall discuss how to define custom color maps.

8.3 Small changes

You may now start from a familiar situation, such as the above `BARELIFE` experiment, and try to make a few small changes or additions.

To avoid spoiling your distribution file BARELIFE.EXP , copy it first to a new file, say, PLAY.EXP ; a few keypresses from the control panel will do (cf. Section 6.1):

- Type $\boxed{\text{E}}$`barelife` to visit the first file.
- Type $\boxed{\text{Ctrl-C}}$`play` to copy this file to a new file called PLAY.EXP .
- As soon as the copy is ready the editor will take you directly to screen 1 of the new file; you may safely play with it.

You'll certainly be tempted to try some variants of "life," for instance permuting or complementing some of the 0 s and 1 s that appear in the CASE list (between braces in Figure 8.1). After making each change, escape from the editor with $\boxed{\text{Esc}}$, load the PLAY file again (using $\boxed{1}$) and run a few steps to see the results.

Suppose, now, that you want to add the ECHO feature[*CAM Book* 3.2] to "life;" that is, at each step, as CAM overwrites the state of plane 0 with a new state as specified by LIFE , the old state should be saved in plane 1—so that this plane always reflects the situation of plane 0 one step before.

In CAM Forth, one way to achieve this is to add (a) a separate table descriptor for the evolution of plane 1 and (b) a separate MAKE-TABLE command to translate this descriptor and write the resulting information to CAM's lookup tables:

```
                                       : ECHO
                           CENTER >PLN1 ;
   MAKE-TABLE ECHO
```

Another way is to replace the LIFE descriptor by one that specifies the evolution of plane 1 as well[*CAM Book* 7.6], as follows

```
                                       : LIFE ( with ECHO)
 8SUM {0 0 CENTER 1 0 0 0 0 0 } >PLN0
                           CENTER >PLN1 ;
```

With this table descriptor, the command MAKE-TABLE LIFE will fill both PLN0 and PLN1 columns of the lookup tables. Similar considerations apply to the TRACE feature[*CAM Book* 3.3], defined as follows

```
                                       : TRACE
                  CENTER CENTER' OR >PLN1 ;
```

(with TRACE, a bit in plane 1 is turned on if the corresponding bit in plane 0 was on—as with ECHO—but then it *stays* on, so that moving objects in plane 0 leave a continuous trace on plane 1).

Note that the words ECHO and TRACE (of plane 0 onto plane 1), as well as FREEZE and BARE (two more handy rule components which respectively "freeze" and "keep clear" plane 1's contents) are predefined in the CAM software. Thus, if you want plane 1 to "hold on" to whatever pattern you choose to store there—while plane 0 runs a rule—it is enough to put the command

```
MAKE-TABLE FREEZE
```

in the source file, since the definition

```
: FREEZE
     CENTER' >PLN1 ;
```

is already present in the CAM Forth dictionary. You use BARE in a similar way to restore column PLN1 of the lookup table to its default, "all zeros" state (cf. 8.1).

8.4 Customizing the control panel

We have seen how to change some aspects of a rule—for instance to go from "bare life" to "life with echo" to "life with trace" and back to "bare life"—by editing the experiment file and loading it again. This, however, re-initializes the whole experiment, clearing whatever pattern you have in the bit-planes.

Since in this case all you want to do is modify the lookup tables, a better way is to press the $\boxed{\text{f}}$ key to go to the Forth interpreter, type, say, MAKE-TABLE ECHO , and go back to the control panel via the $\boxed{\text{Esc}}$ key. Wouldn't it be much easier if by just pressing a control-panel key one could switch on-the-fly from one variant of the rule to the other?

In CAM Forth, the UNSHIFTED, SHIFTED, and CONTROL versions of each key are permanently associated with standard CAM or PC functions, as documented in Parts I and II. On the other hand, the *alternate* (Alt-) version of each key (obtained by pressing that key while holding the ALT key down) may be temporarily attached to custom, user-defined functions through the "alias" mechanism explained below.

Suppose you want to devote the $\boxed{\text{Alt-T}}$ key to turning on the TRACE mode. To attach the command MAKE-TABLE TRACE to this key, you first define a new Forth word expressing that command, and *immediately after this definition* you write ALIAS T , as follows

```
: Trace
     MAKE-TABLE TRACE ;   ALIAS T
```

We have chosen the name Trace (in lower case, to distinguish it from TRACE) for mnemonic reasons; any other name would do. From this moment, every time you type $\boxed{\texttt{Alt-T}}$ from the control panel the word `Trace` is immediately executed—just as if you had typed it from the Forth interpreter.

As long as they are in force, custom control-panel commands have exactly the same status as standard commands.[3] In particular, after attaching the word `Trace` to the $\boxed{\texttt{Alt-T}}$ key as above, the **ALTERNATE** menu, which you get by typing 5$\boxed{\texttt{m}}$ (Section 2.2) and is initially empty, will display the item

```
  T Trace
```

Moreover, the name of this custom control-panel command—namely `Trace` —will be echoed on the terminal whenever you hit the $\boxed{\texttt{Alt-T}}$ key from the control panel, exactly as it happens for the standard commands.

A custom command remains in force until you attach a new custom command to the same key, or execute `NEW-EXPERIMENT` (in which case the key to which it was attached reverts to "Undefined").

Aliases are liberally used in the experiment of Figure 8.2. Two alternative rule-components for plane 0 are defined; one is the usual `LIFE` , the other, called `OTHER-LIFE` , is a slight variation on the same theme. For plane 1 we have the choice between `BARE` , `ECHO` , `TRACE` , and `FREEZE` .

Note that, if we ignore for a moment the last line, this experiment file gives a lot of definitions but does not actually send any tables to CAM. Once the file is loaded and the happy-face prompt reappears, it will be your responsibility to select a rule, by typing, for example, $\boxed{\texttt{Alt-O}}\boxed{\texttt{Alt-E}}$ for "other-life with echo."

To make sure that at the beginning of the experiment a nonvacuous rule will be present in CAM, the last line of Figure 8.2, namely `Life Bare` , makes an initial selection ("plain life") for you, just as if you had hit the custom keys $\boxed{\texttt{Alt-L}}\boxed{\texttt{Alt-B}}$.

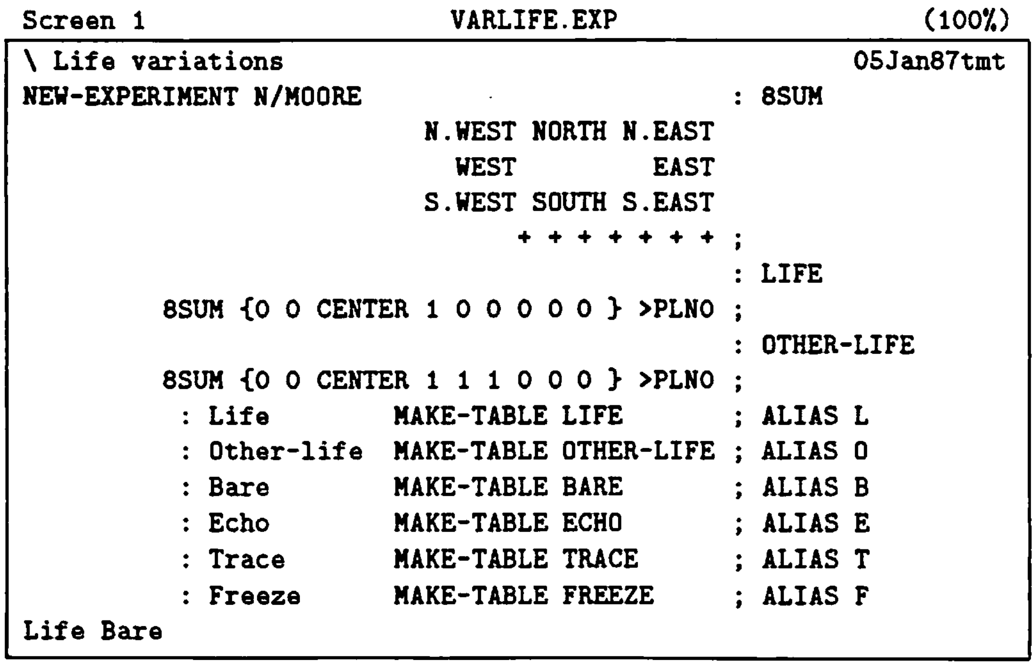
Run the experiment and play with the custom keys just introduced. Instead of a random soup as an initial configuration, try also one containing glider guns (file `GUNS.PAT`), which you can load into plane 0 by typing 0$\boxed{\texttt{G}}$guns .

8.5 Props

If you are giving a critical demo, you don't want to spend your time fumbling with keys or trying to remember what you had called a certain pattern file to be used in the experiment.

Suppose, for example, that you intend to play the glider-breeding experiment discussed in[*CAM Book* 3.4]. You want to run "life" starting from a primeval

[3]In fact, the standard commands of CAM's control panel were attached to their keys by the very same "alias" mechanism, during system generation. Look, for example, at the source file `CAM-KEYS.4TH`.



```
Screen 1                     VARLIFE.EXP                    (100%)
\ Life variations                                        05Jan87tmt
NEW-EXPERIMENT N/MOORE                         : 8SUM
                          N.WEST NORTH N.EAST
                            WEST        EAST
                          S.WEST SOUTH S.EAST
                                + + + + + + + ;
                                               : LIFE
         8SUM {0 0 CENTER 1 0 0 0 0 0 } >PLN0 ;
                                               : OTHER-LIFE
         8SUM {0 0 CENTER 1 1 1 0 0 0 } >PLN0 ;
          : Life        MAKE-TABLE LIFE        ; ALIAS L
          : Other-life  MAKE-TABLE OTHER-LIFE  ; ALIAS O
          : Bare        MAKE-TABLE BARE        ; ALIAS B
          : Echo        MAKE-TABLE ECHO        ; ALIAS E
          : Trace       MAKE-TABLE TRACE       ; ALIAS T
          : Freeze      MAKE-TABLE FREEZE      ; ALIAS F
Life Bare
```

Figure 8.2: Source screen for LIFE variations.

soup, and after a second or two you want to AND the pattern on the screen with a circular mask, thus erasing everything except the central portion of the picture. From the shores of this "island," free-swimming gliders are likely to emerge.

The idea is to have the experiment start not only with the LIFE rule in the lookup tables and a few custom keys defined, but also with the appropriate initial configuration already in plane 0 and the desired mask pattern already stored in planes 0's buffer. When your audience is ready, you hit $\boxed{S}$ to start running. After a few seconds, without stopping the simulation, you hit the custom key $\boxed{\texttt{Alt-M}}$ to mask out the desired portion of the screen; a few gliders will hopefully emerge. Hit the custom key $\boxed{\texttt{Alt-R}}$ to repopulate plane 0 with a new random configuration: in this way, several breeding trials can be shown in close succession by alternating $\boxed{\texttt{Alt-R}}$ and $\boxed{\texttt{Alt-M}}$.

Here is the relevant portion of an experiment file for this (we assume that LIFE has already been defined):

```
             : Mask                 0 AND>PL ;  ALIAS M
             : Random          HALF 0 RND>PL ;  ALIAS R

OPEN-PATTERN DISK.PAT  \ Open desired pattern file
0 FILE>PL              \ Move patt. from file, via plane 0
```

```
0 PL>PB                 \    to buffer 0

MAKE-TABLE LIFE         \ Compile and download lookup tables
MAKE-TABLE ECHO

Random                  \ Start with random soup
```

Refer to the glossary for a detailed explanation of the new Forth words used here. Briefly,

- `Mask` will AND buffer 0 to plane 0, and is attached to the `Alt-M` key.
- `Random` will populate plane 0 with 50% randomness (the first argument of `RND>PL` is the expected number of `1` s; one-half of 256×256 is a round 8000 in hexadecimal, or 37,768 in decimal—and is predefined for convenience as a constant called `HALF`), and is attached to the `Alt-R` key.
- `OPEN-PATTERN` creates a file control block for a pattern file (in this case `DISK.PAT`) and makes it the *current pattern* file, i.e., the one to which operations such as `FILE>PL` implicitly refer.
- `FILE>PL` with argument 0 moves this pattern from its file to plane 0, and similarly `PL>PB` moves it from plane 0 to plane 0's buffer (no words are provided in CAM Forth for moving a pattern directly from disk to a plane buffer).
- Note that, although we have attached `Random` to a control-panel key, nothing prevents us from also using it from the interpreter level, as we do in the last line in order to start the experiment with a random soup already on the screen.

Before being filled with the random soup, plane 0 will for a short time contain the disk pattern (in transit for buffer 0). To avoid distracting your audience (the pattern would briefly flash on the CAM screen), you can momentarily inhibit the CAM display and then restore it by using the commands `TRUE INVISIBLE` and `FALSE INVISIBLE` .

The CAM words that are most likely to be useful in creating custom commands are discussed under the following glossary headings

```
ALIAS ARG ARG?
AREA AND>PL NOT-PL RND>BUF
OPEN-PATTERN FILE>IMAGE FILE>PLN
MAKE-CMAP COLOR-MASK SHOW-FUNCTION SHOW-STATE
MAKE-CYCLE NEXT-STEP STEP
SHIFTS
```

```
EVENT-COUNT
OPEN-DATA
OPEN-TABLE MAKE-TABLE
```

8.6 On-line analysis

Let us consider, for a change, the "brain" rule of [*CAM Book* 6.1]—a neural network whose neurons communicate with their neighbors by immediate contact. A `1` in plane 0 means that the corresponding neuron is firing; a `1` in plane 1 means that the neuron is in a refractory state ("recovering"). The rule is repeated here in Figure 8.3.

If we start this rule with 90% of the neurons firing, this fraction will quickly drop; if we start with 1%, this fraction will climb. In either case, the fraction will converge toward an equilibrium value of approximately 2%. We'd like to monitor in real-time the number of neurons that are firing, by literally counting them at each step. By plotting this number vs time, one can get an idea of how fast the system approaches equilibrium, and of the typical size of the fluctuations about equilibrium.

The firing of a neuron is a simple example of a *cell event*, i.e., of a situation that can be recognized by local and uniform mechanisms similar to those used for updating a cell (see Sections 9.6 and 9.7 for more details).

CAM comes equipped with a real-time event counter, i.e., one that can count cell events at the same rate as they occur. For design simplicity, the input of this counter is permanently connected to the *I* ("intensity") output of the color map; that is, at the end of each active step the number stored in the counter tells us how many intensified pixels appeared on the screen during that step.

The main programming issue, then, is how to translate occurrences of the desired kind of cell event into `1` s on the *I* line. In the above "brain" case the events to be counted are just the `1` s in plane 0; thus, it is sufficient to use a color map where the *I* line directly reflects the contents of plane 0. Clearly, the *IRGB* color map (Section 3.2) will do.

A second issue has to do with the proper reading of the counter. During each active step (in which CAM scans and updates the entire cell array) CAM's hardware counter is also updated—as the sought-for events are encountered.[4] At the end of the step, the contents of this hardware counter is transferred to a *count register* in the PC. In what follows, by the '(event) count' or the 'value of the (event) counter'

[4]During idle steps, in which CAM displays but does not update the cell array, no counting is done.

```
Screen 1                     BRAIN.EXP                    (100%)
\ Monitoring the BRAIN                                04Jan87tmt
NEW-EXPERIMENT N/MOORE                                : STIMULUS
    NORTH SOUTH WEST EAST N.WEST N.EAST S.WEST S.EAST
                                        + + + + + + +
                                { 0 0 1 0 0 0 0 0 0 } ;
                                                      : BRAIN
          STIMULUS CENTERS 0= AND >PLN0  CENTER >PLN1 ;
MAKE-TABLE BRAIN
MAKE-CMAP IRGB-MAP                                    : FUD.
                         0 0 AT  (UD.) TYPE-S  8 0 AT ;
                                                      : COUNTING
                                STEP EVENT-COUNT FUD. ;
            : Counting  DARK 8 0 AT   NEW-EVENTS STEP
                                  MAKE-CYCLE COUNTING ;  ALIAS C
            : Normal    DARK    IDLE EVENT-COUNT FUD.
                                      MAKE-CYCLE STEP ;  ALIAS N
```

Figure 8.3: Source screen for monitoring the firings of BRAIN.

we shall mean the contents of this software register, which is readily accessible and remains constant from the end of one active step to the end of the next one.

We also have to decide what to do with the event counts as they are read. We could accumulate them in a memory buffer of reasonable size, and once in a while flush this buffer to disk (Section 9.11); such a permanent record may be used for off-line analysis. We could plot them vs time as they come—it is even possible to use a spare bit-plane of CAM as a real-time plotting device (see sample file Q2REOS.EXP). In the present case, however, we will just print these counts on the terminal. Thus, we'll have to tell the system that the default action attached to one tick of the cellular-automaton clock (cf. Section 9.9), namely "step," should be replaced by the sequence "step, read count, print count." An appropriate setup for the above experiment is given by the Forth screen of Figure 8.3.

The first part of this screen, up to MAKE-TABLE BRAIN , is by now routine; the command

```
MAKE-CMAP IRGB-MAP
```

makes CAM use the *IRGB* color map (Section 3.2), so that the contents of plane 0 will go verbatim to the *I* beam, to which the counter is attached.

The word COUNTING defines a *run cycle*[*CAM Book* 11.5] suitable for our counting needs (see Section 9.9). In particular, STEP asks CAM to start a new step as soon as possible (the previous step may still be in progress at the moment `STEP` is executed) and waits until the new step is actually ready to start; `EVENT-COUNT` picks up the value of the event counter and places it on the the stack as a *double number*.[5] Finally, `FUD.` prints this number on the terminal in a special way that will be discussed in a moment together with the words `DARK` and `AT` —which also have to do with printing.

The word `Counting` , attached to the custom key $\boxed{\texttt{Alt-C}}$, contains (besides the phrase `NEW-EVENTS STEP` , discussed below) the command `MAKE-CYCLE COUNTING` , which replaces the default run cycle by the custom one just defined by us[*CAM Book* 11.5]—namely `COUNTING` . The custom key $\boxed{\texttt{Alt-N}}$ restores the default run cycle—which is simply `STEP` —after performing some final servicing of the counter.

Load this experiment, put a random configuration in plane 0, and run the rule. After a few seconds hit $\boxed{\texttt{Alt-C}}$ to turn on the counting feature: as the simulation proceeds, a stream of numbers will start pouring out of the terminal

```
1990 1981 2007 1962 1991 1970 2012 ...
```

where each number represents how many cells were firing at the corresponding step. To turn off this stream, hit $\boxed{\texttt{Alt-N}}$.

An astute observer may notice that when one stops the simulation while `COUNTING` is active, the count last printed on the screen does not refer to the last step executed but to the previous one: one count still has to come out. This occurs because the count is double-buffered, so that a new step can be started before analysis of the previous step is performed.[6] The existence of this double-buffering mechanism (a first-in/first-out pipeline with a two-item capacity) also explains the need for the phrase `NEW-EVENTS STEP` in the above definition of `Counting` — which serves to prime the pipeline—and the phrase `IDLE EVENT-COUNT FUD.` — which pulls the last count out of the pipeline.

Details on the stepping process, the make-up of the run cycle, and counting options are given in Sections 9.9 and 9.7.

The reason why we have to make recourse to a special printing procedure is that the DOS terminal output routines are very inefficient, and would take over a sixtieth of a second to print the count on the screen, thus slowing down the simulation. `FUD.` makes use of BIOS routines that are much faster.

[5] A Forth cell, consisting of 16 bits, can represent an unsigned integer in the range 0–65,535 (0000–FFFF in hex). On the other hand, the number of cells in the array—and thus the maximum value of an event count—is 65,536, so that two cells are needed to store this count. If you know for some reason that a count of 65,536 can never arise, you can convert the double number to a single number using `DROP` ; otherwise you must use double-precision arithmetic (Section 5.16).

[6] This arrangement allows one to do a significant amount of analysis even while CAM is running at full speed.

The F83 word `(UD.)` converts a double number on the stack to a character string. `TYPE-S` prints this string starting from the current cursor position. Since the BIOS routine called by `TYPE-S` is not smart enough to start a new line when the current line is full, we have to explicitly position the cursor before each count is printed, using `AT` preceded by column and row number. We chose to position the cursor at the upper-left corner of the screen when printing the count, and reposition it 8 characters to the right for the normal control-panel messages. `DARK` clears the screen of any garbage left over by previous control-panel activity.

In spite of the above precautions, you'll notice that once in a while CAM still idles for one step when `COUNTING` is active: some terminal-output activity is keeping the PC busy right at the moment when a new step should be scheduled, and the step is missed.

The essential tasks that the PC must perform at each step in order to keep CAM running at full speed are handled by interrupts, are essentially invisible to the user, and take up only a small fraction of one-sixtieth of a second (the duration of one CAM step). If you avoid getting bogged down in input/output, there is plenty of time left during each step for the PC to do substantial data analysis using a custom run cycle.

If at any time the on-line data analysis tasks performed by the PC should take longer than one CAM step (e.g., when you have to transfer a memory buffer to disk), or are poorly timed (as in the above case) nothing bad will happen: CAM will wait until the PC is done (in increments of one step time) and the simulation will just slow down.

8.7 Data logging

In the course of an experiment it may be useful to record on disk in a continuous, incremental way some of the data produced by real-time analysis (cf. previous section), or read from disk a stream of data to be used as time-varying parameters for the experiment itself.

The CAM software provides basic interfacing with *data files* for these and similar uses (see Section 9.11 and `OPEN-DATA` in the Glossary). The experiment `Q2REOS` provides a serious example of use of data files. Here we will briefly illustrate the most basic concepts.

In the `BRAIN` experiment of Section 8.6 we showed how to count the number of cells that are firing at each step, and how to display the sequence of such counts on the PC screen. Suppose, now, that we want to make runs of 1024 steps of `BRAIN` and log the resulting counts, step by step as they come, on a disk file for subsequent analysis.

The new version of the experiment is listed in Figure 8.4.

First, we have to create the data file that will hold the counts. Since these counts come as double numbers (cf. Section 5.16), and thus consist of four bytes each, the experiment will generate 4096 bytes of data, equivalent to 4 blocks

```
    4 CREATE-FILE BRAIN.DAT
```

Then, before the experiment starts running, we open this file as a *data* file

```
    OPEN-DATA BRAIN.DAT
```

Among other things, this reserves a block-size buffer in memory to temporarily hold data on their way to the disk (we don't want to slow down the simualtion by making a separate disk access at every step), and initializes the associated pointer to point to the very beginning of the file.

In Figure 8.3, every time that during the course of the experiment we got a count from `EVENT-COUNT` we displayed it on the screen by means of the word `FUD.` . Here, the word `PUT-COUNT` gets the event-count data and appends it to the currently open data file as a record consisting of two Forth cells (i.e., the size of a double number).

The experiment is set up so that the run cycle runs the entire 1024 steps and closes the data file (`CLOSE-DATA`) at the end of it, thus insuring that the memory buffer is safely stored to disk. Load this experiment and hit the $\boxed{S}$ key. The simulation will run the 1024 steps (the disk light will blink a few times) and then stop by itself, with the logged counts recorded in the `BRAIN.DAT` file. `REWIND` brings the file pointer back to the beginning of the file, ready for another run.[7] A new run will, of course, overwrite the data. In between runs, you may want to examine the logged data, do some further analysis on them, or copy them to another file (cf. Section 4.2).

[7]In spite of its the name, `CLOSE-DATA` actually leaves the file open.

```
Screen .                   BRAINLOG.EXP                    (100%)
\ Monitoring the BRAIN                                  04Jan87tmt
NEW-EXPERIMENT N/MOORE                                  : STIMULUS
   NORTH SOUTH WEST EAST N.WEST N.EAST S.WEST S.EAST
                + + + + + + +  { 0 0 1 0 0 0 0 0 0 } ;
                                                        : BRAIN
         STIMULUS CENTERS 0= AND >PLN0  CENTER >PLN1 ;
MAKE-TABLE BRAIN
MAKE-CMAP IRGB-MAP                                      : PUT-COUNT
                       EVENT-COUNT PUT-DATA PUT-DATA ;
                                                        : LOGGING
                          NEW-EVENTS STEP  1023 0 DO
                                 STEP PUT-COUNT LOOP
                                 IDLE PUT-EENT STOP
                                   CLOSE-DATA REWIND ;
4 CREATE-FILE BRAIN.DAT   OPEN-DATA BRAIN.DAT
MAKE-CYCLE LOGGING   HALF 0 RND>PL
```

Figure 8.4: Source screen for logging the firing counts of BRAIN.

Chapter 9

Resources

The present chapter briefly reviews—directly or through pointers to other material—the hardware resources of CAM and the software commands that control them. Most of these resources are adequately discussed elsewhere (the previous and following chapters of this manual, the glossary, the *CAM Book*, and the annotated experiment files). Here we provide connecting material and some technical details.

9.1 The lookup tables

As explained in Chapter 7 of the *CAM Book*, CAM consists of *two* independent halves, called CAM-A and CAM-B, which may be used separately or coupled together. Each half, consisting of two bit-planes, has its own lookup table (Figure 9.1)—whose main function is to to turn the data currently in the bit-planes into new data according to a specified dynamics. Each entry of this table consists of four bits; thus, the table can be visualized as four *columns* of data. The table contains 65,536 (=2^{16}) entries (or *rows*); a particular entry is accessed by a given combination of `0` s and `1` s on the table's 16 address lines. Depending on the currently selected neighborhood, the sixteen bits that make up a lookup table's *input* may come from any of several sources (Table 9.1):

- Some signals originate from within the CAM card, and are extracted from
  - The two bit-planes of this particular half CAM.
  - The two bit-planes of the other half CAM.
  - The horizontal and vertical address of a cell (*spatial phases*).
- Some signals have an external origin:

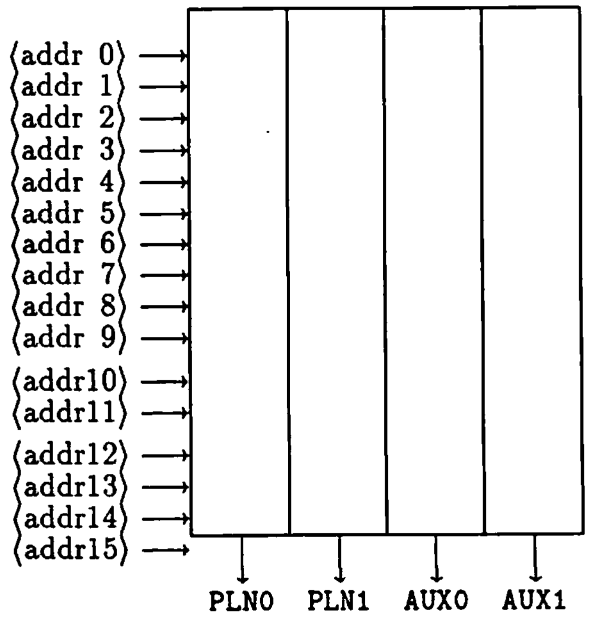


Figure 9.1: The lookup table of CAM-A (for that of CAM-B, replace 0 and 1 in the output labels by 2 and 3). The sixteen address lines can be connected to a variety of signals.

  - *Temporal phase* signals are supplied by the PC; they are generated in real-time by the CAM software as specified by the *run cycle* (Section 9.9).
  - Signals coming from the most disparate sources can be fed to the *user connector* (Chapter 10); these sources may include other CAM cards, special-purpose hardware attachments, a video camera, etc.

- The above signals are used for deciding which table entry to *read*, instant by instant, during a simulation. There are also provisions for deciding in which table entry to *write* something—but this only concerns the low-level software routines in charge of downloading the tables' contents.

In turn, the four bits that appear at the table's *output* may go to different destinations:

- The first two columns of a table (also called *regular* tables) are generally used for determining the new state of the two bit-planes of the corresponding CAM half.

- The remaining two columns (*auxiliary* tables) may

  - Be left unused.

- Replace the regular tables in determining the new state of the two bit-planes (Section 9.8).
- Be used for computing an output function (Section 9.7), to be used in conjunction with the color map.

One of the functions of the CAM Forth software is to provide simple and intuitive ways to exercise the above options.

9.2 Neighborhoods

The CAM neighborhoods provide a way of physically connecting different subsets of signals to the tables' inputs, and attaching appropriate mnemonic names to the signals so selected, as explained in Chapter 7 of the *CAM Book*. The generally useful neighborhood assigments are listed in the following tables.[1]

MAJOR NEIGHBORHOODS

addr	N/MOORE	N/VONN	N/MARG	N/MARG-PH	N/MARG-HV	N/USER
0 ==	CENTER	CENTER	CENTER	CENTER	CENTER	CENTER
1 ==	CENTER'	CENTER'	CENTER'	CENTER'	CENTER'	CENTER'
2 ==	S.EAST	EAST'	CW	CW	CW	⟨user 2⟩
3 ==	S.WEST	WEST'	CCW	CCW	CCW	⟨user 3⟩
4 ==	N.EAST	SOUTH'	OPP	OPP	OPP	⟨user 4⟩
5 ==	N.WEST	NORTH'	CW'	CW'	CW'	⟨user 5⟩
6 ==	EAST	EAST	CCW'	CCW'	CCW'	⟨user 6⟩
7 ==	WEST	WEST	OPP'	OPP'	OPP'	⟨user 7⟩
8 ==	SOUTH	SOUTH	⟨user 8⟩	PHASE	HORZ	⟨user 8⟩
9 ==	NORTH	NORTH	⟨user 9⟩	PHASE'	VERT	⟨user 9⟩

(9.1)

MINOR NEIGHBORHOODS

addr	&/CENTERS	&/PHASES	&/HV	&/USER
10 ==	&CENTER	&PHASE	&HORZ	⟨user 10⟩
11 ==	&CENTER'	&PHASE'	&VERT	⟨user 11⟩

[1]There happen to be a more neighborhoods available. One is used for the expanded view provided by the control-panel's $\boxed{x}$ key; two others are components of **N/MOORE** and **N/VONN**, with lines 6, 7, 8, and 9 left as user neighbors. All the twelve-bit neighborhoods are listed in the glossary. A set of extended neighborhoods that make use of the upper four address bits will be defined in future versions of the software. With sixteen address bits a four-plane margolus neighborhood is possible by appropriately setting the extended registers on CAM-PC. See the *CAM-PC Hardware Manual* for more details.

Neighbor assignments have both a *software* and a *hardware* effect. The software effect is to make the appropriate neighbor words available—and mask out the others, for your protection. The hardware effect, which takes place immediately, is to wire the appropriate signal sources to the specified table inputs.

If you are perverse you can defeat the software protection—which is based on the use of vocabularies. For example, the word `S.EAST` belongs to a small, specialized vocabulary which is included in the search order only when the `N/MOORE` assignment is in effect. You can make a duplicate of this word in the main *Forth* vocabulary, as follows

```
N/MOORE                 : S.EAST
                S.EAST ;
```

Now, if you say

```
N/VONN
```

and try to define a rule in terms of `S.EAST` (which is meaningful only in the `N/MOORE` neighborhood), the software will let you do it without warning.

In certain circumstances it is meaningful to use a rule that uses two different neighborhoods (one at a time; say, on alternating steps) depending on the value of a phase bit that is visible in both neighborhoods (Section 9.8). In this case, the neighbor declarations that have been used (for their software effects) in different places in the definition of the rule must be issued again in the appropriate sequence (for their hardware effects) by the run cycle that drives the simulation.

In essence, the software effect of `N/MOORE` is just to introduce, say, `S.EAST` as a mnemonic for "table input no. 2;" this mnemonic can be applied by you indifferently to the table of CAM-A or that of CAM-B—which have identical structure. On the other hand, the hardware effect of `N/MOORE` is restricted to the currently selected CAM half (specified by the command `CAM-A` or `CAM-B`); however, the command `CAM-AB` —which is the default one—lets subsequent neighborhood assignments go to *both* CAM halves at once (until overridden by `CAM-A` or `CAM-B`). Moreover, if you have more that one CAM card, the neighborhood declaration's hardware effect will apply only to the currently selected card (cf. Section 11.2).

Many pairs of neighbor words (such as `CENTER` and `CENTER'` , or `HORZ` and `VERT`) are accompanied by a *joint* version, which is always a two-bit variable. The least significant bit is the value of the first element of the pair, the next bit that of the second element (in arithmetic notation, "the first bit plus twice the second"). Thus, CENTERS=CENTER+2×CENTER' , and HV=HORZ+2×VERT .

By default (after `NEW-EXPERIMENT`) the major neighborhood assignment is `N/USER` and the minor one `&/USER` : the corresponding table address lines are connected to input pins on the user connector via a TTL buffer. If you are curious, a *floating* TTL input (i.e., one that is left unconnected) looks like a `1` —not like a 0 ; thus, if you forget to make a neighborhood assignment your rule will run as if all the neighbors are in the 1 state.[2]

Note that, after you have explicitly made a minor neighborhood assignment, you cannot make a new one without explicitly making a new major assignment first (even if this is the same as the previous one). Thus, if during the course of an experiment you want to change the minor neighborhood from &/CENTERS to &/PHASES , you cannot say

```
    N/MOORE &/CENTERS
    ...
    &/PHASES
```

You have to say

```
    N/MOORE &/CENTERS
    ...
    N/MOORE &/PHASES
```

Moreover, a new major assignment also resets the previous *minor assignment*. Thus, after

```
    N/MOORE &/CENTERS
    ...
    N/VONN
```

the current minor assignment is no longer &/CENTERS but the default &/USER .

9.3 Table generation

CAM Forth encourages you to give a *structural* description for your cellular-automaton rules, using an algorithmic language and appropriate mnemonics; this description is then translated by the software into a lookup table. The best way to fully understand the magic that turns a CAM Forth rule descriptor in the PC into a cellular-automaton rule in the CAM card is to have a conceptual picture of that translation process.

In CAM Forth, neighbor names such as NORTH , SOUTH , etc. are used to describe the new state of a cell as a logical function of neighbors, and thus specify a cellular-automaton rule. "Column dispatcher" words such as >PLN0 and >AUX0 specify which column of the lookup table each part of the result goes to. MAKE-TABLE is then used to evaluate a rule for all possible neighborhood configurations, in order to generate new entries for all of the indicated subtables. For example,

[2]Except for CENTER and CENTER' (lines 0 and 1), which are internally connected to the bit-planes in all neighborhoods—including N/USER.

```
                              : TRACE
        CENTER CENTER' OR >PLN1 ;
    MAKE-TABLE TRACE
```

would produce a table-column for plane 1 instructing each bit of this plane to become a 1 if the bit itself or the corresponding one on plane 0 is a 1 . As a result, during the course of the simulation plane 1 will contain a "check mark" for every cell whose bit in plane 0 has *ever* been on, even if it was subsequently turned off (assuming we start with a clear plane 1).

MAKE-TABLE goes through all possible neighborhood configurations involving 12 neighbors. [Future versions of the software will include options for the remaining 4 bits. At the present time, these are only accessible through low level routines.] For each configuration, it assigns appropriate values to the neighbor words, which are really table-compilation variables. It also assigns a default result for the table entry being constructed: the old entry. Words such as >PLN0 each change one bit of the table entry; then the entry is stored in the table and the next configuration is considered. Words such as PLN0 and AUX0 can be used to read the current value of the corresponding bit of the entry—either the old value, or the value last assigned with >PLN0 , etc. Thus, one can even generate tables as a function of their previous contents.

9.3.1 Details

The following notes are chiefly meant for the advanced user.

The sixteen bits that collectively make up the lower input to a lookup table can be read (in binary) as a number ranging from 0 through 65,535 (in decimal). During table compilation CAM Forth's internals use a dedicated variable, called X , to point at a given entry of the table. Another variable, called Y , serves as a scratchpad for constructing the entry itself; the eight least significant bits of Y correspond to the eight columns of the lookup tables (the four columns of CAM-A's table and the four columns of CAM-B's).

The command MAKE-TABLE ⟨table descriptor⟩ sets up a loop where X starts at 0 and runs all the way up to 4095. For each value of X , the scratchpad Y is initialized with the present contents of the X-th table entry; then the table-descriptor word is executed—and this usually affects some of the bits of Y ; finally, the new value of Y is stored back in the X -th entry. During the execution of the table descriptor, the words responsible for changing some bits of Y are the "column dispatcher" words >PLN0 , >PLN1 , >AUX0 , etc. (A word such as PLN0 puts on the stack the current value of that bit of Y that would be overwritten by >PLN0 . Thus, one can explicitly use the current contents of the lookup table in generating its new contents.) It is in this way that MAKE-TABLE TRACE will fill *all* of column PLN1 , without changing the contents of the other columns.

Ultimately, a table-descriptor specifies how `Y` should be changed as a function of `X`. The bits of `X` represent a given neighborhood configuration, and the bits of `Y` represent the values that we want the tables to return when that configuration is encountered during the scanning of the bit-planes. However, instead of explicitly handling `X` and `Y`, we use mnemonic words that refer to particular bits of `X` and `Y`. The words that refer to bits of `Y` are the column dispatchers, as we have just seen; the words that refer to bits of `X` are the neighbor and pseudo-neighbor words, such as `NORTH` and `PHASE`.

In what follows, we'll suppose that `MAKE-TABLE` is in the process of generating entry no. 5 of the tables, for which `X` has the value (in binary) `000000000101`. What state of a cell's neighborhood would yield this combination of bits as an input to the lookup table—and thus would produce, as a result of the lookup, the contents of this entry?

If the nine outputs (a 3×3 window) from each plane were permanently wired as inputs of the lookup table, this question could be answered once and for all. However, since the overall number of plane outputs is 36 (nine neighbors times four planes)[3] and the number of table inputs is only 12, a selection must be made. As explained in the previous section, the various CAM neighborhoods provide different selections (by means of hardware multiplexers), and for each selection the wiring (and thus the meaning) of the 12 input lines changes correspondingly: some way is needed to automatically keep track of all of this.

Suppose the neighborhood assignment is `N/MOORE`. A *neighbor word* such as `S.EAST` is made available by the software only in a neighborhood where the corresponding "south-east" plane output is wired to the table. This word knows to which of the sixteen available address positions this output is wired: as you can see from Table 9.1, in `N/MOORE` the `S.EAST` neighbor indeed appears as address line 2. With this information, `S.EAST` looks at the current value of `X`, namely 000000000<u>1</u>01, and returns the bit in position 2 of this string (count from the right, starting from 0)—which we have underlined. In this case, the phrase

```
    S.EAST >PLN0
```

will extract bit 2 from `X` and put it in the "plane-0" bit of `Y`. If you said `CCW` instead of `S.EAST`, you'd get an error message, since the neighbor word `CCW` is inaccessible in the `N/MOORE` context. If you said

```
    LIFE
```

(where `LIFE` is as defined in Figure 8.1), bits 0 and 2–9 of `X` would be added (by `8SUM`), giving 2 as a result; this 2 as an argument to the case statement (also in Figure 8.1) would cause `CENTER` to extract bit 0 of `X`; the result, namely `1`,

[3]And there are many more signals on the board that one might on occasion want to feed to the table.

would be placed by `>PLN0` in the corresponding bit-position in `Y`, so that when `Y` is copied to the table, this `1` would appear in column `PLN0` of entry 5.

Note that columns `PLN0` and `PLN1` belong to CAM-A, while `PLN2` and `PLN3` belong to CAM-B. When we compute a bit as a function of the entry address `X`, we obtain a different rule depending on whether this bit is sent to a table-column belonging to CAM-A or CAM-B; all the rest of the above construction is independent of whether we are programming CAM-A or CAM-B. In particular, the neighbor word `S.EAST` knows that it refers to input 2 of a table, but does *not* know whether we have in mind the table for CAM-A or that for CAM-B. Thus, to make `LIFE` run on CAM-B it is sufficient to replace the `>PLN0` in its definition by `>PLN2` : the neighbor names do not have to be changed.

Though it is possible to use a single table descriptor for filling all of the table-columns at once (all eight bits of `Y` for each value of `X`), if CAM-A and CAM-B use different neighborhoods it is usually easier to write separate table descriptors for the two CAM halves.

9.4 Phases

Besides the contents of the bit-planes and signals provided through the user connector, there are other quantities, called *phases*, that may be used as lookup-table inputs (cf. Table 9.1 and [*CAM Book* 11.1,11.2]). The *temporal* phases are transmitted to CAM by the software, and remain constant for the duration of a step (thus the same value is seen by all cells); the *spatial* phases are generated by CAM's address circuitry, but their "phasing" with respect to the origin of the array may be controlled by the software. To each phase quantity there corresponds a *pseudo-neighbor* used during table generation (see below).

What the user software reads or writes is not directly the hardware bits corresponding to the temporal phases or the "phasing" of the spatial phases, but shadows of these bits in the PC memory, which we shall call *pseudo-variables*; lower levels of the software take care of synchronizing the transfer of these data to CAM at each step.

The following words are used for controlling phase pseudo-variables:

```
<&PHASE>
<&PHASE'>
<PHASE>
<PHASE'>
<TAB-A>
<TAB-B>

<ORG-H>
```

```
    <ORG-V>
```

To set to `1`, say, the bit corresponding to `&PHASE` you say

```
    1 IS <&PHASE>
```

(Since only the least significant bit is used, the strange-looking phrase

```
    37 IS <&PHASE>
```

would give the same result.) To read whatever value you last gave to this pseudo-variable (in case you didn't bother to keep a record) you just say `<&PHASE>`, and its value will be pushed on the stack (as the least significant bit of a cell containing zeros in all other positions). Thus, if you want to complement the value of this phase you can say

```
    <&PHASE> NOT IS <&PHASE>
```

The above pseudo-variables are naturally grouped in pairs, and the two elements of a pair can be jointly controlled by the following words

```
    <&PHASES>
    <PHASES>
    <TAB-AB>

    <ORG-HV>
```

which return a two-bit value on the stack and are assigned a two-bit value by the `IS` construct. In each case, the least significant bit corresponds to the first element of the pair. Thus

```
    2 IS <&PHASES>
```

is equivalent to

```
    0 IS <&PHASE>
    1 IS <&PHASE'>
```

Note that the phase bit corresponding to `<TAB-A>` (and similarly for `<TAB-B>`) never appears as one of the 12 inputs of CAM-A's lookup table; rather, it decides whether the regular or the auxiliary tables are to be used for determining a cell's new state (Section 9.8).

One more pseudo-variable with a function somewhat analogous to that of a phase is controlled by the words `SHOW-STATE` and `SHOW-FUNCTION` (see Section 9.7).

During table generation, the pseudo-neighbor words referring to those phase signals that may appear as lookup-table inputs (cf. Table 9.1) are

```
    &PHASE
    &PHASE'
```

```
PHASE
PHASE'

HORZ
VERT
```

which return one-bit values, and their joint versions

```
&PHASES
PHASES

HV
```

which return two-bit values.

9.4.1 Spatial phases

As explained in [*CAM Book* 11.1], the CAM hardware assigns values of the spatial phases to each cell of the cell array; namely, `HORZ` takes on alternating values :.01010:. on consecutive cells of a row, and `VERT` alternating values :.01010:. on consecutive cells on a column. In most situations these values are not explicitly fed to the lookup table, but affect what is fed to it by controlling which three of the eight cells that surround a given center cell will be seen as its "Margolus neighbors" `OPP` , `CW` , and `CCW` (the fourth Margolus neighbor, `CENTER` , is the cell itself, and is always available).

By means of `<ORG-H>` and `<ORG-V>` the user can only control the "phasing" of these phases, i.e., the values that they will take on the top-left cell of the array—and consequently which of the four possible ways of partitioning the array into 2×2 Margolus blocks is active during a given step.

Explicit use of the pseudoneighbors `HORZ` and `VERT` (or their aliases `&HORZ` , `&VERT`) is necessary only if one wants to use the "absolute" Margolus neighbors `UL` , `UR` , `LL` , and `LR` (cf. [*CAM Book* 12.5] and the Glossary), or if one wants to make use of horizontal or vertical parities (even or odd rows, even or odd columns) outside of a Margolus-neighborhhod context.

We mentioned in Section 3.9 that the grid made visible on the CAM screen in *Expanded mode* partitions the screen into 2×2 blocks; these blocks coincide with the current Margolus blocks (as determined by `<ORG-H>` and `<ORG-V>`). Since the values of *all* PC-*controlled variables relevant to the next step* (in particular, the phases) are loaded into CAM immediately after the execution of the previous step (cf. Section 9.9.2), whenever you stop the simulation the Margolus blocks that you see on the *Expanded mode* screen are, in normal circumstances, those that will be used during the *next step*. In other words, during rule debugging you can "preview" (and possibly modify) the Margolus partitioning before using it.

9.5 Using precompiled tables

The translation of a table descriptor into one or more columns of a lookup table may take from a fraction of a second to several seconds, depending on the speed of your PC, the number of table columns involved, and the complexity of the description algorithm. Once an experiment has been thoroughly debugged, it may occasionally be convenient to save the tabular version of the rule on a disk file (default extension: `TAB`), and load the tables directly from this "memory dump" whenever the experiment has to be performed. This eliminates annoying waiting times in canned demos. The following commands are used for this purpose at the control-panel level

TAB	*Save tables.* Save CAM's lookup tables to a specified file (default extension: `TAB`). The tables occupy 4096 bytes.
BACKTAB	*Load tables.* Load CAM's lookup tables from a specified file (default extension: `TAB`).

Within an experiment file, the corresponding Forth commands are `TAB>FILE` and `FILE>TAB` (cf. 9.11 and glossary).

Note that the entire contents of the lookup tables (4K nybbles for each CAM half, for a total of 4K bytes) is transfered to or from disk—even if all you are interested in is a column or two. Note also that *only* the table is transfered; the hardware neighborhood and any run cycle must be set up separately.

9.6 The color map

The *color map* is a small lookup table with four inputs (called `ALPHA` , `ALPHA'` , `BETA` , and `BETA'`)—which for the moment we shall assume come from bit-planes 0, 1, 2, and 3 (but see below)—and four outputs (called `INTEN` , `RED` , `GREEN` , and `BLUE`) which go directly to the the four color-monitor beams (cf. Section 3.2). The color map is written by the command `MAKE-CMAP` ⟨table descriptor⟩, where ⟨table descriptor⟩ is a Forth word that describes the table's contents. For example, the *IRGB* map of Section 3.2 is described in an obvious way as follows

```
                    : IRGB-MAP
ALPHA   >INTEN
ALPHA'  >RED
BETA    >GREEN
BETA'   >BLUE  ;
```

The conventions for a color-map descriptor are similar to those used for a cellular-automaton rule descriptor. For each column (or output line) of the color-map table we define a function of the available inputs (ALPHA etc., which play a role analogous to that of the neighbor words of Section 9.2) and specify (by means of a "dispatcher" word such as >INTEN) what column of the table this data should go to. The actual shipping of data to the color-map hardware table is done by the command

```
MAKE-CMAP IRGB-MAP
```

The default—or *standard*—color map (Section 8.2) performs some logic on the planes' contents before displaying it:

```
                                             : STD-MAP
ALPHA      ALPHA'      AND >RED
ALPHA      ALPHA' NOT  AND >GREEN
ALPHA NOT  ALPHA'      AND >BLUE
BETA       BETA'       OR  >INTEN ;
```

That is, a pixel will be *red* if both the bits in planes 0 and 1 are on; *green* if only plane 0 (plane 0 *and not* plane 1) is on; and *blue* if only plane 1 is on. Moreover, the *intensity* "beam" is turned on if either or both of the bits of planes 2 and 3 are on.

In many experiments the individual bit-planes represent distinct features of the model under study (e.g., dynamical variables, short-term memory, obstacles, thermal reservoir, etc.). By attaching a variety of color maps to the control-panel keys (Section 8.4) one can selectively display those features that are of interest at any given moment (or mask out irrelevant features, as in the example of Section 9.8). Several examples of color-map programming at this elementary level appear in the sample experiments.

The usefulness of the CAM color table, especially for real-time analysis of a simulation, is greatly enhanced by the following two features

- A real-time event counter is attached to the *Intensity* output of the color map. Thus, by suitably programming the color map one can select what events are to be counted.

- Instead of just the contents of a cell (i.e., one bit from each plane), one can feed to the color map the outputs of the *auxiliary* lookup tables—which in turn can be programmed, just like the regular lookup tables, to compute an arbitrary function of a cell's *neighborhood*. Thus, even as the regular lookup tables compute the array's next state as a function of its current state, one can display (and, if desired, count) by means of the auxiliary tables a *different* function of the current state.

9.7 Displaying an output function

The lookup tables used for constructing a cellular automaton's new state have "twins"—called *auxiliary tables*—which see the same neighborhood data as the regular tables but can be filled with different data (Section 9.1). The auxiliary tables consist of four "columns," namely `AUX0` and `AUX1` for CAM-A and `AUX2` and `AUX3` for CAM-B, which are filled by dispatcher words (`>AUX0` etc.) just as the regular tables.

One use of the auxiliary tables is for display processing, as explained here; another use for them is as an extension of the regular tables, as explained in Section 9.8. These two uses are, by and large, mutually exclusive: if you fill the auxiliary tables with data appropriate for display processing, you can't expect the same data to be useful as an extension of the regular tables for updating purposes (though it is conceivable that one may manage to use some of the auxiliary tables' columns for one purpose and some for the other).

The command `SHOW-FUNCTION` [*CAM Book* 7.7] connects the color map's `ALPHA` input to `AUX0` rather than to plane 0 (and similarly for the other inputs); `SHOW-STATE` restores the default connection to the bit-planes:

SHOW-STATE	SHOW-FUNCTION
plane 0→ALPHA	AUX0→ALPHA
plane 1→ALPHA'	AUX1→ALPHA'
plane 2→BETA	AUX2→BETA
plane 3→BETA'	AUX3→BETA'

Thus, an extra processing stage (represented by the auxiliary tables) may be inserted between the bit-planes and the color map. While the color map can only see one cell at a time when tied directly to the bit-planes, the auxiliary tables can see a whole neighborhood (within the constraints of CAM's neighborhoods).[4] The cascading of auxiliary tables and color map provides substantial power and flexibility for display and counting purposes. Good examples are provided by sections [*CAM Book* 15.5] and [*CAM Book* 17.6] of the *CAM Book* (the source code for these examples is included in CAM's software).

9.8 Doubling up the lookup tables

A *phase* pseudo-neighbor(cf. 9.4 and [*CAM Book* 11.1,11.2]) allows one to cut up the lookup table into two halves, and use one or the other half at any given step depending on the value that pseudo-neighbor has for the current step (temporal

[4]Note that in each half CAM the auxiliary and the regular tables use the same neighborhood.

phases) or the current cell position (spatial phases). There are times when we wish to alternate in a similar way between two rules but we need a *full-size* table for at least one of the rules.

The command `AUX-TABS` instructs the planes to take their new values from the outputs `AUX0` ... `AUX3` of the auxiliary tables rather than from the outputs `PLN0` ... `PLN3` of the regular tables;[5] the command `REG-TABS` restores the default situation (new state from the regular tables).

As an example, let us give an explicit rule for Pomeau's original version of the HPP gas[*CAM Book* 16.5]. The four bit-planes represent particles traveling in four directions (east, west, south, and north, respectively for planes 0, 1, 2, and 3). Here a "macro-step" (cf. Section 9.9.2) consists of two "micro-steps:"

Step 1: Each particle takes one step in the appropriate direction; i.e., all of plane 0 shifts one position eastward, plane 1 westward, etc.

Step 2: The contents of a cell is examined and possibly modified, without looking at the neighboring cells: if the cell contains exactly one particle in plane 0 and one in plane 1, these two particles are transfered to planes 2 and 3 (intuitively, they collide and come out at right angles from the original directions); the opposite transfer occurs if the cell contains exactly one particle in plane 2 and one in plane 3.

The experiment file of Figure 9.2 shows a compact way of describing this rule to CAM. The run cycle `HPP-CYCLE` takes care of switching between regular and auxiliary tables on alternating steps.

The `SHIFT` micro-step needs the `N/VONN` neighborhood on both halves of the machine; the `COLLIDE` micro-step needs `&/CENTERS` on both halves; since these two neighborhoods are compatible with each other, we simply program the machine with both of them. (Had they been incompatible, we would have used separate assignments for the two micro-step words, and switched neighborhoods as well as tables in the run cycle.)

In writing this rule, we took advantage of the ambiguity of the neighbor words with respect to which half of CAM they refer to (cf. Section 9.2). In `SHIFT` , note that `SOUTH'` and `NORTH` refer to CAM-B (planes 2 and 3 respectively), since they are dispatched to `AUX2` and `AUX3` . A similar game is played in `COLLIDE` . Take the collision case (when the `IF` condition is true), for example; the neighbor words `&CENTER` and `&CENTER'` , which are duplicated by `2DUP` , generically refer to "the other half" of CAM, and specifically mean the contents of planes 0 and 1 when they are dispatched to planes 2 and 3, and vice versa. In `COLLIDE` , we could have written more concisely

[5]See `<TAB-A>` in the glossary for the separate control of this feature in CAM-A and CAM-B.

```
Screen 1                          HPP.EXP                          (100%)
\ Original HPP rule, with a 2-step cycle                         04Jan87tmt
NEW-EXPERIMENT  N/VONN &/CENTERS
                                                      : SHIFT
                   WEST EAST' NORTH SOUTH'
                   >AUX3 >AUX2 >AUX1 >AUX0 ;
                                                      : COLLIDE
                  CENTER  CENTER' =
                 &CENTER &CENTER' = AND IF
                     &CENTER &CENTER' ELSE
                      CENTER  CENTER' THEN
              2DUP >PLN3 >PLN2 >PLN1 >PLN0 ;
MAKE-TABLE SHIFT
MAKE-TABLE COLLIDE
                                                      : HPP-CYCLE
              AUX-TABS STEP  REG-TABS STEP ;
MAKE-CYCLE HPP-CYCLE
```

Figure 9.2: Source screen for the HPP-gas rule, slow version (two-step cycle).

```
         ... IF
  &CENTERS ELSE
   CENTERS THEN
DUP >PLNB >PLNA
```

When SHIFT is compiled, we will be warned that a word with the same name already exists. Since we don't use this old meaning in this experiment, we may ignore the warning.

If we used the *IRGB* map for this rule, particles going in different directions would have different colors. A more appropriate color map would give to each cell an intensity proportional to the number of particles (0, 1, 2, 3, or 4) present in it. For this we need *five* levels, but with the PC's color monitor only four intensity levels are possible (black, grey, white, bright white) The following color map makes do by using blue as an intermediate level between grey and white.

```
                                                  : DENSITY
          ALPHA ALPHA' BETA BETA' + + + + ;   ( -- 0|1|2|3|4)
: L0  0 >INTEN  0 >RED  0 >GREEN  0 >BLUE ; \ black
: L1  1 >INTEN  0 >RED  0 >GREEN  0 >BLUE ; \ gray
: L2  0 >INTEN  0 >RED  0 >GREEN  1 >BLUE ; \ blue
: L3  0 >INTEN  1 >RED  1 >GREEN  1 >BLUE ; \ white
: L4  1 >INTEN  1 >RED  1 >GREEN  1 >BLUE ; \ light white
```

```
                                            : DENSITY-MAP
                     DENSITY { L0 L1 L2 L3 L4 } ;
  MAKE-CMAP DENSITY-MAP
```

Note that in this implementation of the HPP rule each macro-step consists of two steps. Using the external connector in order to synthesize an ad-hoc neighborhood for this rule, it is possible to run twice as fast: both the shifting and the colliding are done in a single step, as explained in Section 10.

9.9 The run cycle

Each step of CAM is an activity that takes as an input the current contents of the bit-planes and returns as an output their new contents. The bit-planes are always available for this activity; however, any additional input information must be set up by the software before the step starts, and any additional output information must be picked up by the software after the step is over. As explained in Chapter 11 of the *CAM Book*, the chief function of the *run cycle* is to provide a regular, automated service for such "input delivery" and "output pick-up." For example, in the experiment of Section 8.6 the run cycle contains provisions for reading the event count after each step; in that of Section 9.8 (see also [*CAM Book* 11.13] the run cycle is used to toggle the value of a phase parameter before each step—so that two distinct dynamics take place in alternation at even and odd steps.

9.9.1 The run cycle as a co-process

A run cycle is a Forth word, say, MY-CYCLE , consisting of "segments" separated by occurrences of STEP (or IDLE [6]), used as an "exit marker" (see below). The command

```
MAKE-CYCLE MY-CYCLE
```

makes this word the current run cycle, and executes the first segment of it (i.e., up to the first occurrence of the marker); as soon as the marker word itself is encountered, the execution of the run cycle is suspended.

Thereafter, at every tick of the cellular-automaton clock (represented by the command NEXT-STEP), execution of the marker word that had suspended the cycle is resumed (thus, a step is actually initiated only at this moment) and the following segment is executed, up to the next exit marker. Execution of the run cycle continues in this way, driven by the NEXT-STEP clock. If and when the end of the run-cycle word is reached, execution wraps around to its beginning; the end

[6]The word IDLE is in all respects identical to STEP, except that it does not update the bit-planes.

of the word is *not* treated as an exit marker. Thus, the run cycle is a *co-process*, i.e., a separate process that runs in alternation with the main Forth process (and has its own program pointer and data stack).

When you again use a MAKE-CYCLE command, either with the same run-cycle word or a different one, the current run-cycle process is abandoned and a fresh one is started.

Typically, the user will never explicitly issue a NEXT-STEP instruction, since this instruction is automatically generated by the control panel—singly by the $\boxed{s}$ command (*Step*), and repeatedly by the $\boxed{S}$ command (*Run*). However, NEXT-STEP may be useful for "tracing" the behavior of a complex experiment—by single-stepping through it by hand from the Forth interpreter.

Since the run cycle is executed as a co-process, if you have to do some low-level debugging of its Forth code (e.g., checking what is on the stack after each word is executed, perhaps using DEBUG) you should execute the run cycle directly as a foreground process. If you type the name of a run cycle at the Forth interpreter level, the run cycle will be executed just as an ordinary Forth word. STEP knows that in this context it does not have to wait for a NEXT-STEP prompt.

9.9.2 Micro-steps and macro-steps

Usually, one pass through the run cycle will not represent the entire course of an experiment, but rather a small, self-contained portion of it lasting a few steps and repeated over and over; in this case the run cycle can be seen as a way of synthesizing a "macro-step" out of a short sequence of "micro-steps."

It is often the case that the macro-step should be regarded as an *indivisible* sequence of operations (cf. the two-step cycle of the HPP gas, Section 9.8). It is then useful to have a way to insure that one macro-step has been completed before analyzing the data produced by it or starting a macro-step of a different nature. The word NEXT-CYCLE runs without stopping from the current marker position on through whatever portion of the run cycle is left, wrapping around the end once and finally stopping exactly where MAKE-CYCLE would have left off, i.e., at the first exit marker. On the other hand, the word FINISH-CYCLE stops at the end of the run-cycle word—without wrapping around. Thus, the sequence

```
                                : SAGA
       MAKE-CYCLE HPP-CYCLE      \ attach MY-CYCLE and run initial part
    99 0 DO NEXT-CYCLE LOOP      \ go 99 times through entire MY-CYCLE
               FINISH-CYCLE ;    \ terminate the cycle
```

will execute the macro-step defined by HPP-CYCLE a total of 100 times—since initialization and termination, used in this way, together add up to exactly one pass through the cycle—without any gap or overlap. Specifically, in the HPP-gas rule of Section 9.8, the "initialize" part executes

```
    AUX-TABS
```

each iteration of the loop executes

```
                    STEP   REG-TABS STEP
    AUX-TABS
```

and the "terminate" part executes

```
                    STEP   REG-TABS STEP
```

The advantage of letting MAKE-CYCLE execute all the preparatory operations for the first step, stopping just before starting the step (and similarly for each occurrence of NEXT-CYCLE), is that the state of the whole machine can be examined exactly as it would be during the execution of the step. If you show the expanded view (using $\boxed{x}$) with a grid, the position of the grid will reflect the spatial-phase origin to be used in the step; the color map will be the one that will be used during the step, etc.

If the auxiliary tables are filled with a copy of the regular tables and the machine is left idle after MAKE-CYCLE , the display will show the state of the planes as they are before taking the first step. Typing SHOW-FUNCTION at this point will show the state of the planes as they would be if the step were executed (cf. Sections 9.7 and 9.10.1), *but without actually altering the planes*; in fact, typing SHOW-STATE will show again the actual state of the planes. In this way one can have at any moment a non-destructive preview of the exact action of a step, and compare at leisure the "before" with the "after"—which may be useful for debugging an experiment.

9.10 The event counter

Since the event counter is permanently connected to the color map's *Intensity* output, the main issue is how to characterize the events to be counted and route them to this output. This is done by programming the color map and, if desired, the auxiliary tables—as explained in the previous two sections.

The software machinery for reading the event counter is adequately illustrated, for most circumstances, by the example of Section 8.6. Should a more sophisticated use of the event-counter pipeline be necessary, refer to the information given under EVENT-COUNT in the glossary.

Note that a count of 65,536 (256×256) is just one too many to be contained in a 16-bit counter. Instead of distinguishing a count of 0 from a count of 65,536 by means of an overflow bit, CAM uses a flip-flop that is set when *at least one event* occurred (thus, this flip-flop is set for any count from 1 through 65,536). This flip-flop has separate service routines that are used by the "stop-on-event" facility, discussed under EVENT-STOP in the glossary; this flip-flop is also of course used by EVENT-COUNT when necessary.

9.10.1 What is counted—and when

By contrasting two trivial situations, the following remarks will help you construct a mental model of what actually goes on during display and counting.

When the four bit-planes are directly connected to the color map (this is the default situation, explicitly selected by the command `SHOW-STATE`), a pixel appears on the screen at the same time as the corresponding cell is accessed by the updating machinery, and displays the contents of the cell as of *before* the updating. Thus, at the end of an active step the screen will have displayed what the state of the array was just before that step. If you program the intensity bit to count the number of `1` s in a plane (as in Section 8.6) and take *a single*, isolated step from a given initial configuration, the following will happen:

- During the 1/60-th of a second in which the step is performed, the screen will show the given configuration and the counter will count the number of `1` s in this configuration—while the updating machinery constructs a new configuration.

- During each of the following 1/60-ths of a second the system will take an *idle* step: the display will repeatedly show the newly constructed (now current) configuration; the counter will remain stopped; and the updating machinery will stay idle.

If you read the hardware counter at any moment now, its value will refer to what was counted during the active step—and thus to the initial configuration rather than to the one now appearing on the screen.

Suppose we are running the `PARITY` rule[*CAM Book* 4.2], defined by

```
CAM-A N/VONN
                                          : 5HOOD
     CENTER NORTH SOUTH WEST EAST ;
                                          : PARITY
      5HOOD XOR XOR XOR XOR >PLNO ;
MAKE-TABLE PARITY
```

Let us fill column 0 of the auxiliary tables with exactly the same function of the neighbors, i.e.,

```
                                          : AUX-PARITY
      5HOOD XOR XOR XOR XOR >AUXO ;
MAKE-TABLE AUX-PARITY
```

so that the new value for the bit of plane 0, (which comes out of `PLNO`) appears in duplicate at `AUXO` (cf. end of Section 9.9.2).

Now, let's say `SHOW-FUNCTION` and run the simulation. The end result is simply that during each step the color map will see the *terminal* state of the array rather than the *initial* one. Using, for instance, the *IRGB* map, the event counter will give us the number of `1` s present in plane 0 at the *end* of a step—rather than at the *beginning* of the step as in Section 8.6.

9.11 Pattern, data, and table files

Pattern files can be created, edited, saved, and loaded from the control panel, as explained in Chapter 3. They can be automatically recalled and loaded by an experiment file, as in Section 8.5; see also `FILE>PL` and `FILE>IMAGE` in the glossary.

If several pattern files will be needed in the course of a demo, they should all be mentioned in `OPEN-PATTERN` statements, usually at the beginning of the experiment. If you say, outside of a colon definition,

```
OPEN-PATTERN FIG1.PAT
OPEN-PATTERN FIG2.PAT
     ...
OPEN-PATTERN FIGn.PAT
```

each file is in turn assigned a file control block, opened, and made the current *pattern* file (i.e., the file that operations such as `PL>FILE` and `FILE>PL` use as source or destination). At the end of this sequence, all n files will have been assigned a control block and will be open (but only the last one will be the "current pattern file").

After this, and at any moment in the course of the experiment, whenever the Forth word `FIG2.PAT` is executed (either from the interpreter or inside a colon definition) the corresponding pattern file is made the current pattern file. There is no need to close a pattern file, even if new data has been written to it.

The command `OPEN-DATA` is used in a similar way for making *data files* accessible and/or current. The file control block for a data file includes a pointer that is used for sequential access through the file, and pointer operations (see `DATA-PTR` and `GET-DATA` , and `PUT-DATA` in the glossary, for details) always refer to the current data file.

Typical uses for a data file are

- To accumulate selected analysis data produced by a simulation (Section 8.6).

- To provide a sequence of input parameters (e.g., phases) for successive steps of an experiment.

- To recall data generated by a previous run of an experiment, in order to compare or correlate them, step by step, with those generated by the current run.

In a similar way, for the table files (cf. Section 9.5) we have the words `OPEN-TABLE` , `TAB>FILE` and `FILE>TAB` .

9.12 Initialization

Now that we are familiar with most of the resources of CAM, it will be useful to know how these resources are affected when the machine is initialized.

The command `NEW-EXPERIMENT` will

1. Bring the Forth dictionary to its pristine initial state, forgetting any definitions that were subsequently loaded from files or typed at the terminal.
2. Clear the 4 bit-planes.
3. Clear the CAM lookup tables.
4. Load the color-map table with the default color map, `STD-MAP` .
5. Reset the display mode to the default one (see `LONG?` in the glossary).
6. Select CAM 0 (if more than one CAM card is present).
7. Set up to apply hardware effect of neighborhood assignments to both halves of CAM (i.e., execute `CAM-AB`).
8. Select the major neighborhood `N/USER` and the minor neighborhood `&/USER` .
9. Reset the run cycle to `STEP` .
10. Flush and reset the event-counter pipeline.
11. Turn off control-panel options (grid, magnification, dot mode, cage mode, etc.).
12. Reset default `TAB` , `PAT` , and `DAT` files to be undefined.

On the other hand, the contents of the plane buffers and of the cage are not affected by `NEW-EXPERIMENT` , and thus provide a way of passing plane contents from one experiment to another without going through a disk file.

Chapter 10

The user connector

Let's go back to the HPP-gas rule, in the version of Section 9.8; can we compress the two-step cycle into a single step? Instead of waiting for the particles from the north, south, west, and east to move into the 1×1 scope of the center cell in order to permute them as specified by the collision rule, can't we catch them when they are still one cell away, and permute them while we are bringing them in? (After all CAM has a built-in 3×3 scope of vision.)

Though we only need *four* neighbors, none of the standard CAM neighbors provide these particular four, namely,

0. the *west* bit from plane 0,

1. the *east* bit from plane 1,

2. the *north* bit from plane 2,

3. the *south* bit from plane 3.

In fact, the `N/VONN` neighborhood on CAM-A makes items 0 and 1 (but not 2 or 3) visible for the updating of planes 0 and 1; while the same neighborhood on CAM-B makes items 2 and 3 (but not 0 or 1) visible to planes 2 and 3. That is, two of the neighbors we want are missing from CAM-A, and two from CAM-B.

The neighbors we need are actually present on the *user connector*; in fact, all nine neighbors from each of the four planes—for a total of 36—plus a number of additional signals are present as *outputs* there. Moreover, the `N/VONN` major neighborhood assignment leaves us with two spare address lines (10 and 11) on each lookup table, which can be programmed by a minor neighborhood assignment (cf. Section 9.2 and [*CAM Book* 7.3.2]). What we'll do is program these spare lines as *user neighbors* (with the `&/USER` assignment); that is, they will be directly tied to input pins UA10, UA11 (for CAM-A) and UB10, UB11 (for CAM-B) of the user connector. We'll install four wire jumpers to connect to these input pins the desired plane outputs, coming out of pins N2, S3, W0, and E1 of the user connector.

The overall situation is the following:

CAM-A

BIT NEEDED	SEEN AS	VIA
west of plane 0	WEST	N/VONN
east of plane 1	EAST'	N/VONN
north of plane 2	UA10 ("&NORTH")	&/USER
south of plane 3	UA11 ("&SOUTH'")	&/USER

CAM-B

BIT NEEDED	SEEN AS	VIA
west of plane 0	UB10 ("&WEST")	&/USER
east of plane 1	UB11 ("&EAST'")	&/USER
north of plane 2	NORTH	N/VONN
south of plane 3	SOUTH	N/VONN

We have chosen mnemonic names for the new custom neighbors by analogy with the rest of the naming scheme; for instance, &SOUTH' signifies the *south* bit from the *primed* plane (i.e., the odd-numbered one) of the *other* CAM half, namely CAM-B—and therefore the south bit of plane 3. These names, to be used in the rule descriptor, are assigned to the appropriate address lines by the == assignment. Figure 10.1 shows the entire source code for this experiment.

In that figure, COLL-A and COLL-B represent the conditions under which a collision (and thus the reshuffling) occurs, as seen respectively from CAM-A and CAM-B. As in Figure 9.2, we have made life simpler by including in the collision condition the two cases where there are four particles or no particles, since the reshuffling of identical particles or of empty cells is harmless.

Of course, in place of HPP-FAST we could have written, perhaps more clearly, separate rule descriptors for CAM-A and CAM-B.

The signals on the user connector are discussed in CAM's *CAM-PC Hardware Manual*. The connector's pinout is given in file PINOUT.DOC . All 36 neighbors from the planes are brought out as outputs on this connector, as well as all table address lines for both lookup tables as inputs (except lines 0 and 1, which are permanently connected to CENTER and CENTER'). Thus, by placing a few jumpers on the connector it is possible to complement or replace the standard CAM neighborhoods by custom neighborhoods.

It is also possible to completely bypass the built-in lookup tables and replace them with external ones: the "new state" inputs for all four planes can be made to come, under software control, from four pins on the user connector.

Other output signals of general interest (cf. *CAM-PC Hardware Manual*) include:

```
Screen 1                   HPP-FAST.EXP                    (100%)
\ Fast version of the HPP rule (1-step cycle)         04Jan87tmt
NEW-EXPERIMENT  N/VONN &/USER    DECIMAL \ to make sure 10, 11
10 == &NORTH    10 = &WEST       \ mean the right thing
11 == &SOUTH'   11 = &EAST'      \ in case you were in HEX
                                              : COLL-A
         WEST  EAST' =  &NORTH &SOUTH' =  AND ;
                                              : COLL-B
        &WEST &EAST' =   NORTH  SOUTH' =  AND ;
                                              : HPP-FAST
                        COLL-A IF
              &NORTH &SOUTH' ELSE
                   WEST EAST THEN >PLN1 >PLN0
                        COLL-B IF
                &WEST &EAST' ELSE
                NORTH SOUTH' THEN >PLN3 >PLN2 ;
MAKE-TABLE HPP-FAST
```

Figure 10.1: Source screen for a fast realization of the HPP gas, using the user connector to provide the required extra neighbors.

- The spatial phases and the temporal phases.

- Four pseudo-neighbors that take on the value 1 on, respectively, the top and bottom rows and the leftmost and rightmost columns of the bit-plane array. These are useful, for example, for generating a "NOW" line for one-dimensional cellular automata (cf. [*CAM Book* 9.7]) without wasting one bit-plane for this purpose; for constructing reflecting boundaries (in place of the default wraparound) around the bit-planes; for injecting a flow from one edge and draining it from the opposite edge ("wind tunnel"); and similar situations where a different dynamics is required on a distinguished line drawn across the array.

- Gating signals, to tell external circuitry when the array is actively updating and when it is undergoing maintenance tasks (such as horizontal or vertical retrace). In this way, external counters or event detectors can be prevented from recognizing spurious events.

It must be noted that any jumpers or other circuitry connected to the user connector for special applications may be left in place even when not needed, since it's up to the software to decide whether or not to look at the user-connector signals: all experiments that do not attempt to use this connector run as if nothing was connected to it.

If applications requiring different wiring schemes are to be run on the same CAM, prewired headers (duly labeled and documented) should be used instead of individual jumpers, to facilitate switch-over. If signals have to travel between CAM and external custom hardware, one can use ribbon cable terminated by standard 50-pin headers.

Chapter 11

Advanced interconnection techniques

This chapter introduces techniques for going beyond the standard 256×256×4, toroidal wrap-around format—by interconnecting in special ways the bit-planes of one or more CAM cards.

11.1 Edge gluing

CAM's normal mode of operation, without any external connections, is as four planes of 256×256 bits. In order to simulate larger systems, provision is made for *gluing* the edges of bit-planes together to form larger bit planes. We shall first describe how gluing is performed on a single CAM card to allow it to simulate two planes of 512×256 bits or (with some limitations on the neighborhoods that can be used) one plane of 512×512 bits. Then we shall describe how the bit-planes of one card may be glued to those of another, for example allowing four cards to simulate four planes of 512×512 bits (with no limitations, in this case, on the choice of neighborhood).

Gluing a single CAM card requires hardware and software actions. The hardware action, to be described in more detail below, consists of appropriately cabling certain *glue* connectors on the card; the software action consists of issuing commands to turn on the horizontal- and/or vertical-glue bits of the *control and configuration register*, after the experiment has been initialized by `NEW-EXPERIMENT` . Turning on these bits causes CAM to listen to the signals on the glue connectors, which it would otherwise ignore.

Each CAM card contains two 10-pin *horizontal-glue* connectors, one for input and one for output, and a similarly organized set of *vertical-glue* connectors; you can use single-wire jumper cables to connect planes 0 to 2 and 1 to 3; or to connect planes 0 to 1 and 2 to 3 (see file `PINOUT.DOC` for details).

11.1.1 Horizontal glue

To configure CAM horizontally into two planes of 512×256 bits, use four insulated, single-wire jumpers to connect the individual input-output pairs as follows: hglue-out0 to hglue-in2, hglue-out2 to hglue-in0, hglue-out1 to hglue-in3, and hglue-out3 to hglue-in1. Carefully follow the diagram in the file `PINOUT.DOC` so that the outputs are not shorted to ground.

In software, issue the command `HGLUE SET-CCR` , to turn on the horizontal-glue bit in the configuration control register. Leave the vertical glue connector uncabled and the vertical glue bit clear. Planes 0 and 2 will now be joined horizontally, as will planes 1 and 3. In other words, CAM-B is now a horizontal extension of CAM-A. Therefore, in running an experiment, it is necessary to specify the same neighborhood and rule table for CAM-B as for CAM-A, which can conveniently be accomplished by defining the neighborhood and table in CAM-A and then issuing the command `B=A` . By the same token, the minor neighborhood selection `&/CENTERS` should be avoided since it will now refer to the site 256 units away from `CENTER` in what is now the *same* bit plane.

Although CAM is now logically connected as a 512×256 array, the display still treats the bit planes as before, so that the two halves of the larger array will be seen superimposed on the screen. For example, using the *IRGB* color map, the two halves of glued plane 0–2 will be shown as grey and green superimposed, while the two halves of glued plane 1–3 will be shown as red and blue superimposed. If this proves confusing, one can turn off the display of planes 2 and 3 (e.g., by hitting the color toggle keys $\boxed{\texttt{f8}}$ and $\boxed{\texttt{f10}}$) and one will be left with a display that shows a 256×256 window into the 512×256 system. The arrow keys can be used to shift this window.

11.1.2 Vertical glue

To connect CAM as a single 512×512 system, install the jumpers as described above on the horizontal glue connector, and install additional wires in an analogous manner on the vertical glue connectors.

In software, issue the commands `HGLUE SET-CCR` and `VGLUE SET-CCR` to enable gluing both horizontally and vertically. Since all four planes are now logically connected into one large torus, the neighborhood selection must be one that makes the same neighbors available on all four bit planes. This means that `N/VONN` or any of the Margolus neighborhoods can be used, but `N/MOORE` cannot. The transition rule must of course also be the same for all bit planes: plane 1's new state must be the same function of the neighbors `NORTH'` , `SOUTH'` , etc.

as plane 0's is of the neighbors `NORTH` , `SOUTH` , etc. The following is a typical example, in which the nonmonotonic majority function `5ANNEAL` [*CAM Book* 5.4,8.3] is applied to all 4 bit planes.

```
    NEW-EXPERIMENT
    HGLUE SET-CCR  VGLUE SET-CCR
    N/VONN                                            : 5ANNEAL
                                  + + + + { 0 0 1 0 1 1 } ;
                                                      : RULE
       NORTH  SOUTH  EAST  WEST  CENTER  5ANNEAL >PLN0
       NORTH' SOUTH' EAST' WEST' CENTER' 5ANNEAL >PLN1 ;

MAKE-TABLE RULE
B=A
```

For display, toggling off all but one of the colors will provide a 256×256 window into the 512×512 array, which can again be shifted using the arrow keys.

11.1.3 Gluing cards together

Gluing can also be performed between bit planes of *different* CAM cards. We describe how to configure two CAMs to operate as a 512×512 array with two bits per site and no restriction on the main neighborhood selection. Other configurations are analogous.

First, each of the two cards is cabled individually as described above for a 512×256 array.

Next these two 512×256 systems are glued together vertically into a single 512×512 system by using two 10-conductor gluing cables to cross-cable between the boards, each cable running from one board's vertical-input glue connector to the vertical-output glue connector on the other board.

Finally, the two cards are connected by a master/slave cable (also available as an option), installed in two slots of the same PC, and their video connectors are daisy-chained together. The setup of the slave and monitors is described in the next section on multiple CAMs. In the software, commands are issued to initialize both CAMs, turn on vertical and horizontal gluing, and enable the desired monitor configuration. A typical format is given below.

```
    NEW-EXPERIMENT   2 CAMS
    N/MOORE                              : RULE
                                       . . .
                          an arbitrary rule
                  depending on Moore neighbors
                                       . . . ;
```

```
MAKE-TABLE RULE
B=A
TAB>BUF
                                              : INIT-ALL
                                    FOR-ALL-CAMS
            VGLUE SET-CCR  HGLUE SET-CCR
                 CAM-AB N/MOORE  BUF>TAB
                                       NEXT-CAM ;
INIT-ALL
```

As explained in the next section, the loop FOR-ALL-CAMS ... NEXT-CAM issues the same information (neighborhood, glue-bits, and rule table) to the two CAMs, only one of which can be spoken to at a time.

11.2 Multiple CAMs

Up to eight CAM cards can be ganged togetherand operated jointly in a relatively straightforward manner; in this way, one can obtain CAM systems having substantially greater capabilities. Typical applications of multi-card systems are

- The use of one CAM card to supply good-quality, finely-tuneable random numbers to another.
- Simulating two-dimensional automata with more than 16 states per site.
- Simulating three-dimensional automata.
- Simulating two-dimensional arrays larger than 256×256 while retaining all of the features available with a single card (cf. 11.1).

Operating multiple CAMs for any of these reasons is not difficult, but it requires attention to several kinds of physical connection among the cards and appropriate commands for communicating with them individually or collectively from the PC.

The most important physical connection among the cards (aside from the shared PC bus into which they are all plugged) is a chain of master/slave connections which serves to distribute a common clock signal (that of the "master" CAM) to all the other "slave" CAMs. In addition, on the slaves, a clock jumper (W1) must be moved and address switches must be set to map each CAM into a different region of the PC memory—with the master occupying addresses DF800 to DFFFF and the slaves (up to eight) occupying consecutively lower 2K byte regions.[1]The

1

whole 16K byte region (normally DC000–DFFFF) of the address space reserved for the eight CAMs can be remapped by means of programming switches in order to avoid conflicts with other cards. Cf. the *CAM-PC Hardware Manual* and CAM-BASE in the glossary. The clock jumber provides a clean signal for reliable operation.

To set up the connections, the sixteen-pin *master-out* connector of one CAM (which becomes CAM 0—the *master*) is cabled to the *slave-in* connectors of the other CAMs. The address switches of each of the slave CAMs must be set to successively lower addresses to become CAM1, CAM2, etc.

A second kind of connection is generally needed to handle the displays. As explained in Appendix A, each CAM has a nine-pin video-in connector and a matching video-out connector, which can be switched under software control between displaying that CAM's own video signal or simply passing on a video signal received from some other CAM (or from the PC color card) via the card's video-in connector. If all the CAMs are daisy-chained together, with the video output of one going to the video input of another, and the output of the last CAM going to the monitor, then it becomes possible to display any CAM's video-output at will under software control.

A third kind of connection will depend very much on the particular application of the multi-CAM system. This is the connection among the user connectors of the various CAMs. Since all the CAMs are operating under the same clock, plane-output lines (e.g., C1, designating the center cell on plane 1) from one CAM may be wired to table address inputs (e.g., UA10, designating the tenth user input to CAM-A's lookup table) of another CAM, in the same manner as custom neighbor connections are made on the user connector of a single CAM (see Section 10). In this way, it becomes possible, using n CAMs, to simulate cellular automata rules depending on up to $4n$ bits per site, or, alternatively, to simulate a three-dimensional array of $256\times256\times4n$ with one bit per site. In addition to individual connections on the user connector, the *depth-glue* connectors provide a simple way to the exchange center bits among multiple CAMs. (Cf.*CAM-PC Hardware Manual.*)

A fourth kind of connection has already been discussed: the gluing cables used to connect bit planes (in the same or different CAMs) into arrays larger than 256×256 sites. These connections are made only when gluing is desired.

Having made the necessary hardware connections, it becomes necessary to communicate with the multiple CAMs via software. Immediately after NEW-EXPERIMENT one should include the command 'n CAMS' (e.g., 3 CAMS) to tell the PC how may CAMs there are. As a rule, communication (specifically, loading a table or color map, setting bits of registers such as CCR, PCA, and PRA, operations such as RND>PL on data in the bit planes, and the assignment of values to pseudo-neighbors such as <ORG-HV> and <PHASES>) takes place with only one CAM at a time; uder the control of the word `CAM-SELECT` . For example, `0 CAM-SELECT` causes subsequent commands to refer to CAM 0, the master, until a subsequent command such as `2 CAM-SELECT` , after which communications would be directed at CAM 2. The only exceptions to the rule that commands are directed to one CAM at a time are the commands `NEW-EXPERIMENT` , which resets all CAMs, and `STEP` , which causes all CAMs to step. When uniform data must be directed at all the CAMs (e.g., giving them all the same rule) the command in question (such as `VGLUE SET-CCR`) is put in a loop demarcated by the words `FOR-ALL-CAMS` ... `NEXT-CAM` . An example is given at the end of the previous section on gluing. Note in particular that neighborhood selections must be repeated.

If the video-in and -out connectors on the CAMs are properly daisy-chained together, the command `SHOW-CAM` can switch the display from one CAM to another. Thus, `0 SHOW-CAM` causes the master to be displayed, and `1 SHOW-CAM` displays the first slave. It is possible, though perhaps confusing, to choose one CAM for communication and another for display. Keyboard commands such as $\boxed{;}$ (*Random*) then appear to have no effect, but in fact they are operating on a CAM that is not being displayed. The control-panel $\boxed{a}$ command alleviates this problem by performing the functions both of `SHOW-CAM` and `CAM-SELECT` ; for example, typing 1$\boxed{a}$ causes CAM 1 to be displayed and at the same time selects that CAM for communications with the PC.

11.3 Kicking

A particularly useful consequence of the fact that the PC communicates with each of the several CAMs individually is the ability to rapidly generate high-quality random numbers on one CAM for use by another CAM. The numbers are generated by a particle-conserving lattice-gas rule on one CAM (say, the slave) for use by another (the master). To reduce correlations normally found in the lattice gas (due to the locality of the rule of motion of the particles), the plane containing the lattice gas is *kicked* after every step; this kicking consists of adding a random offset to the row and column address registers (PRA and PCA) of the slave CAM. This causes the entire gas configuration to shift by a large random amount (much as in the "hyperspace" option in the game *Asteroids*) at each step, in addition to its normal dynamics, and nearly obliterates the local correlations that would be exhibited by an unkicked lattice gas used as a source of noise. The enclosed source file `CAM1RAND.4TH` , intended to be included in experiments requiring random numbers, sets up the two-CAM configuration, loads the lattice gas rule for CAM 1, and redefines the run cycle so as to cause stepping and kicking of the lattice gas in CAM 1 each time a step is taken by the master CAM 0 . The file also defines pseudo-neighbors `RAND0`, ..., `RAND3`, available in CAM 0 under the minor neighborhood selection `&/USER`, and intended to be connected (via the user connector) to the four bit-planes of CAM 1. The enclosed experiment file `CANIS2.EXP` makes use of `CAM1RAND.4TH` to generate random numbers for a simulation of the canonical-ensemble Ising model.

Of course, kicking can also be performed on a single CAM. This approach is illustrated in the accompanying annotated experiment `CANIS1.EXP`, which is similar to `CANIS2.EXP` but uses only one CAM— containing both the lattice gas and the Ising configuration. The disadvantage of using only one CAM is that—since the kicking shifts all bit-planes of a CAM card by the same amount—any information that one wishes not to be kicked (in this case the Ising configuration) must be swapped out of CAM (into a buffer in the PC) each time a kick is performed. This slows the simulation several-fold with respect to a one-CAM system without kicking (which however has significantly distorted behavior due to the correlations in the "random" numbers) or to a two-CAM system with one CAM dedicated to the kicked lattice gas.

Part IV

Appendices

Appendix A

Configuring the monitor(s)

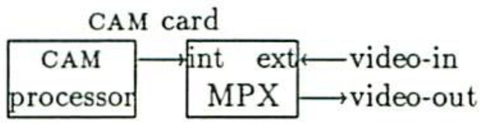
CAM produces a video signal intended to directly drive a standard PC color monitor,[1] and thus a color graphics adapter card is not needed. However, when using CAM it is desirable to retain the ordinary text terminal functions of the PC, and this can be done by either having a separate monitor for the PC or sharing a monitor between the PC and the CAM card(s).

A.1 Display multiplexing

The most trivial hook-up entails leaving the PC connected to its own monitor (either monochrome, through a monochrome card, or color, through a color card), and adding a separate color monitor for CAM (or for each of the CAMs if you have more than one), connected to the card's *video-out* port.

On the other hand, for economy, portability, or compactness you may want to use only one monitor for both CAM and the PC. Moreover, if you have more than one CAM card installed, it may be impractical to use a separate monitor for each card.

In order to provide maximum flexibility, the CAM card has a *video-in* as well as a *video-out* connector, and is equipped with a two-input/one-output "daisy-chain" multiplexer, denoted by 'MPX' in the following diagram



(*video-out* is a female 9-pin connector, *video-in* a male one). Depending on the setting ('int' or 'ext') of a software switch, the signal appearing at the video-out

[1]I.e., TTL RGBI, NTSC-compatible scanning format, with 60 Hz vertical sync and 15,750 Hz horizontal sync—such as produced by a CGA card and accepted by any CGA-compatible monitor, including EGA monitors and most multi-sync monitors.

port will be either the CAM processor's *internal* video signal or a (TTL-buffered) copy of whatever *external* signal is coming through the video-in port.

Thus, for the trivial hook-up mentioned above the switch would be permanently set on the 'int' position and CAM's color monitor would be plugged into the video-out port, while the video-in port would be left disconnected. The PC's output would go directly to its own monitor. This mode of operation is selected by the command `2 DISPLAYS` .

On the other hand, if your PC comes equipped with a *color* card, you can use a single color monitor and time-share it between CAM and the PC. In this case, the output of the color card would go to CAM's video-in port via the *routing cable.*[2] This mode of operation is selected by the command `1 DISPLAYS` . In this mode, pressing the $\boxed{\text{a}}$ key from CAM's control panel will toggle the display source between 'int' (the CAM signal) and 'ext' (the PC signal). Control-panel keys that request keyboard input, such as $\boxed{\text{P}}$ and $\boxed{\text{G}}$ (but not $\boxed{\text{f}}$) temporarily switch back to the PC display for this input.

As distributed, the CAM program defaults to this "`1 DISPLAYS`" time-shared mode (cf. Section 2.4), and initially selects the PC as the display source. See below and Section B.1 for ways to customize the software so as to reflect your actual monitor hook-up.

Warning: *Do not attempt to share a single display with a* monochrome *card, unless you have a special, multi-purpose monitor that can indifferently accept color or monochrome signals. Many color monitors (and in particular the standard* IBM *one) can be permanently damaged when driven by the 18-KHz horizontal sync provided by the monochrome card.*

For similar reasons, do not connect a standard PC monochrome *monitor to* CAM*'s output (which is equivalent to that of a color card). It won't work, and may be permanently damaged.*

A.2 Multiple CAMs

If your PC is equipped with two or moreCAM cards[3] the fully nonmultiplexed display mode is selected by the command n+1 `DISPLAYS` , where n is the number of cards you have (e.g., `5 DISPLAYS` for four cards). The video signals from the n cards will go directly to the n color monitors, while that from the PC will go to the (n+1)-th monitor (color or monochrome).

If you want to use one color monitor for all the CAMs and a separate monitor

[2]Supplied with CAM. This is just a foot or so of 9-conductor cable, with a male header at one end and a female at the other. Pin 1 goes straight to pin 1, and so on.

[3]The software must be made aware of this by the command n `CAMS`; see Section 11.2.

for the PC, daisy-chain the CAMs only and use the 2 DISPLAYS mode. Typing 3$\boxed{\text{a}}$ from the control panel will route to the one color monitor the video signal from CAM 3 (remember that the numbering of CAM cards starts from 0!); typing just $\boxed{\text{a}}$ will have no effect.

If you want to share that one color monitor also with the PC, chain the PC's video output to the last CAM's video-in port and use the 1 DISPLAYS mode. Typing 3$\boxed{\text{a}}$ from the control panel will route to the monitor the video signal from CAM 3; subsequently, typing just $\boxed{\text{a}}$ will toggle the display between the PC and CAM 3.

A bit called CAMOUT in CAM's CCR register (*control and configuration register*) determines whether the external (0) or the internal (1) video signal is fed to the video-out port. To set this bit for, say, CAM 3 you do

```
 3 CAM-SELECT  CAMOUT SET-CCR
```

and similarly you clear it with CLR-CCR .

These primitives are called by the Forth word DISPLAYS and the control-panel functions attached to the $\boxed{\text{a}}$ key; they can be used directly for special configuration needs.

When you are in a fully non-multiplexed mode, the control panel will not change any CAMOUT bits. Thus, if you want to directly manipulate these bits for several CAMs you should begin by telling the software that you have $n+1$ displays.

Appendix B

Customizing the software

A number of features of the CAM Forth system may be personalized according to your taste, your needs, and your resources. These changes can be made on the fly, from the Forth interpreter, and forgotten at the end of the session; they can be retained from session to session, by saving the customized version of the system to disk; or they or can be given a permanent status in the software by entering them in the appropriate source files and recompiling the relevant part of the system.

B.1 Personalized features

The following considerations apply indifferently to `F83.EXE` or `CAM.EXE` —which is a superset of the latter; we'll use `CAM` in the examples.

Fire up `CAM.EXE` from DOS, as in Section 1.3. You'll recall that the screen editor places a stamp with user identification and date on each screen you create or modify (Section 6.2). To tell `CAM` who you are, type

```
I'M xyz
```

where `I'M` is a Forth word that enters `xyz` (or any other three-letter choice) as your identifier.

If your PC system has no clock, it is a good idea to include the DOS `DATE` and `TIME` commands in the `AUTOEXEC.BAT` file (which is executed whenever the PC is booted) to encourage you to enter the current date.

The default base for numeric conversion (when you type in numbers directly to the Forth interpreter, or when you want it to print out numbers) is ten. If you prefer base sixteen, type

```
' HEX IS SET-BASE
```

As usual in Forth, you can change to any base at any time, depending on the current needs; the base you choose with `SET-BASE` is that which is selected by default whenever you enter the interpreter, and restored when an error occurs, you hit the $\boxed{\text{BREAK}}$ key, or load a file (with $\boxed{\text{l}}$ or $\boxed{\text{L}}$) from the control panel.

The default choice is that upper- and lower-case names are distinct (and some existing Forth words of no concern to you make use of this distinction). If you wish to be able to type upper- and lower-case interchangeably (which we, however, discourage), give the command

```
CAPS ON
```

and everything you type thereafter will be read as upper-case.

If you are using separate monitors for CAM and the PC, type `2 DISPLAYS` to prevent the software from trying to share the CAM monitor with the PC (Appendix A).

If you have a Centronix-like—rather than Epson-like—printer (for other printers see Section B.2.4), type

```
' CENTRONICS IS INIT-PR
```

Section B.3 discusses how to change the defaults for where the system expects the source files to be. Section 6.1 mentions the `VARIABLE AUTO-X` , whose default value you may want to change. Similarly for the `VARIABLE INSO` , which controls whether you are in insert mode when you enter the screen editor..

If you have made any of these changes, you can save your customized version of `CAM` just by typing

```
SAVE-SYSTEM C:CAM.EXE
```

or whatever name and drive you choose. Provided you keep a copy of the original `CAM.EXE` (or `F83.EXE` , if that's what you are customizing), no irreversible damage can be done if you somehow mangle this process.

For documentation and repeatability, the above operations or any number of other minor changes or additions can of course be edited first on a screen, and then loaded into the system (cf. Section B.2.4).

B.2 Recompiling the CAM system

If you want to extend or improve CAM Forth you are provided with all the information and the tools to do it.

Starting from Forth's *kernel*[1] you can regenerate the whole `CAM` system from scratch—after having made any desired changes in the appropriate source files. For less extensive changes, you may want to start your regeneration process at a

[1]This is a minimal, executable Forth system containing just enough resources to be able to bootstrap itself by compilation of additional Forth screens.

later stage. On the other hand, if you really want you can start at an even earlier stage: using the current version of Forth and the provided metacompiler you can generate a new kernel—one that speaks a somewhat different dialect of Forth, or perhaps a radically different one.

B.2.1 Generating a new CAM.EXE

If you only want to change something in the CAM portion of the source code (files of the form CAM-*.4TH) you only need to recompile this part of the system.

With all of the CAM-*.4TH files on the default drive, in the default directory (here we'll use C:) start up the F83.EXE program from DOS by typing

```
   C>f83 cam-load.4th
```

(of course, you could give a path-name for F83.EXE), and once you are in Forth type OK .

This loads all of the required files, and (if everything runs without errors) creates a new version of CAM.EXE . If one of your changes causes an error during loading, you will be placed in the editor at the point that needs to be corrected; once you are ready to try again, exit to DOS and begin anew.

B.2.2 Generating a new F83.EXE

If you have made any changes in the files EXTEND86.4TH , CPU8086.4TH , UTILITY.4TH or PC.4TH , to regenerate the CAM system you must first generate a new F83.EXE and then follow the procedure of the preceeding section.

If you already have the program KERNEL.EXE in your files you can generate F83 directly, otherwise you will have to run a meta-compilation first (see next section) to generate KERNEL.EXE .

With the four source files mentioned above all on the default drive, in the default directory (say C:) start KERNEL from DOS with

```
 C>kernel extend86.4th
```

and once you are in Forth type OK . This loads all extensions and creates a new version of F83.EXE on the default drive.

Say BYE to Forth, and you're done and back in DOS. Note that if you get any compilation errors before about the middle of the file PC.4TH there won't yet be a screen editor loaded, and so Forth will only be able to print out the file, screen number, and line number where the error was detected—you'll have to exit to DOS and start up a working F83 or CAM to make corrections.

B.2.3 Generating a new KERNEL.EXE

If you want to change something in the file `KERNEL86.4TH` (which defines Forth "in Forth" for the metacompiler), you will have to go through all three stages of the compilation process in order to get a new working `CAM` system; this only takes a few minutes.

Make sure `META86.4TH` and `KERNEL86.4TH` are on the default drive (again, we'll use `C:`) and start up either `F83` or `CAM` (the metacompiler can be run from either of these). Type

```
C>f83 meta86.4th
```

(or `cam meta86.4th` if you're starting from `CAM.EXE`). Once you are in Forth, type `OK` and the meta-compilation will begin. When it is done, say `BYE` to leave Forth. You now have a new `KERNEL.EXE` file, and can proceed to produce the rest of the system.

B.2.4 Custom screens

If you are recompiling `F83` , you may first want to customize certain default parameters by editing the appropriate source screens (the most relevant options are on screen 13 of `EXTEND86.4TH`), so that the changes will be permanently incorporated in the software. For example, you might want to define your own version of `NORMAL` ,

```
                                      : (MY-NORMAL)
                    (NORMAL) REVERSE ON ;
   ' (MY-NORMAL) IS NORMAL
```

which would have you normally work in reverse video (black characters on a white background).

To produce pleasing listings with a printer different from the default one (Epson-like), you should find out how to put your printer in compressed mode. If your printer is a Centronics, for example, you should comment out the command

```
' EPSON IS INIT-PR
```

on screen 13, and activate the (normally commented-out) command

```
' CENTRONICS IS INIT-PR
```

on the same screen (the appropriate `CENTRONICS` word is already defined in the system).

If your printer is something else, you will need to define a word to set it up. Suppose your printer is a Kludge-97 and wants to receive a CONTROL-C character to put it in compressed mode; then you should say

```
                                        : KLUDGE-97
                           CONTROL C EMIT ;
   ' KLUDGE-97 IS INIT-PR
```

Even if you don't want to recompile the system at this moment, you can edit screen 13 of EXTEND86.4TH , load just this screen by hand (using LOAD), and type

```
SAVE-SYSTEM CAM.EXE
```

This gives you an executable version of CAM in which INIT-PR is vectored to KLUDGE-97 ; moreover, since the screen already has the desired changes in it, if you ever recompile the system you won't have to redo these changes—they'll be automatically incorporated.

B.3 Installing the system on a hard disk

If you have a hard disk, it is desirable to transfer to it all of the CAM software (or at least the files with extension EXE and 4TH).

Make a new directory in the hard disk—CAM is as good a name as any for this directory—and copy to it all the distribution software:

```
C>CD \
C>MD CAM
C>CD CAM
C>COPY A:*.*/V
```

(we assume the hard disk is drive C: and the floppies use A:). The last line will have to be repeated after inserting each floppy.

To run CAM , boot the computer in the usual way, go to directory CAM in drive C: , and type CAM .

If you have a hard disk (or large-capacity floppy-disk drive) and have moved all of the sources to one drive, you should tell the VIEW facility (Section 7.1) where to find the source files.[2] Assuming the hard disk is in drive C: , type

```
DRIVE C: IS-HOME-OF KERNEL86.4TH
DRIVE C: IS-HOME-OF EXTEND86.4TH
DRIVE C: IS-HOME-OF CPU8086.4TH
DRIVE C: IS-HOME-OF UTILITY.4TH
DRIVE C: IS-HOME-OF PC.4TH
```

for the F83 sources, and similarly for the CAM sources. In general, if you are keeping the source code on-line but have moved some of the files from the drive

[2]As supplied, VIEW and FIX will look for the sources on the *default* drive.

where the system expects it to be to another drive, you'll have to do something similar.[3] However, information about where each source file resides is automatically updated if you re-generate the system (cf. Section B.2).

If you are using DOS version 3.0 (or later), you can put the source files in a separate directory, and give that directory a drive letter using the SUBST command.

B.4 Mouse

When using the plane editor, some of the keyboard functions can be complemented by a mouse used as a pointing device (Section 3.10). For this purpose, CAM Forth supports a MOUSE SYSTEMS mouse, plugged into serial port 1. No software installation for the mouse is needed.

The mouse connects to serial port 1 (device COM1), and uses interrupt no. 4. There are no provisions for setting it up as COM2; on the other hand, the mouse interrupt number can be changed (should there be an interrupt-number conflict with other hardware sitting on the PC bus) by writing the desired value in the VARIABLE MOUSE-IRQ# (see Glossary).

In later versions of the software we may allow a variety of mice by having a separately installable mouse driver. Check the file READ-ME.DOC for the latest information.

[3]Eventually, this problem will be solved more satisfactorily with a variant of the DOS PATH command, which will let you specify the paths to be searched when source code is being sought; for the moment, directory paths are not supported from within Forth.

Appendix C

The assembler

A complete assembler for the full 8086 instruction set is included in the F83 package. The source code is in the file `CPU8086.4TH` .

This assembler is written in Forth (the original version is by Michael Perry). The instructions are organized in different classes, and a defining word is created for each class. Each Intel mnemonic corresponds to a Forth word which, when interpreted, will assemble machine-language code in the dictionary. If the mnemonic appears in a COLON definition, it will assemble code every time that definition is executed. Thus COLON words can act as assembler macros.

Structured conditionals are supported by this assembler. Structures such as `BEGIN ... WHILE ... REPEAT` , `IF ... ELSE ... THEN` , and `DO ... LOOP` are provided; they compile appropriate branches which test processor flags. In our experience, the use of these words simplifies one of the most error-prone aspects of assembler-language programming. Note that these words belong to the `ASSEMBLER` vocabulary (Section 5.3), and are quite distinct from their synonyms in the main `FORTH` vocabulary. The latter may still be used during assembly (for example, in macros that create repetitive portions of assembly code)—but explicit vocabulary switching is then mandatory.

By recoding into assembly language just a few words that appear in frequently executed inner loops of a Forth program, execution speed is often dramatically improved.

C.1 Symbols and syntax

The Forth assembler uses the same mnemonics as the Intel assembler (you can use a conventional programmer's reference card for the 8086, such as that printed by MICRO LOGIC, Hackensack, NJ, for a complete list of mnemonics, opcodes, operands, and meaning of machine instructions); however, the reverse-Polish notation of Forth suggests a different order for the operands. Namely, the Intel order

⟨mnemonic⟩⟨destination⟩⟨source⟩

of the instruction `MOV BX,AX` (move contents of the AX register to the BX register) becomes, in Forth, `AX BX MOV` (read "from AX to BX move"), with the syntax

⟨source⟩⟨destination⟩⟨mnemonic⟩.

The only exception to this rule is the `OUT` instruction, which takes its arguments in the same order as the `IN` instruction.

For reference, a complete synopsis of legal instruction formats for the Forth assembler is given in Section C.8.

Single-operand instructions do the obvious thing. For example, `AX POP` assembles an instruction to pop the top item of the stack into the AX register. Allowed operands are:

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

[BX+SI]  [SI+BX]  [SI]   #
[BX+DI]  [DI+BX]  [DI]   #)
[BP+SI]  [SI+BP]  [BX]  S#)
[BP+DI]  [DI+BP]  [BP]
```

The operands in the first group are complete by themselves, while those in the second group expect a parameter on the stack, as in the following examples

`3 # AX MOV`	Move the immediate number 3 into the AX register.
`AX 0 [DI] MOV`	Move the contents of AX into the location pointed to by the DI register, with a displacement of 0 added to the address; use the default DS segment.
`ES: AX 0 [DI] MOV`	Same as above, but addressing relative to the ES segment.
`3 # 300 #) MOV`	Move the immediate number 3 into location 0300 relative to the DS register.
`300 #) JMP`	Jump to location 0300 relative to the CS register.

Note that `#` signals a number, `#)` an address, and `S#)` a segment address. The only instructions that can use a segment address are `JMP` and `CALL`.

Each of the allowed operands leaves a 16-bit word on the stack during assembly as an argument for the mnemonic, which comes last. This is the operand's only effect. For instance, in the last example above, `#)` leaves a flag on the top of the stack which tells the assembler word JMP that the next item on the stack is an address.

Special Cases

The "shift" mnemonics each assemble a shift by one position unless CL is on the top of the stack:

AX SHL	Shift AX left by 1.
AX CL SHL	Shift AX by CL positions.

The "string" mnemonics CMPS , MOVS and SCAS each assemble an operation on 16-bit words unless preceded by BYTE , in which case they operate on bytes.

The "string" mnemonics STOS and LODS expect either AX or AL on the stack, to tell them whether to perform a word or byte operation, as in AX LODS , AL LODS , AX STOS , and AL STOS .

Note that the word BYTE used above, as well as FAR used with the jumps and calls below, set flags rather than leave anything on the stack.

C.2 Exotic jumps, calls, and returns

The 8086 machine language includes a variety of exotic intersegment hops (supported by the Intel assembler) that will probably not be used by those writing CODE words for Forth. However, for completeness the corresponding Forth assembler constructs are described here.

Subroutines have to know whether they are returning to a point within the same segment or in a different segment. A FAR return expects both a segment and an offset on the stack, while a normal return expects only an offset (don't blame us!).

RET	Normal return.
FAR RET	Intersegment return.
3 +RET	Return and then pop 3 values off of the stack.
2 FAR +RET	Intersegment return, and pop 2 values.

The CALL and JMP instructions have normal and FAR forms, and some choices about how to get the destination address.

`1234 #) CALL`	Assembles unconditional call to address 1234, with the address stored as a relative displacement.
`1234 S#) CALL`	Same as above, but the address is stored as an absolute offset from beginning of segment (useful in relocatable code, I assume).
`AX CALL`	The address is in AX.
`0 [BX] CALL`	BX points to the address
`1234 400 #) FAR CALL`	Direct intersegment call to segment 400, offset 1234.
`1234 S#) FAR CALL`	The four bytes starting at location 1234 contains the segment/offset for a FAR call.
`0 [BX] FAR CALL`	BX points to the segment/offset double word.

Similar considerations apply to JMP (same addressing modes).

C.3 Using machine language routines in Forth

Some the of resources of the 8086 microprocessor are used in running the Forth *virtual machine*, and the assembler has been complemented by words that make reference to these resources convenient.

In particular, Forth uses the SI register for its *Instruction Pointer* and the BP register for its *Return-stack Pointer*; the words IP and RP have been introduced as synonyms for, respectively SI and BP. These registers (as well as CS, SS, DS, and the data direction flag) must be maintained across Forth words (i.e., any CODE word that changes them must restore them before returning control to the inner interpreter (see below). All other registers (including ES) may be used freely by Forth CODE words.

We shall discuss later on the mechanism of the Forth inner interpreter, which constitutes the Forth virtual machine. For the moment, it will be more useful to give some examples of typical CODE words, taken from the file KERNEL86.4TH .

```
\ Drop the top element of the stack

  CODE DROP ( n)
   AX POP     \ Pop one element off data stack (8086 stack)
```

```
     NEXT     \ Return control to the inner interpreter.
   END-CODE   \ Exit the assembler mode of interpretation
```

Note that NEXT is a macro. It assembles a jump to a piece of machine code—the actual inner interpreter—located at an address which we shall symbolically call >NEXT .

```
\ Fetch a 16-bit value from addr

              CODE @ ( addr -- n)
          BX POP
     0 [BX] PUSH
   NEXT  END-CODE
```

```
\ Store a 16-bit value at addr

              CODE ! ( n addr)
          BX POP
      0 [BX] POP
   NEXT  END-CODE
```

Note that [BX] needs a displacement as an argument, even though we only wanted a displacement of zero.

```
\ Duplicate top of stack

              CODE DUP ( n -- n n)
          AX POP
         AX PUSH
           1PUSH     \ Push AX, return to inner interpreter
          END-CODE
```

The macro 1PUSH compiles a jump to location APUSH —which is located just before >NEXT and contains the instruction AX PUSH .

```
\ Exchange top two elements of stack

              CODE SWAP ( n n' -- n' n)
          DX POP
          AX POP
           2PUSH     \ Push DX and AX, return to inn. interpr.
          END-CODE
```

The macro 2PUSH compiles a jump to location DPUSH —which is located just before APUSH and contains the instruction DX PUSH .

```
\ Copy second element from top

              CODE OVER ( n n' -- n n' n)
          DX POP
          AX POP
         AX PUSH
           2PUSH
          END-CODE

\ Replace top two elements by their sum

              CODE + ( n n' -- n+n')
          BX POP
          AX POP
       BX AX ADD
           1PUSH
          END-CODE

\ Replace top 2 elements by their difference

              CODE - ( n n' -- n-n')
          BX POP
          AX POP
       BX AX SUB
           1PUSH
          END-CODE
```

Note that `BX AX SUB` means "subtract `BX` from `AX` and leave the result in `AX`." This is a bit awkward, since the order of the arguments in the assembler code is the inverse of the usual one, but is a consequence of interchanging the order that Intel uses in their menmonics. Attention should be paid to this notational quirk.

The same problem arises with the compare instruction, `CMP`. Let's define two pieces of code that return on the stack the quantities `TRUE` and `FALSE`:

```
           LABEL YES
   -1 # AX MOV
         1PUSH
           LABEL NO
    0 # AX MOV
         1PUSH
```

We then define the Forth words `<` and `>`:

```
\ Return TRUE if n<n'
```

```
        CODE <  ( n n' -- flag)
    AX POP
    BX POP
 AX BX CMP
    YES JL
 NO #) JMP
    END-CODE
```

```
\ Return TRUE if n>n'
```

```
        CODE >  ( n n' -- flag)
    AX POP
    BX POP
 AX BX CMP
    YES JG
 NO #) JMP
    END-CODE
```

After the instruction AX BX CMP in the code for < , the conditional jump YES JL will occur only if BX is less than AX, which is the opposite of what one might expect. But by assigning AX to n' and BX to n we interchanged the order of the arguments, and the overall test for "less than" works fine.

Note that conditional jumps do not take a usual argument, but just an address on the stack.

C.4 The inner interpreter

And now we come to the machine code which makes up the inner interpreter. Of course this code was not produced by the Forth assembler (which wouldn't be able to run if the inner interpreter weren't already in place), but by the *meta-assembler*—a tool available to the metacompiler during system generation (see Section B.2). However, you might want to write your own customized version of the inner interpreter (for instance, for debugging purposes) and switch to it when desired. For this reason, in the version that follows we have avoided using meta-assembler constructs.

```
      LABEL DPUSH \ Label for jump in 2PUSH macro
   DX PUSH
     ( LABEL APUSH) \ Target location for jump
                    \   in 1PUSH macro
   AX PUSH
     ( LABEL >NEXT) \ Target location for jump
```

```
                        \   in >NEXT macro
      AX LODS           \ These three lines are the inner interpreter
     AX W MOV           \   A detailed explanation
    0 [W] JMP           \   is given below
```

The word W which appears above is a synonym for BX; in the Forth virtual machine, this register is used as the *Word pointer*. The appropriateness of this appellation will become apparent.

Note that we didn't insert actual labels to mark locations APUSH and >NEXT , since we wanted contiguous machine code. We make that up with the following:

```
DPUSH 1+ CONSTANT APUSH  \ to mimic LABEL APUSH
DPUSH 2+ CONSTANT >NEXT  \ to mimic LABEL >NEXT
```

And then we define the macros that we use all the time:

```
                  : 2PUSH
  DPUSH #) JMP ;
                  : 1PUSH
  APUSH #) JMP ;
                  : NEXT
  >NEXT #) JMP ;
```

If we replace by brute force the contents of locations >NEXT and following by a jump to our own custom inner interpreter, we can make a Forth program do all sorts of strange things! This is in fact how the Forth DEBUG utility works.

The actual code for the inner interpreter is in the three lines following the comment (LABEL >NEXT) . It begins by picking up a word at the location pointed to by the Forth IP (instruction pointer) register (i.e., the 8086's SI register).

A Forth COLON word (i.e., one compiled using :) is essentially a list of pointers to other Forth words. The IP normally sits pointing to the word in the list after the one that is currently being executed. By entering >NEXT , we turn our attention to this next word. Its address is copied into AX by an AX LODS instruction. This instruction also increments the IP by 2, so that it is left pointing to the "new" next word in our list.

As a first approximation, Forth replaces a series of machine-language subroutine calls with a list of addresses to branch to, and an inner-interpreter which gets the next address on the list and branches to it. When this routine is done, it returns to the inner interpreter. If this were the whole story, we would simply be saving a little space (the missing CALL instructions) at the expense of a little time (the overhead in performing the CALL simulation).

What is actually done is only slightly more complicated than this. Instead of getting the next address and jumping to it, we jump indirectly via it (AX W MOV

0 [W] JMP). What this means is that instead of a list of addresses to be branched to, we have a list of pointers to Forth words. Each of these Forth words in turn contains the actual address to branch to in order to execute that word, as well as some parameter information of interest to the execution routine. For instance, suppose SCR is a Forth VARIABLE. Then our list might contain a pointer to SCR 's first cell (called the *code field*), which in turn points to the code which is shared by all VARIABLE's. This code uses the contents of the W register, which still points to SCR 's code field which is being executed, to construct a pointer to SCR 's data cell; it is this pointer that is then left on the stack, as the result of calling SCR .

In a CODE word, the code field is followed by executable machine code and points directly to the first instruction of it.

In a COLON word, the code field is followed by a list of data cells containing addresses of other words. Many of these words were produced by another COLON definition, and thus contain in turn list of addresses of other words: the definition was recursive, and the execution must be recursive as well.[1] To achieve this purpose, as soon as activated the COLON interpreter (i.e., the code pointed to by a COLON word's code field) parks in the return stack the currently active value of IP (which was pointing to the next cell in the list being processed by its caller), loads IP with the address of the first cell of its own list, and jumps to >NEXT (the inner interpreter). Eventually we will arrive at a CODE word, which we can execute directly. At the end of each COLON list we will usually find an "end of list" word, called UNNEST , which will pop the return stack into the IP and make us climb up one recursion level.

In conclusion, all that the inner interpreter does is "find the next word and execute it."

C.5 Structured programming

In conventional assemblers, program-flow control is achieved by means of jumps or calls to suitably labeled locations. Forth-style assemblers prefer structured control constructs (such as IF ... ELSE ... THEN). Structuring leads to clearer code and fewer errors; what is more, structured style lends itself quite naturally to the incremental, one-pass compilation approach of Forth.

[1]'Recursive' here means that at each level of the calling hierarchy a new copy of the defining or executing mechanism is created and activated, so that many copies of this mechanism may be active at the same time—each having progressed a certain way through its task. In the Forth word RECURSE and RECURSIVE, explained in the Glossary, the term 'recursive' is used in a more limited sense, i.e., it refers to words that call *themselves* rather than some other word; since Forth temporarily masks out the name of the word that is undergoing definition, if one want to refer to just that word one must "unmask" it.

The control constructs used in the Forth assembler are analogous to those used in Forth itself. However, the machine-language conditional jumps compiled by the assembler will, at run time, test the processor status register rather than the Forth data stack (as is done by Forth `IF`) or some other registers of the Forth virtual machine (as is done by Forth `LOOP`).

For example, the following assembler excerpt

```
\ Wait for ready flag

                  382 CONSTANT PORTC
                          CODE ?OK
        PORTC # DX MOV BEGIN
                  DX AL IN
     80 # AL TEST  0<> UNTIL
                        NEXT
                    END-CODE
```

should produce the same code as (in Intel style)

```
         MOV  DX,#382
  START: IN   AL,DX
         TEST AL,#80
         JZ   START
         JMP  >NEXT
```

where `>NEXT` is a label.

The following conditions can be tested

(C.1)

`<`	less
`>`	greater
`<=`	less/equal
`>=`	greater/equal
`0=`	zero
`0<>`	nonzero
`0<`	negative
`0>=`	nonnegative
`=`	synonym for `0=`
`<>`	synonym for `0<>`

`U<`	unsigned `<`
`U>`	unsigned `>`
`U<=`	unsigned `<=`
`U>=`	unsigned `>=`
`OV`	overflow
`NOV`	no overflow
`CARRY`	carry set
`NCARRY`	carry clear
`EVENL`	low byte is even
`ODDL`	low byte is odd
`CXNZ`	CX register $\neq$ zero

(note that `EVENL` and `ODDL` sense whether there is an even or odd *number of 1s* in the byte, not just whether the byte is evenly divisible by 2).

In a similar way one could have, say,

```
    BEGIN ... U< WHILE ... REPEAT
```

or

```
    OV IF ... ELSE ... THEN
```

These conditionals can be arbitrarily nested.

The construct

```
   234 DO ... LOOP
```

can be used to execute some piece of code 234 times—the index is in the CX register, starting off at 234 and being decremented by 1 each time LOOP is executed. If the count is already in CX, the construct

```
    HERE ... LOOP
```

can be used. Since there is only one CX register, it must be pushed on the stack or otherwise saved in order to perform nested DO loops:

```
            234 DO
              ...
         CX PUSH
           40 DO
           ...
            LOOP
          CX POP
           ...
               LOOP
```

C.6 Simple examples

EXAMPLE 1: *Count the number of* 1 *bits in some range of memory locations.*

Assume that BIT-COUNT-TABLE is a Forth word that returns the starting address of a table 256 bytes long, already initialized to contain the number of 1 s in the binary numbers from 0 to 255 (thus, its entries are 0 1 1 2 ... 6 7 7 8).

The following Forth CODE word makes use of this table to count the number of 1 s in a given area of memory:

```
                         CODE COUNT-BITS
   CX POP  DI POP  ES POP     ( segm offs len -- count)
     DX DX XOR  AH AH XOR
 BIT-COUNT-TABLE # BX MOV
                     HERE
 ES: 0 [DI] AL MOV XLAT
              AX DX ADD
              DI INC LOOP
```

```
                DX PUSH  NEXT
                         END-CODE
```

We begin by putting the length in CX, which will be decremented by LOOP in order to keep track of how many times the loop has been executed (LOOP keeps looping until it decrements CX to zero). We use DI to point to the next byte to be examined (we initialize it to point to the first byte to be examined). We allow the section of memory to be examined to lie in any segment, and use the ES segment register to hold the segment specified. DX will contain our running total of 1 bits, and is initialized to zero. AH is initialized to zero, and will remain zero since the XLAT instruction will only affect AL. BX is initialized to point to the BIT-COUNT-TABLE for the benefit of the XLAT instruction.

The body of the loop consists of four instructions: (a) Pick up the byte pointed to by ES:DI (note the use of a segment override prefix to make the data access relative to the ES segment, rather than the default DS segment); (b) Look up the bit-count using the XLAT instruction (replace AL with the byte from the table pointed to by BX that corresponds to AL's original value); (c) Add the bit-count to the running total being kept in DX; and (d) Make DI point to the next byte to be examined.

When the loop is done, the result contained in DX is pushed onto the stack, and we return to the inner interpreter.

EXAMPLE 2: *Call a BIOS function to send a character to a communications port.*

The fact that the machine stack is the same as the Forth parameter stack makes it very easy to build machine-language words which make BIOS or DOS functions immediately available as part of Forth.

```
   HEX
                 CODE COM-EMIT
     DX POP  AX POP     ( char port# -- status flag)
        1 # AH MOV
            14 INT
               CWD
        AX DX XCHG
             2PUSH
          END-CODE
```

Look at the PC *Technical Reference* manual for details about BIOS functions. The parameter information for the call is arranged in the appropriate registers, and then we call the BIOS function by issuing a soft interrupt. After exiting this function, the high-order bit of AX is set if the routine was unable to transmit the byte of data over the line; the rest of AH contains the status of the port, and AL still contains the character.

The order of the arguments to our word is conventional: source followed by destination. The `CWD` instruction converts a 16-bit word in the `AX` register into a double word by extending its sign bit throughout `DX`—thus `DX` becomes a Forth logical flag. We exchange `AX` and `DX` so that we can use `2PUSH` to put the flag and status information on the stack with the flag on top, which will be the most useful order for subsequent use. Notice that the low byte of our status word still contains the byte we were trying to send, while our `COM-EMIT` word ignores the high byte of its character argument, so that it's very easy to use this word in a loop that retries the operation.

For further examples, look at the source code in the files `KERNEL86.4TH` , `CAM-IO.4TH` , and `CAM-BUF.4TH` . The file `CAM-INT.4TH` contains the source code for a long interrupt routine (the one that interacts with CAM every 60-th of a second to control its operation). This source provides a good example of how to use Forth as a macro assembler.

C.7 Notes and thoughts on the F83 Assembler

1. Some additional constants have been defined to complete the set of condition codes used as branch conditions in the July 1984 version of F83, namely `NOV` (=70), `CARRY` (=73), `NCARRY` (=72), `EVENL` (=7B), `ODDL` (=7A), and `CXNZ` (=E3); moreover, `=` and `<>` have been added as synonyms for `0=` and `0<>` respectively.

2. Reversing the roles of `#)` and `S#)` in `FAR CALL`'s would lead to a more regular mnemonic interpretation of `#)` across the entire instruction set. The current assignment of meaning to these two words has been acknowledged as awkward by the authors of F83, and may be revised in a later version of F83.

3. We have added a defining word `SUBR` which makes self-compiling subroutines.

```
                              : SUBR
          CREATE ASSEMBLER DOES>
                      #) CALL ;
```

   In this way, after the definition

```
                            SUBR GET1
                        1 AL IN
                     2 # AL TEST
                            RET
```

any occurrence of GET1 compiles a subroutine call to the above code whenever it appears in a CODE word.

4. The meaning of the condition-code words in the ASSEMBLER vocabulary used in comparisons could be exchanged between elements of a pair such as > and < , so that one could write the two compared quantities in the traditional Forth order. For example AX BX CMP < IF ⟨action⟩ THEN would execute ⟨action⟩ if AX is less than BX. (The meaning of condition codes after a SUB works fine; the effect of CMPS and SCAS on comparisons changes correspondingly.)

C.8 Instruction templates

Here we give templates for all legal 8086 Forth assembler instructions.

Pairs of lower-case letters are used to stand for appropriate arguments, according to the following table (for example, rx stands for any of the four general-purpose word registers AX, BX, CX, or DX):

rx → AX BX CX DX (general register, restricted choice)
sr → CS DS ES SS (segment register)
rr → AX BX CX DX AH AL BH BL CH CL DH DL (general register)
xx → SI DI BP BX BX+SI BX+DI BP+SI BP+DI (index register)
mm → a one-word address
ss → a one-word segment number
dd → a one-word displacement
ii → a one-word value
bb → a one-byte value

As an example, the instruction SS 20 [BX+SI] MOV follows the template
sr dd [xx] MOV .

With these conventions, we now list all legal instructions:

AAA	ASCII adjust for add
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
rr rr ADC	Add with carry
rr mm #) ADC	
mm #) rr ADC	
rr dd [xx] ADC	
dd [xx] rr ADC	
ii # rr ADC	

`ii # mm #) ADC`	
`ii # dd [xx] ADC`	
`rr rr ADD`	Add
`rr mm #) ADD`	
`mm #) rr ADD`	
`rr dd [xx] ADD`	
`dd [xx] rr ADD`	
`ii # rr ADD`	
`ii # mm #) ADD`	
`ii # dd [xx] ADD`	
`rr rr AND`	Logical AND (clears carry flag, overflow flag)
`rr mm #) AND`	
`mm #) rr ADD`	
`rr dd [xx] AND`	
`dd [xx] rr ADD`	
`ii # rr AND`	
`ii # mm #) AND`	
`ii # dd [xx] AND`	
`mm #) CALL`	Call procedure (pushes return addr)
`dd S#) CALL`	
`rx CALL`	
`dd [xx] CALL`	
`dd [xx] FAR CALL`	
`mm ss #) FAR CALL`	
`mm S#) FAR CALL`	
`CBW`	Convert byte to word (extend sign bit of `AL` thru `AH`)
`CLC`	Clear carry flag
`CLD`	Clear direction flag
`CLI`	Clear interrupt enable flag (turns off interrupts)
`CMC`	Complement carry flag
`rr rr CMP`	Compare dest with source (used with conditional jumps
`rr mm #) CMP`	such as `JG`, and with structured conditionals
`mm #) rr CMP`	such as `IF`, `WHILE`, etc.)
`rr dd [xx] CMP`	Note that the order of arguments is the opposite
`dd [xx] rr CMP`	of what you might expect from Forth:
`ii # rr CMP`	after `AX BX CMP`, a `JG` (jump if greater-than)
`ii # mm #) CMP`	would be executed if `BX` is greater than `AX`.
`ii # dd [xx] CMP`	
`BYTE CMPS`	Compare bytes; `DS:SI`=src, `ES:DI`=dest,
`CMPS`	Compare words; postincrement `SI` and `DI`.
`CWD`	Convert word to double word (extend `AX` sign thru `DX`)
`DAA`	Decimal adjust for addition
`DAS`	Decimal adjust for subtraction
`mm #) DEC`	Decrement by one

dd [xx] DEC	
rr DEC	
SP DEC	
BP DEC	
SI DEC	
DI DEC	
rr DIV	Divide unsigned: if divisor is a byte operand,
BYTE mm #) DIV	then AL=AX/source, AH=remainder. If word op,
mm #) DIV	then AX=DX:AX/source, DX=remainder.
BYTE dd [xx] DIV	
dd [xx] DIV	
rx 0 ESC	Escape for instruction to coprocessor
rx 1 ESC	
rx 2 ESC	
rx 3 ESC	
rx 4 ESC	
rx 5 ESC	
rx 6 ESC	
rx 7 ESC	
mm #) 0 ESC	
mm #) 1 ESC	
mm #) 2 ESC	
mm #) 3 ESC	
mm #) 4 ESC	
mm #) 5 ESC	
mm #) 6 ESC	
mm #) 7 ESC	
dd [xx] 0 ESC	
dd [xx] 1 ESC	
dd [xx] 2 ESC	
dd [xx] 3 ESC	
dd [xx] 4 ESC	
dd [xx] 5 ESC	
dd [xx] 6 ESC	
dd [xx] 7 ESC	
HLT	Halt and wait for interrupt
rr IDIV	Signed divide: see DIV
BYTE mm #) IDIV	
mm #) IDIV	
BYTE dd [xx] IDIV	
dd [xx] IDIV	
rr IMUL	Multiply signed. For byte operand,
BYTE mm #) IMUL	AX=AL×source. For word operand,
mm #) IMUL	DX:AX=AX×source.

BYTE dd [xx] IMUL	
dd [xx] IMUL	
bb AL IN	Input from port
bb AX IN	
DX AL IN	
DX AX IN	
BYTE mm #) INC	Increment by one
mm #) INC	
BYTE dd [xx] INC	
dd [xx] INC	
rr INC	
SP INC	
BP INC	
SI INC	
DI INC	
bb INT	Interrupt
INTO	Interrupt (4) if overflow
IRET	Return from interrupt (restores flags from stk)
dd JA	Jump if above (unsigned)
dd JAE	Jump if above or equal (unsigned)
dd JB	Jump if below (unsigned)
dd JBE	Jump if below or equal (unsigned)
dd JCXZ	Jump if CX register is zero
dd JE	Jump if equal (zero flag = 1)
dd JG	Jump if greater (signed)
dd JGE	Jump if greater or equal (signed)
dd JL	Jump if less (signed)
dd JLE	Jump if less or equal (signed)
mm #) JMP	Jump unconditionally
dd S#) JMP	
rr JMP	
dd [xx] JMP	
dd [xx] FAR JMP	
mm ss #) FAR JMP	
mm S#) FAR JMP	
dd JNE	Jump if not equal (zero flag = 0)
dd JNO	Jump if no overflow (ovrflw flag = 0)
dd JNS	Jump if not sign (sign flag = 0)
dd JO	Jump if overflow (overflow flag = 1)
dd JPE	Jump if parity even (parity flag = 1)
dd JPO	Jump if parity odd (parity flag = 0)
dd JS	Jump if sign (sign flag = 1)
LAHF	Load AH from low byte of flags
mm S#) rx LDS	Load double-word pointer from memory into DS:rx

`mm #) rx LEA`	Load effective address
`dd [xx] rx LEA`	
`mm S#) rx LES`	Load double-word pointer from memory into ES:rx
`BYTE LODS`	Load string byte at DS:SI into AL (postincr.)
`LODS`	Load string word at DS:SI into AX (postincr.)
`dd LOOP`	Decrement CX by 1 and jump if not zero
`dd LOOPE`	Loop while equal (zero flag $= 1$, CX $\neq 0$)
`dd LOOPNE`	Loop while not equal
`rr rr MOV`	Move from source to destination
`rr mm #) MOV`	
`mm #) rr MOV`	
`ii # mm #) MOV`	
`rr dd [xx] MOV`	
`dd [xx] rr MOV`	
`ii # dd [xx] MOV`	
`ii # SP MOV`	
`ii # BP MOV`	
`ii # SI MOV`	
`ii # DI MOV`	
`dd S#) AL MOV`	(byte to AL, address rel to seg start)
`dd S#) AX MOV`	(word to AX, address rel to seg start)
`AL dd S#) MOV`	(AL, address rel to seg start)
`AX dd S#) MOV`	(AX, address rel to seg start)
`rr sr MOV`	
`mm #) sr MOV`	
`dd [xx] sr MOV`	
`sr rr MOV`	
`sr mm #) MOV`	
`sr dd [xx] MOV`	
`BYTE MOVS`	Move string byte; DS:SI=src, ES:DI=dest
`MOVS`	Move string word (postincrement SI and DI)
`rr MUL`	see IMUL
`BYTE mm #) MUL`	
`mm #) MUL`	
`BYTE dd [xx] MUL`	
`dd [xx] MUL`	
`rr NEG`	Negate (subtract from zero)
`BYTE mm #) NEG`	
`mm #) NEG`	
`BYTE dd [xx] NEG`	
`dd [xx] NEG`	
`NOP`	No operation
`rr NOT`	Logical NOT (ones' complement)
`BYTE mm #) NOT`	

`mm #) NOT`	
`BYTE dd [xx] NOT`	
`dd [xx] NOT`	
`rr rr OR`	Logical OR (clears carry, overflow flags)
`rr mm #) OR`	
`mm #) rr OR`	
`rr dd [xx] OR`	
`dd [xx] rr OR`	
`ii # rr OR`	
`ii # mm #) OR`	
`ii # dd [xx] OR`	
`bb AL OUT`	Output to port
`bb AX OUT`	
`DX AL OUT`	
`DX AX OUT`	
`mm #) POP`	Pop word from stack (postincrement SP)
`dd [xx] POP`	
`rr POP`	
`sr POP`	
`SP POP`	
`BP POP`	
`SI POP`	
`DI POP`	
`POPF`	Pop flags from stack
`mm #) PUSH`	Push word onto stack (predecrement SP)
`dd [xx] PUSH`	
`rr PUSH`	
`sr PUSH`	
`SP PUSH`	
`BP PUSH`	
`SI PUSH`	
`DI PUSH`	
`PUSHF`	Push flags onto stack
`rr RCL`	Rotate thru carry left by one position
`mm #) RCL`	
`dd [xx] RCL`	
`rr CL RCL`	Rotate thru carry left by CL positions
`mm #) CL RCL`	
`dd [xx] CL RCL`	
`rr RCR`	Rotate thru carry right by one position
`mm #) RCR`	
`dd [xx] RCR`	
`rr CL RCR`	Rotate thru carry right by CL positions
`mm #) CL RCR`	

dd [xx] CL RCR	
RET	Return from proceedure
FAR RET	Return from FAR proceedure
ii +RET	Return and pop ii items off stack
ii FAR +RET	Return FAR and pop items
rr ROL	Rotate left by 1 position
mm #) ROL	
dd [xx] ROL	
rr CL ROL	Rotate left by CL positions
mm #) CL ROL	
dd [xx] CL ROL	
rr ROR	Rotate right by 1 position
mm #) ROR	
dd [xx] ROR	
rr CL ROR	Rotate right by CL positions
mm #) CL ROR	
dd [xx] CL ROR	
SAHF	Store AH into low byte of flags
rr SAL	Shift arithmetic left by one (zero fill)
mm #) SAL	
dd [xx] SAL	
rr CL SAL	Shift arithmetic left by CL (zero fill)
mm #) CL SAL	
dd [xx] CL SAL	
rr SAR	Shift arithmetic right by one (sign fill)
mm #) SAR	
dd [xx] SAR	
rr CL SAR	Shift arithmetic right by CL (sign fill)
mm #) CL SAR	
dd [xx] CL SAR	
rr rr SBB	(destination minus source)
rr mm #) SBB	
rr dd [xx] SBB	
mm #) rr SBB	
dd [xx] rr SBB	
ii # rr SBB	
ii # mm #) SBB	
ii # dd [xx] SBB	
BYTE SCAS	Scan string byte: CMPs ES:DI with AL (dest)
SCAS	Scan string word. AX acts as CMP dest.
rr SHL	Same as SAL
mm #) SHL	
dd [xx] SHL	
rr CL SHL	

`mm #) CL SHL`	
`dd [xx] CL SHL`	
`rr SHR`	Shift logical right one position (zero fill)
`mm #) SHR`	
`dd [xx] SHR`	
`rr CL SHR`	Shift logical right CL positions (zero fill)
`mm #) CL SHR`	
`dd [xx] CL SHR`	
`STC`	Set carry flag
`STD`	Set direction flag (string ops will decr)
`STI`	Set interrup enable flag (allow interrupts)
`BYTE STOS`	Store AL in string byte at ES:DI (postincr.)
`STOS`	Store AX in string word at ES:DI (postincr.)
`rr rr SUB`	(destination minus source)
`rr mm #) SUB`	
`mm #) rr SUB`	
`rr dd [xx] SUB`	
`dd [xx] rr SUB`	
`ii # rr SUB`	
`ii # mm #) SUB`	
`ii # dd [xx] SUB`	
`rr rr TEST`	(affects only flags)
`mm #) rr TEST`	
`dd [xx] rr TEST`	
`ii # rr TEST`	
`ii # mm #) TEST`	
`ii # dd [xx] TEST`	
`WAIT`	CPU enters WAIT state
`rr rr XCHG`	Exchange contents of dest and source
`mm #) rr XCHG`	
`dd [xx] rr XCHG`	
`SP AX XCHG`	
`BP AX XCHG`	
`SI AX XCHG`	
`DI AX XCHG`	
`XLAT`	Translate: AL+BX points to new value for AL
`rr rr XOR`	Logical XOR
`rr mm #) XOR`	
`mm #) rr XOR`	
`rr dd [xx] XOR`	
`dd [xx] rr XOR`	
`ii # rr XOR`	
`ii # mm #) XOR`	
`ii # dd [xx] XOR`	

C.8.1 Prefixes

Prefixes are instructions that modify the action of the following instruction. They comprise the lock prefix,

LOCK	Lock bus during execution of next instruction

the repeat prefixes, and the segment-overrride prefixes.

The repeat prefixes are used with LODS, MOVS, STOS, CMPS, and SCAS. After each repetition, the CX register is decremented: execution continues until either CX reaches zero, or the repeat test (equal or not equal) fails. Source and destination pointers (SI and DI) end up pointing one position past the end of string(s), or else one position past the first point of match/mismatch.

REP	Repeat next instruction
REPZ	Repeat while zero flag on (equal)
REPNZ	Repeat while zero flag off (unequal)

The segment-override prefixes are used to override the 8086 default assumptions concerning which segment register to use with the next instruction. The normal default is the DS segment, and this can always be overridden. Memory references using the BP pointer take the stack segment (SS) as their default, but this can also be overridden. The only defaults which can't be overridden are: string destinations (ES), stack operations (SS), and instruction fetches (CS).

CS:	Use code segment register with next instruction
DS:	Use data segment register with next instruction
ES:	Use extra segment register with next instruction
SS:	Use stack segment register with next instruction

C.9 Interface with DEBUG

One of the programs which is supplied as part of DOS is an 8088/86 assembly language programming aid called DEBUG (this should not be confused with the word-debugging utility of F83—also called DEBUG). This program provides facilities for tracing and disassembling machine language programs, which may be used in conjunction with Forth if you are having difficulty getting a machine language routine to work within F83 or CAM . The DEBUG program is extensively documented in the DOS manual, but we will provide here a brief illustration of how to use it with Forth.

To invoke DEBUG for use with a version of the Forth system, say, CAM.EXE , you would simply type from DOS

```
C>debug cam.exe
```

You will find yourself at the `DEBUG` command level, with a `-` ("dash") as a prompt. (In the following examples, this dash has been indicated for clarity, but should not be typed by you.)

The most useful commands will be `G` (Go), `D` (Dump), `R` (show/modify Registers), `T` (Trace), and `U` (Unassemble).

To begin executing the Forth program, simply type

```
-G
```

This will cause execution to begin at location `0100` (hex) within the segment containing the `CAM.EXE` program, which is the normal point where a `EXE` file begins execution. Execution will proceed normally (with about 20K of the PC's RAM taken up by `DEBUG`) until you type `G` at the Forth level (or execute `G` as part of another Forth word). At this point you will find yourself back in `DEBUG` .

C.9.1 Unassembling code

To unassemble code starting at location `0567` (hex) type

```
-U 567
```

This will unassemble a screenful of code; to continue to unassemble following lines, just type

```
-U
```

for another screenful.

To dump values from memory starting at location `07A2`, type

```
-D 7A2
```

You can return to Forth by typing

```
-G
```

This will cause you to continue where you left off. Thus, you can unassemble code with `DEBUG` almost as if this were a Forth utility.

C.9.2 Tracing

If you want to trace program execution, you should set a breakpoint while in `DEBUG` . If you want to start tracing at location `7A9C`, restart Forth using

```
-G 7A9C
```

When the execution of Forth reaches this location you will exit to `DEBUG` . The status of all registers and the next instruction to be executed will be displayed. All the `DEBUG` commands are available; in particular, typing

```
-T
```

will cause the next instruction to be executed in *trace mode*. This means that we will return to DEBUG after the instruction has been completed, and the status of all registers and the instruction to be executed next will be shown. You can single-step through the execution with more T 's, or you can type, say,

```
-T 20
```

to trace 20 (hex) instructions. At any time you can continue with Forth, by typing

```
-G
```

or continue executing until another breakpoint occurs (at location 82D7, say) by typing

```
-G 82D7
```

You can end a trace by cold-starting Forth with

```
-G =100
```

(execution resumes at location 0100) or with

```
-G =100 7AD3
```

(cold-start, and leave a breakpoint at 7AD3).

Before cold-starting Forth, the code-segment register must have the correct value. If you are tracing a DOS interrupt or some other routine located outside of Forth, you should restore the CS register using the R command to its normal value. The command would be

```
-R CS
```

The program will respond with its current value, and give you an opportunity to enter a new value, e.g.

```
CS 917
:
```

You can respond after the : prompt with a new value, or press RETURN to leave the register unchanged.

C.9.3 Leaving DEBUG

Forth normally takes over interrupt vectors that it uses, saving their initial values during cold-start, and restoring them when you execute BYE . Thus you should normally exit debug by typing BYE from Forth and then Q (Quit) from DEBUG .

If you have cold-started Forth from DEBUG more than once since the beginning of the session, incorrect information may be saved about the original values of these vectors. In this case you should exit by rebooting the PC.

Appendix D

Software interface with CAM

The CAM software, as currently realized, consists of eight source files that are loaded on top of the Forth system; the latter is a modified version of F83 (see Introduction and Section 4.7).

The CAM part of the system serves not only as a shared language for use by the CAM community, but also—for those who might want to implement their own CAM driving software in some other language—as an example of how to make CAM perform various functions.

Details concerning data formats used by this software (which should be useful to those wishing to write programs (in Forth or other languages) to perform data analysis, pattern generation, etc.) are provided in the next section. In this section, we provide a brief overview of the structure and organization of the CAM-specific part of the software.

The lowest level of the software is contained in the file CAM-INT.4TH . This contains the interrupt driver for CAM that normally gets control during every CAM vertical blanking interval; interrupts that invoke this driver are generated by the master CAM (CAM number 0) every sixtieth of a second. The principal function of this routine is to maintain the shadow registers that constitute the primary real-time interface between application programs and the CAM hardware. For each installed CAM, the interrupt driver copies control-register data and color-map data to the machine, and retrieves any accumulated counts from the event counters before scheduling a new step (if one is pending). Since the high-level word STEP waits for this copying to occur before proceeding, any settings of neighborhoods, color maps, phase bits, etc. that are made before STEP is executed will have actually been sent to CAM before the application can try to change them further. The actual interlock between the interrupt routines and the mainline routines is handled by the intermediate level word PEND , which sets a flag that is cleared by the interrupt routine when the copying has actually taken place, and the word WAIT-FOR-PEND , which waits for this flag to be cleared. The interrupt routine can also read or write a limited amount of plane data during the vertical blanking time (see R/FLY and W/FLY in the glossary); it is this facility that is used to draw the dot and cage cursors used by the control-panel program. The file CAM-INT.4TH contains shadow screens and is carefully annotated; it provides an example of how well suited Forth is to being used as a macro-assembler language.

The rest of the CAM source files are documented principally by the glossary, some sections of this manual, and the *CAM Book*. After the interrupt level routines are loaded, the next-lowest level of routines (contained in the file CAM-IO.4TH) is added. These input/output routines are used to interface with the shadow variables that are maintained by the interrupt, and also to do plane and table I/O, which involves inhibiting the interrupt routine and taking direct control over CAM hardware registers. The software is designed to handle up to eight CAMs simultaneously; at the interrupt level all CAMs are always treated on an equal footing, except that step-requests are sent only to the master CAM. At the I/O level, we adopt the strategy of only communicating with one machine at a time (see CAM-SELECT), avoiding the need to have all I/O routines take a CAM number as an argument. The interlock words PEND and WAIT-FOR-PEND are defined in this file.

The next level of the software, contained in the file CAM-BUF.4TH , defines words for performing memory management outside of the 64K segment containing the Forth system: allocating and de-allocating buffers. Most of the extra buffer areas used by the remainder of the routines are defined here, as well as operations on and between buffers addressed by segment and offset (including random number generation) and operations between buffers and planes or tables.

At the next level (CAM-HOOD.4TH) we worry about table generation. We define the word == , and use it to define all of the compilation variables used in table generation; we define >> and use it to define all of the words used to modify table entries during table generation; we define all of the major and minor neighborhood selectors; and we define MAKE-TABLE . Similarly, we define the words used in color-map generation.

With this background, we are ready to worry about making CAM run steps (CAM-STEP.4TH). We define a stepping coroutine called NEXT-STEP , and the words STEP and IDLE , which cause a coroutine exit to the main program when used within the Forth word attached to this coroutine. NEXT-STEP resumes the coroutine where you last left off, beginning by causing the active step (or idle step) where you exited to actually start. MAKE-CYCLE is used for attaching a Forth word to the stepping coroutine. In this file, we also make provision for service steps (see BEGIN-SERVICE-STEPS) such as shifts of bit-planes, which can be interposed between two steps of an experiment: CAM stepping control and status information (as well as tables and color maps) is restored (by END-SERVICE-STEPS) when done. Provision for showing an expanded version of the central portion of the screen using CAM itself as the display device while running is also made here.

The penultimate file is `CAM-EDIT.4TH`. It contains all of the routines used for the limited graphic editing facilities provided. Cursors are drawn by using the facility for reading and writing plane data during the vertical blanking time without turning off CAM's display (as is required by the normal plane I/O routines). This file also contains an interrupt driver for a MOUSE SYSTEMS serial mouse, which can be used instead of the arrow keys in the control panel program to move the dot-editing cursor.

Finally, the file `CAM-KEYS.4TH` defines a key-interpreter loop that constitutes the control-panel program. When a key is pressed, a set of vocabularies (`GENERAL` , `PLANE-OPS` , etc.) is searched for that key name. If it appears as a Forth word in one of these vocabularies, then this word is executed. Key names are put in these vocabularies by the word `ALIAS` , which makes the key name be essentially a synonym for another Forth word. When the key name is executed, the name of the associated Forth word is printed, and then this word is executed. For each of the `ALIAS` vocabularies, a listing of key names and the names of associated Forth words is made available as a sort of menu. The order of items in these menus reflects the order in which the keys were added to the dictionary. User defined `ALIAS`es are put in the `ALTERNATE` vocabulary, and (to avoid confusion with system-defined keys) are always attached to `Alt-` keys.

All of these files are loaded by the master file `CAM-LOAD.4TH` , which also defines how CAM is initialized when it first starts up, and what clean-up activities are done when you end the CAM program.

Appendix E

Data formats

There are three main kinds of files supported directly by this software for storing and retrieving information relevant to CAM experiments. If you wish to manipulate these files from other languages (for example, to perform data analysis or pattern generation using a more familiar or appropriate language than Forth) this information on data formats should be helpful.

E.1 Plane pattern files

Plane pattern files have a default extension of `PAT` , and are always a multiple of 8K bytes. These files contain no header information: the number of bit-planes that are stored as a pattern (1, 2, 3, or 4) is determined entirely by the file size (8192, 16384, 24576, or 32768 bytes).

All data corresponding to a single bit-plane is stored contiguously: the first byte of each 8K record corresponds to the eight bits on the left end of the top row of CAM's screen. Subsequent bytes within this record are mapped in a left-to-right, top-to-bottom order (the first 32 bytes correspond to the top row, the next 32 to the next row, etc.). Note that within each byte, the least significant bit of the byte appears as the leftmost on the screen, with the more significant bits appearing at consecutive locations *to its right*. This may be the opposite of what you expect from ordinary arithmetic notation.

See `IMAGE>FILE` in the glossary for a discussion of how 8K records are mapped onto planes.

E.2 Table files

The lookup tables used by CAM may be saved to or restored from disk (see `FILE>TAB` and `TAB>FILE` in the glossary). The default extension for table files is TAB . Each lookup table is exactly 4096 bytes long and contains only table data: no header data is included. Neighborhood and run-cycle information must be provided separately. Each byte in a table file (default extension is .TAB) contains information for 8 subtables (columns). Bit positions 0, 2, 4, and 6 refer to the regular tables for planes 0, 1, 2, and 3 respectively; positions 1, 3, 5, and 7 refer to the auxiliary tables for planes 0, 1, 2, and 3 respectively. Each entry (byte) corresponds to a particular set of input values (see Table 9.1). For example, entry 000 refers to the case when all neighbors return zeros; entry FFF (hex) refers to the case when all neighbors return ones.

E.3 Data files

The default extension for data files is DAT . 16-bit values stored in or retrieved from such files using PUT-DATA and GET-DATA appear in consecutive locations, each value consisting of two bytes (the low-order byte is stored in a position closer to the beginning of the file than the high-order byte). Data files in this implementation are always padded to be a multiple of 1K (1024) bytes in length—you may want to start your file with a count of data items if this is a variable quantity.

Appendix F

Forth-83 Standard

We insert here some verbatim excerpts from the *Forth-83 Standard*, retaining the numbering of the original sections. The material we include is intended to help make clear what parts of this system are standard Forth, and what parts are extensions.

Forth standardization efforts began in the mid-1970's by the European Forth Users Group (EFUG); this effort resulted in the Forth-77 Standard. As the language continued to evolve, an interim Forth-78 Standard was published by the Forth Standards Team. The Forth-79 Standard was published in 1980, and the Forth-83 Standard in 1983.

FORTH STANDARDS TEAM
P.O. BOX 4545
MOUNTAIN VIEW, CA 94040
USA

ISBN 0-914699-03-2

F.5 Definition of Terms

These are the definitions of the terms used within this Standard.

address, byte
    An unsigned 16-bit number that locates an 8-bit byte in a standard FORTH address space over the range {0..65,535}. It may be a native machine address or a representation on a virtual machine, locating the addr-th byte within the virtual byte address space. Addresses are treated as unsigned numbers. See: "arithmetic, two's complement"

address, compilation
    The numerical value compiled for a FORTH word definition which identifies that definition. The address interpreter uses this value to locate the machine code corresponding to each definition.

address, native machine
    The natural address representation of the host computer.

address, parameter field
    The address of the first byte of memory associated with a word definition for the storage of compilation addresses (in a colon definition), numeric data, text characters, etc.

arithmetic, two's complement
    Arithmetic is performed using two's complement integers within a field of either 16 or 32 bits as indicated by the operation. Addition and subtraction of two's complement integers ignore any overflow condition. This allows numbers treated as unsigned to produce the same results as if the numbers had been treated as signed.

block
    The 1024 bytes of data from mass storage which are referenced by block numbers in the range {0..the number of blocks available -1}. The actual amount of data transferred and the translation from block number to device and physical record is a function of the implementation. See: "block buffer" "mass storage"

block buffer
    A 1024-byte memory area where a block is made temporarily available for use. Block buffers are uniquely assigned to blocks. See: "9.7 Multiprogramming Impact"

byte
    An assembly of 8 bits. In reference to memory, it is the storage capacity for 8 bits.

character

A 7-bit number the signification of which is given by the ASCII standard. When contained in a larger field, the higher order bits are zero. See: "6. REFERENCES"

compilation

The action of converting text words from the input stream into an internal form suitable for later execution. When in the compile state, the compilation addresses of FORTH words are compiled into the dictionary for later execution by the address interpreter. Numbers are compiled to be placed on the data stack when later executed. Numbers are accepted from the input stream unsigned or negatively signed and converted using the value of BASE. See: "number" "number conversion" "interpreter, text"

defining word

A word that, when executed, creates a new dictionary entry in the compilation vocabulary. The new word name is taken from the input stream. If the input stream is exhausted before the new name is available, an error condition exists. Examples of defining words are: : CONSTANT CREATE

definition

See: "word definition"

dictionary

A structure of word definitions in computer memory which is extensible and grows toward higher memory addresses. Entries are organized in vocabularies to aid location by name. See: "search order"

display

The process of sending one or more characters to the current output device. These characters are typically displayed or printed on a terminal. The selection of the current output device is system dependent.

division, floored

Integer division in which the remainder carries the sign of the divisor or is zero, and the quotient is rounded to its arithmetic floor. Note that, except for error conditions, n1 n2 SWAP OVER /MOD ROT * + is identical to n1. See: "floor, arithmetic"

Examples:

dividend	divisor	remainder	quotient
10	7	3	1
-10	7	4	-2
10	-7	-4	-2
-10	-7	-3	1

equivalent execution

A standard program will produce the same results, exclusive of timing dependencies, when given the same inputs on any Standard System which has sufficient resources to execute the program. Only standard source programs are transportable.

error condition
An exceptional condition which requires action by the system which may be other than the expected function. Refer to the section "10. Error Conditions".

false
A zero number represents the false state of a flag.

flag
A number that may have one of two logical states, false or true. See: "false" "true"

floor, arithmetic
If z is any real number, then the floor of z is the greatest integer less than or equal to z.

```
        The floor of +.6 is 0
        The floor of -.4 is -1
```

free field format
Numbers are converted using the value of BASE and then displayed with no leading zeros. A trailing space is displayed. The number of characters displayed is the minimum number of characters, at least one, to uniquely represent the number. See "number conversion"

glossary
A set of explanations in natural language to describe the corresponding computer execution of word definitions.

immediate word
A word which executes when encountered during compilation or interpretation. Immediate words handle special cases during compilation. See, for example, IF LITERAL ." etc.

input stream
A sequence of characters available to the system, for processing by the text interpreter. The input stream conventionally may be taken from the current input device (via the text input buffer) and mass storage (via a block buffer). BLK , >IN , TIB and #TIB specify the input stream. Words using or altering BLK , >IN , TIB and #TIB are responsible for maintaining and restoring control of the input stream.

The input stream extends from the offset value of >IN to the size of the input stream. If BLK is zero the input stream is contained within the area addressed by TIB and is #TIB bytes long. If BLK is non-zero the input stream is contained within the block buffer specified by BLK and is 1024 bytes long. See: "11.8 Input Text"

interpreter, address
The machine code instructions, routine or other facilities that execute compiled word definitions containing compilation addresses.

interpreter, text
The word definition(s) that repeatedly accepts a word name from the input stream, locates the corresponding compilation address and starts the address interpreter to execute it. Text from the input stream interpreted as a number leaves the corresponding value on the data stack. Numbers are accepted from the input stream unsigned or negatively signed and converted using the value of BASE . See: "number" "number conversion"

layers
The grouping of word names of each Standard word set to show like characteristics. No implementation requirements are implied by this grouping.

layer, compiler
Word definitions which add new procedures to the dictionary or which aid compilation by adding compilation addresses or data structures to the dictionary.

layer, devices
Word definitions which allow access to mass storage and computer peripheral devices.

layer, interpreter
Word definitions which support vocabularies, terminal output, and the interpretation of text from the text input buffer or a mass storage device by executing the corresponding word definitions.

layer, nucleus
Word definitions generally defined in machine code that control the execution of the fundamental operations of a virtual FORTH machine. This includes the address interpreter.

load
Redirection of the text interpreter's input stream to be from mass storage. This is the general method for compilation of new definitions into the dictionary.

mass storage
Storage which might reside outside FORTH's address space. Mass storage data is made available in the form of 1024-byte blocks. A block is accessible within the FORTH address space in a block buffer. When a block has been indicated as UPDATEed (modified) the block will ultimately be transferred to mass storage.

number

When values exist within a larger field, the most-significant bits are zero. 16-bit numbers are represented in memory by addressing the first of two bytes at consecutive addresses. The byte order is unspecified by this Standard. Double numbers are represented on the stack with the most-significant 16 bits (with sign) most accessible. Double numbers are represented in memory by two consecutive 16-bit numbers. The address of the least-significant 16-bits is two greater than the address of the most-significant 16 bits. The byte order within each 16-bit field is unspecified. See: "arithmetic, two's complement" "number types" "9.8 Numbers" "11.7 Stack Parameters"

number conversion

Numbers are maintained internally in binary and represented externally by using graphic characters within the ASCII character set. Conversion between the internal and external forms is performed using the current value of BASE to determine the digits of a number. A digit has a value ranging from zero to the value of BASE-1. The digit with the value zero is represented by the ASCII character "0" (position 3/0 with the decimal equivalent of 48). This representation of digits proceeds through the ASCII character set to the character "9" corresponding to the decimal value 9. For digits with a value exceeding 9, the ASCII graphic characters beginning with the character "A" (position 4/1 with the decimal equivalent 65) corresponding to the decimal value 10 are used. This sequence then continues up to and including the digit with the decimal value 71 which is represented by the ASCII character "~" (position 7/14 with a decimal equivalent 126). A negative number may be represented by preceding the digits with a single leading minus sign, the character "-".

number types

All number types consist of some number of bits.` These bits are either arbitrary of are weighted.

Signed and unsigned numbers use weighted bits. Weighted bits within a number have a value of a power of two beginning with the rightmost (least-significant) bit having the value of two to the zero power. This weighting continues to the leftmost bit increasing the power by one for each bit. For an unsigned number this weighting pattern includes the leftmost bit; thus, for an unsigned 16-bit number the weight of the leftmost bit is 32,768. For a signed number this weighting pattern includes the leftmost bit, but the weight of the leftmost bit is negated; thus, for a signed 16-bit number the weight of the leftmost bit is -32,768. This weighting pattern for signed numbers is called two's complement notation.

Unspecified weighted numbers are either unsigned numbers or signed numbers; program context determines whether the number is signed or unsigned. See: "11.7 Stack Parameters"

pictured numeric output
    The use of numeric output definitions which convert numerical values into text strings. These definitions are used in a sequence which resembles a symbolic 'picture' of the desired text format. Conversion proceeds from least-significant digit to most-significant digit, and converted characters are stored from higher memory addresses to lower.

program
    A complete specification of execution to achieve a specific function (application task) expressed in FORTH source code form.

receive
    The process of obtaining one character from the current input device. The selection of the current input device is system dependent.

recursion
    The process of self-reference, either directly or indirectly.

return
    The means of indicating the end of text by striking a key on an input device. The key used is system dependent. This key is typically called "RETURN", "CARRIAGE RETURN", or "ENTER".

screen
    Textual data arranged for editing. By convention, a screen consists of 16 lines (numbered 0 through 15) of 64 characters each. Screens usually contain program source text, but may be used to view mass storage data. The first byte of a screen occupies the first byte of a mass storage block, which is the beginning point for text interpretation during a load.

search order
    A specification of the order in which selected vocabularies in the dictionary are searched. Execution of a vocabulary makes it the first vocabulary in the search order. The dictionary is searched whenever a word is to be located by its name. This order applies to all dictionary searches unless otherwise noted. The search order begins with the last vocabulary executed and ends with FORTH , unless altered in a system dependent manner.

source definition
    Text consisting of word names suitable for compilation or execution by the text interpreter. Such text is usually arranged in screens and maintained on a mass storage device.

stack, data
    A last in, first out list consisting of 16-bit binary values. This stack is primarily used to hold intermediate values during execution of word

definitions. Stack values may represent numbers, characters, addresses, boolean values, etc.

When the name 'stack' is used alone, it implies the data stack.

stack, return
A last in, first out list which contains the addresses of word definitions whose execution has not been completed by the address interpreter. As a word definition passes control to another definition, the return point is placed on the return stack.

The return stack may cautiously be used for other values.

string, counted
A sequence of consecutive 8-bit bytes located in memory by their low memory address. The byte at this address contains a count {0..255} of the number of bytes following which are part of the string. The count does not include the count byte itself. Counted strings usually contain ASCII characters.

string, text
A sequence of consecutive 8-bit bytes located in memory by their low memory address and length in bytes. Strings usually, but not exclusively, contain ASCII characters. When the term 'string' is used alone or in conjunction with other words it refers to text strings.

structure, control
A group of FORTH words which when executed alter the execution sequence. The group starts and terminates with compiler words. Examples of control structures: DO ... LOOP   DO ... +LOOP   BEGIN ... WHILE ... REPEAT   BEGIN ... UNTIL   IF ... THEN   IF ... ELSE ... THEN   See: "9.9 Control Structures"

transportability
This term indicates that equivalent execution results when a program is executed on other than the system on which it was created.
See: "equivalent execution"

true
A non-zero value represents the true state of a flag. Any non-zero value will be accepted by a standard word as 'true'; all standard words return a 16-bit value with all bits set to one when returning a 'true' flag.

user area
An area in memory which contains the storage for user variables.

variable, user
A variable whose data storage area is usually located in the user area. Some system variables are maintained in the user area so that the words may be re-entrant to different users.

vocabulary
An ordered list of word definitions. Vocabularies are an advantage in separating different word definitions that may have the same name. More than one definition with the same name can exist in one vocabulary. The latter is called a redefinition. The most recently created redefinition will be found when the vocabulary is searched.

vocabulary, compilation
The vocabulary into which new word definitions are appended.

word
A sequence of characters terminated by one blank or the end of the input stream. Leading blanks are ignored. Words are usually obtained via the input stream.

word definition
A named FORTH execution procedure compiled into the dictionary. Its execution may be defined in terms of machine code, as a sequence of compilation addresses, or other compiled words.

word name
The name of a word definition. Word names are limited to 31 characters and may not contain an ASCII space. If two definitions have different word names in the same vocabulary they must be uniquely findable when this vocabulary is searched. See: "vocabulary" "9.5.3 EXPECT"

word set
A named group of FORTH word definitions in the Standard.

word set, assembler extension
Additional words which facilitate programming in the native machine language of the computer which are by nature system dependent.

word set, double number extension
Additional words which facilitate manipulation of 32-bit numbers.

word set, required
The minimum words needed to compile and execute Standard Programs.

word set, system extension
Additional words which facilitate the access to internal system characteristics.

word, standard
A named FORTH procedure definition, in the Required word set or any extension word sets, formally reviewed and accepted by the Standards Team.

F.12 Required Word Set

F.12.1 The Required Word Set Layers

The words of the Required Word Set are grouped to show like characteristics. No implementation requirements should be inferred from this grouping.

Nucleus layer

```
! * */ */MOD + +! - / /MOD 0< 0= 0> 1+ 1-
2+ 2- 2/ < = > >R ?DUP @ ABS AND C! C@ CMOVE
CMOVE> COUNT D+ D< DEPTH DNEGATE DROP DUP EXECUTE EXIT
FILL I J MAX MIN MOD NEGATE NOT OR OVER PICK R>
R@ ROLL ROT SWAP U< UM* UM/MOD XOR
```

Device layer

```
BLOCK BUFFER CR EMIT EXPECT FLUSH KEY SAVE-BUFFERS
SPACE SPACES TYPE UPDATE
```

Interpreter layer

```
# #> #S #TIB ' ( -TRAILING . .( <# >BODY >IN
ABORT BASE BLK CONVERT DECIMAL DEFINITIONS FIND FORGET
FORTH FORTH-83 HERE HOLD LOAD PAD QUIT SIGN SPAN TIB
U. WORD
```

Compiler layer

```
+LOOP , ." : ; ABORT" ALLOT BEGIN COMPILE CONSTANT
CREATE DO DOES> ELSE IF IMMEDIATE LEAVE LITERAL LOOP
REPEAT STATE THEN UNTIL VARIABLE VOCABULARY WHILE [ [']
[COMPILE] ]
```

F.12.2 The Required Word Set Glossary

```
!               16b addr --                    79                "store"
```
16b is stored at addr.

```
#               +d1 -- +d2                     79                "sharp"
```
The remainder of +d1 divided by the value of BASE is converted to an ASCII character and appended to the output string toward lower memory addresses. +d2 is the quotient and is maintained for further processing. Typically used between <# and #> .

```
#>              32b -- addr +n                 79        "sharp-greater"
```
Pictured numeric output conversion is ended dropping 32b. addr is the

```
    address of the resulting output string.  +n is the number of characters
    in the output string.  addr and +n together are suitable for TYPE .

#S              +d -- 0 0                         79               "sharp-s"
    +d is converted appending each resultant character into the pictured
    numeric output string until the quotient (see: # ) is zero.  A single
    zero is added to the output string if the number was initially zero.
    Typically used between <# and #> .

#TIB             -- addr                          U,83        "number-t-i-b"
    The address of a variable containing the number of bytes in the text
    input buffer.  #TIB is accessed by WORD when BLK is zero.  {{0..capacity
    of TIB}}  See:  "input stream"

'                -- addr                          M,83                "tick"
    Used in the form:
            ' <name>
    addr is the compilation address <name>.  An error condition exists
    if <name> is not found in the currently active search order.

(                --                               I,M,83             "paren"
                 --    (compiling)
    Used in the form:
            ( <ccc>)
    The characters <ccc>, delimited by ) (closing parenthesis), are
    considered comments.  Comments are not otherwise processed.  The blank
    following ( is not part of <ccc>.  ( may be freely used while
    interpreting or compiling.  The number of characters in <ccc> may be from
    zero to the number of characters remaining in the input stream up to the
    closing parenthesis.

*               w1 w2 -- w3                       79                "times"
    w3 is the least-significant 16 bits of the arithmetic product of w1 times
    w2.

*/              n1 n2 n3 -- n4                    83          "times-divide"
    n1 is first multiplied by n2 producing an intermediate 32-bit result.  n4
    is the floor of the quotient of the intermediate 32-bit result divided by
    the divisor n3.  The product of n1 times n2 is maintained as an
    intermediate 32-bit result for greater precision than the otherwise
    equivalent sequence:  n1 n2 * n3 / .  An error condition results if the
    divisor is zero or if the quotient falls outside of the range
    {-32,768..32,767}.  See: "division, floored"

*/MOD           n1 n2 n3 -- n4 n5                 83      "times-divide-mod"
    n1 is first multiplied by n2 producing an intermediate 32-bit result.  n4
    is the remainder and n5 is the floor of the quotient of the intermediate
    32-bit result divided by the divisor n3.  A 32-bit intermediate product
    is used as for */ .  n4 has the same sign as n3 or is zero.  An error
```

```
     condition results if the divisor is zero of if the quotient falls outside
     of the range {-32,768..32,767}.  See: "division, floored"

+              w1 w2 -- w3                      79                "plus"
     w3 is the arithmetic sum of w1 plus w2.

+!             w1 addr --                       79          "plus-store"
     w1 is added to the w value at addr using the convention for + .  This sum
     replaces the original value at addr.

+LOOP          n --                             C,I,83       "plus-loop"
               sys --    (compiling)
     n is added to the loop index.  If the new index was incremented across
     the boundary between limit-1 and limit then the loop is terminated and
     loop control parameters are discarded.  When the loop is not terminated,
     execution continues to just after the corresponding DO .  sys is balanced
     with its corresponding DO .  See: DO

,              16b --                           79               "comma"
     ALLOT space for 16b then store 16b at HERE 2- .

-              w1 w2 -- w3                      79               "minus"
     w3 is the result of subtracting w2 from w1.

-TRAILING      addr +n1 -- addr +n2             79       "dash-trailing"
     The character count +n1 of a text string beginning at addr is adjusted to
     exclude trailing spaces.  If +n1 is zero, then +n2 is also zero.  If the
     entire string consists of spaces, then +n2 is zero.

.              n --                 M,          79                 "dot"
     The absolute value of n is displayed in a free field format with a
     leading minus sign if n is negative.

."             --                               C,I,83       "dot-quote"
               --    (compiling)
     Used in the form:
             ." <ccc>"
     Later execution will display the characters <ccc> up to but not including
     the delimiting " (close-quote).  The blank following ." is not part of
     <ccc>.

.(             --                               I,M,83       "dot-paren"
               --    (compiling)
     Used in the form:
             .( <ccc>)
     The characters <ccc> up to but not including the delimiting ) (closing
     parenthesis) are displayed.  The blank following .( is not part of <ccc>.

/              n1 n2 -- n3                      83              "divide"
     n3 is the floor of the quotient of n1 divided by the divisor n2.  An
     error condition results if the divisor is zero or if the quotient falls
     outside of the range {-32,768..32,767}.  See: "division, floored"

/MOD           n1 n2 -- n3 n4                  83             "divide-mod"
     n3 is the remainder and n4 the floor of the quotient of n1 divided by the
     divisor n2.  n3 has the same sign as n2 or is zero.  An error condition
     results if the divisor is zero or if the quotient falls outside of the
     range {-32,768..32,767}.  See: "division, floored"

0<             n -- flag                       83              "zero-less"
     flag is true if n is less than zero (negative).

0=             w -- flag                       83            "zero-equals"
     flag is true if w is zero.

0>             n -- flag                       83           "zero-greater"
     flag is true if n is greater than zero.

1+             w1 -- w2                        79               "one-plus"
     w2 is the result of adding one to w1 according to the operation of + .

1-             w1 -- w2                        79              "one-minus"
     w2 is the result of subtracting one from w1 according to the operation of
     - .

2+             w1 -- w2                        79               "two-plus"
     w2 is the result of adding two to w1 according to the operation of + .

2-             w1 -- w2                        79              "two-minus"
     w2 is the result of subtracting two from w1 according to the operation of
     - .

2/             n1 --n2                         83             "two-divide"
     n2 is the result of arithmetically shifting n1 right one bit.  The sign
     is included in the shift and remains unchanged.

:              -- sys                          M,79                "colon"
     A defining word executed in the form:
             : <name> ... ;
     Create a word definition for <name> in the compilation vocabulary and
     set compilation state.  The search order is changed so that the first
     vocabulary in the search order is replaced by the compilation vocabulary.
     The compilation vocabulary is unchanged.  The text from the input stream
     is subsequently compiled.  <name> is called a "colon definition".  The
     newly created word definition for <name> cannot be found in the
     dictionary until the corresponding ; or ;CODE is successfully processed.

     An error condition exists if a word is not found and cannot be converted
     to a number or if, during compilation from mass storage, the input stream
     is exhausted before encountering ; or ;CODE .  sys is balanced with its
     corresponding ; .  See:  "compilation" "9.4 Compilation"

;               --                          C,I,79      "semi-colon"
                sys --   (compiling)
     Stops compilation of a colon definition, allows the <name> of this
     colon definition to be found in the dictionary, sets interpret state and
     compiles EXIT (or a system dependent word which performs an equivalent
     function).  sys is balanced with its corresponding : .  See: EXIT  :
     "stack, return" "9.4 Compilation"

<               n1 n2 -- flag               83             "less-than"
     flag is true if n1 is less than n2.
             -32768 32767 < must return true.
             -32768 0 < must return true.

<#              --                          79            "less-sharp"
     Initialize pictured numeric output conversion.  The words:
             #  #>  #S  <#  HOLD  SIGN
     can be used to specify the conversion of a double number into an ASCII
     text string stored in right-to-left order.

=               w1 w2 -- flag               83                "equals"
     flag is true if w1 is equal to w2.

>               n1 n2 -- flag               83          "greater-than"
     flag is true if n1 is greater than n2.
             -32768 32767 > must return false
             -32768 0 > must return false

>BODY           addr -- addr2               83               "to-body"
     addr2 is the parameter field address corresponding to the compilation
     address addr1.  See: "9.2 Addressable Memory"

>IN             -- addr                     U,79               "to-in"
     The address of a variable which contains the present character offset
     within the input stream {{0..the number of characters in the input
     stream}}.  See: WORD

>R              16b --                      C,79                "to-r"
     Transfers 16b to the return stack.  See:  "9.3 Return Stack"

?DUP            16b -- 16b 16b              79        "question-dupe"
            or  0 -- 0
     Duplicate 16b if it is non-zero.

@               addr -- 16b                 79                 "fetch"
     16b is the value at addr.

ABORT                                       79
```

Clears the data stack and performs the function of QUIT . No message is displayed.

```
ABORT"          flag --                     C,I,83      "abort-quote"
                --    (compiling)
```

Used in the form:

```
        flag ABORT" <ccc>"
```

When later executed, if flag is true the characters <ccc>, delimited by " (close-quote), are displayed and then a system dependent error abort sequence, including the function of ABORT , is performed. If flag is false, the flag is dropped and execution continues. The blank following ABORT" is not part of <ccc>.

```
ABS             n -- u                      79             "absolute"
```

u is the absolute value of n. If n is -32,768 then u is the same value. See: "arithmetic, two's complement"

```
ALLOT           w --                        79
```

Allocates w bytes in the dictionary. The address of the next available dictionary location is updated accordingly.

```
AND             16b1 16b2 -- 16b3           79
```

16b3 is the bit-by-bit logical 'and' of 16b1 with 16b2.

```
BASE            -- addr                     U,83
```

The address of a variable containing the current numeric conversion radix. {{2..72}}

```
BEGIN           --                          C,I,79
                -- sys   (compiling)
```

Used in the form:

```
        BEGIN ... flag UNTIL
     or
        BEGIN ... flag WHILE ... REPEAT
```

BEGIN marks the start of a word sequence for repetitive execution. A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false. The words after UNTIL or REPEAT will be executed when either loop is finished. sys is balanced with its corresponding UNTIL or WHILE . See: "9.9 Control Structures"

```
BLK             -- addr                     U,79              "b-l-k"
```

The address of a variable containing the number of the mass storage block being interpreted as the input stream. If the value of BLK is zero the input stream is taken from the text input buffer. {{0..the number of blocks available -1}} See: TIB "input stream"

```
BLOCK           u -- addr                   M,83
```

addr is the address of the assigned buffer of the first byte of block u.

If the block occupying that buffer is not block u and has been UPDATEed it is transferred to mass storage before assigning the buffer. If block u is not already in memory, it is transferred from mass storage into an assigned block buffer. A block may not be assigned to more than one buffer. If u is not an available block number, an error condition exists. Only data within the last buffer referenced by BLOCK or BUFFER is valid. The contents of a block buffer must not be changed unless the change may be transferred to mass storage.

BUFFER u -- addr M,83

Assigns a block buffer to block u. addr is the address of the first byte of the block within its buffer. This function is fully specified by the definition for BLOCK except that if the block is not already in memory it might not be transferred from mass storage. The contents of the block buffer assigned to block u by BUFFER are unspecified.

C! 16b addr -- 79 "c-store"

The least-significant 8 bits of 16b are stored into the byte at addr.

C@ addr -- 8b 79 "c-fetch"

8b is the contents of the byte at addr.

CMOVE addr1 addr2 u -- 83 "c-move"

Move u bytes beginning at address addr1 to addr2. The byte at addr1 is moved first, proceeding toward high memory. If u is zero nothing is moved.

CMOVE> addr1 addr2 u -- 83 "c-move-up"

Move the u bytes at address addr1 to addr2. The move begins by moving the byte at (addr1 plus u minus 1) to (addr2 plus u minus 1) and proceeds to successively lower addresses for u bytes. If u is zero nothing is moved. (Useful for sliding a string towards higher addresses).

COMPILE -- C,83

Typically used in the form:

```
: <name> ... COMPILE <namex> ... ;
```

When <name> is executed, the compilation address compiled for <namex> is compiled and not executed. <name> is typically immediate and <namex> is typically not immediate. See: "compilation"

CONSTANT 16b -- M,83

A defining word executed in the form:

```
16b CONSTANT <name>
```

Creates a dictionary entry for <name> so that when <name> is later executed, 16b will be left on the stack.

CONVERT +d1 addr1 -- +d2 addr2 79

+d2 is the result of converting the characters within the text beginning at addr1+1 into digits, using the value of BASE , and accumulating each into +d1 after multiplying +d1 by the value of BASE . Conversion

```
     continues until an unconvertible character is encountered.  addr2 is the
     location of the first unconvertible character.

COUNT           addr1 -- addr2 +n                    79
     addr2 is addr1+1 and +n is the length of the counted string at addr1.
     The byte at addr1 contains the byte count +n.  Range of +n is {0..255}.
     See: "string, counted"

CR              --                                   M,79                "c-r"
     Displays a carriage-return and line-feed or equivalent operation.

CREATE          --                                   M,79
     A defining word executed in the form:
             CREATE <name>
     Creates a dictionary entry for <name>.  After <name> is created, the
     next available dictionary location is the first byte of <name>'s
     parameter field.  When <name> is subsequently executed, the address of
     the first byte of <name>'s parameter field is left on the stack.
     CREATE does not allocate space in <name>'s parameter field.

D+              wd1 wd2 -- wd3                       79               "d-plus"
     wd3 is the arithmetic sum of wd1 plus wd2.

D<              d1 d2 -- flag                        83          "d-less-than"
     flag is true if d1 is less than d2 according to the operation of < except
     extended to 32 bits.

DECIMAL         --                                   79
     Set the input-output numeric conversion base to ten.

DEFINITIONS     --                                   79
     The compilation vocabulary is changed to be the same as the first
     vocabulary in the search order.  See: "vocabulary, compilation"

DEPTH           -- +n                                79
     +n is the number of 16-bit values contained in the data stack before +n
     was placed on the stack.

DNEGATE         d1 -- d2                             79             "d-negate"
     d2 is the two's complement of d1.

DO              w1 w2 --                             C,I,83
                -- sys    (compiling)
     Used in the form:
             DO ... LOOP
          or
             DO ... +LOOP
     Begins a loop which terminates based on control parameters.  The loop
     index begins at w2, and terminates based on the limit w1.  See LOOP and
     +LOOP for details on how the loop is terminated.  The loop is always
     executed at least once.  For example:  w DUP DO ... LOOP executes 65,536
     times.  sys is balanced with its corresponding LOOP or +LOOP .
     See:  "9.9 Control Structures"

     An error condition exists if insufficient space is available for at least
     three nesting levels.

DOES>           -- addr                          C,I,83             "does"
                --    (compiling)
     Defines the execution-time action of a word created by a high-level
     defining word.  Used in the form:
             : <namex> ... <create> ... DOES> ... ;
     and then
             <namex> <name>
     where <create> is CREATE or any user defined word which executes CREATE .

     Marks the termination of the defining part of the defining word <namex>
     and then begins the definition of the execution-time action for words
     that will later be defined by <namex>.  When <name> is later executed,
     the address of <name>'s parameter field is placed on the stack and then
     the sequence of words between DOES> and ; are executed.

DROP            16b --                           79
     16b is removed from the stack.

DUP             16b -- 16b 16b                   79                 "dupe"
     Duplicate 16b.

ELSE            --                               C,I,79
                sys1 -- sys2   (compiling)
     Used in the form:
             flag IF ... ELSE ... THEN
     ELSE executes after the true part following IF .  ELSE forces execution
     to continue at just after THEN .  sys1 is balanced with its corresponding
     IF .  sys2 is balanced with its corresponding THEN .  See: IF  THEN

EMIT            16b --                           M,83
     The least-significant 7-bit ASCII character is displayed.  See: "9.5.3
     EMIT"

EXECUTE         addr --                          79
     The word definition indicated by addr is executed.  An error condition
     exists if addr is not a compilation address.

EXIT            --                               C,79
     Compiled within a colon definition such that when executed, that colon
     definition returns control to the definition that passed control to it by
     returning control to the return point on the top of the return stack.  An
     error condition exists if the top of the return stack does not contain a
```

```
        valid return point.  May not be used within a do-loop.
        See: ;  "stack, return" "9.3 Return Stack"

EXPECT          addr +n --                          M,83
        Receive characters and store each into memory.  The transfer begins at
        addr proceeding towards higher addresses one byte per character until
        either a "return" is received or until +n characters have been
        transferred.  No more than +n characters will be stored.  The "return" is
        not stored into memory.  No characters are received or transferred if +n
        is zero.  All characters actually received and stored into memory will be
        displayed, with the "return" displaying as a space.  See: SPAN
        "9.5.2 EXPECT"

FILL            addr u 8b --                        83
        u bytes of memory beginning at addr are set to 8b.  No action is taken if
        u is zero.

FIND            addr1 -- addr2 n                    83
        addr1 is the address of a counted string.  The string contains a word
        name to be located in the currently active search order.  If the word is
        not found, addr2 is the string address addr1, and n is zero.  If the word
        is found, addr2 is the compilation address and n is set to one of two
        non-zero values.  If the word found has the immediate attribute, n is set
        to one.  If the word is non-immediate, n is set to minus one (true).

FLUSH           --                                  M,83
        Performs the function of SAVE-BUFFERS then unassigns all block buffers.
        (This may be useful for mounting or changing mass storage media).

FORGET          --                                  M,83
        Used in the form:
                FORGET <name>
        If <name> is found in the compilation vocabulary, delete <name> from
        the dictionary and all words added to the dictionary after <name>
        regardless of their vocabulary.  Failure to find <name> is an error
        condition.  An error condition also exists if the compilation vocabulary
        is deleted.  See: "10.2 General Error Conditions"

FORTH           --                                  83
        The name of the primary vocabulary.  Execution replaces the first
        vocabulary in the search order with FORTH .  FORTH is initially the
        compilation vocabulary and the first vocabulary in the search order.  New
        definitions become part of the FORTH vocabulary until a different
        compilation vocabulary is established.  See:  VOCABULARY

FORTH-83        --                                  83
        Assures that a FORTH-83 Standard System is available, otherwise an error
        condition exists.
```

```
HERE            -- addr                             79
     The address of the next available dictionary location.

HOLD            char --                             79
     char is inserted into a pictured numeric output string.  Typically used
     between <# and #> .

I               -- w                                C,79
     w is a copy of the loop index.  May only be used in the form:
             DO ... I ... LOOP
          or
             DO ... I ... +LOOP

IF              flag --                             C,I,79
                -- sys    (compiling)
     Used in the form:
             flag IF ... ELSE ... THEN
          or
             flag IF ... THEN
If flag is true, the words following IF are executed and the words
     following ELSE until just after THEN are skipped.  The ELSE part is
     optional.

     If flag is false, words from IF through ELSE , or from IF through THEN
     (when no ELSE is used), are skipped.  sys is balanced with its
     corresponding ELSE or THEN .  See: "9.9 Control Structures"

IMMEDIATE       --                                  79
     Marks the most recently created dictionary entry as a word which will be
     executed when encountered during compilation rather than compiled.

J               -- w                                C,79
     w is a copy of the index of the next outer loop.  May only be used within
     a nested DO-LOOP or DO-+LOOP in the form, for example:
             DO ... DO ... J ... LOOP ... +LOOP

KEY             -- 16b                              M,83
     The least-significant 7 bits of 16b is the next ASCII character received.
     All valid ASCII characters can be received.  Control characters are not
     processed by the system for any editing purpose.  Characters received by
     KEY will not be displayed.  See: "9.5.1 KEY"

LEAVE           --                                  C,I,83
                --      (compiling)
     Transfers execution to just beyond the next LOOP or +LOOP .  The loop is
     terminated and loop control parameters are discarded.  May only be used
     in the form:
             DO ... LEAVE ... LOOP
       or
```

```
                DO ... LEAVE ... +LOOP
        LEAVE may appear within other control structures which are nested within
        the do-loop structure.  More than one LEAVE may appear within a do-loop.
        See: "9.3 Return Stack"

LITERAL         -- 16b                          C,I,79
                16b --    (compiling)
        Typically used in the form:
                [ 16b ]  LITERAL
        Compiles a system dependent operation so that when later executed, 16b
        will be left on the stack.

LOAD            u --                            M,79
        The contents of >IN and BLK , which locate the current input stream, are
        saved.  The input stream is then redirected to the beginning of screen u
        by setting >IN to zero and BLK to u.  The screen is then interpreted.  If
        interpretation from screen u is not terminated explicitly it will be
        terminated when the input stream is exhausted and then the contents of
        >IN and BLK will be restored.  An error condition exists if u is zero.
        See: >IN  BLK  BLOCK

LOOP            --                              C,I,83
                sys -- (compiling)
        Increments the DO-LOOP index by one.  If the new index was incremented
        across the boundary between limit-1 and limit the loop is terminated and
        loop control parameters are discarded.  When the loop is not terminated,
        execution continues to just after the corresponding DO .  sys is balanced
        with its corresponding DO .  See: DO

MAX             n1 n2 -- n3                     79                  "max"
        n3 is the greater of n1 and n2 according to the operation of > .

MIN             n1 n2 -- n3                     79                  "min"
        n3 is the lesser of n1 and n2 according to the operation of < .

MOD             n1 n2 -- n3                     83
        n3 is the remainder after dividing n1 by the divisor n2.  n3 has the same
        sign as n2 or is zero.  An error condition results if the divisor is zero
        or if the quotient falls outside of the range {-32,768..32,767}.
        See:  "division, floored"

NEGATE          n1 -- n2                        79
        n2 is the two's complement of n1, i.e., the difference of zero less n1.

NOT             16b1 -- 16b2                    83
        16b2 is the one's complement of 16b1.

OR              16b1 16b2 -- 16b3               79
        16b3 is the bit-by-bit inclusive-or of 16b1 with 16b2.
```

```
OVER            16b1 16b2 -- 16b1 16b2 16b3       79
     16b3 is a copy of 16b1.

PAD                  -- addr                      83
     The lower address of a scratch area used to hold data for intermediate
     processing.  The address or contents of PAD may change and the data lost
     if the address of the next available dictionary location is changed.  The
     minimum capacity of PAD is 84 characters.

PICK                 +n -- 16b                    83
     16b is a copy of the +nth stack value, not counting +n itself.  {0..the
     number of elements on stack-1}
             0 PICK is equivalent to DUP
             1 PICK is equivalent to OVER

QUIT                 --                           79
     Clears the return stack, sets interpret state, accepts new input from the
     current input device, and begins text interpretation.  No message is
     displayed.

R>                   -- 16b                       C,79            "r-from"
     16b is removed from the return stack and transferred to the data stack.
     See: "9.3 Return Stack"

R@                   -- 16b                       C,79            "r-fetch"
     16b is a copy of the return stack.

REPEAT               --                           C,I,79
                     sys --     (compiling)
     Used in the form:
             BEGIN ... flag WHILE ... REPEAT
     At execution time, REPEAT continues execution to just after the
     corresponding BEGIN .  sys is balanced with its corresponding WHILE .
     See: BEGIN

ROLL                 +n --                        83
     The +nth stack value, not counting +n itself is first removed and then
     transferred to the top of the stack, moving the remaining values into the
     vacated position.  {0..the number of elements on the stack-1}
             2 ROLL is equivalent to ROT
             0 ROLL is a null operation

ROT             16b1 16b2 16b3 -- 16b2 16b3 16b1  79                 "rote"
     The top three stack entries are rotated, bringing the deepest to the top.

SAVE-BUFFERS    --                                M,79       "save-buffers"
     The contents of all block buffers marked as UPDATEed are written to their
     corresponding mass storage blocks.  All buffers are marked as no longer
     being modified, but may remain assigned.
```

```
SIGN            n --                                83
     If n is negative, an ASCII "-" (minus sign) is appended to the pictured
     numeric output string.  Typically used between <# and #> .

SPACE           --                                  M,79
     Displays an ASCII space.

SPACES          +n --                               M,79
     Displays +n ASCII spaces.  Nothing is displayed if +n is zero.

SPAN            -- addr                             U,83
     The address of a variable containing the count of characters actually
     received and stored by the last execution of EXPECT .  See: EXPECT

STATE           -- addr                             U,79
     The address of a variable containing the compilation state.  A non-zero
     content indicates compilation is occurring, but the value itself is
     system dependent.  A Standard Program may not modify this variable.

SWAP            16b 16b2 -- 16b2 16b1               79
     The top two stack entries are exchanged.

THEN            --                                  C,I,79
                sys --   (compiling)
     Used in the form:
             flag IF ... ELSE ... THEN
          or
             flag IF ... THEN
     THEN is the point where execution continues after ELSE , or IF when no
     ELSE is present.  sys is balanced with its corresponding IF or ELSE .
     See:  IF  ELSE

TIB             -- addr                             83              "t-i-b"
     The address of the text input buffer.  This buffer is used to hold
     characters when the input stream is coming from the current input device.
     The minimum capacity of TIB is 80 characters.

TYPE            addr +n --                          M,79
     +n characters are displayed from memory beginning with the character at
     addr and continuing through consecutive addresses.  Nothing is displayed
     if +n is zero.  See: "9.5.4 TYPE"

U.              u --                                M,79            "u-dot"
     u is displayed as an unsigned number in a free-field format.

U<              u1 u2 -- flag                       83        "u-less-than"
     flag is true if u1 is less than u2.

UM*             u1 u2 -- ud                         83          "u-m-times"
```

ud is the unsigned product of u1 times u2. All values and arithmetic are unsigned.

```
UM/MOD          ud u1 -- u2 u3                    83        "u-m-divide-mod"
```

u2 is the remainder and u3 is the floor of the quotient after dividing ud by the divisor u1. All values and arithmetic are unsigned. An error condition results if the divisor is zero or if the quotient lies outside the range {0..65,535}. See: "floor, arithmetic"

```
UNTIL           flag --                           C,I,79
                sys --    (compiling)
```

Used in the form:

```
        BEGIN ... flag UNTIL
```

Marks the end of a BEGIN-UNTIL loop which will terminate based on flag. If flag is true, the loop is terminated. If flag is false, execution continues to just after the corresponding BEGIN . sys is balanced with its corresponding BEGIN . See: BEGIN

```
UPDATE          --                                79
```

The currently valid block buffer is marked as modified. Blocks marked as modified will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block or upon execution of FLUSH or SAVE-BUFFERS .

```
VARIABLE        --                                M,79
```

A defining word executed in the form:

```
        VARIABLE <name>
```

A dictionary entry for <name> is created and two bytes are ALLOTted in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack.

```
VOCABULARY      --                                M,83
```

A defining word executed in the form:

```
        VOCABULARY <name>
```

A dictionary entry for <name> is created which specifies a new ordered list of word definitions. Subsequent execution of <name> replaces the first vocabulary in the search order with <name>. When <name> becomes the compilation vocabulary new definitions will be appended to <name>'s list. See: DEFINITIONS "search order"

```
WHILE           flag --                           C,I,79
                     sys1 -- sys2    (compiling)
```

Used in the form:

```
        BEGIN ... flag WHILE ... REPEAT
```

Selects conditional execution based on flag. When flag is true, execution continues to just after the WHILE through to the REPEAT which then continues execution back to just after the BEGIN . When flag is false, execution continues to just after the REPEAT , exiting the control structure. sys1 is balanced with its corresponding BEGIN . sys2 is balanced with its corresponding REPEAT . See: BEGIN

```
WORD            char -- addr                    M,83
```

Generates a counted string by non-destructively accepting characters from the input stream until the delimiting character char is encountered or the input stream is exhausted. Leading delimiters are ignored. The entire character string is stored in memory beginning at addr as a sequence of bytes. The string is followed by a blank which is not included in the count. The first byte of the string is the number of characters {0..255}. If the string is longer than 255 characters, the count is unspecified. If the input stream is already exhausted as WORD is called, then a zero length character string will result.

If the delimiter is not found the value of >IN is the size of the input stream. If the delimiter is found >IN is adjusted to indicate the offset to the character following the delimiter. #TIB is unmodified.

The counted string returned by WORD may reside in the "free" dictionary area at HERE or above. Note that the text interpreter may also use this area. See: "input stream"

```
XOR             16b1 16b2 -- 16b3               79                "x-or"
```

16b3 is the bit-by-bit exclusive-or of 16b1 with 16b2.

```
[               --                              I,79        "left-bracket"
                --    (compiling)
```

Sets interpret state. The text from the input stream is subsequently interpreted. For typical usage see LITERAL . See:]

```
[']             -- addr                         C,I,M,83    "bracket-tick"
                --    (compiling)
```

Used in the form:

```
        ['] <name>
```

Compiles the compilation address addr of <name> as a literal. When the colon definition is later executed addr is left on the stack. An error condition exists if <name> is not found in the currently active search order. See: LITERAL

```
[COMPILE]       --                              C,I,M,79 "bracket-compile"
                --    (compiling)
```

Used in the form:

```
        [COMPILE] <name>
```

Forces compilation of the following word <name>. This allows compilation of an immediate word when it would otherwise have been executed.

```
]               --                              79         "right-bracket"
```

Sets compilation state. The text from the input stream is subsequently compiled. For typical usage see LITERAL . See: [

F.13 Double Number Extension Word Set

F.13.1 Double Number Extension Word Set Layers

Nucleus layer

```
2!   2@   2DROP   2DUP   2OVER   2ROT   2SWAP   D+   D-   D0=   D2/
D<   D=   DABS   DMAX   DMIN   DNEGATE   DU<
```

Device Layer

none

Interpreter layer

```
D.   D.R
```

Compiler layer

```
2CONSTANT  2VARIABLE
```

F.13.2 The Double Number Extension Word Set Glossary

```
2!              32b addr --                         79             "two-store"
     32b is stored at addr.  See: "number"

2@              addr -- 32b                         79             "two-fetch"
     32b is the value at addr.  See: "number"

2CONSTANT       32b --                              M,83        "two-constant"
     A defining word executed in the form:
             32b 2CONSTANT <name>
     Creates a dictionary entry for <name> so that when <name> is later
     executed, 32b will be left on the stack.

2DROP           32b --                              79              "two-drop"
     32b is removed from the stack.

2DUP            32b -- 32b 32b                      79              "two-dupe"
     Duplicate 32b.

2OVER           32b1 32b2 -- 32b1 32b2 32b3         79              "two-over"
     32b3 is a copy of 32b1.

2ROT            32b1 32b2 32b3 -- 32b2 32b3 32b1    79              "two-rote"
```

The top three double numbers on the stack are rotated, bringing the third double number to the top of the stack.

```
2SWAP           32b1 32b2 -- 32b2 32b1        79            "two-swap"
```
The top double numbers are exchanged.

```
2VARIABLE       --                            M,79      "two-variable"
```
A defining word executed in the form:

```
        2VARIABLE <name>
```
A dictionary entry for <name> is created and four bytes are ALLOTted in its parameter field. This parameter field is to be used for contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When <name> is later executed, the address of its parameter is placed on the stack.
See: VARIABLE

```
D+              wd1 wd2 -- wd3                79
```
See the complete definition in the Required Word Set.

```
D-              wd1 wd2 -- wd3                79             "d-minus"
```
wd3 is the result of subtracting wd2 from wd1.

```
D.              -- d                          M,79             "d-dot"
```
The absolute value of d is displayed in a free field format. A leading negative sign is displayed if d is negative.

```
D.R             d +n --                       M,83           "d-dot-r"
```
d is converted using the value of BASE and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if d is negative. If the number of characters required to display d is greater than +n, an error condition exists. See: "number conversion"

```
D0=             wd -- flag                    83       "d-zero-equals"
```
flag is true if wd is zero.

```
D2/             d1 -- d2                      83        "d-two-divide"
```
d2 is the result of d1 arithmetically shifted right one bit. The sign is included in the shift and remains unchanged.

```
D<              d1 d2 -- flag                 83
```
See the complete definition in the Required Word Set.

```
D=              wd1 wd2 -- flag               83             "d-equal"
```
flag is true if wd1 equals wd2.

```
DABS            d -- ud                       79          "d-absolute"
```
ud is the absolute value of d. If d is -2,147,483,648 then ud is the same value. See: "arithmetic, two's complement"

```
DMAX            d1 d2 -- d3                  79              "d-max"
     d3 is the greater of d1 and d2.

DMIN            d1 d2 -- d3                  79              "d-min"
     d3 is the lesser of d1 and d2.

DNEGATE         d1 -- d2                     79
     See the complete definition in the Required Word Set.

DU<             ud1 ud2 -- flag              83           "d-u-less"
     flag is true if ud1 is less than ud2.  Both numbers are unsigned.
```

F.14 Assembler Extension Word Set

F.14.1 The Assembler Extension Word Set Layers

Nucleus layer

none

Device layer

none

Interpreter layer

ASSEMBLER

Compiler layer

;CODE CODE END-CODE

F.14.2 Assembler Extension Word Set Usage

Because of the system dependent nature of machine language programming, a Standard Program cannot use CODE or ;CODE .

F.14.3 The Assembler Extension Word Set Glossary

```
;CODE           --                           C,I,79   "semi-colon-code"
                sys1 -- sys2   (compiling)
     Used in the form:
          : <namex> ... <create> ... ;CODE ... END-CODE
     Stops compilation, terminates the defining word <namex> and executes
     ASSEMBLER.  When <namex> is executed in the form:
          <namex> <name>
     to define the new <name>, the execution address of <name> will
     contain the address of the code sequence following the ;CODE in <namex>.
```

```
    Execution of any <name> will cause this machine code sequence to be
    executed.  sys1 is balanced with its corresponding : .  sys2 is balanced
    with its corresponding END-CODE .  See: CODE  DOES>

ASSEMBLER       --                              83
    Execution replaces the first vocabulary in the search order with the
    ASSEMBLER vocabulary.  See:  VOCABULARY

CODE            -- sys                          M,83
    A defining word executed in the form:
        CODE <name> ... END-CODE
    Creates a dictionary entry for <name> to be defined by a following
    sequence of assembly language words.  Words thus defined are called code
    definitions.  This newly created word definition for <name> cannot be
    found in the dictionary until the corresponding END-CODE is successfully
    processed (see: END-CODE ).  Executes ASSEMBLER .  sys is balanced with
    its corresponding END-CODE .

END-CODE        sys --                          79                "end-code"
    Terminates a code definition and allows the <name> of the corresponding
    code definition to be found in the dictionary.  sys is balanced with its
    corresponding CODE or ;CODE .  See: CODE
```

F.15 The System Extension Word Set

F.15.1 The System Extension Word Set Layers

```
Nucleus layer

    BRANCH   ?BRANCH

Device layer

    none

Interpreter layer

    CONTEXT   CURRENT

Compiler layer

    <MARK   <RESOLVE   >MARK   >RESOLVE
```

F.15.2 System Extension Word Set Usage

```
    After BRANCH or ?BRANCH is compiled, >MARK or >RESOLVE is executed.  The
    addr left by >MARK is passed to >RESOLVE .  The addr left by <MARK is
    passed to <RESOLVE .  For example:
```

```
: IF      COMPILE ?BRANCH    >MARK    ; IMMEDIATE
: THEN    >RESOLVE   ; IMMEDIATE
```

F.15.3 The System Extension Word Set Glossary

```
<MARK            -- addr                    C,83        "backward-mark"
     Used at the destination of a backward branch.  addr is typically only
     used by <RESOLVE to compile a branch address.

<RESOLVE         addr --                    C,83     "backward-resolve"
     Used at the source of a backward branch after either BRANCH or ?BRANCH .
     Compiles a branch address using addr as the destination address.

>MARK            -- addr                    C,83         "forward-mark"
     Used at the source of a forward branch.  Typically used after either
     BRANCH or ?BRANCH .  Compiles space in the dictionary for a branch
     address which will later be resolved by >RESOLVE .

>RESOLVE         addr --                    C,83      "forward-resolve"
     Used at the destination of a forward branch.  Calculates the branch
     address (to the current location in the dictionary) using addr and places
     this branch address into the space left by >MARK .

?BRANCH          flag --                    C,83      "question-branch"
     When used in the form:  COMPILE ?BRANCH  a conditional branch operation
     is compiled.  See BRANCH for further details.  When executed, if flag is
     false the branch is performed as with BRANCH .  When flag is true
     execution continues at the compilation address immediately following the
     branch address.

BRANCH           --                         C,83
     When used in the form:  COMPILE BRANCH  an unconditional branch operation
     is compiled.  A branch address must be compiled immediately following
     this compilation address.  The branch address is typically generated by
     following BRANCH with <RESOLVE or >MARK .

CONTEXT          -- addr                    U,79
     The address of a variable which determines the dictionary search order.

CURRENT          -- addr                    U,79
     The address of a variable specifying the vocabulary in which new word
     definitions are appended.
```

F.16 Controlled Reference Words

The Controlled Reference Words are word definitions which, although not required, cannot be present with a non-standard definition in the vocabulary FORTH of a Standard System. These words have present usage and/or are candidates for future standardization.

```
-->             --                              I,M,79       "next-block"
                --   (compiling)
     Continue interpretation on the next sequential block.  May be used within
     a colon definition that crosses a block boundary.

.R              n +n --                         M,83              "dot-r"
     n is converted using BASE and then displayed right aligned in a field +n
     characters wide.  A leading minus sign is displayed if n is negative.  If
     the number of characters required to display n is greater than +n, an
     error condition exists.  See:  "number conversion"

2*              w1 -- w2                        83            "two-times"
     w2 is the result of shifting w1 left one bit.  A zero is shifted into the
     vacated bit position.

BL              -- 32                           79                  "b-l"
     Leave the ASCII character value for space (decimal 32).

BLANK           addr u --                       83
     u bytes of memory beginning at addr are set to the ASCII character value
     for space.  No action is taken if u is zero.

C,              16b --                          83              "c-comma"
     ALLOT one byte then store the least-significant 8 bits of 16b at
     HERE 1- .

DUMP            addr u --                       M,79
     List the contents of u addresses starting at addr.  Each line of values
     may be preceded by the address of the first value.

EDITOR          --                              83
     Execution replaces the first vocabulary in the search order with the
     EDITOR vocabulary.  See: VOCABULARY

EMPTY-BUFFERS   --                              M,79      "empty-buffers"
     Unassign all block buffers.  UPDATEed blocks are not written to mass
     storage.  See: BLOCK

END             flag --                         C,I,79
                sys --    compiling
     A synonym for UNTIL .

ERASE           addr u --                       79
     u bytes of memory beginning at addr are set to zero.  No action is taken
     if u is zero.
```

HEX -- 79

Set the numeric input-output conversion base to sixteen.

INTERPRET -- 79

Begin text interpretation at the character indexed by the contents of >IN relative to the block number contained in BLK , continuing until the input stream is exhausted. If BLK contains zero, interpret characters from the text input buffer. See: "input stream"

K -- w C,83

w is a copy of the index of the second outer loop. May only be used within a nested DO-LOOP or DO-+LOOP in the form, for example:

```
        DO ... DO ... DO ... K ... LOOP ... +LOOP ... LOOP
```

LIST u -- M,79

The contents of screen u are displayed. SCR is set to u. See: BLOCK

OCTAL -- 83

Set the numeric input-output conversion base to eight.

OFFSET -- addr U,83

The address of a variable that contains the offset added to the block number on the stack by BLOCK or BUFFER to determine the actual physical block number.

QUERY -- M,83

Characters are received and transferred into the memory area addressed by TIB . The transfer terminates when either a "return" is received or the number of characters transferred reaches the size of the area addressed by TIB . The values of >IN and BLK are set to zero and the value of #TIB is set to the value of SPAN . WORD may be used to accept text from this buffer. See: EXPECT "input stream"

RECURSE -- C,I,83
-- (compiling)

Compile the compilation address of the definition being compiled to cause the definition to later be executed recursively.

SCR -- addr U,79 "s-c-r"

The address of a variable containing the number of the screen most recently LISTed.

SP@ -- addr 79 "s-p-fetch"

addr is the address of the top of the stack before SP@ was executed.

THRU u1 u2 -- M,83

Load consecutively the blocks from u1 through u2.

U.R u +n -- M,83 "u-dot-r"

u is converted using the value of BASE and then displayed as an unsigned number right aligned in a field +n characters wide. If the number of characters required to display u is greater than +n, an error condition exists. See: "number conversion"

Appendix G

Glossary

The present glossary comprises two sections; the first deals with CAM-specific words, while the second is devoted to general-purpose F83 words that have been included in the generic Forth system (F83.EXE) upon which the CAM software is built. The CAM section is fairly exhaustive: besides providing a handy reference for the casual user, it lists all of the software primitives that may be of interest to the sophisticated user. On the other hand, the F83 section only lists words that are deemed notable for some reason. A number of these words are original with the present implementation; a few F83 words have been renamed to avoid confusion with hexadecimal constants;[1] some entries are of a tutorial nature, as a complement to Chapter 5; finally, some words are listed just to make you aware of the useful functions they provide.

Each entry of the glossary is headed by a Forth word—or by several words that are closely related in meaning. The header entry contains several fields:

- The Forth word in question. This may be followed by '`name`' or '`filename`', denoting that, besides any arguments on the stack, the word also expects a character string (usually the name of a Forth word or of a file) to be read from the input stream.

- Stack comment, in parentheses (cf. Section 5.11). No stack comment appears if the word does not expect or leave anything on the stack. The dash in a stack comment is dropped if the word does not leave anything on the stack.

- The compiler which was responsible for creating the word.

- The source file where the word is defined.

For example, the following header

[1] All single-digit hex numbers (`0` through `F`) are defined as constants; no two-digit hex numbers (i.e., `00` through `FF`) are used as names of words in the `FORTH` vocabulary.

```
    CENTERS  ( -- 2bits)           :      CAM-HOOD
```

means that CENTERS takes no arguments from the stack; returns a number (all arguments or results named in stack comments are 16-bit quantities unless otherwise specified) of which only the two least significant bits are relevant (the others are all zeros); is defined by the COLON compiler; and is loaded into the system from the source file CAM-HOOD.4TH . On the other hand, the header

```
    INCLUDE  filename                 :      CAM-BUF
```

means that INCLUDE will look for a file name in the input stream (in this case, the source-file text immediately following, since INCLUDE can be used only in the interpretive mode), and will not affect the stack.

Among the abbreviations used within the stack comments, the most common are `n` , for a generic number—often the number of a bit-plane; `d` , for a *double* number (two 16-bit values, with the 16 most significant bits on top of the stack); `f` , for a logical flag (TRUE or FALSE); `mask` for a bit mask; `len` for a length (e.g., number of bytes in a string); `str` for the address of a *counted string* (the characters that make up the string are preceded by a *count* byte); `addr` , for a 16-bit address; `seg` , for an 8086 segment; `offs` for an offset. A mnemonic such as `row#` means "row number" (i.e., which row), while `#rows` means "number of rows" (i.e., how many rows).

Note that 'variable' generically refers to a quantity whose value may change during the execution of a program. We consistently use the small-caps spelling when we mean a Forth VARIABLE (i.e., a word that returns the *address* of its data cell, as explained in Section 5.8).

Strictly speaking, CAM neighbor or pseudo-neighbor words such as NORTH or VERT are compilation variables, and never know anything about the data actually present on the CAM card (say, in the bit-planes or in the address registers). However, in describing their function we use the convenient fiction that NORTH , for example, "returns the value of the the north-neighbor bit in plane 0 or 2;" what we mean is, of course, that "the value returned by NORTH will be used to compile a table entry which will be accessed during the simulation just when the north-neighbor bit has that value" (cf. Section 9.3).

Many pairs of neighbor words (such as CENTER and CENTER' , or HORZ and VERT) are accompanied by a *joint* version, which is always a two-bit variable. The least significant bit is the value of the first element of the pair, the next bit that of the second element (in arithmetic notation, "the first bit plus twice the second"). Thus, CENTERS=CENTER+2×CENTER' , and HV= HORZ+2×VERT . Similar considerations apply to the phase pseudo-neighbors (such as PHASES) and to the words that control the phase pseudo-variables (such as <PHASES>), as explained in Section 9.4.

Entries are in ASCII alphabetic order. Major neighborhood assignments begin with `N/` , minor assignments with `&/` . Pseudo-variables which are mapped into CAM shadow variables have the form `<...>` ; words such as `>PLN0` and `>BODY` are read "to plane 0" and "to body." Variables used as flags end with a question mark. Routines whose actions depend on such flags have names that start with a question mark. Other conventions used in names either are generic Forth conventions (see Appendix F), or are explained along with the relevant words.

G.1 CAM words

#CAMS (-- n) : **CAM-IO**

Returns the number of CAMs that the software believes are installed. This number can be changed using the word **CAMS**.

#IDLES (-- addr) **VARIABLE CAM-STEP**

This variable controls the rate at which steps occur when using the control-panel *Run* command $\boxed{S}$. Speed is controlled by adjusting the number of **IDLE** (i.e., display-only) steps inserted between each active **STEP**.

&/CENTERS : **CAM-HOOD**

Minor neighborhood assignment for weak coupling between CAM-A and CAM-B. Makes the neighbors **&CENTER** and **&CENTER'** available as part of the current neighborhood for either CAM-A, CAM-B, or both (see **CAM-A**). The corresponding joint version **&CENTERS** is of course also available.

NOTE: Two consecutive minor assignments should not be used without an intervening major assignment.

&/HV : **CAM-HOOD**

Minor neighborhood assignment. Makes **&HORZ**, **&VERT**, and **&HV** available. See also **CAM-A** and **&/CENTERS**.

&/PHASES : **CAM-HOOD**

Minor neighbor assignment. Makes **&PHASE**, **&PHASE'**, and **&PHASES** available. See also **CAM-A**.

&/USER : **CAM-HOOD**

Minor neighborhood assignment. Connects address lines 10 and 11 of the current CAM half to input pins UA10, UA11 (for CAM-A) or UB10, UB11 (for CAM-B) of the user connector. This is the default after **NEW-EXPERIMENT** and after each major neighborhood assignment, if no explicit minor assignment is made. Only **CENTER**, **CENTER'**, and **CENTERS** are always available. See the description of == for an example of how to give a name to a user neighbor for use in rules. See also **CAM-A**.

&CENTER (-- bit) == **CAM-HOOD**
&CENTER'
&CENTERS (-- 2bits) : **CAM-HOOD**

Neighbors made available by the minor neighborhood assignment **&/CENTERS**. The meaning of **&CENTER** is "The center bit of the *even*-numbered plane in the other half of CAM." Thus, in a rule-component that is sent to CAM-A (see **>PLN0** and **>PLN1**) it returns a bit from plane 2.

Similarly, **&CENTER'** refers to the *odd*-numbered plane. **&CENTERS** is the joint version of **&CENTER** and **&CENTER'**

&HORZ (-- bit) == **CAM-HOOD**
&VERT
&HV (-- 2bits) : **CAM-HOOD**

Pseudo-neighbors made available by the minor neighborhood assignment `&/HV`. The word `&HORZ` returns the parity of the *horizontal* spatial position of the cell currently being updated, relative to an origin set by `<ORG-HV>`.

Similarly, `&VERT` returns the *vertical* parity. `&HV` is the joint version of this pair.

`&PHASE ( -- bit)` == CAM-HOOD
`&PHASE'`
`&PHASES ( -- 2bits)` : CAM-HOOD

Pseudo-neighbors made available by the minor neighborhood assignment `&/PHASES`. During a step, all cells see for `&PHASE` the value that was assigned to the Forth word `<&PHASE>` when the `STEP` command was issued.

Similarly, `&PHASE'` gets its value from `<&PHASE'>`. `&PHASES` is the joint version of this pair.

`&VERT` see `&HORZ`

`*BEAM ( n)` : CAM-HOOD

Toggle the visibility of beam *n* (0, 1, 2, 3 = blue, green, red, intensity). See `COLOR-MASK`.

`+BEAM ( n)` : CAM-HOOD

Make beam *n* visible if it was temporarily turned off by `*BEAM` (0, 1, 2, 3 = blue, green, red, intensity); this cannot reverse the effect of `INVISIBLE`. See `COLOR-MASK`.

`-BUF>PDAT ( pl.mask src.seg offs r0 c0 #rows #byts row.incr)` CODE CAM-IO

Low-level primitive—alternative entry point to `BUF>PDAT`. This routine sets the direction flag so that string moves from memory run backwards. Since bytes are always written to CAM forwards, this allows strings of bytes to be reversed. The offset given here should point to the last byte of the buffer to be sent. See also `-PDAT>BUF`.

`-PDAT>BUF ( pl.mask dest.seg offs r0 c0 #rows #byts row.incr)` CODE CAM-IO

Low-level primitive—alternative entry point to `PDAT>BUF`. Sets up the PC to perform string moves backwards. Since CAM always reads the data it sends forwards, this results in bytes being stored in reverse order. The offset should point to the last byte of the data buffer. See also `-BUF>PDAT`. (This routine is called by `S/S-PL`).

`-STEP#` : CAM-STEP

Negates the value of the doubleword variable `STEP-NUMBER`; useful in conjunction with reversible rules. More at `STEP-NUMBER`.

`/-PL` see `NOT-PL`

`2ARG ( -- d)` : CAM-KEYS

Used within control-panel key definitions (see `ALIAS`) to allow parameters to be passed to Forth words attached to keys. Works exactly like `ARG`, but returns a double number on the stack.

`<&PHASE> ( -- bit)` =AAR CAM-HOOD
`<&PHASE'>`
`<&PHASES> ( -- 2bits)`

`<&PHASE>`, `<&PHASE'>`, and their joint version `<&PHASES>` are pseudo-variables distinct from but similar in behavior to `<PHASE>` and `<PHASE'>`. Set using `IS` as in `1 IS <PHASES>`.

`<ORG-H> ( -- bit)` =AAR CAM-HOOD
`<ORG-V>`
`<ORG-HV> ( -- 2bits)`

The pseudo-variable <ORG-H> corresponds to the least significant bit of the horizontal value of the space-grid origin. A 0 corresponds to an even position, a 1 to an odd one. The pseudo-neighbor word HORZ, used during lookup table compilation, refers to this pseudo-variable. Set using IS.

Similarly, <ORG-V> corresponds to vertical origin. <ORG-HV> is the joint version of this pair, returning <ORG-H>+2×<ORG-V>. See HORZ and &HORZ. Set using IS, but see also EVEN-GRID and ODD-GRID.

<PHASE> (-- bit) =AAR CAM-HOOD
<PHASE'>
<PHASES> (-- 2bits)

The <PHASE> pseudo-variable represents a phase bit present on the CAM card that can be controlled at will from the PC. The pseudo-neighbor word PHASE, used during lookup table compilation, refers to this pseudo-variable.

Similarly, <PHASE'> represents another phase bit. <PHASES> is the joint version of <PHASE> and <PHASE'>. See PHASE', <&PHASE>. Set using IS.

<TAB-A> (-- bit) =AAR CAM-HOOD
<TAB-B>
<TAB-AB> (-- 2bits)

The pseudo-variable <TAB-A> refers to a CAM bit that controls whether the regular (0) or the auxiliary (1) tables of CAM-A are in use (see >PLN0, >AUX0). For example, 0 IS <TAB-A> sets CAM-A to use normal tables.

Similarly for <TAB-B> and CAM-B.

<TAB-AB> is the joint version of this pair. A value of 0 means "use the regular tables for both CAM-A and CAM-B;" a value of 3,"use the auxiliary tables for both" (bits 0 and 1 refer to CAM-A and CAM-B respectively). For example, 1 IS <TAB-AB> sets CAM-A to use the auxiliary tables and CAM-B to use regular ones.

The words REG-TABS and AUX-TABS can also be used to set TAB-AB to 0 or 3, respectively.

=2ARG (addr) : CAM-KEYS

Used for passing parameters to routines attached to control panel keys. Works like =ARG, except that it stores a double number.

== name (n) : CAM-HOOD

This word associates the given name to table address line *n*. It can be used to name the "unnamed" neighbors provided by N/USER, &/USER, and N/MARG. Example: (in decimal)

```
NEW-EXPERIMENT CAM-A  N/MOORE  10 == RAND1  11 == RAND2
```

would establish the Moore neighborhood on CAM-A, with two external neighbors named RAND1 and RAND2 (perhaps coming from an external random-number generator) attached to the UA10 and UA11 inputs of the user connector. Note that the default minor neighborhood &/USER was automatically selected when a new major neighborhood was established.

=ARG (addr) : CAM-KEYS

Used for passing parameters to routines attached to control panel keys. Works like **ARG**, except that it expects an address on the stack, and it takes any argument typed in and stores it as a 16-bit value in the location specified by **addr** (usually put on the stack by a VARIABLE). *If no argument was typed in, the value at* **addr** *is unchanged!* This is convenient to use with keys which have a default parameter: if no argument is given, the last argument previously given remains in force.

>AUX0 (bit) >> CAM-HOOD
>AUX1
>AUX2
>AUX3
>AUXA (2bits) : CAM-HOOD
>AUXB

Column dispatcher for the auxiliary tables. **>AUX***n* works just like >PLN*n*, but the result for the neighborhood case being considered is sent to a table that will be used to update plane *n* only if <TAB-A> is set appropriately.

>AUXA is the joint version of **>AUX0** and **>AUX1** (and works like **>PLNA**); **>AUXB** that of **>AUX2** and **>AUX3**.

>BLUE (bit) >> CAM-HOOD
>GREEN
>INTEN
>RED

Dispatchers for individual color-map columns. Low bit of value on the stack indicates whether or not the named beam should be on or off for the case under consideration. See **MAKE-CMAP** and >IRGB.

>GREEN >INTEN see >BLUE

>IRGB (4bits) : CAM-HOOD

Dispatcher for a whole color-map entry (see **MAKE-CMAP**). The four lower bits of the value on the stack are used to determine the color to be associated with a given cell-state (see the table in Section 9.6). See also **IRGB**.

>PLN0 (bit) >> CAM-HOOD
>PLN1
>PLN2
>PLN3
>PLNA (2bits) : CAM-HOOD
>PLNB

Column dispatchers for the regular tables (see **MAKE-TABLE**). The low bit of the argument to >PLN0 is used as the value to be sent to plane 0 when the neighborhood configuration currently under consideration is found.

Similarly for **PLN1**, **PLN2**, and **PLN3**. See also **PLN0**.

>PLNA is the joint version of >PLN0 and >PLN1. Least significant bit of argument is used for plane 0, the next for plane 1—the rest is ignored. Similarly, >PLNB is the joint version of >PLN2 and >PLN3.

>RED see >BLUE.

?CAGE*PLNS : CAM-KEYS

If **CFLAG** is set, exchange the contents of the cage buffer (see **CBUF-SEG** and **CAGE-AREA**) with the active region of CAM's planes. Since most plane operations work on this active region, this causes subsequent plane operations to act on the cage, with the area swapped into the cage-buffer playing the role of the buffer. Execute this a second time to "turn off" the cage.

In performing the exchange of plane and buffer data, this routine uses as an intermediate buffer the second half of the 8K temporary buffer at **TEMP-SEG**, whose contents is thus destroyed.

AAR (-- offset) **EQU CAM-IO**

Offset of CAM's "auxiliary address" register within each CAM's shadow segment. See **<ORG-H>**, **<PHASES>**, **<&PHASES>**, and **C!CAM**.

ALIAS name **: CAM-KEYS**

Used to attach a Forth routine to a key of the control panel. User routines are normally attached to **Alt-** keys; all of the alphabetic **Alt-** keys plus $\boxed{\texttt{Alt-F1}}$...$\boxed{\texttt{Alt-F10}}$ are available for use by the user. **ALIAS** attaches the named key to the most recently defined Forth word. For example,

```
                          : Example
  ." This is an example" ;   ALIAS E
```

would attach the Forth word "**Example**" to the $\boxed{\texttt{Alt-E}}$ key. If this example were compiled, subsequently pressing $\boxed{\texttt{Alt-E}}$ would cause the letter **E** to appear, then the word "**Example**," and finally the word would execute, producing its message "**This is an example**."

Note that you should use upper-case letter names, and upper-case **F1**–**F10** for the key names following **ALIAS**. All user-defined keys automatically appear in the **ALTERNATE** menu of the control panel.

For those who are curious, **ALIAS** is implemented by means of the Forth **VOCABULARY** mechanism. Each time **ALIAS** is executed, it puts an entry in the current alias vocabulary (see **ALIASES** and **ORDER**). When an **Alt-** key is pressed, the key name is searched for as a Forth word in this vocabulary, using **FIND**; if found, it is executed. All the rest of the control-panel keys are attached in a similar manner, using other **ALIAS** vocabularies, such as **GENERAL**, **PLANE-OPS**, etc.

ALIASES **: CAM-KEYS**

You probably don't want to use this word—it sets the aliases vocabulary to be whatever is the context vocabulary when **ALIASES** is executed. The CAM program puts key definitions into alias vocabularies: the vocabulary that is used when the system is first booted is the **ALTERNATE** vocabulary. All user definitions in this **ALTERNATE** vocabulary are invoked by **Alt-** keys—all reserved for user definitions. If **ALIASES** is used to change the aliases vocabulary, system keys may become redefined and hence unavailable; and non-standard keys will not all be automatically put into the **ALTERNATE** submenu. See also **ALIAS** and **ORDER**.

ALPHA (-- bit) **== CAM-HOOD**
ALPHA'
BETA
BETA'
ALPHAS (-- 2bits) **: CAM-HOOD**
BETAS

The four color-map inputs. **ALPHA** usually returns the state of plane 0 and **ALPHA'** that of plane 1—but see **SHOW-FUNCTION**. Similarly, **BETA** and **BETA'** refer to planes 2 and 3. Used in color-map generation (see **MAKE-CMAP**).

ALPHAS is the joint version of **ALPHA** and **ALPHA'**. Similarly for **BETAS**.

`ALTERNATE` VOCABULARY CAM-KEYS

This is the `VOCABULARY` in which user-defined `Alt-` keys are put.

```
AND>PL  ( n)                                        :  CAM-BUF
OR>PL
XOR>PL
P*BUF
PL>PB
PB>PL
PLS>PBS  ( --)
PBS>PLS
```

`AND>PL` performs the logical AND function between plane *n* and the corresponding plane buffer; the result is stored back in the plane. Similarly for `OR>PL` and `XOR>PL`.

`P*BUF` exchanges the plane with its buffer.

`PL>PB` copies the plane to its buffer; vice versa for `PB>PL`. `PLS>PBS` and `PBS>PLS` (with no arguments on the stack) do the copying for all four bit-planes.

See also `PERMUTE` and `AREA`

All the above operations between planes and buffers affect only the *active* region of a plane and the associated buffer (see `NOT-PL`).

`ANDS` see `STORES`

`AREA ( seg h w)` : CAM-BUF

This word is used to set the values of the height and width of the active region of the bit-planes, and to associate a buffer-segment with this region. Operations on planes, such as `/-PL`, `NOT-PL`, `RND>PL`, etc. (but not `FILE>PL` and other disk I/O words) operate only on the active region of the bit-planes. Similarly, operations between planes and buffers treat the planes as if they were only the size of their active regions, and use a correspondingly small contiguous area of buffers starting at the beginning of the given segment.

Two special cases which call `AREA` are `WHOLE-AREA` and `CAGE-AREA`. The former sets the area to be the full screen, and the associated buffer-segment to be the `PBUF-SEG`—this is the default situation which is generally in effect. `CAGE-AREA` uses the `CBUF-SEG` and height and width taken from variables `C-HEIGHT` and `C-WIDTH`. The active region's height, width, and associated segment are available to any routine: they are supplied by CONSTANTs `R/8`, `C/8` and `A-SEG`, which respectively return the height and width in units of eight cell positions, and the associated segment.

As an example, if you wanted to define 4 extra plane buffers and copy the planes to them, you could use (numbers are in hexadecimal)

```
                                  8000 SEGMENT EXTRA-BUFFERS
    EXTRA-BUFFERS 20 20 AREA
    PLS>PBS
```

Note that the height and the width of the full screen (in units of eight-cell positions) are both 20 (hex) (or 32 in decimal). As a further example,

```
    TEMP-SEG 4 6 AREA
    0 PL>PB    2 PL>PB
```

would transfer data from a 32-high, 48-wide area in the center of the screen to two areas of **TEMP-SEG**. The data from plane 0 will go to offset 0, the data from plane 2 will go to offset 384 and following locations (4×6×8 bytes of buffer alloted per plane).

ARG (-- n) : CAM-KEYS

Used within the definition of a control-panel key (see **ALIAS**) to take a numerical parameter from previous digit keystrokes. If an argument was typed at the control-panel prompt immediately before **ARG** is executed, this argument is returned by **ARG** as a 16-bit value on the stack. If there was no argument, a value of 0 is put on the stack. See also **2ARG**, **ARG?**, **=ARG**, and **=2ARG**.

ARG? (-- f) : CAM-KEYS

Can be used by routines which are attached to control-panel keys (see **ALIAS**). It returns a logical flag to indicate whether an argument was typed for this key (see **ARG**).

AUX-TABS : CAM-HOOD

Used to select the auxiliary tables for both CAM-A and CAM-B. This is useful in a step-cycle—see also **REG-TABS**, **<TAB-A>**,and **MAKE-CYCLE**.

AUX0 (-- bit) =NEW CAM-HOOD
AUX1
AUX2
AUX3
AUXA (-- 2bits) : CAM-HOOD
AUXB

AUX0 is a table compilation variable; it returns the value that will be sent to the **AUX0** column of the current entry in the lookup table. Similarly for **AUX1** etc. See **>AUX0** and **>PLN0**.

AUXA is the joint version of **AUX0** and **AUX11** (=**AUX0**+2×**AUX1**); similarly for **AUXB**.

B!CELL (bit r c n) : CAM-IO

Stores the bit contained in the first argument to row r, column c of bit-plane *n*. Wrap-around occurs if r or c have values that exceed 255.

B=A : CAM-HOOD

Copy the current rule and neighborhood assignment used by CAM-A into CAM-B.

B@CELL (r c n -- bit) : CAM-IO

Returns on the stack the value of the bit in row r, column c of bit-plane *n*. Wrap-around occurs if r or c have values that exceed 255.

BARE : CAM-HOOD

MAKE-TABLE BARE will set the rule for plane 1 to return zeros for all cases.

BEGIN-MACROS : CAM-INT

Begin defining macros—this allots some space for the machine code, and puts subsequent Forth definitions after this space, until **USE-MACROS** is executed. See also **FORGET-MACROS**, which erases all macros, leaving only the code generated using them.

BEGIN-SERVICE-STEPS : CAM-STEP

Used when CAM must perform "service" steps extraneous to the simulation proper; for instance, shifting of bit-planes is usually done with service steps (see **SEND-SHIFTS**). Switches all CAMs to begin using an extra set of shadow areas, and saves all tables for all CAMs.

Initially the extra shadow areas are a copy of the normal shadows, so that color maps, etc., start off the same. You may change tables, neighborhoods, and phases; when you are done, **END-SERVICE-STEPS** will restore CAM's previous state. All tables, event counts, neighborhoods, etc. (everything but the contents of the planes) will be back to what they were before the service steps.

`BETA BETA' BETAS` see `ALPHA`

`BIT-GET ( bit# n -- bit)` CODE CAM-IO

Returns the specified bit of the 16-bit quantity *n*.

`BIT-PUT ( bit bit# n -- n')` CODE CAM-IO

Store a bit into an indicated position of the 16-bit quantity *n*. The modified quantity is returned on the stack.

`BLACK` : CAM-HOOD

Use during color-map generation (see **MAKE-CMAP**). Sets the result for all colors to be off, so that one does not have to explicitly program any colors that are not used.

`BUF-AND ( s.seg offs d.seg offs #byts)` CODE CAM-BUF
`BUF-OR`
`BUF-XOR`

BUF-AND is the logical AND of the source segment:offset with the destination. Corresponding bytes are ANDed, and the results are stored back into the destination. Similarly for **BUF-OR**, etc.

`BUF>MAP` : CAM-HOOD

Copies the color-map buffer into the color map—see **MAP>BUF**.

`BUF>PDAT ( pl.msk src.seg offs r0 c0 #rows #byts row.incr)` CODE CAM-IO

Low-level primitive for all transfers of data from PC memory to CAM. The plane-mask argument indicates which planes the data will be written to simultaneously (a value of 3 indicates planes 0 and 1, for example). The source segment and offset point to the start of the data, `r0` and `c0` give the upper left corner on the screen of the region to be written—`r0` is the row, and `c0` is the column divided by 4 (since columns use nybble addressing). The number of rows and bytes to be written on each row are given, and then the row increment, i.e., the number of CAM rows to skip at the end of each row before writing the next one.

See also **-BUF>PDAT**, **PB>PL**, and **TEMP>PL**.

`BUF>TAB` : CAM-BUF

Used to copy a precompiled table from a buffer in the PC's memory to CAM's lookup tables. All 8 columns (**PLN0–PLN3** and **AUX0–AUX3**) are copied. The buffer starts at **TBUF-SEG:0000**. Cf. **TAB>BUF** and **BUF>TDAT**.

`BUF>TDAT ( buf.seg buf.offs first.page #pages)` CODE CAM-IO

Low-level primitive used for all writing of table data from PC to CAM. Complete tables are 10 (hex) pages long, where each page is 100 (hex) bytes. A source segment and offset are given, and the first page of the destination is indicated (usually 0). See also **TDAT>BUF**, **BUF>TAB**, and **MAKE-TABLE**.

BUTTONS (-- 3bits) : CAM-EDIT

Returns the value (0–7) of the mouse buttons' status. The high, mid, and low bits correspond to the left, middle, and right buttons. 0 means button pressed, 1 means button up.

C!CAM (val loc#) : CAM-IO

Used to talk directly to CAM shadows and hardware. Can be used in hardware debugging (naive users should not need to use this). Normally writes val to shadow-register loc#, which is sent to CAM during its next interrupt. Used in conjuction with HARD or with CAM data areas (PDAT or TDAT) it writes directly to CAM. See also CCR, AAR, TAA, TBA, PCA, and PRA.

C@CAM (loc# -- val) : CAM-IO

Used to talk directly to CAM shadows and hardware. Can be used in hardware debugging (naive users should not need to use this). Normally reads val from shadow-register loc#, which was sent to CAM during its last interrrupt. Used in conjuction with HARD or with CAM data areas (PDAT or TDAT) it reads directly from CAM. See also C!CAM.

CAGE-AREA : CAM-BUF

See AREA.

CAM : CAM-IO

Select CAM (rather than the PC) as the video source seen on the display; ignored if you have separate monitors for CAM (or CAMs) and the PC. See DISPLAYS and PC.

CAM# (-- addr) VARIABLE CAM-IO

This is the number of the currently selected CAM (ranging from 0 to 7, where 0 is the master CAM and 1 thru 7 are slaves). Most operations only affect the currently selected CAM. See FOR-ALL-CAMS and NEXT-CAM for a discussion of how to change information for all CAMs.

CAM-A : CAM-HOOD
CAM-B
CAM-AB

CAM-A selects CAM-A as the object of future neighborhood assignments. Similarly for CAM-B.

CAM-AB selects *both* halves of the machine as the target of neighborhood assignments; this is the default situation, in effect after executing NEW-EXPERIMENT.

CAM-BASE (-- addr) VARIABLE CAM-INT

This variable contains the base segment address used by CAM-INIT to set up all addressing of CAM data. Its default value is DC00. This indicates that up to eight CAMs occupy consecutive 2K blocks of memory beginning at address DC000, with the master CAM (number 0) last and the seventh slave first. This is the variable that must be changed if the addressing jumpers on CAMs are changed, to remap CAMs into another 16K block of memory. To use a block beginning at D0000, for example, you could type D000 IS CAM-BASE and then save the system using SAVE-SYSTEM.

CAM-IRQ# (-- addr) VARIABLE CAM-INT

This variable is used by CAM-INIT to set up CAM's vertical-blank interrupt. The default value is 2 (CAM will use IRQ-2). This value must be changed if the IRQ jumper on CAM is changed. Note that only the jumper on CAM 0 (the master) matters (none of the slaves will cause interrupts). See also CAM-BASE.

`CAM-PTR ( -- addr)` VARIABLE CAM-IO

Pointer to the selected CAM. See CAM-SELECT.

`CAM-RESUME ( -- addr)` EQU CAM-INT

This constant points to the beginning of a machine-language subroutine which causes operation of the CAM interrupt service routines to resume after being suspended by the CAM-SUSPEND routine. This is used to prevent the interrupt from interfering with foreground machine-code routines which must temporarily take control of CAM machine registers.

`CAM-SEG ( -- segment)` : CAM-IO

Returns on the stack the segment value of selected CAM's memory mapped area. This is used to access CAM's registers as memory locations.

`CAM-SELECT ( cam#)` : CAM-IO

Select the active CAM, to which commands are directed. After RESET-CAMS, CAM 0 (the master) will be the selected one. To perform an operation on all installed CAMs, see FOR-ALL-CAMS. *Note that the command* STEP *is always directed to all* CAMs, *no matter which of them is currently selected.*

See also SHOW-CAM, CAM, and PC.

`CAM-SUSPEND ( -- addr)` EQU CAM-INT

Suspend CAM operation—see CAM-RESUME.

`CAMOUT ( -- mask)` EQU CAM-IO

"CAM output selection" bit of CAM's configuration and control register. See SET-CCR.

`CAMS ( #cams)` : CAM-STEP

Used to tell the system how many CAMs are present. Non-existent CAMs will be ignored as far as event detection is concerned, and shadows will not be copied to them (this allows more time during the CAM interrupt for other things—see R/FLY for example). The number of CAMs is also used by FOR-ALL-CAMS, which you can use to perform operations on all CAMs. If you set this number to more CAMs than are actually present, the software will service these non-existent CAMs exactly as if they were there—you can do this to try out programs that are intended to use more CAMs than you actually have.

If there are more than four CAMs present in the system, the vertical blank interval will be increased by 8 scan-lines, to give the interrupt more time. This interval can be set to this longer value manually by executing DPYNRM CLR-CCR. The default length of this interval is determined by the state of the VARIABLE LONG?, and is restored whenever you execute NEW-EXPERIMENT or RESET-CAMS.

`CBUF-SEG ( -- segment)` SEGMENT CAM-BUF

This is the 2K segment used for storing the cage's contents. See also CAGE-AREA.

`CCR ( -- offset)` EQU CAM-IO

Offset of the "configuration control" register within each CAM's shadow segment. See SET-CCR and C!CAM.

CCW CCW' CCWS see CW

CCW-PL see NOT-PL

CENTER (-- bit) == CAM-HOOD
CENTER'
CENTERS (-- 2bits) : CAM-HOOD

The neighbor CENTER (the current center of attention) is available as part of all neighborhoods. Like all other neighbor names, this is a compilation variable which is set by MAKE-TABLE at the beginning of each of the 4096 cases that must be considered to generate a complete table. The rule is a Forth word that is a function of these neighbor names, and MAKE-TABLE evaluates it for all possible values of the neighbors.

For a rule-component that is sent to CAM-A (see >PLN0, >AUX0) CENTER corresponds to the center cell bit of plane 0. Exactly the same tables can be sent to CAM-B (see >PLN2, >AUX2), in which case CENTER corresponds to the center cell of plane 2. For example,

```
    : ID-02   CENTER DUP  >PLN0  >PLN2 ;
```

would send the identity rule to planes 0 and 2. In fact, as long as CAM-A and CAM-B have the same neighborhood, it is always possible to just duplicate the rule result and send it to corresponding subtables on the two halves of the machine.

Similarly for CENTER', referring to planes 1 or 3.

The "plural" version CENTERS is the joint form of CENTER and CENTER', and returns the value CENTER+2×CENTER'.

CLOSE-DATA : CAM-BUF

Close the currently active data file, and write all changes to disk. (If the file isn't closed, some changes may not get written). Each open data file must be selected separately and then closed. See OPEN-DATA.

CLR-CCR (mask) CODE CAM-IO

Works like SET-CCR, but clears the indicated bits to zero, leaving the rest unchanged. Bits to be cleared are indicated by 1s in the mask.

CMAP (-- offset) EQU CAM-IO

Offset within each CAM's shadow segment of the beginning of its color map. See MAKE-CMAP, BUF>MAP, MAP>BUF, and C!CAM.

COL (-- 2bits) : CAM-HOOD

Neighbor in the N/XPAND n-hood. Current cell's column position on the screen, mod 4. See <ORG-HV> and CENTER.

COLOR-MASK (-- addr) VARIABLE CAM-HOOD

VARIABLE used by MAKE-CMAP to generate color maps that have some of the beams temporarily turned off. The words *BEAM and +BEAM act on the low byte of this mask; the word INVISIBLE acts on the high byte. During color-map generation, the low and the high bytes of this mask are ORed together, and bits 0–3 of the result are used as the final mask. Any 0 in the mask leaves the corresponding beam (0, 1, 2, 3 = blue, green, red, intensity) unaffected, any 1 turns the corresponding beam off. Color maps are stored in duplicate—an active map and a copy. Beams are turned off only in the active map, so that they can later be turned back on by referring to the copy.

After NEW-EXPERIMENT, the COLOR-MASK is off. If any of the beams have been turned off, all color maps will be generated with the appropriate beams turned off in the active map. To cause subsequent color maps to be generated with all beams turned on, you can use COLOR-MASK OFF.

CW (-- bit) == CAM-HOOD
CCW
OPP
CW'
CCW'
OPP'
CWS (-- 2bits) : CAM-HOOD
CCWS
OPPS

Neighbors made available by the N/MARG, N/MARG-HV and N/MARG-PH major neighborhood assignments. CW returns on the stack the value, for the case under consideration, of the clockwise cell in the Margolus neighborhood (plane 0 for CAM-A, 2 for CAM-B); CCW, of the counter-clockwise neighbor; OPP, of the diagonally opposite. (Cf. CENTER and MAKE-TABLE).

Similarly for CW', CCW', and OPP'—all referring to planes 1 or 3.

CWS, etc are joint versions.

CW-PL see NOT-PL

DATA-ADDR (n -- addr) : CAM-BUF

Returns the address in memory where the next *n* bytes of data from the active data-file (see OPEN-DATA) can be found, and advances the data pointer for that file by *n*. Uses Forth block-I/O routines, and so data must be accessed using a record size that divides evenly into blocks of size 1K (e.g., *n* could be 8 each time). See also OPEN-DATA, CLOSE-DATA, GET-DATA, PUT-DATA, and REWIND. The active data pointer is kept in the double-number VARIABLE DATA-PTR—this can be manipulated directly to perform random access.

DATA-PTR (-- addr) 2VARIABLE CAM-BUF

This is the data pointer used by the OPEN-DATA command. It is a double-number VARIABLE which points to the next byte of the file to be retrieved or written. It is auto incremented as you read or write data; if you modify it directly, be sure to give it an even value, so that word reads and writes won't straddle two blocks. See also DATA-ADDR, REWIND, GET-DATA and PUT-DATA.

DIAG (-- mask) EQU CAM-IO

Diagnostic bit within CAM's step status register—see C@CAM.

DISPLAY-CONTROL VOCABULARY CAM-KEYS

VOCABULARY containing the display control ALIAS keys.

DISPLAYS (#displays) : CAM-STEP

Initialize the hardware for the selected number of displays. If only a single display is available, it will be shared between CAM and the PC. If two displays are available, one is assumed to be dedicated to the PC and the other is shared among the CAMs. If three or more displays are available, the PC and each CAM are treated as if they each have their own display (see DISPLAY). In this case, the control panel will never change the setting of the display-multiplexing hardware, though the user is allowed to. See CAMOUT and Appendix A.

DOS... : CAM-KEYS

Prints the message `(type 'exit' to return)` and then executes `COM` to invoke a copy of the DOS command interpreter.

`DOT ( row col)` : `CAM-EDIT`

Changes the bit at the indicated row and column without turning off the display. Only changes one bit per 60-th of a second—you probably want to use `B!CELL` and `B@CELL` to draw lines. The plane number is selected by changing the variable `DOT-PLN`.

`DOT-PLN ( -- addr)` `VARIABLE` `CAM-EDIT`

This VARIABLE contains the current plane number used by `DOT`, and is used to display the dot-cursor by the CAM control panel program.

`DOTS,SHIFTS` `VOCABULARY` `CAM-KEYS`

`VOCABULARY` containing the graphic editing and shift-key `ALIAS` definitions.

`DPYLIN ( -- mask)` `EQU` `CAM-IO`

"Display line" bit of step status register—see `C@CAM`.

`DPYNRM ( -- mask)` `EQU` `CAM-IO`

"Display normal" bit of CAM's configuration control register—see `SET-CCR` and `C!CAM`.

`DPYRQ ( -- mask)` `EQU` `CAM-IO`

"Display request" bit of CAM's configuration and control register—see `SET-CCR` and `C!CAM`.

`EAST EAST' EASTS` see `NORTH`

`ECHO` : `CAM-HOOD`

`MAKE-TABLE ECHO` makes the rule for plane 1 be "copy the `CENTER` of plane 0."

`ECNT-1 ( -- offset)` `EQU` `CAM-IO`
`ECNT-2`

See `EVENT-COUNT`.

`ECNT-TOT ( -- offset)` `EQU` `CAM-IO`

Global variable in the `STEP-SEG`. See `EVENT-TOT`.

`EDITING,RUNNING` `VOCABULARY` `CAM-KEYS`

`VOCABULARY` containing the `ALIAS` key definitions for editing screens, loading files, and running experiments.

`END-SERVICE-STEPS` : `CAM-STEP`

See `BEGIN-SERVICE-STEPS`.

`EQU name ( value)` : `CAM-INT`

Makes `name` a constant and sets it to the given value.

`EVEN-GRID` : `CAM-HOOD`

Sets both the horizontal and the vertical grid origin position to an even cell. This word is equivalent to `0 IS <ORG-HV>` .

`EVENT ( -- mask)` `EQU` `CAM-IO`

"Event detected" bit in CAM's step status register—see `EVENT-COUNT`, `EVENT-TOT` and `C@CAM`.

`EVENT-COUNT ( -- d)` CODE CAM-STEP

The number of occurrences of an event, returned as a double number. Correct usage is to use `NEW-EVENTS` to reset event pointers, and then run two steps before calling `EVENT-COUNT`. If you only start one step and then immediately call `EVENT-COUNT`, the step you started won't yet have finished, and so there won't yet be anything to count. (This is a problem of pipelining your computation to run at full speed—you must always set up the info for the next step before the current step has finished.)

If `EVENT-COUNT` is executed and no counts have accumulated since `NEW-EVENTS` or `EVENT-COUNT` was last executed, an error message will be printed and execution will be aborted. If either one or two counts have accumulated, all is well (the earlier of two counts will be returned the first time `EVENT-COUNT` is called, the later the next time it is called); these two counts are kept in `ECNT-1`, `ECNT-2`. If more than two counts have accumulated, again an error is signaled.

The one-byte variable `EVENT-LEN` in the `STEP-SEG` keeps track of the number of counts that have been accumulated. If you wish to just read an event count without keeping track of exactly which step it is associated with, you can set this variable to one, as in

```
    1 STEP-SEG EVENT-LEN C!L    EVENT-COUNT
```

which will read the most recent count. A word which reads the count at the end of one step, and starts a new step could be defined as

```
                          : READ-AND-STEP  ( -- d)
    STEP  EVENT-COUNT ;
```

Although this looks like we are stepping and then reading, the count read here by `EVENT-COUNT` will be the one produced by an earlier step: we are just starting the new step before we look at what we read, so as not to slow things down. A loop which runs 100 steps and examines all the counts could look like this

```
                                  : ANALYZE-EVENTS
    NEW-EVENTS STEP  99 0 DO
  READ-AND-STEP ANALYZE LOOP
    IDLE EVENT-COUNT ANALYZE ;
```

Here `ANALYZE` is some user-defined word which takes a double number on the stack and does something with it. The `IDLE` is there to wait for the last active step to finish—we could instead have made the loop 100 long, and not read the extra count at the end.

See `OPEN-DATA` if you wish to accumulate data on disk; see also `EVENT-TOT` and `EVENT-STOP`.

`EVENT-HANDLER` DEFER CAM-IO

This is a `DEFER`red word that will be called by `WAIT-FOR-PEND` (and thus indirectly by `STEP` and `IDLE`) if CAM stepping has been stopped because an event of a particular type has occurred (see `EVENT-MASK` and `EVENT-TYPE`). Initially this word is set by `NEW-EXPERIMENT` to be a `NOOP` (null operation). By setting this word to use a definition of your own, you can take special action when an event of interest has occured. After `EVENT-HANDLER` has executed, CAM stepping will resume. See `EVENT-STOP`.

`EVENT-LEN ( -- offset)` EQU CAM-INT

Offset to the one byte variable in the **STEP-SEG** which keeps track of how many counts have accumulated since the last execution of **NEW-EXPERIMENT**, **NEW-EVENTS**, or **EVENT-COUNT**. This variable is incremented each time a new count becomes available, and decremented whenever **EVENT-COUNT** reads a count—it allows **EVENT-COUNT** to maintain a two-item event stack in order to ensure that no counts are lost inadvertently.

EVENT-MASK (-- offset) EQU CAM-INT

This is a byte in the **STEP-SEG** which controls which CAM's events can cause stepping to stop.

The least significant bit corresponds to CAM 0 (the master). The most significant bit corresponds to CAM 7 (the seventh slave). A value of 1 indicates that event detection for the corresponding CAM during the next step may cause stepping to stop (see **EVENT-TYPE**, **EVENT-STOP**, and **EVENT-HANDLER**).

EVENT-STOP (-- offset) EQU CAM-INT

Because of the overlapping of step scheduling and event-count reading that is necessary to run at full speed, events that occur during one step will not be detected using **EVENT-COUNT** until after a new step has begun. This means that the state of CAM when the event happened is no longer available. If one desires to examine the states of CAMs in which certain events happened, a stop-on-event facility is available which doesn't slow CAM down.

To use this, you should first program the color map and auxiliary tables so that you detect the event of interest during the step in which the configuration of interest is being constructed by the updating (see **SHOW-FUNCTION**). The variables **EVENT-MASK** and **EVENT-TYPE** allow you to control which sorts of events (on up to eight CAMs simultaneously) will request that stepping be interrupted. The variable **EVENT-STOP** (within the **STEP-SEG**) indicates which CAM or CAM's requested stepping to be interrupted (bit 0 corresponds to the master CAM, etc.). As long as the **EVENT-STOP** byte is non-zero, stepping will be inhibited: the word **WAIT-FOR-PEND**, which is called by **STEP** and **IDLE**, examines this byte. If it is non-zero, a **DEFER**red word **EVENT-HANDLER** is executed, and then **EVENT-STOP** is cleared. The normal definition of **EVENT-HANDLER** is a null operation—by having it execute your own routine you can have whatever action you wish taken whenever an event of interest occurs.

EVENT-TOT (-- quad-number) : CAM-STEP

This returns the cumulative count (**ECNT-TOT**) of all events that have occurred since the last time **NEW-EXPERIMENT** or **NEW-EVENTS** was called. Before reading the count, it waits for the completion of up to two steps, namely, (a) a step that may currently be in progress and/or (b) one to whose execution the step-scheduling mechanism is irrevocably committed (i.e., one for which the **STEP** command has already been issued).

The total returned on the stack is a *quadruple* number—the high-order double number is on the top of the stack. Arithmetic operators for manipulating quad numbers can be made available by **INCLUDE**ing the file **QUAD.4TH**. See also **EVENT-COUNT**.

EVENT-TYPE (-- offset) EQU CAM-INT

This is a byte in the **STEP-SEG** segment which, together with **EVENT-MASK**, controls which CAM's events can cause stepping to stop.

The least significant bit corresponds to CAM 0 (the master). The most significant bit corresponds to CAM 7 (the seventh slave). A value of 0 or 1 indicates what type of condition (event or non-event) for the corresponding CAM during the next step may cause stepping to stop (see **EVENT-TYPE** and **EVENT-HANDLER**).

EXEC (str)	**CODE PC**

Given a pointer to a counted string (which must be immediately followed by a carriage return character, not counted as part of the string), **EXEC** will try to execute it as a DOS command. A blank string invokes the DOS command interpreter (from which you return to Forth by typing **exit**). Command lines should look like

```
/c  copy a:example b:
```

(note the extra '/c' at the beginning). After the DOS command, execution proceeds with the next instruction after **EXEC**, with the stack as it was before the call (except, of course, that **str** has been removed). See also **>DOS** and **COM**.

FILE>IMAGE (n #planes)	**: CAM-BUF**

Reads an image from disk to CAM (used with **OPEN-PATTERN**). Plane n is read from the n-th 8K record in the file, then plane $n+1$ from record $n+1$, etc., for the specified number of planes. If the file is too short it is wrapped around. Thus, for example, if a 2-record file is read to 4 planes, the 2 records are used for the first two planes, and then again for the second two. As a second example, if a single plane starting at 2 is read from a 32K file, then only record 2 (starting the count from 0) is read and it goes into plane 2. See also **IMAGE>FILE**.

FILE>PL (n)	**: CAM-BUF**

Used with **OPEN-PATTERN**. Copy data from the pattern file into the selected CAM plane. The file is treated as divided into 8K-byte records (this is the size of one bit-plane). The record n is copied to the plane n. File is wrapped around if n is too large. See also **PL>FILE**.

FILE>TAB	**: CAM-BUF**

Load entire lookup tables from current **TAB** file. Use with **OPEN-TABLE**.

FINISH-CYCLE	**: CAM-STEP**

This word is used for cleanly changing from one run cycle to another (see **MAKE-CYCLE**). The current run cycle is run until the end of the Forth word currently attached as the cycle is encountered. Then, instead of looping back to the beginning as usually happens with a step cycle, execution of the cycle ends and you exit from **FINISH-CYCLE**. You don't have to execute this word before executing **MAKE-CYCLE** on a new word—usually there is no reason to.

FOR-ALL-CAMS	**: CAM-STEP**

Begins a **DO** loop in which the index **I** ranges over the numbers of all installed CAMs. During each iteration, the **I**-th CAM is selected. The end of the loop is indicated by the word **NEXT-CAM**. As with ordinary **DO** loops, a **FOR-ALL-CAMS** loop can only be used inside of a colon definition.

FORGET-MACROS	**: CAM-INT**

Forget all Forth words defined since **BEGIN-MACROS** was executed, but leave the code generated in the dictionary.

FREEZE	**: CAM-HOOD**

MAKE-TABLE FREEZE makes the rule for plane 1 be the identity rule—no other planes are affected.

GENERAL	**VOCABULARY CAM-KEYS**

Vocabulary containing general-purpose key **ALIAS**'es, such as numbers and a key to exit to Forth.

GET-DATA (-- n) : CAM-BUF

Used with OPEN-DATA . Get the next sequential 16-bit quantity from the current data file.

GET-TIME (-- d) CODE CAM-LOAD

Returns time of day as a double number on the stack. The top number contains the hours (high byte) and minutes (low byte); the other number contains seconds (high byte) and 1/100-th seconds (low byte).

GO : CAM-KEYS

Resume stepping. See STOP.

GOTO-K : CAM-KEYS

Go to the control-panel program. Exits from whatever operation was going on (such as loading) and starts up the control panel.

H1 (-- bit) == CAM-HOOD

Pseudo-neighbor available as part of the N/XPAND neighborhood. Returns the second least significant bit of the horizontal position of the center cell. See also HORZ and COL.

HALF CONSTANT CAM-KEYS

Returns the constant value 32,762 (8000 in hex), which is half the number of cells on the CAM screen.

HARD (loc -- hard-loc) : CAM-IO

Used in conjuction with C@CAM and C!CAM to talk directly to hardware registers. For example,

```
5 AAR C!CAM
```

would write 5 to the shadow variable associated with the AAR register (which will be sent to CAM when it interrupts next), while

```
5 AAR HARD C!CAM
```

would write directly to the CAM hardware AAR register of the currently selected CAM.

HGLUE (-- mask) EQU CAM-IO

"Horizontal glue" bit of configuration control register. Set it for all CAM's that are to be glued horizontally—they will then make use of the signals coming into their horizontal glue connectors. Vertical gluing works in a similar fashion. See VGLUE, SET-CCR, and C!CAM.

HMASK (-- addr) VARIABLE CAM-STEP

Hold mask. The four low-order bits of this variable constitute a mask that is used by the CAM control panel program when it calls SEND-SHIFTS in response to the arrow keys. Normally, tables for four different shifts are sent for each bit-plane; if a bit of HMASK is set, an identity rule (no change) is sent as the table for the corresponding plane. Thus the indicated planes are held fixed, while the rest of the planes are free to shift.

HORZ (-- bit) == CAM-HOOD
VERT
HV (-- 2bits) : CAM-HOOD

Pseudo-neighbors made available by the N/MARG-HV major neighborhood assignment. HORZ returns the horizontal parity of the position of the cell currently being updated, relative to an origin set by <ORG-HV>. VERT does the same for the vertical position, and HV returns the joint value HORZ+2×VERT.

HRUN (-- mask) EQU CAM-IO

"Horizontal run" bit of the step status register. See SSR.

HV see HORZ

IDLE : CAM-STEP

Execute one *idle* step on CAM, during which display occurs but there is no updating of cell states. All shadow data is transferred, just as for an active step (color maps, neighborhood assignment, reading or writing planes during the vertical blank interval, etc); event-counts and step-counts are not affected. IDLE steps can be (and are) used for controling CAM's speed. When IDLE is used inside of a run cycle, it causes an exit just as STEP does. See also IDLES.

IDLES (n) : CAM-STEP

Execute IDLE *n* times.

IMAGE>FILE (n #planes) : CAM-BUF

Used with OPEN-PATTERN. Saves the image as consecutive 8K records, with no special header information. See FILE>IMAGE.

IMOVE (src.seg offs dest.seg offs #byts) CODE CAM-BUF

The same as LMOVE, but no interrupts are allowed to occur during the move—the block of data is moved as an indivisible unit.

INCLUDE name : CAM-BUF

Used to include the entire contents of another file as if it appeared at the indicated point. Used, for example, with AUTOCORR.4TH when you want to run an autocorrelation experiment, and with DOS2.4TH if you want to use the DOS2 file interface. (Also QUAD.4TH to load the quadruple-precision arithmetic words).

INVISIBLE (mask) : CAM-HOOD

TRUE INVISIBLE temporarily turns off all beams of the display, while FALSE INVISIBLE turns them back on without losing track of which beams were toggled off by *BEAM. See COLOR-MASK.

IRGB (-- 4bits) CODE CAM-HOOD

Color-map compilation variable. Returns the value that will be used as the current entry if no further changes are made. See also >IRGB and MAKE-CMAP.

IRGB-MAP : CAM-HOOD

Color map (see MAKE-CMAP) that associates a separate color "beam" with each of the four bit-planes (or auxiliary table outputs—see SHOW-FUNCTION). Planes 0, 1, 2, 3 correspond to intensity, red, green, and blue respectively.

LDTBL (-- mask) EQU CAM-IO

"Load table" bit of the configuration control register. See MAKE-TABLE and CCR.

LL LR LL' LR' LLS LRS see UL

LONG? (-- addr) VARIABLE CAM-STEP

VARIABLE used by **RESET-CAMS** to determine the default number of lines in a scan frame. If more than four CAMs are installed, this variable is ignored and the longest frame is used. On monitors where the top or bottom edges of the plane array are not visible, adding extra blank lines to the frame makes more of the display visible (you may have to adjust the vertical-size control in the monitor to take full advantage of this feature).

See **CAMS**.

MAKE-CMAP name : **CAM-HOOD**

Makes a new color map (see Section 9.6). **MAKE-CMAP** actually generates two copies of the color map (each nybble representing a color is stored twice, so that each of the 16 cases takes up a byte). One of the copies is the active map, which is sent to CAM; the other is used to restore the active map when beams have been temporarily turned off (see **COLOR-MASK**, ***BEAM**, and **INVISIBLE**).

MAKE-CYCLE name : **CAM-STEP**

Used to attach a Forth word as the active run cycle (see Section 9.9. The run cycle will be executed as a co-routine by the control panel. The purpose of the run cycle is to modify CAM parameters in a regular cycle in between steps. For example,

```
                              : BB-CYCLE
                EVEN-GRID STEP
                 ODD-GRID STEP ;
    MAKE-CYCLE BB-CYCLE
```

will attach **BB-CYCLE** as the active run cycle. Each time the Forth word **NEXT-STEP** is executed, **BB-CYCLE** will continue execution up to the next occurence of the word **STEP**, at which point execution of the cycle will be suspended with all parameters for the next step prepared, but the new step not yet started.

When **BB-CYCLE** is first attached by **MAKE-CYCLE**, **EVEN-GRID** is executed and the execution is suspended. The first time that **NEXT-STEP** is executed, **STEP** and **ODD-GRID** are executed, and then execution is suspended. The second time **NEXT-STEP** is executed, **STEP** and **EVEN-GRID** are executed, etc. (when you reach the end of **BB-CYCLE**, execution wraps around to the beginning). Note that:

- **NEXT-STEP** is the word that is executed when you run steps from the control panel.
- This co-routine has a separate stack from the rest of Forth.
- The default run cycle, which is attached whenever you execute **NEW-EXPERIMENT**, is just the word **STEP**.
- Since the run cycle begins to execute when you say **MAKE-CYCLE** (up to the first occurrence of **STEP** or **IDLE**) you should be careful that the experiment does not modify parameters (such as neighborhood assignments) initialized by the cycle.

MAKE-TABLE name : **CAM-HOOD**

Used to generate a lookup table from a rule-word, and to send it to CAM (see Section 9.3).

MAP>BUF : **CAM-HOOD**

Saves the color map table in the color-map buffer. See **BUF>MAP**.

MOUSE-INIT : **CAM-EDIT**

Initialize the mouse interface if present.

`MOUSE-IRQ#` VARIABLE CAM-EDIT

Variable containing the interrupt-request number for the mouse. Default value is 4.

`MOUSE-POS ( -- row col)` : CAM-EDIT

Returns current mouse row and column (or position determined by arrow keys).

`MOUSE>ORG ( -- delta-row delta-col)` : CAM-KEYS

Resets the mouse to be at the origin (center of the screen), and returns the amount of row- and column-shift used.

`N.EAST`
`N.WEST`
`S.EAST`
`S.WEST`

Neighbors made available by the `N/MOORE` and the `N/CORNERS` major assignments (cf. `CENTER`). During table compilation, `N.EAST` returns the value of the north-east neighbor on plane 0 or 2, etc.

No primed or joint versions of these neighbors exist in any of the standard neighborhoods.

`N/CORNERS` : CAM-HOOD

Major neighborhood assignment, consisting of `CENTER`, `CENTER'`, four corners (`N.EAST`, etc.), and four user neighbors (address lines 6, 7, 8, and 9). See `==`.

`N/MARG` : CAM-HOOD

Major neighborhood, consisting of the neighbors `CENTER CENTER' CW CW' CCW CCW' OPP OPP'` along with their joint versions. Neighbors number 8 and 9 are left as user neighbors—see `==`.

`N/MARG-HV` : CAM-HOOD

`N/MARG` neighborhood, with `HORZ` and `VERT` (and their joint version `HV`) added.

`N/MARG-PH` : CAM-HOOD

`N/MARG` neighborhood, with the pseudo-neighbors `PHASE` and `PHASE'` (and the joint version `PHASES`) added.

`N/MOORE` : CAM-HOOD

Major neighborhood, consisting of the nine neighbors in a 3×3 region on plane 0 (CAM-A) or plane 2 (CAM-B), plus the center cell on the other plane in the same half of CAM. The neighbors consist of the four corner neighbors of `N/CORNERS` plus the four side neighbors `NORTH SOUTH WEST EAST`, and of course `CENTER` and `CENTER'`, which are part of every neighborhood. Only one joint neighbor, `CENTERS`, is available.

`N/NSWE'` : CAM-HOOD

Major neighborhood, consisting of the four side neighbors `NORTH' SOUTH' EAST' WEST'`, the usual center cells, and four user neighbors (address lines 6, 7, 8, and 9).

`N/USER` : CAM-HOOD

Major neighborhood; attaches address lines 2–9 to user-connector inputs UA2–UA9 (for CAM-A) or UB2–UB9 (for CAM-B); as in all other neighborhoods, lines 0 and 1 are connected to **CENTER** and **CENTER'**. This is the default neighborhood used if no other major assignment is made.

See the description of == for an example of how to give names to user neighbors for use in rules.

N/VONN : **CAM-HOOD**

Major neighborhood, consisting of **NORTH NORTH' SOUTH SOUTH' WEST WEST' EAST EAST'** along with the centers and the joint neighbors for all of these (**NORTHS** etc.).

N/XPAND : **CAM-HOOD**

Major neighborhood. This is a specialized neighborhood that is supported mainly for use in the expanded-display mode. It makes available the neighborhood words **ROW** and **COL**, which allow the rule to know the position (mod 4) of the current cell on the screen, and the neighbors **NORTH' SOUTH' WEST' EAST' CENTER'** and **CENTER**.

NEW-EVENTS : **CAM-STEP**

Initializes event counts and stack information that tells subsequent **EVENT-COUNT** whether or not any events have been missed (because you didn't read them). It also resets **EVENT-TOT**, the cumulative count, to zero.

NEW-EXPERIMENT : **CAM-KEYS**

This is the word that is normally used to begin an experiment. It uses **NEWX** to erase all definitions from the previous experiment (but see **PERMANENT**) and resets CAM to a standard state with **RESET-CAMS**.

NEW=0 : **CAM-HOOD**

Used during table generation to clear all parts of the current entry (corresponding to all primary and all auxiliary table outputs) to zero. Otherwise, all parts of the current entry which are not explicitly set to some new value are left unchanged. See **MAKE-TABLE**.

NEW=AA **CODE** **CAM-HOOD**

Used within a table definition. Copies the CAM-A part of a table entry into the CAM-B part. For example, **MAKE-TABLE NEW=AA** would make the table for CAM-B be a copy of that for CAM-A.

Note: This copies the table, but not the neighborhood assignment. See **B=A** for a complete copy.

NEW=BA **CODE** **CAM-HOOD**

Interchanges the CAM-A and CAM-B halves of a table entry. **MAKE-TABLE NEW=BA** would interchange the complete tables. See **NEW=AA**.

NEW=BB : **CAM-HOOD**

Make both halves of the table entry copies of the CAM-B half. See **NEW=AA**.

NEW=TBUF : **CAM-HOOD**

Copies the new entry from the table buffer. This allows you to generate tables that are a function of the contents of the table buffer.

NEWX **DEFER** **CAM-KEYS**

This is an abbreviated version of **NEW-EXPERIMENT** that doesn't reset CAM—it just erases the old experiment from the Forth dictionary.

NEXT-CAM : **CAM-STEP**

End of a FOR-ALL-CAMS ... NEXT-CAM cycle. See FOR-ALL-CAMS.

NEXT-CYCLE : CAM-STEP

Continue executing the run cycle without stopping until you reach the end of it (see MAKE-CYCLE) and wrap around to the beginning of it; then stop at the first occurence of STEP or IDLE).

NEXT-STEP CREATE-PRO CAM-STEP

Continue executing the run cycle (see MAKE-CYCLE), stopping at the next occurence of STEP or IDLE).

```
NORTH  ( -- bit)                                          ==  CAM-HOOD
SOUTH
EAST
WEST
NORTH'
SOUTH'
EAST'
WEST'
NORTHS  ( -- 2bits)                                        :  CAM-HOOD
SOUTHS
EASTS
WESTS
```

NORTH is a neighbor for plane 0 (CAM-A) or plane 2 (CAM-B) made available by the major neighborhood assignments N/VONN and N/MOORE; NORTH' is for planes 1 or 3 (cf. CENTER). Similarly for SOUTH, EAST, and WEST.

NORTHS etc. are joint versions.

```
NOT-PL  ( n)                                               :  CAM-BUF
\-PL
/-PL
CW-PL
CCW-PL
RND>PL
S/S-PL
T/B-PL
VAL>PL  ( val n)
RND>PL  ( exp n)
```

NOT-PL performs the logical NOT function on plane *n*.

\-PL flips the plane across the main diagonal (upper-left/lower-right), /-PL across the secondary diagonal (lower-left/upper-right).

S/S-PL Flips the plane from side to side, T/B-PL from top to bottom.

CW-PL Rotates the plane one-quarter turn clockwise, CCW-PL counter-clockwise.

VAL>PL Fills the plane with repetitions of the 16-bit pattern val.

RND>PL Generates a random pattern where the expected number of 1s (over the whole bit-plane) is exp.

See also AND>PL.

All the above plane operations affect only the *active* region of the plane, determined by AREA or a word that calls AREA (such as WHOLE-AREA or CAGE-AREA). See AREA for details.

ODD-GRID : **CAM-HOOD**

Sets both horizontal- and vertical-grid origins to the odd value. This word is equivalent to

```
3 IS <ORG-HV>
```

OPEN-DATA filename : **CAM-BUF**

Open a *data* file named **filename** (to be used for reading or numbers as 16-bit words using **GET-DATA** and **PUT-DATA**) and create a Forth word **filename** that will be used for making this file the "current" data file. The declaration **OPEN-DATA filename** *must not* occur inside of a COLON definition. No default extension is supplied; if an extension is desired (**DAT** is the recommended extension for data files) it should be typed as part of **filename**.

As many data files as desired may be simultaneously open. Whenever executed, the word **filename** (which *may* occur in COLON definitions) changes the currently selected data file to **filename** and restores the position pointer within the file to the place where it was at when you last selected that data file (since it was opened by **OPEN-DATA**). When a data file are opened, the pointer is at set at the beginning of the file. You should use the Forth word **CREATE-FILE** (it takes the size in 1K blocks as an argument) to create the file before you use it. See also **CLOSE-DATA** and **DATA-ADDR**.

OPEN-PATTERN filename : **CAM-BUF**

Open a *pattern* file named **filename** for reading or writing plane data (using **FILE>IMAGE**, **IMAGE>FILE**, **FILE>PL**, and **PL>FILE**. Works analogously to **OPEN-DATA**. The recommended extension for pattern files is **PAT** (this extension is provided by default when you *Get from disk* or *Put to disk* CAM planes directly from the control panel; cf. Section 3.14).

OPEN-TABLE filename : **CAM-BUF**

Open a *table* file named **filename** for reading or writing precompiled lookup tables (using **FILE>TAB** and **TAB>FILE**). Works analogously to **OPEN-DATA**. The recommended extension for table files is **TAB** (this extension is provided by default when you *Save tables* or *Load tables* directly from the control panel; cf. Section 9.5).

OPP OPP' OPPS see **CW**

OR>PL see **AND>PL**

ORDER : **CAM-KEYS**

Lists the **CURRENT** and **CONTEXT** vocabularies, and also which vocabulary **ALIAS**'es are put into.

ORS see **STORES**

P*BUF see **AND>PL**

PAIR (a b -- a+2b) **CODE** **CAM-HOOD**

Used to construct joint neighbors, as in

```
: CENTERS  CENTER CENTER' PAIR ;
```

It can also be used repeatedly to join a group of one-bit items on the stack into a single binary number.

PB>PL PBS>PLS see **AND>PL**

PBUF-SEG (-- segment) **SEGMENT** **CAM-BUF**

Returns on the stack the value of the plane-buffer memory segment. This buffer is a memory area of 4×8K bytes that starts at location **PBUF-SEG:0000**.

PC : **CAM-IO**

Use the video output of the PC rather than CAM as the signal to be sent to the monitor. This has effect only when a single display is being shared between CAM and the PC. See DISPLAYS, CAM, CAM-SELECT, and SHOW-CAM.

PCA (-- offset) EQU **CAM-IO**

Offset of "plane column address" register within each CAM's shadow segment. This register always contains the column address that will result in a stationary picture if used for the next step; thus, it can be incremented or decremented to obtain desired effects. For example, PCA C@CAM 2+ PCA C!CAM would result in the configuration on the currently selected CAM being shifted 2 nybbles (an 8-cell width) to the left. Note that the column address register contains a nybble address; this is different from the row address register PRA which contains a bit address: PRA C@CAM 2+ PRA C!CAM would result in a shift up of only two cell positions.

PDAT (-- offset) EQU **CAM-IO**

CAM's plane data area. See BUF>PDAT, B!CELL, and C!CAM.

PDAT>BUF (n dest.seg offs r0 c0 #rows #byts row.incr) CODE **CAM-IO**

Low-level primitive used for all reading of data from plane *n* into the PC memory. Arguments have the same meanings as for BUF>PDAT (except that the segment is a destination, rather than a source). See also -PDAT>BUF, PL>PB, and PL>TEMP.

PEND : **CAM-IO**

Sets the STEP-PENDING shadow variable to TRUE. This variable will be cleared the next time the shadows are copied to CAM (usually every 60-th of a second). See WAIT-FOR-PEND. This word is used by STEP and IDLE.

PERMANENT : **CAM-LOAD**

NEW-EXPERIMENT normally erases all user-defined words from the Forth dictionary (as does NEWX). However, if the word PERMANENT is executed, all words in the dictionary at that moment become permanent, and will not be forgotten if NEW-EXPERIMENT or NEWX is subsequently executed. To forget words you have added to the permanent part of the dictionary, you can use the word FORGET, as in

```
    FENCE OFF  FORGET name  PERMANENT
```

where name could be any Forth word. To make changes to the system more permanent, you can save a new version of the system using SAVE-SYSTEM.

PERMUTE (new3 new2 new1 new0) : **CAM-KEYS**

The four numbers on the stack specify which planes should to be copied as the new contents of the four planes. For example,

```
    3 2 0 1 PERMUTE
```

would interchange planes 0 and 1 and leave the other two unchanged;

```
    1 1 1 1 PERMUTE
```

isn't really a permutation—plane 1 is copied onto all planes. Note that, as with other plane operations, the action of **PERMUTE** is restricted to the *active* region of the planes (cf. **NOT-PL**); the remainder of each plane (if any) is left unchanged.

PHASE (-- bit) == **CAM-HOOD**
PHASE'
PHASES (-- 2bits) : **CAM-HOOD**

Pseudo-neighbors made available by **N/MARG-PH**. The value that the Forth pseudo-variable **<PHASE>** had immediately before a step is begun can be seen during that step by all cells through this neighbor. Similarly for **PHASE'**. **PHASES** is their joint version. Cf. **&PHASE**.

PL>FILE (n) : **CAM-BUF**

Used with **OPEN-PATTERN** . See **FILE>PL** .

PL>PB see **AND>PL**

PL>TEMP (n) : **CAM-BUF**

Moves data from the active region of plane *n* to the temporary-buffer area. See **/-PL**.

PLANE-OPS **VOCABULARY CAM-KEYS**

Vocabulary containing control-panel commands for operations on CAM's bit-planes (cf. Chapter 3).

PLN0 (-- bit) **=NEW CAM-HOOD**
PLN1
PLN2
PLN3
PLNA (-- 2bits) : **CAM-HOOD**
PLNB

The current value, in the entry under construction, of the bit(s) to be sent to the correponding table column(s). See **>PLN0** and **MAKE-TABLE**.

PLS>PBS see **AND>PL**

PRA (-- offset) **EQU CAM-IO**

Offset of "Plane row address" register. See **PCA**.

PUT-DATA (n) : **CAM-BUF**

See **GET-DATA** , and **REWIND** . Write *n* to next location of active data file and autoincrement **DATA-PTR** .

R-INIT : **CAM-BUF**

Initializes the random number generator to a standard state.

R/FLY (data-buf.seg data-buf.offs n ptr-buf #items len) : **CAM-IO**

Read CAM data on the fly. Used to read plane data during vertical-blank interrupt without affecting stepping or display. Along with **W/FLY**, this is used to produce the dot and cage cursors.

The pointer buffer **ptr-buf** located within the **FORTH-SEG** contains one row/column pointer for each data item to be read. As each data item is read, **len** bytes are appended to the data buffer which starts at **data-buf.seg:data-buf.offs**. All items are read from the same plane *n*.

The arguments and operation of `W/FLY` (write on the fly) are similar—only the direction of data movement between planes and data buffer is reversed. In both cases, the high-order byte of `#items` is used as an additional argument. If this byte is zero, the transfer of the color maps for all CAM's from the shadow registers is suppressed before the step during which the `R/FLY` or `W/FLY` operation takes place, to give extra time during the interrupt for reading or writing plane data. The maximum number of items which can be read or written during a single interrupt is 10 (hex).

See also `DOT`, `B@CELL`, and `PL>PB`. Cumulative information about plane contents can also be obtained without slowing down the updating or hindering the display using the event counters (see `EVENT-COUNT`).

`REG-TABS` : `CAM-HOOD`

Used to select the regular tables for both CAM-A and CAM-B. This is useful in a run cycle—see also `AUX-TABS`, `<TAB-A>` and `MAKE-CYCLE`.

`RESET-CAMS` : `CAM-STEP`

This word is called by `NEW-EXPERIMENT` and by `DISPLAYS`. It resets all CAMs to a standard state, by doing the following:

- setting up the default neighborhoods `N/USER` and `&/USER`
- setting up the default tables (zero for all planes – see `MAKE-TABLE`)
- setting up the default run cycle (to `STEP`; see `MAKE-CYCLE`)
- setting up the default color map (to `STD-MAP`; see `MAKE-CMAP`)
- clearing all planes (but not the plane buffers) to zeros
- setting up the default number of lines in a frame (see `LONG?`).
- selecting CAM 0 (the master CAM) as the current one (see `CAM-SELECT`)

`REWIND` : `CAM-BUF`

Used to reset `DATA-PTR` to zero. See `OPEN-DATA` .

`RND ( -- n)` `CODE` `CAM-BUF`

Returns a 16-bit random number.

`RND>BUF ( n dest.seg dest.offs len)` `CODE` `CAM-BUF`

Fill a buffer of the given size, at the given position, with bits that have a probability of $n/65536$ of being a `1`. (n is the expected number of `1`s for a buffer the size of a bit plane). Note that `len` is in bytes.

`RND>PL` see `NOT-PL`

`ROW ( -- 2bits)` : `CAM-HOOD`

Neighbor in the `N/XPAND` neighborhood. Current cell's row position on the screen, mod 4.

`S.EAST S.WEST` see `N.EAST`

`S/S-PL` see `NOT-PL`

`SEG= ( seg1 offs1 seg2 offs2 len -- flag)` `CODE` `CAM-KEYS`

Compares two strings of length `len`, starting at the given segments and offsets—returns a `TRUE` flag on the stack if the strings are identical.

SEND-SHIFTS (hold.mask) : CAM-STEP

Overwrites the lookup tables of all CAMs with a "service" rule that is used for shifting the bit-planes. Usually used within a BEGIN-SERVICE-STEPS ...END-SERVICE-STEPS pair—which takes care of saving and then restoring the current rule.

A rule that depends on &PHASES is sent to all CAMs; depending on the value of these phases, we get different shift directions (see SHIFT). Only the regular tables are used for shifting; the auxiliary tables are given the value that the selected CAM's table had, to allow SHOW-FUNCTION (see) to keep working even during shifts. (Actually, the auxiliary tables sent by SEND-SHIFTS are constructed from the table saved by BEGIN-SERVICE-STEPS (see also TSAV-SEG); if SEND-SHIFTS is not being used within the "SERVICE-STEPS" construct, these auxiliary tables will be invalid.)

This is a lower-level word than SHIFTS. See the description of the hold.mask under SHIFTS; see also SHIFT.

SET-CCR (mask) CODE CAM-IO

Uses the mask to set all indicated bits of the CCR to 1 (all those which have 1s in corresponding positions in the mask). From most to least significant, the CCR bits have the names USETAB, DPYRQ, VGLUE, HGLUE, LDTBL, INTENB, DPYNRM, and CAMOUT; most of these are documented in this glossary. Only VGLUE and HGLUE will be typically controlled by explicitly using SET-CCR. See also C@CAM and CLR-CCR.

SHAD-PTR (-- addr) VARIABLE CAM-IO

Pointer to the shadow registers corresponding to the selected CAM. See CAM-SELECT.

SHAD-SEG (-- segment) : CAM-IO

Returns on the stack the value of selected CAM's shadow-memory segment. This is used to access CAM's shadow variables directly, rather than via the effects of words such as N/MOORE, EVEN-GRID, MAKE-CMAP, etc. See also STP0-SEG and STP1-SEG.

SHIFT (NSWE.mask) : CAM-KEYS

Assumes that the shift-tables have been downloaded (see SEND-SHIFTS). Causes a shift of one position in the direction indicated by the shift mask. For example, 8 indicates a shift up of all CAMs, 4 a shift down, 2 a shift left, 1 a shift right, 9 a shift up-and-right, etc. All CAMs are shifted simultaneously.

SHIFTS (delta-row delta-col hold.mask) : CAM-KEYS

Loads the shift tables and performs the indicated shifts of all bit-planes which aren't held fixed by the hold.mask argument: 1s in the mask cause corresponding planes (bit position 0 corresponds to plane 0, etc.) to be held fixed. Note that MOUSE>ORG produces arguments in the correct format for SHIFTS.

SHOW-CAM (n) : CAM-IO

Assumes that all CAM video-outputs are daisy-chained and connected to a single monitor. SHOW-CAM will cause the indicated CAM's internal video signal to be the one passed through the chain to the monitor. This word doesn't change the selected CAM (see CAM-SELECT)—it only controls which output is visible.

SHOW-FUNCTION : CAM-HOOD

Direct the output of the auxiliary tables to be taken as the input to the color map, in place of the four bits of the center cell. See also SHOW-STATE.

SHOW-STATE : CAM-HOOD

Cause the four bit cell values to be used directly as the inputs to the color map—see **SHOW-FUNCTION**.

SIG (n -- +1|0|-1) : CAM-EDIT

Returns the sign of the number on the stack—0 gives 0.

SOUTH SOUTH' SOUTHS see **NORTH**

SRQ (-- offset) EQU CAM-IO

Pointer to the hardware "step request" register. Since step requesting is normally mediated by the CAM interrupt service routine, this register would only be directly written to in order to test the hardware. See **C!CAM**, **CAM-BASE**.

SSR (-- offset) EQU CAM-IO

This points to the hardware "step status" register, which is normally monitored by the CAM interrupt service routine. It can be read directly using **C@CAM**, as in **SSR C@CAM**. See also **CAM-BASE**.

The low-order bit is a diagnostic bit, **DIAG**, used to test the CAM hardware. The high-order bit is the "event detected" bit, **EVENT**, which indicates whether or not the intensity output of the color map was *ever* on during the most recent step (this is used by **EVENT-STOP** and by **EVENT-COUNT**). The other bits reflect the dynamic progress of the step as it runs (Is there a step running? waiting to run? is the display on? are we within the active portion of the line right now? etc.).

STD-MAP : CAM-HOOD

Standard color map, which assigns the three primary colors to CAM-A. This map is useful when we are primarily interested in planes 0 and 1; planes 2 and 3 are only used for highlighting.

STEP : CAM-STEP

Triggers one CAM cell-updating step. All of the CAM parameters should have been set up before this (such as neighborhood, phases, spatial origin, color map, etc.—see **N/MOORE** etc., **<PHASE>** etc.). Execution of the Forth word following **STEP** will not proceed until shadow variables containing these CAM parameters have been copied to CAM, and the actual step has begun (see **PEND** and **WAIT-FOR-PEND**, which **STEP** makes use of).

In order to allow steps to run at full speed (every 60-th of a second) the parameters for the next step must be set up while the current step is running. This must be kept in mind when reading event-count information—see **EVENT-COUNT**. If no instruction to run an active (updating) step has been sent to CAM before the step that is running is completed, an idle (display-only) step will be scheduled.

Used in conjunction with a run cycle (see **MAKE-CYCLE**), **STEP** causes a coroutine exit to the main program—the run cycle routine is resumed by executing **NEXT-STEP**. If **STEP** is executed in the main program, no exit occurs. **IDLE** behaves like **STEP** in all respects (including the coroutine exit) except that no updating occurs: all shadow parameters (in particular, color maps and neighborhood assignments) are transfered and an idle step is begun before execution of the instruction following **IDLE** occurs. When multiple CAMs are used together, all CAMs respond as a group to a stepping request: they all either step or idle at the same time.

See also **STEPS**, **STEP-NUMBER**, **EVENT-HANDLER**, **BEGIN-SERVICE-STEPS**, and **SHIFTS**.

STEP-BASE (-- addr) VARIABLE CAM-INT

Contains the base address for the step status and control information for all CAMs (including shadow registers). It is changed by **BEGIN-SERVICE-STEPS** to use a distinct set of control information for service steps. See **STP1-SEG** and **STEP-SEG**.

STEP-COUNT (-- offset) EQU CAM-INT

Offset within the **STEP-SEG** of a double-number system variable which is incremented each time an active step (as opposed to an idle step in which display but no updating occured) is completed. This counter is incremented by the CAM interrupt routine.

STEP-NUMBER (-- addr) 2VARIABLE CAM-STEP

This double-number VARIABLE is incremented each time the word **STEP** is executed. It is used by the CAM control-panel program to run a specified number of steps by starting it with a negative value: when this number reaches zero, **STEP** turns off the **STEPPING?** flag (see **STOP**).

When running a reversible rule, this stopping mechanism can be quite convenient. Start the count at zero, and negate it when you reverse direction—when you reach zero (the starting configuration) the stepping will halt.

This variable is maintained by the word **STEP**—it is distinct from the doubleword **STEP-COUNT** variable within the **STEP-SEG**, which is incremented by the CAM interrupt service routine each time an active step is started. See also **-STEP#**.

STEP-PENDING (-- offset) EQU CAM-INT

This is the offset into the **STEP-SEG** of a one-byte system variable. It is not examined by the interrupt, but it is cleared each time shadow information is copied to CAM; this lets the foreground routine know that its safe to change the shadow data. This flag is used as a semaphore by **STEP** and **IDLE**, which set this flag and then wait for it to clear before allowing execution to proceed.

STEP-SEG (-- segment) : CAM-IO

Returns on the stack the segment containing global step information which applies to all CAMs (such as the **STEP-PENDING** flag). See also **STP0-SEG**, **STEP-TYPE**, **STEP-COUNT**, **EVENT-MASK**, **EVENT-TYPE**, **EVENT-LEN**, and **EVENT-STOP**.

STEP-TYPE (-- offset) EQU CAM-INT

This is the offset into the **STEP-SEG** of a one-byte system variable. Whenever shadow data is copied to CAM by the interrupt service routine, this byte is copied to the **SRQ** register of the master CAM. If the high bit of this register is set, an active (updating) step is begun; if this bit is a zero, an idle (display) step occurs, with the new parameters transferred from the shadow registers. **STEP** and **IDLE** use this mechanism. See also **STEP-PENDING**.

STEPPING? (-- addr) VARIABLE CAM-STEP

This variable is used as a flag by the control-panel program to indicate whether CAM is running or stopped. It can be set or cleared with the words **GO** or **STOP**. It is also cleared by the word **STEP** if the doubleword variable **STEP-NUMBER**, which is incremented by **STEP**, becomes zero.

STEPRQ (-- mask) EQU CAM-IO

Step request bit of the step status register—see **SSR**.

STEPS (n) : CAM-STEP

Executes the word **STEP** *n* times.

`STOP` : `CAM-KEYS`

Sets a flag (see `STEPPING?`) used by the control panel to indicate that CAM is not running. `STOP` is used by `Stop.running`, which is invoked by $\boxed{\text{SPACE}}$ from the control panel; any user-defined key can cause stepping to stop by executing this word. `STOP` can even be used inside of a run cycle (see `MAKE-CYCLE`) to cause an experiment to stop itself and wait for the experimenter to restart it.

`STORES ( n dest.seg offs count)` `CODE` `CAM-BUF`
`ANDS`
`ORS`
`XORS`

`STORES` stores the 16-bit value *n* in `count` consecutive memory words.

`ANDS` AND's this value with the contents of those words. Similarly for `ORS` and `XORS`.

`STP0-SEG ( -- segment)` `SEGMENT` `CAM-BUF`
`STP1-SEG`

This is the area which normally contains the global step information and shadow variables used during CAM stepping (this is the normal value returned by `STEP-SEG`). While service steps are being run (see `BEGIN-SERVICE-STEPS`), the `STP1-SEG` becomes the step segment instead—all counts accumulated during these steps accumulate there, and all other changes are made there. This makes it easy to go back to the state the machine had before the service steps began.

`T*BUF` : `CAM-BUF`

Exchanges the content of the lookup tables and the table-buffer.

`T/B-PL` see `NOT-PL`

`TAA ( -- offset)` `EQU` `CAM-IO`
`TBA`

`TAA` is the offset of the "CAM-A table-address source select" register within each CAM's shadow segment—see `CAM-A`, `N/MOORE` (and other neighborhood words) and `C!CAM`. Similarly for `TBA` and `CAM-B`.

`TAB>BUF` : `CAM-BUF`

Saves the lookup tables to their buffer.

`TAB>FILE` : `CAM-BUF`

Save entire lookup tables to current table file. Used with `OPEN-TABLE`.

`TBA` see `TAA`

`TBUF-SEG ( -- segment)` `SEGMENT` `CAM-BUF`

This is the 4K byte segment used by `TAB>BUF` and `BUF>TAB` to save and restore the contents of CAM's lookup tables. See also `TSAV-SEG`.

`TDAT ( -- offset)` `EQU` `CAM-IO`

Table data area within CAM hardware segment. Normally accessed using `MAKE-TABLE`, `TAB>BUF`, and `BUF>TAB`. Can also be accessed directly—see `C!CAM`.

`TDAT>BUF ( buf.seg buf.offs first.page #pages)` `CODE` `CAM-IO`

Low-level primitive for copying table data from CAM to the PC. The arguments have the same order and meaning as for `BUF>TDAT` (except of course that the buffer is a destination this time). See also `TAB>BUF`.

`TEMP-SEG ( -- segment)` SEGMENT CAM-BUF

Returns the segment address of the 8K temporary buffer.

`TEMP>PL ( n)` : CAM-BUF

Copy the contents of the temporary buffer to the active region of plane n. See `NOT-PL`.

`TICK` : CAM-LOAD

Delays the execution by ≈55 msec.

`TICKS ( n)` : CAM-LOAD

Delays the execution by ≈ $n \times 55$ msec.

`TRACE` : CAM-HOOD

`MAKE-TABLE TRACE` will cause the rule for plane 1 to logically OR its previous contents with the current contents of plane 0. Thus plane 1 will act like an infinite-persistence phosphor: once a bit has been "on" on plane 0, the trace will remain lit until cleared.

`TSAV-SEG ( -- segment)` SEGMENT CAM-STEP

This is the segment used by `BEGIN-SERVICE-STEPS` to save the contents of the tables for up to eight CAMs. `END-SERVICE-STEPS` restores the tables from this buffer.

Since this is a very transient buffer, needed only during service steps (such as shifts—see `SHIFTS` and `SEND-SHIFTS`), we put this above all other segments and dynamically allocate and deallocate it as needed. 4K bytes are allocated for every CAM that is present before the tables are saved, and deallocated whenever they are restored. Thus when `EXEC` is executed there is as much memory as possible for executing commands from DOS.

```
UL  ( -- bit)                                    : CAM-HOOD
UR
LL
LR
UL'
UR'
LL'
LR'
ULS  ( -- 2bits)
URS
LLS
LRS
```

Neighbors made available by the `N/MARG-HV` major neighborhood assignment. `UL` returns on the stack the value, for the case under consideration, of the upper-left cell in the Margolus neighborhood (plane 0 for CAM-A, 2 for CAM-B); `UR`, of the upper-right neighbor; `LL`, of lower-left; and `LR` of lower-right. (Cf. `CENTER` and `MAKE-TABLE`).

Similarly for the primed versions—all referring to planes 1 or 3, and the joint versions (e.g., `ULS`=`UL`+2×`UL'`).

USETAB (-- mask) EQU CAM-IO

This is the bit in the configuration control register which decides whether the input to the color map should come from the auxiliary tables or directly from the bit-planes. The words **SHOW-FUNCTION** and **SHOW-STATE** respectively set and clear this bit—see also **SET-CCR**.

V1 (-- bit) == CAM-HOOD

Pseudo-neighbor available as part of the **N/XPAND** neighborhood. Returns the second-least significant bit of the vertical position of the center cell. See also **VERT** and **ROW**.

VAL>PL see **NOT-PL**

VERT see **HORZ**

VGLUE (-- mask) EQU CAM-IO

This bit in the configuration control register controls whether vertical gluing of several CAM's into one longer CAM (via the vertical-glue connector) is enabled. See **HGLUE**.

VRUN (-- mask) EQU CAM-IO

"Vertical run" bit of the step status register—see **SSR**.

W/FLY (data-buf.seg data-buf.offs n ptr-buf #items len) : CAM-IO

Write on the fly—see **R/FLY**.

WAIT-FOR-IDLE CODE CAM-IO

Wait until CAM is inactive—no step in progress and none about to start.

WAIT-FOR-PEND : CAM-IO

Waits until the **STEP-PENDING** shadow variable is reset to **FALSE**, indicating that CAM has been serviced since **STEP-PENDING** was set to **TRUE**. See **PEND**.

WEST WEST' WESTS see **NORTH**

WHOLE-AREA : CAM-BUF

Sets the whole screen to be the active region for plane operations—see **AREA**.

X (-- addr) VARIABLE CAM-HOOD

Index variable used during table generation. Compilation variables return bits of this index variable, which runs through all possible cases as it takes on values from 0 to 4095. Cf. **Y**.

XOR>PL see **AND>PL**

XORS see **STORES**

Y (-- addr) VARIABLE CAM-HOOD

A variable used during table generation as a scratchpad for the current entry. The low eight bits correspond to the eight lookup-table columns (four for CAM-A and four for CAM-B). Cf. tt X.

\-PL see **NOT-PL**

G.2 Notable F83 words

`!L ( n segment offset)` — CODE CPU8086

"Long store:" store *n* at *segment:offset*. While ordinary Forth addresses refer to the 64K segment in which the Forth system resides, this can access any location in the PC address space.

`'USED name` — : PC

This is a utility function, used for finding all definitions that make use of the indicated word. It works by searching through the Forth dictionary, looking for any pointers to the given word. Since the search is crude (it just looks for any 16-bit value in the dictionary that matches the given word's code-field address), some extra matches may be returned—but no true matches will be missed.

See also `VOC`.

`*BREAK*` — : PC

This acts like an execution vector (see `DEFER`), and is given control when $\boxed{\text{BREAK}}$ is pressed. The "break" always occurs *between* two Forth words (it never interrupts a machine-language routine). The action of `*BREAK*` can be modified using the word `IS`. For example, `' TRAP IS *BREAK*` would cause the word `TRAP` to be executed whenever the $\boxed{\text{BREAK}}$ key is invoked (`TRAP` is in fact the normal word executed by `*BREAK*`). See also `BREAK-ENABLE` and `BREAK-DISABLE`.

`+THRU ( n0 n1)` — : EXTEND86

From the current "from" file load screens $n + n_0$ through $n + n_1$, where n is the screen where `+THRU` appears (cf. `THRU`).

`-->` — : EXTEND86

Load next screen, skipping the rest of the current one.

`...` — : PC

Used to re-enter the screen editor from Forth where you last left off. If a new current-file has been established (see `USING`) this file will be edited instead. See also `EDIT`, `M`, and `FIX`.

`.NAME ( nfa)` — : KERNEL86

Print word name, given its name-field address. See `>NAME`.

`2!L ( d seg offs)` — : PC

Double-number "long store" (see `!L`).

`2* ( n -- 2*n)` — CODE KERNEL86

Multiplies by 2 by shifting.

`2@L ( seg offs -- d)` — : PC

Double-number "long fetch" (see `@L`).

`2^N ( n -- 2^n)` — CODE PC

Given a number from 0 to 15 (decimal) this word returns a 16-bit value with the *n*-th bit set to one, the other bits set to zero.

4* **(n -- 4*n)** CODE KERNEL86

Multiply by 4 by shifting.

8* **(n -- 8*n)** CODE KERNEL86

Multiplies by 8 by shifting.

:: : UTILITY

"On-the-fly" compiler, used from the interpreter to execute a phrase containing flow-control directives such as `IF ... THEN` etc., which are not available in the interpretive mode. If you type

```
:: 7F 20 DO I EMIT LOOP ;
```

the interpreter will compile a nameless word identical to `PRINT-ASCII` of Section 5.10 and execute it—without taking any permanent space in the dictionary.

>BODY **(cfa -- pfa)** : KERNEL86

Given the code-field address of a word, returns its parameter-field address, i.e., the address of the first (and possibly the only) data or program cell.

>LINK **(cfa -- lfa)** : KERNEL86

Given the code-field address of a word, returns its link-field address (within each vocabulary, all words form a linked list).

>NAME **(cfa -- nfa)** : KERNEL86

Given the code-field address of a word, returns its name-field address.

>VIEW **(cfa -- vfa)** : KERNEL86

Given the code-field address of a word, returns its view-field address.

@L **(segment offset -- n)** CODE CPU8086

"Long fetch." Fetch *n* from *segment:offset*. While ordinary Forth addresses refer to the 64K segment in which the Forth system resides, this can access any location in the PC address space.

A-EMIT **(char)** : PC

This is the normal function pointed to by the `DEFER`red word `EMIT`. It outputs char to the display screen with the current attribute (see also `TYPER`). If the variable `PRINTING?` is set (see `ON`) the same character is also sent to the standard printer.

ALSO : EXTEND86

Vocabulary search order specifier. See Section 5.17.

ARRAY name **(n)** : PC

Defines, under the given name, a one-dimensional array of Forth cells, consisting of *n* entries. When executed with a number *i* on the stack, the word `name` will return the address of its *i*-th cell.

AT **(col row)** : PC

Positions the display cursor on the PC at the given row and column.

`AUTO-X ( -- addr)` VARIABLE PC

Controls the auto-extend/auto-shrink feature of the Forth screen editor: if non-zero, this feature is enabled.

`BETWEEN ( n min max -- f)` : KERNEL86

Returns a `TRUE` flag if *n* is in the closed interval `min` ≤ `n` ≤ `max`.

`BLINK ( -- addr)` VARIABLE PC

If this variable is set (see `ON`), subsequently `EMIT`ted characters will blink.

`BLOCK ( n -- addr)` : KERNEL86

Makes sure that the *n*-th 1024-byte block of data from the current file is present in a *block buffer* in memory (copying it from disk if necessary), and returns the address of this buffer.

`BODY> ( pfa -- cfa)` : KERNEL86

Given parameter-field address of a word, returns its code-field address (cf. `>BODY`).

`BREAK-DISABLE` : PC

Temporarily suppress the action of the [BREAK] key—see `BREAK-ENABLE` and `*BREAK*`.

`BREAK-ENABLE` : PC

Restore the action of the [BREAK] key. If the key was pressed while the break function was disabled, its effect will take place now. See `BREAK-DISABLE` and `*BREAK*`.

`C!L ( n segment offset)` CODE CPU8086

"Long store" of one byte (cf. `!L`).

`C@L ( segment offset -- n)` CODE CPU8086

"Long fetch" of one byte (cf. `!@`).

`CAPACITY ( -- n)` : KERNEL86

Returns the size (in 1024-byte blocks) of the current file.

`CAPS ( -- addr)` VARIABLE KERNEL86

When this `VARIABLE` is set, characters read by the interpreter are converted to upper case.

`CARRAY ( len)` : PC

Like `ARRAY`, but the array elements are bytes, to be accessed by `C@` and `C!`.

`CENTRONICS` : UTILITY

Similar to `EPSON`, for a Centronics-like printer.

`COM` CREATE PC

Exits to the DOS command interpreter. Upon return, continues with the following instruction, with the stack unchanged. If you return to Forth with the directory changed, Forth will complain when you try to continue editing files in the old directory. Defined using `>DOS` (which in turn calls `EXEC`) as follows:

```
    CREATE COM  ," "  >DOS
```

To see how to define Forth words which invoke DOS commands, see >DOS. Used by DOS... in previous section.

CONVEY (first last) : UTILITY

Copy a consecutive range of screens (see Section 7.6). A second version of this word exists in the SHADOW vocabulary for moving screens along with their documentation 'shadows.' The arguments are the same as above—you give the command as if you were just moving the code screens.

COPY (from to) : UTILITY

Copy the contents of a screen to another screen, updating the disk to reflect the change.

CREATE-FILE filename (#blocks) : EXTEND86

Create a file of given size (in blocks) and given name. If the file already exists, the old version will be deleted.

DARK : PC

Clear the PC screen (to the default attribute).

DEBUG name : UTILITY

Setup to trace the execution of the given word. Tracing will subsequently occur whenever the given word is executed. After each instruction comprising the word is executed, the name of the instruction and the contents of the stack at that point will be displayed; hit SPACE to continue with the execution.

Tracing can be turned off by executing UNBUG, or by pressing $\boxed{\text{BREAK}}$. Tracing can be temporarily suspended by pressing the F key, which puts you in Forth and lets you examine variables. Resume tracing using RESUME.

DEFER name : KERNEL86

Defining word used to construct a vectored-execution word. The vector is set using the word IS, and can be examined using the word SEE. For example, DEFER EMIT defines the word EMIT to be such a deferred word, and

```
' A-EMIT IS EMIT
```

vectors EMIT to execute A-EMIT.

DIR filespec : EXTEND86

List all files in the current directory matching the given file specification (example: DIR *.PAT) . If no specification is given, the whole directory is listed. See also USING.

DISCARD : KERNEL86

The opposite of UPDATE: marks a block buffer as reusable at any time without saving its contents to disk.

>DOS : PC

Used to define Forth words which execute commands at the DOS level. For example,

```
CREATE BACKUP  ," /c copy c:*.exp a:"  >DOS
```

would create a Forth word **BACKUP** which, when executed, would copy all experiment files from drive C: to drive A: to produce a backup. Any executable file can be invoked in a similar manner. **>DOS** is a defining word which causes the most recently defined word to invoke **EXEC** on its parameters when it is executed. **COM** is defined using **>DOS**. See also **DOS...** in the CAM section of the glossary.

DRIVE drivespec (-- n) : **EXTEND86**

Converts a drive specification into a number. For example, **DRIVE B:** leaves the number 2 on the stack. See **IS-HOME-OF**.

DUMP (addr len) : **UTILITY**

Produces a formatted dump of the specified number of memory bytes starting from the given address. See **LDUMP**.

EDIT (n) : **PC**

Begin editing screen *n* of the current file. Note that if editing is initiated from Forth, when you leave the editor (by pressing Esc) you return to Forth.

END **DEFER KERNEL86**

This **DEFER**red word is executed by **BYE** before exiting to DOS. It normally points to a routine which restores the various interrupt vectors that the Forth system program used, and performs other similar clean-up activities.

EPSON : **UTILITY**

Send the character codes necessary to set up an Epson-like printer (such as the standard-issue PC printer) for compressed printing mode. Used by **SHOW** and **LISTING** in order to produce compact listings. See also **CENTRONICS** and **INIT-PR**.

EXPECTR (addr len) **DEFER PC**

Same as Forth's standard **EXPECT**, but complements the reverse-video attribute while it is active, to highlight the input. See **REVERSE**.

FIX name : **PC**

Find the source for the given word, and enter the editor at the appropriate point to edit it. If the word is a system word (part of **F83** or of **CAM**), the appropriate sources must be on-line and on the expected drive (see **IS-HOME-OF**). If the sources aren't online, **FIX** will at least tell you what file and screen the sources were in. If the word is one that you have defined yourself, then it will be found only if it is in the current file (but cf. Section 6.1) . See also **VIEW**, **EDIT**, and **M**.

FLIP (n -- n') **CODE KERNEL86**

Swap high and low byte in given 16-bit quantity.

FORGET name **CODE KERNEL86**

Look up the given word in the current vocabulary, and forget all entries that were made in the dictionary since that word was defined (including **name** itself)

FORM-FEED : **UTILITY**

Send CR and FF codes for printer form feed.

FORTH-SEG (-- seg) **CODE KERNEL86**

Returns the address (in 16-byte "paragraphs") of the segment containing the Forth system proper. This is useful for example in intersegment moves. See also `LMOVE`, `@L`, etc.

`FROM filename` : EXTEND86

Used for loading one file from within another (see also `INCLUDE`) and for copying screens from one file to another, in association with `CONVEY`. Opens the named file as the current input (or "from") file.

`G` CODE PC

Used when Forth is running under the `DEBUG.COM` program, to transfer control to debug by issuing an interrupt 3. If interrupt 3 is not being trapped, `G` does nothing.

`I'M name` : PC

This is used to set the 3-character identifier that appears in the upper right corner of screens modified from the editor. Only screens which have a comment line at the top (one starting with a backslash) are marked.

`I/NORM` : PC

Restore `KEY` and `KEY?` to their default values, in case these `DEFER`red words have been redefined in order to redirect input.

`INIT-PR` DEFER UTILITY

`DEFER`red word, executed to put printer in compressed-text mode. See `EPSON` and `CENTRONICS`.

`INSO ( -- addr)` VARIABLE PC

Variable which controls whether the insert mode is on (non-zero value) or off each time you enter the editor.

`INS-SCR ( n)` : PC

Insert *n* blank screens in the current file, immediately before the screen last accessed by the editor.

`INTENSITY ( -- addr)` VARIABLE PC

This variable controls whether or not subsequently `EMIT`ted characters will be intensified—see `BLINK`.

`IS name ( n)` : KERNEL86

The value on the stack is stored in the data cell associated with the word whose name follows. If this is a VARIABLE, CONSTANT, or DEFERred word, the data cell is simply the word's parameter field. Typically, IS is used for changing the value of a CONSTANT or a pseudo-variable, or for changing the routine pointed to by a DEFERred word.

`IS-HOME-OF ( n)` : EXTEND86

Used to reset the drive specification used by `FIX` and `VIEW` to find a source file. For example,

```
DRIVE B: IS-HOME-OF KERNEL86.4TH
```

can be used to tell these words that this file is now to be found on drive `B:`—rather than whatever drive the file was on when the system was last compiled (usually `C:`).

`KNAME ( pckey -- str|0)` : PC

Used to convert a key code returned by **PCKEY** (which is the default **KEY** routine) into a pointer to a counted string that names the key. If no name has been assigned to this key, 0 is returned.

L : UTILITY
N
P
T
M

L lists the most recently **VIEW**ed screen again, **N** the next, **P** the previous; **T** toggles the shadow twin; and **M** calls the editor for modifying the current view-screen. See **VIEW**.

L>NAME (lfa -- nfa) : KERNEL86

Given the link-field address of a word, returns its name-field address (cf. **>LINK**).

LDUMP (seg offs len) : UTILITY

Like **DUMP**, but with a "long" address consisting of segment:offset.

LINK> (lfa -- cfa) : KERNEL86

Given the link-field address of a word, returns its code-field address (cf. **>LINK**).

LISTING : UTILITY

Used to produce a complete listing of the current file, assumed to have shadow screens. For other files, see **SHOW**.

LMOVE (src.seg offs dest.seg offs #byts) CODE PC

Intersegment move of data from source-segment:offset to destination.

LOWR : PC

Put the PC in lower-case shift state.

M see L

MANY : UTILITY

If you type a single line of the form

    phrase MANY

the given phrase will be interpreted over and over. Hitting any key aborts the loop.

MARK name : EXTEND86

Define a word which, when called, will **FORGET** everything up to but not including the word itself. If you are debugging an application and expect to have to load its source file over and over before it reaches its final form, it's inconvenient to have to **FORGET** by hand the latest version every time you load a new one. Enter **FOOLING** as the first line of your file, type **MARK FOOLING** at the keyboard to put your place marker in the dictionary, and every time you load the file the dictionary will be cleaned up to there as the first thing.

The CAM words **NEWX** and **NEW-EXPERIMENT** make use of this mechanism to clear out old definitions.

N see L

N>LINK (nfa -- lfa) : KERNEL86

Given the name-field address of a word, returns its link-field address (cf. **>LINK**).

`NAME>  ( nfa -- cfa)` : KERNEL86

Given a name-field address, return the corresponding code-field address.

`NNUM` : PC

Set the PC to no-NUMLOCK mode.

`NORMAL` DEFER PC

Reset display attributes and keyboard shift states to a default value.

`NUMBER  ( str -- d)` DEFER KERNEL86

Attempts to convert the given string to a number, always returned in double-number form.

`NUML` : PC

Put the PC into NUMLOCK mode.

`O/NORM` : PC

Reset `EMIT` to point to `A-EMIT`, the normal output routine.

`OFF  ( addr)` : EXTEND86
`ON`

`ON` sets the data cell at addr to `TRUE` (=FFFF); `OFF` sets it to `FALSE` (=0000);

`ONLY` : EXTEND86

Vocabulary search order specifier. See Section 5.17.

`P` see L

`PAGE` : UTILITY

Set up the printer to go to a new page, and appropriately increment or reset the internal page, line, and character counters.

`PCKEY  ( -- char.code)` : PC

The normal input routine, pointed to by `KEY`. All normal ASCII characters are returned unchanged; special PC characters are returned as negative numbers (the negative of the non-zero part of their two-byte codes). Thus all non-printing characters are numerically less than the SPACE character. See also `KNAME` and `PEEK`.

`PEEK  ( -- val)` : PC

This acts the same as `PCKEY`, but doesn't remove the character from the input buffer. If no character is waiting in the input buffer, returns a 0.

`PREVIOUS` : EXTEND86

Vocabulary search order specifier. See Section 5.17.

`RECURSE` : KERNEL86

Within a COLON-compiled word, this is a recursive call to the word itself. See `RECURSIVE`.

`RECURSIVE` : KERNEL86

Used within the definition of a COLON-compiled word, say `MY-WORD`, to let any subsequent occurrences of `MY-WORD` in the definition refer to the very word under compilation rather than a previous word of the same name (if any) already in the dictionary (cf. Section 5.3, and footnote 1 of Chapter C). See `RECURSE`.

RESUME : **UTILITY**

Can be used to continue a Forth trace (see **DEBUG**) after temporarily suspending it to re-enter Forth to examine variables, etc.

REVERSE (-- addr) **VARIABLE PC**

When **REVERSE** is set, subsequent characters **EMIT**ted are in reverse-video. See **BLINK**.

ROOT **VOCABULARY EXTEND86**

This is the *root* vocabulary, included in every search order (but cf. **SEAL**). See Section 5.17.

S-KEYS (-- addr) **VARIABLE PC**

When this variable is **TRUE** (=FFFF), all shift keys behave normally. When this variable is set to F (hex) the action of all shift-lock keys (CapsLock, ScrollLock, NumLock, and Ins) is suppressed.

SAVE-SYSTEM filename : **EXTEND86**

Save "as is" to a file an executable version of the currently running Forth (or CAM Forth) system, with its entire dictionary up to **HERE**; the values of variables and other changeable data cells in the dictionary are saved as well. Data outside the dictionary (such as disk-block buffers) are not saved. Useful for saving minor changes, default settings, etc. without having to recompile the system. F83 takes approximately 31K bytes; CAM Forth approximately 58K.

Typical usage:

```
SAVE-SYSTEM CAM.EXE
```

SCAN (addr len char -- addr' len') **CODE KERNEL86**

In a buffer at position addr, for a length len, scan for the first occurrence of char, returning the position where it was found, and how many characters remain in the buffer from that position on.

SEAL : **EXTEND86**

Vocabulary search order specifier (cf. Section 5.17). It removes the **ROOT** vocabulary from the search order, thus preventing further manipulation of the search order. Used for protected or turn-key applications.

SEE name : **UTILITY**

Decompile the indicated word. This restricted ability to look at word definitions is available without reference to source code. See also **VIEW** and **VOC**.

SEGMENT name (n) : **PC**

Used to reserve space in the PC's memory outside of the segment in which the CAM Forth code resides, for data buffers. Makes the word that follows a Forth constant that returns the segment of an allocated space of *n* contiguous bytes. **SEGMENT** calls DOS function 4A (hex) to inform DOS of the new allocation. For instance, **500 SEGMENT EX-SEG** reserves 500 bytes and sets **EX-SEG** to the memory segment value so that the first byte of the reserved area is at location **EX-SEG:0000**. If n=0, a 64K segment is allocated.

SEG-TOT (-- addr) **VARIABLE PC**

This variable contains the total amount of memory space (in 16-byte paragraphs) used by the Forth system, including the segment where the system itself resides. It is incremented by **SEGMENT**.

SET-BASE — DEFER PC

DEFERred word which is used to set the default base. Usually points to either **DECIMAL** or **HEX**.

SET-MEM (#paragraphs) — : PC

Inform DOS of the current memory allocation that Forth is using. See **SEGMENT**.

SHADOW — VOCABULARY UTILITY

Vocabulary used in conjunction with documentation screens ("shadow" screens). **CONVEY** has a shadow-version which copies both code and shadows simultaneously, when given the parameters for copying the code alone. **SHOW** also has a shadow version, which is used in **LISTING**.

SHOW (first last) — : UTILITY

Listing utility—see Section 7.5.

SKIP (addr len char -- addr' len') — CODE KERNEL86

Scan until the first character that doesn't match **char** (cf. **SCAN**).

SLOW-KEY — : PC

Used with CAM key definitions that are slow to execute—this clears the keyboard buffer so that repeats of the slow key aren't inadvertently accumulated.

T see L

THRU (n0 n1) — : EXTEND86

From the current input (or "from") file, load screens n_0 through n_1.

TIMES (n) — : UTILITY

If you type a single line of the form

```
phrase 7 TIMES
```

the given phrase will be interpreted seven times (cf. **MANY**).

TO 1st.dest (1st.source last.source -- 1st.source last.source) — : UTILITY

Used with **CONVEY**.

TRAP — DEFER KERNEL86

Executed as part of the processing whenever an error or a keyboard $\boxed{\text{BREAK}}$ occurs. This is a DEFERred word.

TYPER (addr len) — : PC

EMIT's a string with the **REVERSE** attribute inverted—used for highlighting.

UEXT (-- addr) — CREATE PC

Points to the default filename and extension used by **USING**.

UM/MOD (d n -- remainder quotient) — CODE KERNEL86

Forth's unsigned-division primitive—a double number is divided by a single number to produce single remainder and quotient.

UNDERLINE (-- addr) VARIABLE PC

When this VARIABLE is set, subsequently EMITted characters will have their *underline* attribute set.

UPDATE : KERNEL86

Marks the current block buffer (see BLOCK) as updated, so that it will be saved to disk.

UPPR : PC

Put the PC in CAPS-LOCK mode.

USING filename : PC

This word is to be used to activate a file for read/write operations and editing directly from Forth. The related F83 word OPEN should be avoided, since it interacts badly with FORGET and NEW-EXPERIMENT.

The default extension in F83 is 4TH, but in the CAM program EXP becomes the default. If no filename is specified, or if the name contains any wildcard characters, a selctive directory listing is given. If the file doesn't exist, USING will offer to create it. For example, USING RHINO opens the file RHINO.EXP for read/write. If an extension is given, it is of course used. See also UEXT.

V-EMIT (repetitions char) : PC

Calls the ROM BIOS video-I/O routine to emit a number of repetitions of a given character, with the current attribute (see BLINK, A-EMIT, etc). Output is very fast, and leaves the cursor back where it started.

VIEW name : UTILITY

This is used for locating and examining source code (see also FIX). VIEW simply lists the screen containing the definition, and makes this file and screen respectively the current *view* file and the current screen. Only the words L, N, P, T, and M make reference to this view-file; the current file used for editing and loading is unaffected by VIEW.

VIEW> (vfa -- cfa) : KERNEL86

Given a view-field address, return the corresponding code-field address.

VOC name : PC

Searches all vocabularies for the given word; the name of each vocabulary that contains a matching entry is printed. If no vocabulary name is printed, the word is undefined.

Y/N (-- f) : PC

Prints the message "(Y/N)" and waits for a key to be pressed. If 'y' or 'Y' is pressed, returns true, else false.

{ (n) : UTILITY

Begins a CASE statement—a sort of computed subroutine call. The *n*-th Forth word in the list between the curly braces is executed (*n* is the number on the stack), and then execution continues with the word immediately following the list. An error is given if the argument is out of range. For example,

```
                                  : -ATTR ( n)
    { REVERSE UNDERLINE BLINK INTENSITY }
                      DUP @ NEGATE SWAP ! ;
```

This word would invert the value of one of four display attributes—the number on the stack tells which one.

All items within the curly braces must be defined Forth words. Note that numbers are not single Forth words, except for those that are defined as CONSTANT's; for convenience, the first sixteen hexadecimal digits are defined as constants (`0` through `F`). Like other words involving branching, this construct can only appear within a compiled word.

} : UTILITY

See {.

Appendix H

Summary of control panel functions

The highest level of the CAM program is a keyboard interpreter that turns the PC keyboard into a dedicated control panel for CAM. A list of the available control-panel commands can be obtained by looking at the various menus, printed by the $\boxed{\texttt{m}}$ key, as explained in Section 2.2. For reference, here we give a brief summary of all control-panel keys, following the order used by the five $\boxed{\texttt{m}}$ menus.

The special keys used by the screen editor for control functions are not part of the control panel and are listed in Chapter 6.

The heading of each of the following sections coincides with the name of the corresponding menu; this, in turn, is but the name of the Forth vocabulary which contains the commands listed in that menu. Similarly, in the listings each control-panel key is accompanied by the name of the Forth word that is executed when the corresponding key is hit; this name (which may slightly differ from the command memonics used in the rest of the manual) is immediately echoed on the PC screen when you hit a key.

H.0 GENERAL

The commands in the GENERAL menu perform functions of general utility that do not affect CAM. The digit keys 0 thru 9 can be used to construct arguments for other keys; after any non-digit key this argument is reset to "no argument," regardless of whether the non-digit key made use of an argument or not. Except when using $\boxed{\#}$, all numeric arguments to keys are read in decimal regardless of the default base (see SET-BASE in the glossary).

$\boxed{m}$ $\boxed{M}$	MENU Show the menu of menus. With a numeric argument, shows one of the submenus.
$\boxed{f}$ $\boxed{F}$	Forth... Leave the control panel and begin a dialogue directly with the Forth interpreter.
$\boxed{Esc}$	(ignored) If you are in Forth, or typing a text argument to any control panel key, this puts you at the top level of the control panel. If you are already at the top level of the control panel, this key is ignored.
$\boxed{\sim}$	DOS... Leave the control panel and begin a dialogue directly with a copy of the DOS command interpreter. Type exit to DOS to return to the control panel.
$\boxed{\#}$	Number: Used to enter arguments to keys that are not in decimal notation. Without a numeric argument, this key accepts a text argument which it interprets as a hexadecimal number. If this key is preceded with a numeric argument, then this number is used as the radix in interpreting the subsequent text argument as a number.

H.1 DISPLAY-CONTROL

These keys affect the display without affecting the contents of the planes or the tables.

$\boxed{a}$	`Toggle.display.source` When sharing a single display, toggles between viewing the PC display and the currently selected CAM display; when PC has its own display, has no effect without an argument. In either case, if multiple CAMs are sharing a display, an argument selects the given CAM for display.
$\boxed{A}$	`Show.PC.display` If CAM and the PC share a display, will show the PC display; else has no effect. With an argument, shows the PC and selects the given CAM as the one to be seen with $\boxed{a}$.
$\boxed{x}$	`Toggle.expanded.view` Toggles between an expanded view of the central portion of the configuration, and the normal view. All keys function in either view.
$\boxed{X}$	`Normal.size` Show normal-sized view of configuration.
$\boxed{'}$	`Toggle.grid` Turn grid on if its off, off if its on. Grid is only visible in the expanded view.
$\boxed{"}$	`Grid.off` Turn the grid off.
$\boxed{f2}$	`IRGB.map` Select the IRGB color map.
$\boxed{F2}$	`Std.map` Select the STD color map.
$\boxed{f4}$	`Toggle.inten` Toggle the visibility of the intensity "beam" of the display.
$\boxed{F4}$	`Inten.on` Turn intensity back on if it is toggled off.
$\boxed{f6}$	`Toggle.red` Toggle visibility of red beam.
$\boxed{F6}$	`Red.on` Turn red beam back on if its off.
$\boxed{f8}$	`Toggle.green` Toggle visibility of green beam.
$\boxed{F8}$	`Green.on` Turn green beam back on if its off.
$\boxed{f10}$	`Toggle.blue` Toggle visibility of blue beam.
$\boxed{F10}$	`Blue.on` Turn blue back on if its off.

H.2 EDITING,RUNNING

These keys control the editing, loading, and running of CAM experiments.

$\boxed{\text{e}}$	`Edit.screen` Continue editing the most recently edited screen. With an argument, will edit the indicated screen of the current file.
$\boxed{\text{E}}$	`Edit.new.file:` Takes a text argument of a new file to edit. If you just hit return, or if the name you give has any wild-card characters, gives a directory listing and asks again for a filename. Default extension is .EXP. If you give a filename that doesn't exist, will offer to create the file. With a numeric argument, editing starts at the indicated screen; otherwise it starts at screen number 1.
$\boxed{\text{l}}$	`Load` Load the current file. With an argument, will load just the indicated screen of the current file.
$\boxed{\text{L}}$	`Load.new.file:` Like `E` , except that the indicated file is loaded, and it won't offer to create any files. With an argument, will just load the indicated screen.
$\boxed{\text{TAB}}$	`Tab.-->.file:` Save the currently active 4K block of the selected CAM's lookup tables to a file. Like $\boxed{\text{E}}$, it can provide a directory listing.
$\boxed{\text{BACKTAB}}$	`Tab.<--.file:` Load the currently active 4K block of the selected CAM's lookup tables from a file. Can provide a directory listing.
$\boxed{\{}$	`Map.-->.file:` Save the selected CAM's color map to a file. This can be useful for generating color hardcopy displays of the screen. Can provide a directory listing.
$\boxed{\}}$	`Map.<--.file:` Load the selected CAM's color map from a file. Can provide a directory listing.
$\boxed{\text{s}}$	`Step(s)` Run one step. With an argument, runs the indicated number of steps. If the argument is zero, starts to run 2^{32} steps, counting as it goes.
$\boxed{\text{S}}$	`Continue.steps...` Resume running steps; continue count where you left off.
$\boxed{,}$	`Slower` Run more slowly.
$\boxed{<}$	`Slowest!` Run about one step a second.
$\boxed{.}$	`Faster` Run faster.
$\boxed{>}$	`Fastest!` Run 60 steps a second.
$\boxed{\text{SPACE}}$	`Stop running` Stop running, and print the current step number.

H.3 PLANE-OPS

Operations on CAM's bit-planes, or between memory buffers and bit-planes. Note that when the cage is active, all operations apply only to the cage region, treating it as if it were the screen, and treating the bit-plane area masked by the cage as if it were the buffer. All operations whose name involves the term 'plane(s)' take an optional plane argument: no argument means operate on all four planes; 0, 1, 2, or 3 indicate a single plane to operate on, and 4 or 5 indicate both planes of either CAM-A or CAM-B respectively. All logical operations involving data from both a buffer and a bit-plane leave their results in the bit-plane.

c	`Toggle.cage` Toggle visibility of the cage. With an argument, creates a new (empty) cage of the given size: first decimal digit in argument is width, second is height, both in units of eight. If only one digit is given, it is used for both height and width.
C	`Cage.off` Hide the cage.
z	`Zero.plane(s)` Clear plane(s) to all zeros.
Z	`Fill.plane(s)` Fill plane(s) with all ones.
r	`CW.plane(s)` Rotate plane(s) clockwise 90°.
R	`CCW.plane(s)` Rotate plane(s) counter clockwise 90°.
\	`\-flip plane(s)` Flip plane(s) across the \ diagonal.
/	`/-flip.plane(s)` Flip plane(s) across the / diagonal.
-	`T/B-flip.plane(s)` Flip plane(s) from top to bottom.
|	`S/S-flip.plane(s)` Flip plane(s) from side to side.
+	`OR.plane(s)` Bit by bit logical OR of bit plane(s) with corresponding buffers.
&	`AND.plane(s)` Logical AND of bit plane(s) with corresponding buffers.
$	`XOR.plane(s)` Logical XOR of bit plane(s) with corresponding buffers.
-	`NOT.plane(s)` Logical NOT of bit plane(s).

$\boxed{g}$	`Get.image.from.buffer` Get bit plane(s) from buffer.
$\boxed{G}$	`Get.image.from.the.file:` Get an image for the entire screen (not just the cage or any other active region) from a file.
$\boxed{p}$	`Put.image.into.buffer` Put bit plane(s) into corresponding buffers.
$\boxed{P}$	`Put.image.into.the.file:` Put the image contained on the entire screen (not just the cage or any other active region) and store it into a file.
$\boxed{*}$	`Exchange.display.and.buffer` Exchange plane(s) with plane buffers.
$\boxed{\%}$	`Percent.of.ones` Set the percent of ones to be used by $\boxed{;}$.
$\boxed{;}$	`Random.configuration` Generate a random configuration on the indicated bit plane(s).
$\boxed{:}$	`Number.of.ones` Set the expected number of ones to be used by $\boxed{;}$.
$\boxed{i}$ $\boxed{I}$	`Init.random.number.generator` Initialize the software random number generator to a standard state. With an argument, uses it as part of its seed.
$\boxed{\hat{}}$	`Permute/copy.planes` Permute or copy the contents of the indicated planes. A four-digit argument indicates the data source (i.e., which plane) for each of the four planes—in the order 0, 1, 2, 3. With no argument, re-uses most recent argument.

H.4 DOTS,SHIFTS

Keys used for dot editing (editing individual bits of CAM cell values). When the dot-cursor is visible, the arrows (and the mouse, if you have one) move the dot cursor rather than the bit-planes. Note that the four corner keys of the numeric keypad are used as additional diagonal-arrow keys by the control panel. On the mouse, the left button is used for drawing horizontal and vertical lines, the right button for diagonal lines, the middle button for unconstrained placement of dots. If the $\boxed{\text{INS}}$ key is held down while the cursor is moved (with the arrows or the mouse) dots will be toggled as the cursor goes by. On some keyboards the auto-repeat function on some keys doesn't work while the insert key is down: any of the shift-lock keys (including $\boxed{\text{CAPSLOCK}}$) can be used within the control panel instead of $\boxed{\text{INS}}$ for this purpose.

$\boxed{\text{d}}$	`Toggle.dots` Toggle dot cursor on or off. With an argument, turns dot cursor on on the indicated bit-plane, otherwise the bit plane in use doesn't change.
$\boxed{\text{D}}$	`Dots.off` Turn dot cursor off.
$\boxed{\text{h}}$	`Hold.fixed` Hold the specified plane(s) fixed during shifts using the arrow keys.
$\boxed{\text{H}}$	`Un-hold` Free the specified plane(s) to shift.
$\boxed{\text{o}}$	`Cursor.to.origin` Move the dot-cursor to the center of the screen.
$\boxed{\text{O}}$	`Shift.to.origin` Shift all planes until the dot cursor is at the center of the screen.
$\boxed{\uparrow}$	`Up` Shift un-held bit planes upward: use auto-repeat of keyboard for continuous shift. With an argument, shift by the indicated amount until the default argument is again changed by giving an explicit argument. When an argument has been given, all subsequent shifts are a multiple of the given value; the inital default is multiples of 8.
$\boxed{\downarrow}$	`Down` Similar to $\boxed{\uparrow}$, but shift down.
$\boxed{\leftarrow}$	`Left` Similar to $\boxed{\uparrow}$, but shift left.
$\boxed{\rightarrow}$	`Right` Similar to $\boxed{\uparrow}$, but shift right.
$\boxed{\text{HOME}}$	`UpLt` Similar to $\boxed{\uparrow}$, but shift diagonally up and to the left.
$\boxed{\text{PGUP}}$	`UpRt` Similar to $\boxed{\uparrow}$, but shift diagonally up and to the right.
$\boxed{\text{END}}$	`DnLt` Similar to $\boxed{\uparrow}$, but shift diagonally down and to the left.
$\boxed{\text{PGDN}}$	`DnRt` Similar to $\boxed{\uparrow}$, but shift diagonally down and to the right.
$\boxed{\text{INS}}$	`Insert.dot` Inserts a dot at the cursor position, and on the cursor bit-plane, even if the cursor isn't visible.

H.5 ALTERNATE

The `Alt-` keys are reserved for application-specific, user-defined commands (custom control-panel commands). When the word `ALIAS` is used to attach a key to a Forth word in an experiment, the name of the key should be written in upper case. (For example, `ALIAS F2` would attach a Forth word to the key $\boxed{\texttt{Alt-F2}}$).

Index

If you don't find an item as a top-level index entry, look at the subentries of the following main entries:
—Control-panel commands (listed by functional groups)
—Control-panel keys (listing of individual keys)
—Editor (listing of individual screen-editor keys)
—Files (terminology, types, operations)
—Forth (terms and concepts of the language)

Forth words are listed here only if they are explicitly discussed in the text. For an extensive listing of Forth words, consult the two sections of the glossary (Appendix G)—of which the first is devoted to CAM-specific words and the second to Forth-generic words.