Panasonic

HHC T.M.

SnapFORTH

VOLUME II: REFERENCE GUIDE

THE HIGH-LEVEL PROGRAMMING LANGUAGE NATIVE TO THE HHC™

RL-S6003S8

SnapFORTH

the high-level programming language native to the HHC™

VOLUME II: REFERENCE

Copyright 1982 by Matsushita Electric Industrial Co., Ltd. All rights reserved. HHC™ is a Trademark of Matsushita Electric Industrial Co., Ltd.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 WHAT THIS DOCUMENT IS FOR 1-1
1.2 HOW THIS DOCUMENT IS ORGANIZED 1-1
1.3 WHAT THIS DOCUMENT ASSUMES 1-2
1.4 THE GLOSSARIES 1-3
1.4.1 How the Glossaries Are Organized 1-3
1.4.2 Glossary Notation 1-4
1.5 HHC ARCHITECTURE 1-5
1.5.1 The Hardware 1-5
1.5.2 The Software 1-7
1.5.3 The Function of Interrupts 1-9
1.6 HOW SnapFORTH DIFFERS FROM FORTH 1-9

CHAPTER 2: GENERAL TECHNICAL INFORMATION

2.1 MEMORY MAP 2-1
2.2 DATA REPRESENTATION 2-3
2.3 STACKS: SOME TERMINOLOGY 2-4
2.4 USE OF THE RETURN STACK IN THE HHC 2-4
2.4.1 Two-Word Entries On the Return Stack 2-5
2.5 THE LATCH BYTE 2-5

CHAPTER 3: INTRODUCTION TO THE SnapFORTH CAPSULE

3.1 COMPILER OPTIONS 3-1
3.2 DICTIONARY VARIABLES 3-3
3.3 BASE 3-4

CHAPTER 4: SnapFORTH CAPSULE; TECHNICAL INFORMATION

4.1 MEMORY LAY-OUT 4-1
4.2 DATA STRUCTURES IN THE SnapFORTH CAPSULE 4-2
4.2.1 SnapFORTH Code 4-2
4.2.2 Implementation of Tag Tables 4-8
4.2.3 Headers and Vocabularies 4-13

CHAPTER 5: CAPSULE GENERATION

5.1 INTRODUCTION 5-1
  5.1.1 Run Time vs Compile Time 5-1
  5.1.2 The "700tags" Tables 5-2
  5.1.3 %INT 5-2
  5.1.4 Variables 5-3
  5.1.5 CREATE 5-3
5.2 EXECUTION OF CAPSULES AND SnapFORTH FILES 5-3
  5.2.1 When Must I Write My Own Capinit Word? 5-4
  5.2.2 The Header of a ROM Capsule 5-5
  5.2.3 Format of a SnapFORTH File Header 5-6
  5.2.4 SnapFORTH Files Must Be Relocatable 5-6
  5.2.5 Capsules Can Be Absolute, But 5-7
5.3 SAVESNAP 5-9
5.4 A SnapFORTH FILE EXAMPLE 5-10
5.5 CREATING A STAND-ALONE CAPSULE 5-10
5.6 USING THE HHC'S EPROM BURNER 5-11
5.7 MAKING A CAPSULE YOURSELF 5-11
5.8 RELOCATING THE TAG TABLE 5-12
5.9 SENDING YOUR PROGRAM TO THE RS232C 5-13
  5.9.1 Structure of a Program for an External RS232C EPROM Burner 5-13
5.10 COMPILATION FOR RAM/ROM 5-14

CHAPTER 6: USING EXTERNAL ROUTINES

6.1 TAG 6-1
6.2 FINDING ANOTHER CAPSULE'S TAG TABLES 6-1

CHAPTER 7: MANAGING DATA IN THE SnapFORTH LANGUAGE

7.1 CONSTANTS 7-1
  7.1.1 Symbolic Constants 7-1
  7.1.2 Colon Definitions As Constants 7-1
  7.1.3 Set.Constant 7-2
7.2 ALLOCATING RAM, PART I: THE TEMPORARY STORAGE AREA (TSA) 7-3
  7.2.1 How To Use a TSA 7-3
  7.2.2 An Example 7-3
  7.2.3 Types Of TSA Variables 7-4
  7.2.4 Notes On Using the TSA 7-4
  7.2.5 Releasing TSAs 7-5
  7.2.6 About the Size of the Temporary Stack 7-5
7.3 ALLOCATING RAM, PART II: DYNAMIC ALLOCATION 7-6
  7.3.1 Is There Enough Room? 7-6
  7.3.2 How Much Room Is There, Anyway? 7-7
7.4 ALLOCATING SPACE IN ROM 7-7
7.5 RESTARTING A PROGRAM 7-8
  7.5.1 The CLEAR Key 7-8
  7.5.2 Changing the Restart Word 7-9
  7.5.3 Software Simulation of the CLEAR Key 7-10
  7.5.4 Inhibiting Hard Clears 7-10
  7.5.5 Ending a Program 7-10

CHAPTER 8: CONTROL FLOW IN THE SnapFORTH LANGUAGE

8.1 TRANSFER OF CONTROL IN SnapFORTH 8-1
8.2 TAG STRUCTURE AND TRANSFER OF CONTROL 8-1
8.3 SOME ADDITIONAL CONTROL STRUCTURES 8-3
8.4 LONG CONTROL STRUCTURES 8-6
  8.4.1 The WHILE Statement 8-7

CHAPTER 9: <BUILDS DOES>

9.1 INTRODUCTION TO <BUILDS DOES> 9-1
9.2 IMPLEMENTATION OF <BUILDS DOES> 9-3
9.3 TAG NUMBERS FOR DEFINING WORDS 9-4

CHAPTER 10: LOADING

10.1 CHANGING IN 10-1
10.2 CHANGING TIB 10-3
10.3 LOADING FROM FILES 10-3
10.4 END OF FILE CONDITION 10-5
10.5 IMPLEMENTATION OF LOAD 10-5
10.6 CHECKING THE LINE LENGTH 10-6

CHAPTER 11: THE HHC KEYBOARD

11.1 HARDWARE ARCHITECTURE 11-1
  11.1.1 Hidden and Immediate Keys 11-1
  11.1.2 Special Keys On the Keyboard 11-1
  11.1.3 "Unhiding" Hidden Keys 11-3
  11.1.4 Keyboard Codes 11-4

11.2 PROGRAM-DEFINED KEYBOARD TRANSLATION TABLES 11-4
11.2.1 Coding a Keyboard Translation Table 11-4
11.2.2 Adopting a New Keyboard Translation Table 11-5
11.2.3 The HHC Character Set 11-7
11.3 EXPECT: KEYBOARD INPUT 11-14
11.3.1 EXPECT Modes 11-17
11.3.2 EXPCT 11-17
11.3.3 Example 1 11-18
11.3.4 Example 2: Editing Text 11-18
11.3.5 Example 3: Changing the End-of-Line Character 11-19
11.4 CHARACTER INPUT: KEY AND ?KEY 11-19
11.4.1 Note on Battery-Conserving Code 11-20
11.5 SIMULATING KEYBOARD INPUT FROM A PROGRAM.......... 11-20
11.5.1 Structure Of the Keyboard Buffer 11-21
11.5.2 The Pushkey Buffer 11-22
11.5.3 Operations On the Buffers 11-22
11.5.4 "Typing" a Function Key 11-23
11.6 KEYBOARD I/O WORDS 11-24
11.6.1 Symbolic Constants For Character Codes .. 11-24
11.6.2 Keyboard Words 11-24
11.6.3 EXPECT Words 11-25

CHAPTER 12: THE LCD DISPLAY AND BEEPER

12.1 THE LCD 12-1
12.1.1 The LCD Character Set: Special Characters 12-2
12.2 I/O WORDS 12-2
12.2.1 EMIT 12-2
12.2.2 "Random Access" Output To the LCD 12-4
12.2.3 Graphics Output 12-6
12.2.4 Program-defined Blips 12-7
12.3 ALTERNATE CHARACTER SETS 12-7
12.3.1 Defining a Character Translation Table 12-8
12.3.2 Informing the HHC of the Table's Address .. 12-10
12.3.3 Defining a Floating Accent Translation Table 12-10
12.3.4 Defining an Alternate Cursor Pattern 12-11
12.4 ROTATION MODE 12-12
12.5 THE STOP/SPEED KEY 12-12
12.6 SLAVING PERIPHERALS TO THE LCD 12-13
12.6.1 FLAME-ON 12-13
12.6.2 Vectored I/O; (EMIT) 12-13
12.7 BLIPS 12-14
12.8 BEEPS AND SQUEAKS 12-15
12.9 DISPLAY AND OUTPUT WORDS 12-15
12.9.1 LCD Words 12-15
12.9.2 Formatted Output Words 12-17
12.9.3 Blip Words 12-17
12.9.4 ASCII Constants 12-17

CHAPTER 13: THE VIRTUAL FILE SYSTEM

13.1 VIRTUAL FILE CONCEPTS 13-1
13.2 FILE TYPES 13-2
13.3 VIRTUAL FILE OPERATIONS 13-2
13.3.1 OPEN: Opening an Existing File 13-2
13.3.2 MAKE: Creating and Opening a New File 13-3
13.3.3 READ: Reading a Text File 13-3
13.3.4 WRITE: Replacing an Existing Record In a Text File 13-4
13.3.5 INSERT: Inserting a New Record In a Text File 13-4
13.3.6 REC-CNT and INSERT: Appending a New Record To a Text File 13-5
13.3.7 DELETE: Deleting a Record From a Text File 13-5
13.3.8 REVISE: Allocating Space in a File 13-5
13.3.9 More REVISE: Adding and Deleting Space In a Non-Text File 13-6
13.3.10 CFILE and DELETE-FILE: Deleting a File 13-6
13.3.11 Renaming a File 13-7
13.3.12 GET-TYPE: Testing and Setting File Type 13-7
13.3.13 LOOKUP: Look Up the Nth File 13-8
13.4 THE FILE SYSTEM EDITOR 13-8
13.5 THE VIRTUAL FILE SPACE 13-9
13.5.1 Structure of Virtual File Space 13-9
13.6 VIRTUAL FILE SYSTEM WORDS 13-12

CHAPTER 14: PERIPHERAL I/O AND TIMER SERVICES

14.1 EVENT CONTROL BLOCKS 14-1
14.1.1 Introducing Asynchronous Processing 14-1
14.1.2 The Event Control Block: Keeping Processes In Step 14-2
14.1.3 Other Uses For ECBs 14-3
14.1.4 More About ECBs 14-3
14.1.5 Using ECBs 14-4

14.2 INTRODUCTION TO PERIPHERAL I/O 14-5
14.2.1 Software: Architecture 14-5
14.2.2 Software: In General 14-6
14.3 PERIPHERAL I/O OPERATIONS 14-7
14.3.1 The ATTACH Operation 14-7
14.3.2 The ATTACHX Operation 14-8
14.3.3 RIP: Request Input 14-8
14.3.4 ROP: Request Output 14-8
14.3.5 Relation of RIP and ROP to KEY and EMIT 14-9
14.3.6 Opening and Closing Devices 14-9
14.3.7 RCTL: Control and Status Operations 14-10
14.3.8 A Note on Multiple Waits 14-11
14.3.9 The System Device Table (SDT) 14-11
14.4 PARAMETER VALUES 14-12
14.4.1 Hardware Device Codes 14-12
14.4.2 Logical Unit Numbers 14-12
14.4.3 ECB Return Codes 14-12
14.5 OUTLINE OF PERIPHERAL I/O WORDS 14-13
14.6 CONTROL CHARACTERS 14-13
14.6.1 ASCII Control Characters 14-14
14.6.2 Escape Control Sequences 14-14
14.6.3 The Keyboard 14-16
14.6.4 The LCD 14-16
14.6.5 The Micro Printer 14-17
14.6.6 Video Interface 14-19
14.6.7 Modem 14-19
14.6.8 Serial Interface Adaptor (RS-232 Interface) 14-24
14.7 TIMER SERVICES 14-29
14.7.1 The Real-time Clock 14-29
14.7.2 Using Timer Facilities 14-30
14.7.3 The Auto-Off Facility 14-30
14.8 I/O AND TIMER SERVICES WORDS 14-31
14.8.1 I/O Words 14-31
14.8.2 Timer Services Words 14-32

CHAPTER 15: FLOATING POINT OPERATIONS

15.1 INTERNAL STORAGE 15-1
15.2 ERROR HANDLING AND SPECIAL CASES 15-1
15.2.1 Defining New Special Case Types 15-3
15.2.2 Some Missing Operations 15-3
15.3 FLOATING POINT OPERATIONS WITHOUT SPECIAL CASE CHECKING 15-4
15.3.1 Calling the Non-Checking Version of F+ 15-4
15.3.2 Calling the Non-Checking Versions of F* and F/ 15-5
15.4 FLOATING POINT WORDS 15-6

CHAPTER 16: THE SnapFORTH ASSEMBLER

16.1 INTRODUCTION 16-1
16.2 INTERFACE WITH FORTH ROUTINES 16-2
16.3 THE RETURN STACK 16-3
16.4 THE PARAMETER STACK 16-3
16.5 SOME MORE EXAMPLES 16-4
16.6 PARAMETER PASSING 16-5
16.6.1 SETUP 16-5
16.6.2 NEXT 16-6
16.6.3 PUSH 16-6
16.6.4 PUT 16-6
16.6.5 CPUSH and CPUT 16-6
16.6.6 POP and 2POP 16-7
16.6.7 PUTFLS 16-7
16.6.8 PUTTRU 16-7
16.7 TEMPORARY STORAGE 16-7
16.8 THE SnapFORTH INSTRUCTION POINTER 16-8
16.9 THE ASSEMBLY PROCESS 16-8
16.10 THE ASSEMBLER DOES NOT WORK IN COMPILE MODE 16-8
16.11 MACROS 16-9
16.12 GETTING IN AND OUT OF THE ASSEMBLER VOCABULARY 16-9
16.13 OPCODES AND ADDRESSING MODES 16-10
16.14 SIMPLE MACHINE INSTRUCTIONS 16-10
16.15 IMMEDIATE ADDRESSING 16-10
16.16 INDEXED ADDRESSING 16-11
16.17 STACK ADDRESSING 16-11
16.18 ZERO PAGE AND ABSOLUTE ADDRESSING 16-12
16.19 INDIRECT X) 16-12
16.20 INDIRECT)Y 16-12
16.21 ACCUMULATOR ADDRESSING 16-12
16.22 SOME MORE INSTRUCTIONS 16-13
16.23 CONTROL FLOW—CONDITIONAL SPECIFIERS 16-13
16.24 CONTROL FLOW—CLAUSES 16-14
16.25 SPAGHETTI 16-14
16.26 JUMP INSTRUCTIONS 16-15
16.27 ;CODE 16-15
16.28 EXAMPLES 16-16

CHAPTER 1: INTRODUCTION

1.1 WHAT THIS DOCUMENT IS FOR

This document describes the HHC computer from the point of view of a programmer writing applications using the SnapFORTH™ capsule. These applications may be distributed in ROM capsules. It also contains information that is useful to developers of lower level software (e.g. device drivers).

1.2 HOW THIS DOCUMENT IS ORGANIZED

1. This introduction describes the HHC in general terms, outlining the major parts of the system and explaining how they relate to each other.
2. GENERAL TECHNICAL INFORMATION discusses various aspects of programming for the HHC, such as memory use, internal program structure, and stack space.
3. INTRODUCTION TO THE SnapFORTH CAPSULE introduces several aspects of the capsule: ROM capsule generation, compiler options and dictionary variables.
4. SnapFORTH CAPSULE; TECHNICAL INFORMATION discusses the principles underlying the SnapFORTH language and its compiler.
5. CAPSULE GENERATION describes how you can make stand- alone applications, running either in ROM capsules or HHC files.
6. USING EXTERNAL ROUTINES explains how to make a program which occupies several ROM capsules, or which uses routines from library capsules like the Scientific Calculator.
7. MANAGING DATA IN THE SnapFORTH LANGUAGE discusses techniques for allocating and using constants, variables, and RAM work space in the HHC's operating system.
8. CONTROL FLOW IN THE SnapFORTH LANGUAGE describes the control structures available in SnapFORTH, some of which are very advanced.
9. <BUILDS DOES> describes the SnapFORTH implementation of the defining words <BUILDS and DOES>.
10. LOADING discusses how the SnapFORTH outer interpreter works, and how you can supply your own input routines if you want to load from a non-standard device.

11. THE HHC KEYBOARD describes the HHC keyboard hardware and software.
12. THE LCD DISPLAY AND BEEPER explains how graphics output can be done, how you define your own character set, etc.
13. THE VIRTUAL FILE SYSTEM describes the part of the nucleus that stores data in RAM. From a software point of view, it is a random access file system. Available file operations are: READ, WRITE, INSERT etc.
14. PERIPHERAL I/O AND TIMER SERVICES is an extension of "General Technical Information" concerned only with these two closely related areas.
15. FLOATING POINT OPERATIONS describes the floating point arithmetic package implemented on the HHC.
16. THE SnapFORTH ASSEMBLER discusses the 6502 assembler which is embedded in the SnapFORTH language, and presents techniques for writing assembler subroutines for use with SnapFORTH.
17. SAMPLE PROGRAMS presents complete application programs written in SnapFORTH, with commentary on important aspects of each.
18. GLOSSARIES contains two glossaries of all the SnapFORTH words involved in the HHC system. The organization of the glossaries is discussed below.

1.3 WHAT THIS DOCUMENT ASSUMES

This document assumes that you have a working knowledge of some dialect of FORTH. If you do not yet have a working knowledge of FORTH, you should first study its accompanying document:

Introduction to SnapFORTH

- A beginners manual of SnapFORTH programming using the SnapFORTH capsule.

Another textbook on FORTH, which covers more advanced subjects, is:

Starting FORTH by Leo Brodie
(Prentice Hall)

Prior to or concurrently with reading this document, you should develop a working familiarity with the HHC from a user's point of view.

1.4 THE GLOSSARIES

1.4.1 How the Glossaries Are Organized

The "GLOSSARIES" section contains two glossaries:

Glossary of SnapFORTH Capsule Words

This glossary is divided into several sections according to the functions of the words. These sections are:

1. ***SnapFORTH capsule words***. Describes all compile time words, such as ':', ';', FORGET, the control structures etc. This chapter is organized as a number of paragraphs, each describing one particular subject. The paragraphs are:
   - Dictionary variables
   - Compiler control bits
   - 'TO' concept words
   - 'O' concept words
   - Nucleus extension words
   - Compile time extensions
   - Dictionary words
   - String handler words
   - Symbol table words
   - Number handler words
   - Tagtable words
   - Defining words
   - AREA words
   - File words
   - Conditional compilation
   - Choice clauses
   - Repetition clauses
   - Compiler/Misc
2. ***Object time words***. These are words in the HHC's vocabulary that are generated by the compiler, but are never written in source programs. They implement program control structures and literal constants.
3. ***Assembler glossary***

Glossary of HHC Words

This glossary describes all executable words you can use in a stand-alone SnapForth program (e.g. OVER, DUP, DROP, READ, WRITE etc.). All words appear in alphabetical order.

In addition, "mini-summaries" containing subsets of the glossaries above are distributed throughout the other chapters of the book.

1.4.2 Glossary Notation

SnapFORTH words are always capitalized. Where there may be confusion about where a word ends and punctuation begins, the word is enclosed in apostrophes. Example: 'REFRESH.' is a word ending with a period.

Glossary Format

Here is an example of a glossary entry with a description of each part:

OVER (X Y --- X Y X)

Copies the second word on the stack to the top of the stack.

OVER

The word being described.

(X Y --- X Y X)

Shows stack input to the left of the hyphens, and stack output to the right. This example shows that the word expects two items on the stack, and leaves three.

Conventions for naming the stack items are intuitive rather than formal. Descriptive names, such as "LEN" for a length, are used where appropriate. Where no conventions are used:

A for an address.

B for a boolean (true/false).

C for a character.

..D for a suffix indicating a double-length (32- bit) operand, e.g., ND for a double-length integer.

FL for a byte of bit flags.

FP for a floating point number.

L for a length or byte count.

N for an integer.

U for an unsigned integer.

X,Y,Z for 16-bit quantities of unspecified type.

[] encloses an operand that may or may not be present. Example: '(X --- X [X])'. X may or may not be duplicated.

The sequence '(... --- [X] B)' customarily means that X is present if B is TRUE, and absent if B is FALSE.

If the same name appears on both sides of the hyphens, it represents the same value (i.e., the value is not changed). If a name appears on the left side of the hyphens, and appears on the right side with a prime after it, it represents a corresponding (but changed) value. Example:

(A L --- A L')

If two or more values of the same type appear in the same specification, they are distinguished by numbers. Example:

(N1 N2 --- N3)

Text Format

In some parts of the text, a less formal kind of notation is used to describe words. The object is to give a description of the linkage in one indented line, without disturbing the flow of the text. Here is an example of this kind of notation:

```
x y OVER x y x
```

OVER copies the second word on the stack to the top of the stack.

1.5 HHC ARCHITECTURE

The HHC's needs for compactness and low power consumption have resulted in a very unusual architecture in both hardware and software. It is important for you to understand the major features of this architecture if you are to program the HHC effectively.

1.5.1 The Hardware

The HHC is a self-contained microcomputer system built around a 1-MHz 6502 microprocessor. It utilizes the full 64K address space. The structure of the address space is complex; various parts of it are occupied by RAM, ROM, plug-in ROM capsules, memory-mapped I/O devices, etc. A very rough map of the address space is shown in a table at the end of this section; a detailed map is shown in Chapter 2.

Compared to other microcomputers, the HHC has much ROM and little RAM. This is due to cost and power constraints, and the need to support a full operating system without peripheral storage.

The HHC's intrinsic[1] I/O devices are a full ASCII keyboard; a one-line LCD display, which can rotate from right to left like a light sign; a set of **LCD indicators** which are displayed as motionless pointers below the text on the LCD display; and a variable-pitch **beeper**.

Intrinsic input consists of a typewriter-like keyboard with several special purpose keys such as "on," "off," "shift," "function," and "help."

Additional devices (peripherals) may be attached through a bus socket on the side of the HHC. Such peripherals are controlled through a standard set of I/O words.

0000-1FFF	Intrinsic (built-in) RAM. Used for variables and scratch space by operating system; also for virtual file space. In some HHCs this part of the address space is not fully populated. The amount of RAM actually available may be 2K (0 through 7FFH), 4K (through 0FFFH), 6K (through 17FFH), or 8K (through 1FFFH).
2000-3FFF	Device control ROMs. A device control ROM is physically part of the device it controls, and so is connected to the HHC when the device is plugged in. The proper device control ROM for a given I/O operation is selected by bank switching. Up to six different ROMs may be bank-switched into this space alternately.
4000-7FFF	Capsule and I/O bank space. ROM capsules are read from this space. The contents of this space are controlled by bank switching; the space may be alternately filled by any of the three capsule sockets on the back of the HHC, and/or sockets in a ROM expander. Bank switching also permits access to the HHC's memory-mapped I/O hardware.
8000-BFFF	Space for RAM expander(s).
C000-FFFF	Intrinsic ROM. Contains the nucleus; interrupt processor; file manager; and intrinsic applications (clock controller, file system editor, and calculator).

[1] - System components that are integral parts of the HHC, or may be treated as though they were when they are attached, are often called "intrinsic." The HHC has "intrinsic RAM," "intrinsic ROM," and several "intrinsic applications," which are application programs stored in intrinsic ROM. Things that are not intrinsic are purchased as accessories and plugged in. For example, additional programs may be purchased in ROM capsules and plugged in; additional RAM may also be plugged in.

1.5.2 The Software

All HHC software is built around an interpretive language called SnapFORTH (or 'SNAP' for short). SnapFORTH is a highly tailored variant of FORTH. Much of the nucleus is written in it; so is virtually all of the applications code.

Major components of the HHC software are listed below. Their relationships are shown graphically in the illustration at the end of this section.

The **SnapFORTH interpreter** is written in low-level code, i.e. in 6502 machine code. It interprets the parts of the software that are written in SnapFORTH.

The **peripheral interface** connects programs to I/O devices. It communicates with programs through a standardized set of SnapFORTH words, and communicates with I/O devices through standardized calls to device-specific I/O driver routines. Its functions include attaching peripherals to programs, logically binding peripherals to one another, and translating I/O requests between the programs' device-independent formats and the I/O devices' device-specific formats. Closely associated with the peripheral interface is the **interrupt handler**.

The **intrinsic I/O drivers** communicate between the peripheral interface and the intrinsic I/O devices. They are sometimes treated as parts of the peripheral interface itself.

The **primary menu handler** is the user's main interface with the operating system. It gains control when the HHC is first powered up, and when the CLEAR key is pressed twice. It cycles through the primary menu and allows the user to choose one of the menu items: intrinsic applications, ROM capsules, etc.

The system's **timer** facility uses a hardware clock that runs continuously, even when the HHC is turned off. This clock can generate an interrupt that turns the HHC on spontaneously if it happens to be off when timer services are needed. The timer facility supports multiple timers that can be set independently

by different parts of software. The facility is used by many parts of the software, such as the clock-controller, the LCD control software, and extrinsic applications.

The **floating point package** supports elementary arithmetic operations, detection of various exception conditions, and conversion to and from ASCII. The floating point format provides 12 decimal digits of precision (a 13th serves as a guard digit). The package is used by the intrinsic calculator and by extrinsic applications.

The **file system** enables programs to do "I/O" to "virtual files" stored in RAM, or to real files if an appropriate peripheral is available.

The items listed above constitute the **nucleus**, that is, the intrinsic portion of the operating system.

The **intrinsic applications** are application programs that are built into the HHC. They are the **calculator**, the **clock controller**, and the **file system editor**.

The **capsule programs** are applications kept in plug-in ROM capsules. When a ROM capsule is plugged into the HHC, it may be bank-switched into the HHC's address space by the nucleus.

Peripheral I/O drivers communicate between the peripheral interface and peripheral devices. They perform the same function as the intrinsic I/O drivers, but for plug-in peripherals instead of intrinsic devices. They reside in ROM that is physically part of each peripheral device. The nucleus bank-switches each device's I/O driver into the HHC's address space when I/O to that device is requested.

Peripheral I/O driver ROMs often contain application programs which appear in the primary menu along with the intrinsic applications and capsule programs. Examples are the telecomputing program, which appears in the modem's ROM, and the serial I/O program, which appears in the serial interface adaptor's ROM.

<table>
<tr><th colspan="3">APPLICATIONS</th></tr>
<tr><td>in I/O driver ROMs</td><td>in ROM capsules</td><td>intrinsic</td></tr>
<tr><td>peripheral I/O drivers</td><td colspan="2">NUCLEUS
interrupt handler; intrinsic I/O drivers; floating point package; timer; menu handler</td></tr>
<tr><th colspan="3">SNAP SnapFORTH INTERPRETER</th></tr>
</table>

Higher "layers" of system components depend on lower ones. The interrupt handler and SnapFORTH interpreter interact closely with all parts of the system.

1.5.3 The Function Of Interrupts

The HHC uses interrupts very extensively. Interrupts can come from two sources: peripherals and the HHC's clock or timers.

The HHC actually turns itself on in response to any event such as a keystroke, processes the event, and then turns itself off again to conserve power. This is an important feature of the HHC's design, since the HHC is powered by 5 AA batteries (IEC designation R6 or equiv.), and these batteries would run the 6502 for only a few hours if used constantly.

Any event which requires the CPU's attention, such as a keystroke, a timer expiration, or a message from a peripheral device, generates an interrupt. This causes the interrupt hardware to turn the CPU on.

Note that the HHC's ON and OFF switches are not electrical controls in the conventional sense; they are ordinary keyboard keys, which the HHC is programmed to respond to by turning itself "on" and "off."

When the HHC is "off," the timer and RAM still receive power. This enables the HHC to "remember" the time of day, appointment calendar, etc., even after the power is turned off. Since the timer and RAM all use CMOS technology, the power consumption is very low.

1.6 HOW SnapFORTH DIFFERS FROM FORTH

SnapFORTH uses a different set of defining words for variables and constants, and has a different set of rules for using them. This is discussed more fully in the chapter on "General Technical Information."

A second major difference between SnapFORTH and FORTH is in SnapFORTH's format for compiled code. Most FORTHs compile a colon definition into a series of two-byte code field addresses, which the interpreter calls like subroutines. SnapFORTH compiles a colon definition into a series of TAGS, which are used to fetch execution addresses from a table at run time. The most commonly used words are represented by one-byte tags, yielding more compact code. The compiled-code format of SnapFORTH is described fully in "Technical Information on SnapFORTH".

CHAPTER 2: GENERAL TECHNICAL INFORMATION

The following pages describe some fundamental truths about how the HHC operates. We briefly describe the memory map for the HHC, and the internal data representation of integers and strings. This information is not unique to SnapFORTH, but is true regardless of the application capsule currently being executed.

2.1 MEMORY MAP

The CPU can address 64K of memory. The address space is partitioned among several different kinds of memory. The major kinds are described in the following memory map.

Address	Description
0000-0380	(The end is approximate.) Nucleus variable space. The parameter stack space is in this area; it is about 0A8H bytes long, and ends at 0FEH (0FFH is used for housekeeping). The parameter stack grows down from the end of the parameter stack space. The return stack space uses bytes 100H to 1FDH. The return stack grows down from the top; the loop stack grows up from the bottom. Should they ever cross, disaster is imminent. Note that this is a deviation from figFORTH, in which loop values are stored on the return stack itself. System variables and various buffers take up the rest of the space from 200H to 380H.
038D-1FFF	(The beginning is equal to the constant FILESPACE.) Intrinsic RAM. Used to hold virtual files and the temporary stack (see the discussion of temporary storage areas and workspace for I/O device routines). Virtual files grow up from the bottom of this space.

The temporary stack grows down from the top.

Note: This space is not fully populated in all HHCs. RAM ends at 7FFH in 2K systems; 0FFFH in 4K systems; 17FFH in 6K systems; and 1FFFH in 8K systems.

2000-3FFF — Device control ROMs. A device control ROM is physically part of the device it controls, and so is connected to the HHC when the device is plugged in. The proper device control ROM for a given I/O operation is selected by bank switching. Up to 6 different ROMs may be bank-switched into this space.

The HHC cannot address all of the ROM that is physically on-line at the same time. It uses bank-switching techniques to "map in" the correct control ROM when I/O is in progress.

4000-7FFF — Capsule space. ROM capsules are accessed through this space.

There are 3 capsule sockets in the back of the HHC, and more in the ROM expander peripheral. Each socket can hold a 16K ROM capsule containing programs or data. The content of capsule space is controlled by bank-switching; the space may contain any of the 3 capsule sockets on the back of the HHC, or the sockets in a ROM expander. In addition, the nucleus allows one capsule to temporarily bank switch in another in order to run programs or access data.

The capsule space can also be bank switched to access the HHC's memory-mapped I/O hardware. This is an operating system function; application programs should not bank-switch the I/O hardware into the capsule space. ***Caution***: attempts to store into this space will address the I/O hardware regardless of what memory bank is switched in. Never try to write to ROM!

8000-BFFF — Extrinsic RAM space. Programmable Memory Peripherals which plug into the HHC bus occupy this part of the address space.

C000-FFFF — Intrinsic ROM. Contains the nucleus; interrupt processor; file manager; and intrinsic applications (clock controller, file system editor, and calculator).

2.2 DATA REPRESENTATION

Integers are represented with the high-order byte oriented toward the top of the address space:

16-bit integer:

```
  in memory: LLLLLLLLHHHHHHHH
             ↑        ↑
             adr      adr+1
  on stack:  LLLLLLLLHHHHHHHH
```

32-bit integer:

```
  in memory: L-L      2-2      3-3      H-H
             ↑        ↑        ↑        ↑
             adr      adr+1    adr+2    adr+3
  on stack:  3-3      H-H               (top)
             L-L      2-2               (sec)
```

Floating point numbers are represented by 8-byte quantities containing a sign bit, an 11-bit biased exponent, and a 52-bit mantissa. See the chapter on floating point arithmetic for more information.

Characters, when stacked, occupy the low-order half of a stack entry. The high-order half is zero.

SNAP has a standardized string data type. A string consists of a length byte giving the number of characters in the string (0 to 255) followed by the string in ASCII. String constants may be defined with the word 'STRING"'; it is used like this:

```
STRING" value" cccc
```

or

```
STRING"  "value"  cccc
```

If 'value' has
leading blanks

'value' is the value of the string constant. The first character follows the word 'STRING"' and one blank; the next '"' marks the end of the string. 'cccc' is the string constant's name.

Alternatively, the word S" may be used to define a string literal:

```
HERE
S" value"
== cccc
```

(This form is used to define tables of strings; for single values, STRING" is preferable.)

Boolean values are 0 for FALSE and non-0 for TRUE. When a SnapFORTH word generates a boolean TRUE, it always uses the value 1.

2.3 STACKS: SOME TERMINOLOGY

In SnapFORTH, as in most stack oriented languages, there are several different stacks; some grow downward, and others grow upward. This tends to create confusion about the meanings of the phrases "top of the stack" and "bottom of the stack". If a certain stack grows downward, is its "top entry" the first one pushed on (which is highest in memory) or the last (which is highest in memory)?

In this document we will define the ***TOP*** of the stack as the end where all of the pushes and pops happen, regardless of whether the stack grows physically from low to high memory or high to low memory. The ***root*** of the stack is the end where the very first entry pushed onto the stack is located.

2.4 USE OF THE RETURN STACK IN THE HHC

In many implementations of FORTH, the DO ... LOOP index and limit are kept on the return stack. In the HHC, loop parameters are kept on a ***separate*** stack. This means that certain coding techniques which work in FORTH will fail on the HHC. An example is using R> and >R to modify a loop's limit while in the loop. Most such coding techniques are bad practice, and should be avoided in any case!

2.4.1 Two-Word Entries On the Return Stack

Each call to a lower level word in SNAP pushes TWO words onto the return stack. The word at the top of the return stack is the BANK ID the return address is in. The word below the top is the return address.

The words 2>R and 2R> are provided to push and pop the bank ID and the return address in one operation.

2.5 THE LATCH BYTE

The **latch byte** is a byte of flags kept in a memory location occupied by special write-only hardware. It is set by nucleus routines, and used directly by other parts of the HHC hardware.

Application programs should avoid setting the latch directly.

Since the latch byte is write-only, its contents cannot be fetched directly. To get the value of the latch byte, fetch LATCHS ("latch shadow"), which is kept equal to the latch byte by the nucleus.

Value (hex)	Word	Meaning
1 2		Indicate which intrinsic capsule socket (if any) is currently bank-switched into the address space. Values 00B, 01B, and 10B represent sockets 0, 1, and 2. Value 11B is used to switch in an external ROM.
4	CAPON	1-> a capsule socket is bank-switched into the address space 4000-7FFF. 0-> the I/O hardware is switched into the address space 4000-7FFF.
8	DISPON	1-> LCD is turned on, 0-> off.
10		1-> addressing slow ROM or I/O ports 0-> normal.
20	BEEPER	High/low DC to speaker. SQUEAK toggles this bit to generate audio.
40	CPUON	1-> CPU power on, 0-> off.
80		Select function of memory mapped area for keyboard/LCD functions. 1-> LCD control, 0->keyboard control.

CHAPTER 3: INTRODUCTION TO THE SnapFORTH CAPSULE

The SnapFORTH capsule is capable of producing programs for 2K, 4K, and 8K-byte EPROM capsules. An HHC EPROM burner is currently under development. When it is available, the user will be able to develop a program with the SnapFORTH capsule, enter the menu of the EPROM burner, and choose activities that will make one or more EPROM capsules containing his program. These capsules will then run in any HHC, even without the SnapFORTH capsule installed. It is also possible to produce the complete memory image of a capsule, and send it to a serially interfaced RS232C EPROM programmer, using the HHC's RS232C peripheral. The Apple II using HHC Development Tool software, and the Apparat EPROM programmer card can be used for this purpose. A section below describes this procedure.

SnapFORTH can compile and assemble either source entered from the keyboard or from a file. The file can be created by the built-in File System editor or by the Portawriter word processor capsule. It is also possible to load a file from a peripheral (e.g. from another computer such as the Apple II) using the DEVLOAD feature.

SnapFORTH programs can call "library" routines from other capsules. In particular, the Scientific Calculator contains a large collection of highly accurate and useful scientific functions that are available for SnapFORTH programs, as long as the Scientific Calculator is present in the HHC.

SnapFORTH is also available through a cross compiler which runs on the APPLE II. This manual, however, describes procedures which are specific to the capsule.

3.1 COMPILER OPTIONS

The SnapFORTH compiler has six options:

%INT - default: set
%FORW - default: clear
%AUTOCAP - default: set
%0OPT - default: set
BEEPER - default: set

The function of each of these options is described below. A separate copy of these control bits is kept for each dictionary file. When you create a new dictionary file the control bits are set to their default values. You can give them new values, and

they will remain their new value next time you enter the dictionary file. To set a compiler control bit, type:

```
SET <name>
```

To clear an option, type:

```
CLR <name>
```

Options can be set and reset only in immediate mode, not inside a definition! For example:

```
CLR BEEPER  SET %FORW
```

%REDEF

When set, SnapFORTH gives a message every time you redefine a name.

%INT

When set, "internal" routines can be used inside a target definition. IF YOU CLEAR %INT, you can prevent an application from using routines which may not be present when the application is executed (i.e. routines from the SnapFORTH capsule itself). This is explained further in the chapter on capsule generation.

%FORW

When set, you can use words before they are defined. Depending on the context in which they appear, the compiler assumes what type the forward definitions must have ('= =' or ':C'). Later occurances of the forward and its definition must agree with this assumption or an error message is given. The next chapter explains in detail how this works.

%AUTOCAP

A stand alone application may need a so-called "CAP-INIT word" to set up the "tag table vector" before it can run on the HHC. When %AUTOCAP is set, the compiler generates the capinit word itself when you execute "TABLES". Your mother word can call this routine through "CAPINIT". If no capinit word is needed (i.e. when you use only one long tag table) CAPINIT does not do anything.

%0OPT

SnapFORTH colon-definitions begin and end with a zero byte. The first zero byte is the 6502 opcode BRK, and the last zero byte is the short tag EXIT. When you define two subsequent colon-definitions, the compiler can perform "Break optimizing"; it does not generate a BRK for the second definition, but rather makes the last byte of the first colon-definition overlap with the first byte of the second definition (since both are zero anyway). This saves one byte. However, the compiler does not really check whether a ":" definition was preceded by another ':' definition; it just checks if the last byte in the dictionary is a zero byte. When this zero byte was actually from a forward reference which will be patched later, the optimizing is illegal and must therefore be turned off (and turned on later when we are out of the danger zone).

BEEPER

When set, the compiler clicks at the end of every line loaded.

3.2 DICTIONARY VARIABLES

For each dictionary file, the SnapFORTH compiler keeps a set of variables describing the dictionary and the state of the compiler. The space for these variables is allocated in the dictionary itself, not in an AREA (otherwise they would not remain when you temporarily leave the dictionary). Of course you cannot use these variables in a stand alone application (they are meant to be compile-time-only anyway).To make a clear distinction between compile time and run time variables, the dictionary variables have a different implementation; dictionary variables use the "TO concept".

When you execute a "TO variable" it leaves its ***value*** on the stack, not its address. '!' And '+!' are not used for these variables; to store into a TO variable type:

```
<value> TO <name>
```

Example:

```
6 TO WIDTH
```

You can add to the contents of a dictionary variable by using '+TO'. Example:

```
2 +TO WIDTH
```

The implementation of TO variables is as follows: a flag called '%VAR' is used to control the operation of the variables. When a TO variable is executed and %VAR is 0 (its usual value), it leaves its value on the stack. When %VAR is +1 the variable takes its new value from the stack (and resets %VAR). When %VAR is -1 the value on the stack is added to the contents of the variable.

The following routines can be used to change the value of %VAR:

TO - Set %VAR to +1
+TO - Set %VAR to -1
FROM - Set %VAR to 0
((- Save %VAR on return stack and reset %VAR.
)) - Restore %VAR from return stack.

You can create your own dictionary variable by typing:

```
VARIABLE <name>
```

It is also possible to apply the same operations to an arbitrary address, 'EXECTO' expects an address on the stack, and fetches/stores a 16-bit value, depending on %VAR.

3.3 BASE

BASE is not a dictionary variable. Instead, BASE is a nucleus variable. This means that there is only one copy of BASE and that its value is not preserved when you leave a dictionary file and enter it at a later time. CLEAR sets ***BASE*** to decimal.

BASE leaves the address of a byte on the stack.

CHAPTER 4: SnapFORTH CAPSULE; TECHNICAL INFORMATION

The SnapFORTH capsule uses two workfiles:

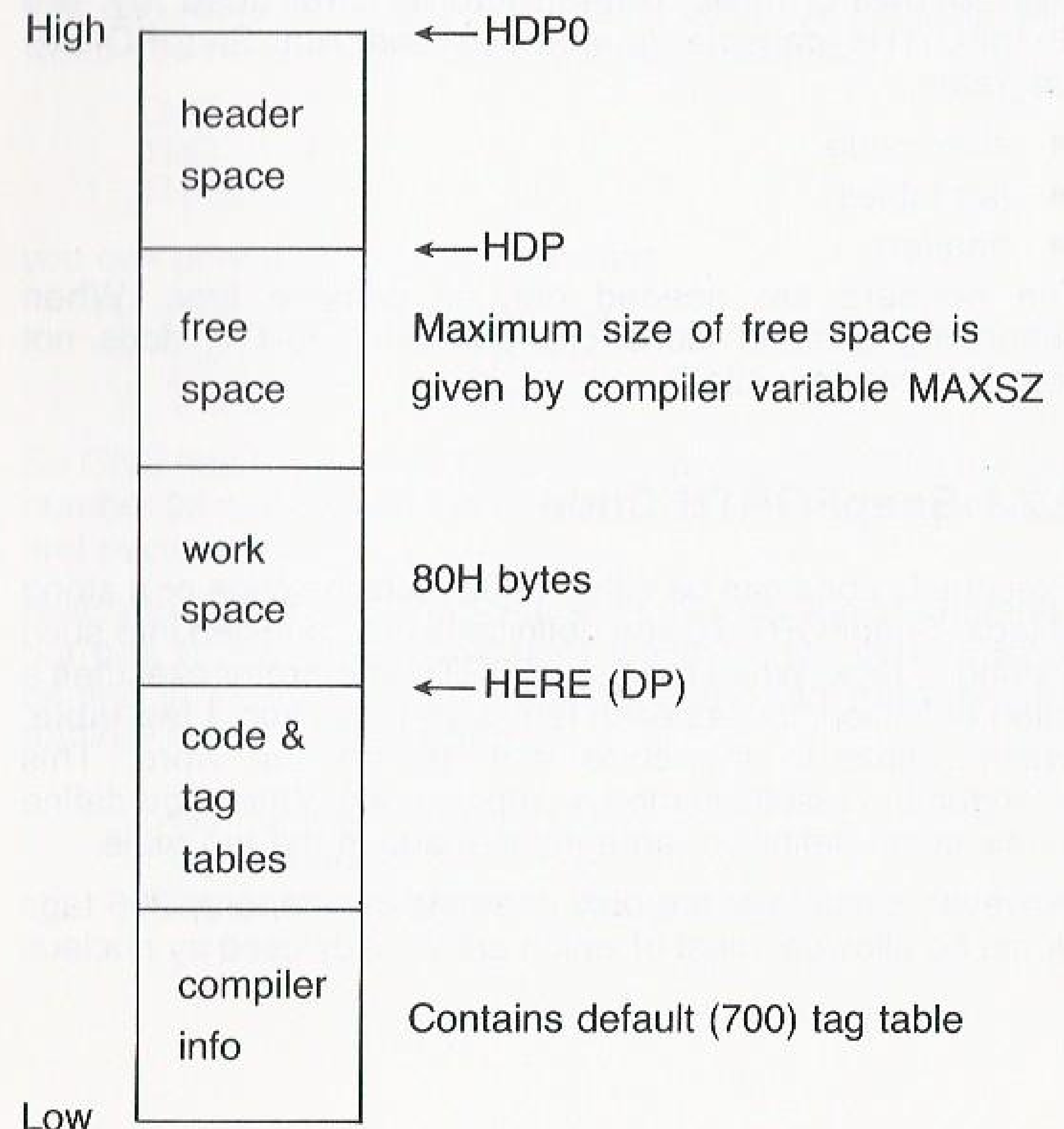
- A dictionary file
- An internal workfile called "S"

You can create an arbitrary number of dictionary files. Only one internal file is needed, it is automatically created by the SnapFORTH capsule when necessary. Therefore, you need not bother about the internal structure of "S".

The dictionary file contains both executable code and symbol table information. The executable code is located in low memory, and the symbol tables are located in high memory, the so-called "Header space". We will call one symbol table entry a **Header**.

4.1 MEMORY LAY-OUT

The following figure illustrates the dictionary lay-out:



The **code & tag tables** part is the one that may eventually be used to generate a stand alone capsule, the other parts are needed only during compilation.

The **compiler info** part is made by the SnapFORTH capsule when the dictionary is created.

The code space grows towards high memory, and the header space grows downwards. When the free space between both becomes too small, the dictionary is extended, and the headers are moved upwards. Since the SnapFORTH capsule knows the internal structure of the header space, it is able to update the links between the headers. After moving, the free space is MAXSZ bytes.

Similarly, when the free space becomes more than MAXSZ, the headers are moved downwards again.

The space above HERE is used by the compiler as workspace.

4.2 DATA STRUCTURES IN THE SnapFORTH CAPSULE

The following three datastructures are used by the SnapFORTH capsule to compile and run SnapFORTH programs:

- snap code
- tag tables
- headers

The headers are needed only at compile time. When generating a stand alone capsule, SnapFORTH does not copy the headers into it.

4.2.1 SnapFORTH Code

Executable code can be either 6502 machine code or a string of tags. SnapFORTH colon definitions are compiled into such a string of tags. When the SnapFORTH interpreter executes a colon definition, it uses each tag as an index into a **tag table**, which points to the code that defines the word. This arrangement results in more compact code. When you define a new colon definition, an entry is made in the tag table.

However, since tags are only one byte in size, only 256 tags would be allowed, most of which are already used by nucleus routines. Therefore the tags 1 to 8 have a special meaning, they are used as **escape tags**; when the SnapFORTH interpreter encounters one of these it uses another tag table to execute the next tag, thus allowing for eight additional, user defined, tag tables.

Thus it takes 2 bytes to execute a tag from one of those additional tag tables (the escape tag and the tag itself). Therefore these tags are called **long tags**. Tags which do not require an escape tag are called **short tags**. Only the first 192 short tags are used by nucleus routines. They are called the **short intrinsic tags**. If you declare a tag table for them, the remaining 64 shorts can be used by your program, thus reducing its size. They are called **short extrinsic tags**.

Two long tables are already in use:

- Table # 1 is used for nucleus words.
- Table # 8 is used by the SnapFORTH capsule.

So if you want to, you can define one short tag table and up to six long tag tables. To make things easier, the SnapFORTH capsule has already reserved 64 tags for you in table # 7, the **700 table**. It is possible to extend to number of tags in the 700 table to 256.

You can retrieve the tag of a routine using **'X**. For example, when you have definined the following routines:

```
: ONE   ;
: TWO   ;
: THREE ;
```

you can print their tags by executing:

```
'X ONE   H.  0007 ok
'X TWO   H.  0107 ok
'X THREE H.  0207 ok
```

So ONE has tag number 00 and escape tag 07, TWO has tag number 02 and escape tag 07 and THREE has tag number 03 and escape tag 07.

Notice that the escape tag actually precedes the tag number, but due to the idiosyncrasies of the 6502 the byte order is printed in reverse. When referring to the **tag number** of a routine, we will use this format:

tag number of one = 700
tag number of two = 701
tag number of three = 702

How to read a compiled routine?

If you define a routine like:

```
: TEST TWO 2+ THREE ;
```

You can dump the body of the routine by:

```
'V TEST DUMP
```

It will read something like:

```
061F   00 07 01 41 ...A
0623   07 02 00 04 ....
```

Where 061F is the address at which TEST's object code is stored. The four two-digit numbers are the contents of the bytes beginning at 61F, and the following charcters are the graphic representations of the ASCII values. Dots signify no graphic representation. Press down arrow to display more bytes, ENTER to terminate.

00 is another special tag which will be discussed later. The routine body breaks down into:

00	6502 machine instruction BRK
07	Introduces a long tag from the 700 table,
01	which happens to be TWO
41	A short tag: 2+
07	Another long one from the 700 table,
02	THREE
00	Another short one: EXIT
04	is the code following the end of TEST's object code.

Short tag #00 is a special one!

As we've discussed earlier, the HHC can either directly execute 6502 machine code or SnapFORTH code (tags). It needs an interpreter to execute the SnapFORTH code. But how does the HHC know if a routine was written in machine code or in Snap? Every routine starts as a machine code routine; when it is actually a SnapFORTH routine, it begins with a single 6502 instruction: **BRK**.

When a SnapFORTH routine is executed by the HHC, the BRK instruction calls the SnapFORTH interpreter which will interpret the remainder of the routine body. Short tag number 00 (nucleus routine **EXIT**) must be executed by the SnapFORTH routine in order to return to the routine which called it.

The opcode of the BRK instruction also happens to be zero, so a SnapFORTH routine both begins and ends with 00!

This fact is (mis)used by the SnapFORTH compiler for **code optimizing**. When it compiles a colon definition it checks if the byte at HERE-1 happens to be zero. If so, it does not compile a BRK for the definition, but uses HERE-1 as its entry point instead, thus saving one byte code. Since colon definitions are often preceded by other colon definitions (ending with EXIT) this is often the case.

There is one serious risk in this strategy; when the zero byte at HERE-1 was not the EXIT tag of a preceding colon definition but an unsolved forward reference instead, it may not remain zero throughout the compilation!

The compiler control bit %0OPT controls this optimizing. The compiler optimizes when you execute:

```
SET %0OPT
```

This is the default setting. You can forbid the optimizing by resetting %0OPT:

```
CLR %0OPT
```

When must I make my own tag table?

While you are developing a program in the dictionary file that you selected when you first entered SnapFORTH, you can ignore tag tables, and other constraints involved in making a capsule or RUN SNAP PROGRAM (executable file). As long as you have a small program that defines fewer than 65 (decimal) tags, you will be making tags in the 700s. More than 64 and you will need to set up a tag table.

How to create a tag table?

As we've explained, tag tables are used by the SnapFORTH interpreter to execute the tags that form a SnapFORTH colon definition. Users of the SnapFORTH capsule can define six tag tables:

- A short tag table, containing up to 64 tags.
- Five long tag tables (numbers 2-6), containing up to 256 tags each.

Tag table #7 has already been created for you by the SnapFORTH capsule.

You can define your tag tables using **TABLES**:

```
#shorts ext# #longs TABLES <name>
```

Where

- **#shorts** is the number of short tags in this table. Typically 64 or zero.
- **ext#** is the extension number (escape tag) of the first long tag. For example, when your long tags start in the 200s, ext# is 2.
- **#longs** is the number of long tags in this table.
- <**name**> is the name of the tag table.

#longs may be more than 100H, for instance, when you execute:

```
HEX 0 2 140 TABLES HOWMANY
```

two long tables are defined: the 200 and the 300 tag table. The latter will contain tags 300H up to 340H.

How many tag tables can I define?

You may define several tag tables, but they may not overlap. Only one of them may contain short tags. Thus, the following is illegal:

```
HEX 0 2 101 TABLES tab1
    0 3 100 TABLES tab2
```

since tag #300 appears in both tables.

It is often more convenient to define all of your tables at once. To reserve all possible tables, execute:

```
HEX   40 2 400   TABLES   MYTAGS
```

Now you have two sets of tables at your disposal:

MYTAGS : Your own table, can hold the following tags:

- 64 short tags
- long tags 200 through 6FF

700TAGS : This table is generated by the SnapFORTH capsule.

- long tags 700 through 73F

You can start using a tag table by executing its name. From that moment on, all new tags will be entered into that table. For example;

```
MYTAGS  : ONE ;
700TAGS : TWO ;
'X ONE H. 00C0
'X TWO H. 0007
```


When to use the '700TAGS' table?

If your program must eventually be burned into a stand-alone capsule, you must make a distinction between routines which are needed only at compile time and routines which will be needed at run time.

Only your own tables are burned into the capsule, the 700TAGS table is not. Therefore you must use the 700TAGS table for compile time routines, otherwise you would be wasting valuable tag table space in your application capsule.

Capsule generation is discussed in depth in the next chapter.

How to create short tags?

By using SnapFORTH compiler directives, you can control which of your tags are assigned short tags, and which are assigned long tags. This is useful, since you can optimize your program for space by assigning short tags to the words that are most frequently ***coded***, or optimize it for speed by assigning short tags to the words that are most frequently ***executed***. (These two goals only conflict if there are not enough short tags to go around.) In general, the speed saved by short tags is not major.

If the current tag table has reserved space for short tags, the SnapFORTH capsule begins assigning short tags to the words it compiles. You can make it switch to compiling long tags using the compiler directive

```
LONG.TAGS
```

And you can switch back again to compiling short tags by:

```
SHORT.TAGS
```

The SnapFORTH capsule remembers for each table whether it must compile long or short tags. For example:

```
MYTAGS SHORT.TAGS          : ONE ;
700TAGS                    : TWO ;
MYTAGS                     : THREE ;
```

ONE and THREE will be assigned short tags, and TWO will be assigned a long tag.

When the short tag table is exhausted, the SnapFORTH capsule will print:

```
out of shorts
```

and automatically switch to compiling long tags.

LONG.TAGS and SHORT.TAGS are global compiler directives, all tags in the current table are assigned long/short tags. It is also possible to assign long/short tags to a specific name. The compiler directive **LONG** guarantees that a specific name will be assigned a long tag:

```
LONG name
```

LONG is used ***before*** the name is defined. It actually reserves space in the current tag table for the specified name, incrementing the counter that determines what tag number will be assigned to the next-defined tag.

Similarly, the compiler directive **SHORT** guarantees that a specific name will be assigned a short tag (unless no more short tags are available in the current tag table):

```
SHORT name
```

Like LONG it is used before the name is defined. LONG and SHORT take precedence over the LONG.TAGS and SHORT.TAGS directives.

It is good programming practice to declare all short tags at the beginning of your program. For example:

```
HEX 40 2 50 TABLES MYTAGS
SHORT short1
SHORT short2
SHORT short3
```

4.2.2 Implementation of Tag Tables

The HHC uses a table called the **Tag Table Vector** in order to execute a long or a short extrinsic tag. The tag table vector contains the addresses and the **bank numbers** of the short extrinsic tag table and the eight long tag tables. The bank number tells in which capsule a tag table can be found, the address tells were it can be found within that capsule.

Before trying to look up a tag in any extrinsic tag table, the inner interpreter checks the table's bank ID to make sure the table is in the address space. If not, it switches the proper memory bank into the address space automatically. It switches back, if necessary, after the tag has been executed.

If an entry in the tag table vector points into capsule ROM, you must add 80H to the bank number. The HHC's intrinsic RAM and ROM (nucleus) both have bank number 0, indicating that they can never be switched out.

The nucleus constant TVECT0 points to the tag table vector. The first thing that must be done by an application capsule is moving the the tag table vector for that particular application to

TVECT0. Only once this has been done, you can execute routines from the application capsule.

Generally speaking, you need not bother about the tag table vector while developing your application with the SnapFORTH capsule, because SnapFORTH has already created a tag table vector for you, and every time you execute TABLES, it sets the appropriate pointers in the tag table vector.

When you turn your program into a stand alone capsule, it may be necesary to initialze the tag table vector yourself. The next chapter discusses how this can be done.

Example

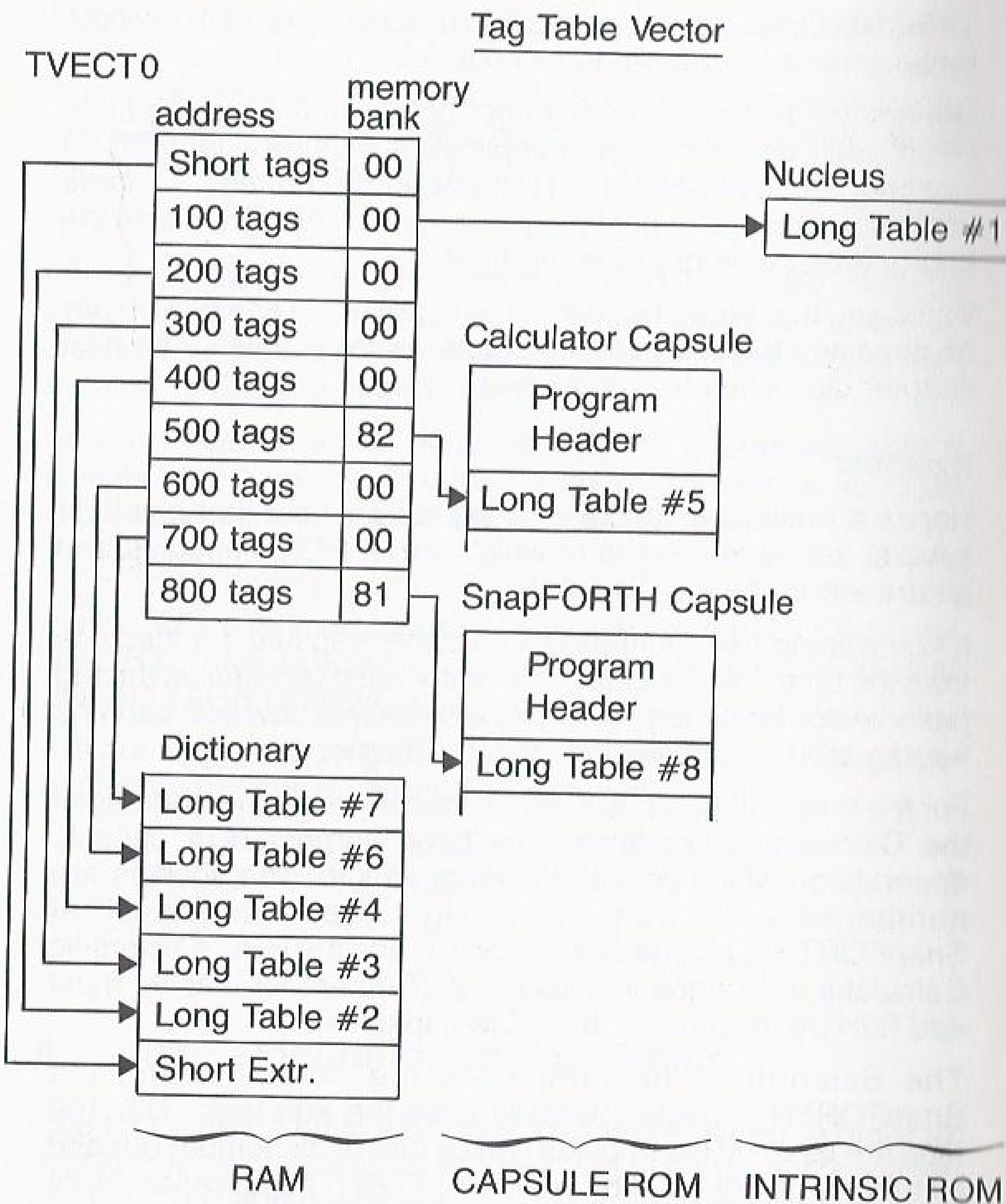
Here's a realistic example of a tag table vector that points to several tag tables, some of which are in ROM capsules and others are in the HHC's RAM.

If you want to use routines from another capsule, for instance from the Scientific Calculator, an entry must be made in the tag table vector for its tag table. We will discuss how you can find the tag table of another capsule in Chapter 6.

For the time being, let us assume that we know the address of the Calculator's tag table. The bank number of a capsule depends on which socket it is plugged into. The sockets are numbered 1,2,0 from left to right. Let's assume that SnapFORTH is plugged into socket 1, and the HHC's Scientific Calculator is plugged into socket 2. (Remember that we must add 80H because both are ROM capsules).

The Scientific Calculator uses the 500 tags and the SnapFORTH capsule internally uses the 800 tags. The 100 tags are used by the nucleus (which has bank number 00) and all other tags are used by your SnapFORTH application. Note that the SnapFORTH dictionary is in RAM, not in the SnapFORTH capsule!

The tag table vector for this configuration is shown in the following figure.



Inside a tag table

The following symbols define the current tag table:

#shorts - Number of shorts in the current table.
#longs - Number of longs
ext# - extension number of the first long tag.
stag# - Next short tag.
ltag# - next long tag.

The short tags are allocated after the longs. You can change this field using TO, but be careful. You could use this for instance to extend the 700 table. For each "TABLES", the end of the tag tables created is marked by two zero bytes.

How to extend the 700 table

When you enter a new dictionary file, the 700tags table is located immediately before HERE. Sixty-four tags are reserved

in the 700 tables. To extend the 700tags table, allot dictionary space for it and update #longs. Don't forget to add the two zero bytes! The tag tables must be initialized with the address of the routine that gives the "UNDEFINED" message. This can be done by copying part of the existing tables into the new table space.

For example, to extend the 700tags table from 64 to 99 entries:

```
99 64 - DUP 2* 2- ALLOT 0 , +TO #LONGS
TT.ORIGIN DUP 2+ #LONGS 1- 2* CMOVE
```

TAG

TAG lets you create a header and a tag for a routine, before either the routine or the tag table in which its tag is to appear are defined. When that tag table is eventually defined, you must skip the entries which you have already allocated yourself, using STAG# or LTAG#. In the following example two long tags and one short tag are predefined using TAG, when the tag table in which they appear is defined we must skip some entries:

```
C0 TAG SHORTC0
0002 TAG LONGZERO
0102 TAG LONGONE
...........
HEX 40 2 100 TABLES MYTABS
1 +TO STAG#          \ Skip C0 entry
2 +TO LTAG#          \ Skip 200 and 201 entries
```

Types of definitions

To be honest, our previous discussion of SnapFORTH code was somewhat oversimplified. Actually SnapFORTH distinguishes three kinds of definitions:

- Colon definitions having a tag (defined by ':')
- Colon definitions without a tag (defined by ':C')
- Literals (defined by '= =' or ':P')

The difference between these is the code generated when you compile them into another definition.

Tags are compiled into an escape tag and a tag number (longs) or only a tag number (shorts). There's a small space overhead in this approach because you need an entry in a tag table in order to execute a tag. When you use a routine only once, it may be more efficient to define it with ':C'. In that case SnapFORTH compiles one short tag called (CALL), followed by the address of the routine. So, the calling sequence takes 3 instead of 2 bytes, but the definition consumes no tag table space.

Literals are compiled into a short tag followed by a value. The value can be either 8 or 16 bits large. The tag can be one of these:

LIT : Push 16 bit value on the stack.
CLIT : Push 8 bit value on the stack.
CLIT2 : Add 200H to 8 bit value and push it.
CLIT3 : Add 300H to 8 bit value and push it.

The compiler chooses whatever tag gives the shortest code.

Literals can be defined by '= ='. For example:

```
999 == NINES
```

Notice that the difference between '= =' and the traditional defining word **CONSTANT** is the code generated when you refer to a word. CONSTANT words compile into a tag, whereas '= =' words compile into the code described above.

:P allows you to define a SnapFORTH word that can be called from a low level (i.e. 6502 assembly language) routine. Since low level routines behave differently from SnapFORTH routines, there is no use in assigning a tag to a ':P' definition. You'll seldom want to use it in a high level routine anyway. A typical calling sequence is:

```
:P HI-LVL EMIT ;P

CODE PRINT ( Address --- )        \ Print string
   BEGIN,   0 X)             LDA,  \ with zero
   0= NOT WHILE,                   \ terminator.
                             DEX,
                             DEX,
               TOP           STA,
               0 #           LDA,
               TOP 1+        STA,
               HI-LVL        JSR,
               0 ,X          INC,
        0= IF, 1 ,X          INC,
        THEN,
   REPEAT,
               POP           JMP,   ENDCODE
```

The address of HI-LVL is stored as a literal (a label) in the header space. If you want to call HI-LVL from within another high level routine, you must say:

```
.... HI-LVL CALL ....
```

':P' and ';P' will be discussed in depth in Chapter 16.

4.2.3 Headers and Vocabularies

All information relevant to the SnapFORTH compiler, such as the names and tag numbers of routines, is stored in so- called "headers". The headers are kept as linked lists, that is, each header contains the address of the previous one. Such a list is called a **vocabulary**. Vocabularies may be nested; a vocabulary may contain several other vocabularies. However, the headers in those vocabularies do not show on the VLIST unless you "activate" a vocabulary by using its name. Example, the ASSEMBLER vocabulary is embedded in the SnapFORTH vocabulary. When you execute VLIST, only the name of ASSEMBLER itself is shown. When you type "ASSEMBLER" and then execute VLIST again, you also see the names in ASSEMBLER (NOP, BEQ, and the like).

Initially, SnapFORTH knows three vocabularies:

- FORTH : The Master vocabulary
- ASSEMBLER : Contained in FORTH
- FORWARD : Used to handle forward references.

The FORWARD vocabulary is a special one; vocabularies are organized tree-wise, all vocabularies are **chained** to their parent and the SnapFORTH vocabulary is the root of the tree. But FORWARD is not chained to any other vocabulary.

Implementation of headers

A header consists of three fields:

- link field: two bytes
- name field: a string
- parameter field: three bytes

You need to know the addresses of these fields to access them. The addresses are called:

LFA: Link Field Address
NFA: Name Field Address
PFA: Parameter Field Address

' (tic) is used to find the PFA of a header, given its name. Example, print the PFA of VLIST:

```
' VLIST H.
```

Once you know the PFA or the LFA, you can compute the other addresses using the following routines:

NFA: (PFA --- NFA)
LFA: (PFA --- LFA)
PFA: (LFA --- PFA)

For instance, to print the PFA, LFA and the NFA of VLIST you can execute:

```
' VLIST     H.
' VLIST LFA H.
' VLIST NFA H.
```

Notice that you can "travel" from LFA to PFA, but you cannot travel from NFA to PFA (but the need will probably never arise).

Headers in the FORWARD vocabulary contain some additional fields.

The link field or LFA

Every header is chained to the previous header in the same vocabulary; when you make a new definition its link is set to the LFA of the previous definition. The links are used by the SnapFORTH compiler to find its way around in the header space.

The SnapFORTH implementation of FORGET requires that all links point "upwards" (to higher addresses than the LFA itself). Since the header space grows downwards, this restriction is always obeyed by routines like ':'. But if you want to make your own headers, you must be careful.

The name field or NFA

The format of the name field is somewhat like that of an ordinary string; a count byte followed by a number of characters. But there are some differences. Only the lower 5 bits of the count byte are used for the length of the name. Bit 7 (the 80's bit) of both the count byte and the last character must be set. All other bytes in the name field must have bit 7 reset. This is useful for finding the beginning and the end of the name.

It may seem superfluous to have two ways of recording the length of a name; the count byte and the two 80 bytes. But SnapFORTH uses this redundancy in a clever way. You can save space by truncating all names down to a certain length. In that case the count byte contains the length of the name as it was before truncating. When SnapFORTH searches for a name in the header space, it first compares the length of the names. If they match, the rest of the names are compared. Comparison stops when an 80's bit is encountered. If we had not used the 80's bit method, we would have to use another byte to record the actual length of the name.

The compiler variable WIDTH controls the truncating of names. For instance, you can have all names truncated down to 3 characters by typing:

```
3 TO WIDTH
```


The names of all nucleus routines are truncated down to 5 characters.

Bit 6 of the count byte is the **immediate bit**. If it is set, the routine is an immediate one, otherwise it isn't.

Bit 5 of the count byte is used to record whether the routine has ever been a forward reference. It is used by FORGET.

The parameter field or PFA

The parameter field falls into two subfields:

- the type field (one byte)
- the value field (two bytes)

The **type** field records the type of the header. Currently, three types are supported.

0: a routine with a tag (e.g. one defined by ':')
1: a literal (defined by '= = ' or :P)
2: a routine without a tag (defined by ':C')

You can implement other values yourself if you like, but routines having other types than 0, 1 or 2 can not be interpreted by SnapFORTH. (But of course you can access these headers using "'" and manipulate them yourself).

The meaning of the **value** field depends on the contents of the type field:

0: a tag number (in the same format as 'X delivers)
1: a value
2: an address.

A routine CFA exists which fetches the execution address of a routine. Its parameters are:

(PFA --- Execution address)

If the type byte at PFA is 1 or 2, CFA fetches the value or address, respectively. Otherwise CFA uses TVECT0 to fetch the execution address from the tag table.

Creating your own headers

You can use (CREATE) to create your own header. (CREATE) expects the value and the type of the header on the stack:

```
<value> <type> (CREATE) <name>
```

For example, the following statement is another way of saying '9 = = CON':

```
9 1 (CREATE) CON
```

FORGET and SHRNK

FORGET is used to remove a range of routines and their headers. For example, when you enter the following definitions:

```
CREATE ONE
CREATE TWO
CREATE THREE
```

and you type:

```
FORGET TWO
```

SnapFORTH removes both THREE and TWO. You can protect routines using **FENCE**. FENCE must be set to the LFA of the first protected routine. For instance, if you had executed:

```
' TWO LFA TO FENCE
```

before you executed FORGET TWO, only THREE would have been forgotten. You can protect everything up to the last definition by:

```
HDP TO FENCE
```

(not by HERE TO FENCE).

The executive routine of FORGET is called **(FORGET)**, it expects and address on the stack which is the PFA+3 of the last routine that may be forgotten.

Since SnapFORTH keeps the headers separate from the code, it is also possible to remove only a range of headers, using **SHRNK**. This is useful when you run short of memory, you can then remove obsolete headers to regain some memory. To call SHRNK, type:

```
SHRNK <firstname> <lastname>
```

Since SHRNK reorganizes the symbol table space, the previous value of FENCE is now garbage. Therefore you must reinitialize FENCE after SHRNK.

When you try to forget a name in ROM (e.g. FORGET FORTH), nothing happens.

Vocabularies: CURRENT and CONTEXT

A vocabulary is a set of related words, you can define one using the defining word VOCABULARY:

```
VOCABULARY MYVOC
```

You can do three things with a vocabulary: enter new definitions into it, list the names in it (VLIST) and search it for definitions that were entered earlier.

You can open a vocabulary for searching and listing by executing its name. For example, when you type:

```
ASSEMBLER
```

the ASSEMBLER vocabulary is opened and the names in it appear in the VLIST. When you open another vocabulary for listing and searching (e.g. MYVOC), the names in ASSEMBLER become invisible again.

After you have opened a vocabulary for listing and searching, you can open it further, in order to enter definitions into it. This is done by the word DEFINITIONS. For example

```
ASSEMBLER DEFINITIONS
```

enables you to enter new definitions into the ASSEMBLER vocabulary. From now on you can open other vocabularies for searching and listing, but as long as you do not execute DEFINITIONS, the compiler continues to enter new definitions into ASSEMBLER.

Let's have a look at the implementation of this mechanism: The compiler variables CURRENT and CONTEXT both contain the address of a vocabulary. Upon entrance of the SnapFORTH capsule, SnapFORTH is both the CURRENT and the CONTEXT vocabulary. CURRENT and CONTEXT are used as follows;

- VLIST lists the CONTEXT vocabulary and all vocabularies chained to it.
- During compilation, the CONTEXT vocabulary and all vocabularies chained to it are searched first, and if a word can't be found there, a secondary search is made starting from CURRENT.
- New definitions are entered into CURRENT.
- When you execute the name of a vocabulary, it becomes CONTEXT.
- DEFINITIONS is defined as:

```
: DEFINITIONS CONTEXT TO CURRENT ;
```

- CODE, CODEC, ;CODE and LABEL set CONTEXT to ASSEMBLER
- Generally speaking, when you set CONTEXT to another vocabulary than CURRENT, you do not want it to remain different forever. It would be cumbersome if you had to reset CONTEXT yourself. Therefore the following words perform CURRENT TO CONTEXT:

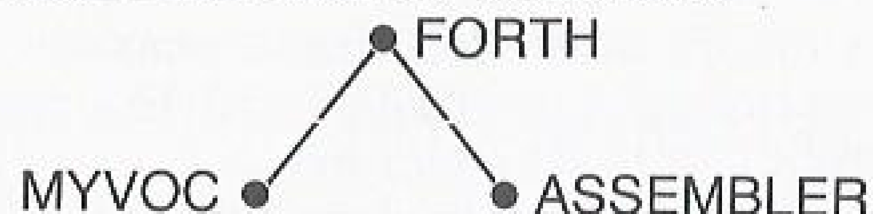
```
ENDCODE : :C :P FORGET
```

Chaining vocabularies

When you manipulate CURRENT and CONTEXT as we described in the preceding section, you have some control over the order in which the SnapFORTH compiler searches through the vocabularies. For instance, if you enter the following commands:

```
FORTH DEFINITIONS
VOCABULARY  MYVOC
```

Your vocabularies are chained as follows:



But it turns out that you do not yet have complete control over the searching order. If you want MYVOC to be searched first and ASSEMBLER to be searched second, you could make MYVOC context and ASSEMBLER current. But since both MYVOC and ASSEMBLER are chained to FORTH, the actual search order is MYVOC FORTH ASSEMBLER FORTH instead of MYVOC ASSEMBLER FORTH.

However, you can change the tree using CHAIN. CHAIN inserts the named vocabulary after the context vocabulary. So, if you execute:

```
MYVOC DEFINITIONS
CHAIN ASSEMBLER
```

the tree looks like:

BEWARE Never chain a vocabulary to itself. If you do so, you may loose the link to FORTH.

Vocabularies: implementation

Vocabularies are implemented by the following data structure:

cfa	cfa+3	cfa+5	cfa+7	cfa+9	cfa+11
JSR Dovoc	Highest	Dummy	Parent	Dummy	V-link

Where:

JSR Dovoc - The executable code for the vocabulary.
Highest - Link to Highest definition in this voc.
Dummies - Dummy names for VLIST
Parent - Vocabulary to which this one is chained
V-link - Link to next vocabulary for FORGET

when you enter the following commands:

```
FORTH DEFINITIONS
VOCABULARY MYVOC
MYVOC DEFINITIONS
CREATE M1
CREATE M2
```

The linkage is as follows:

- The highest field of MYVOC points to the LFA of M2.
- The link field of M2 is linked to the LFA of M1.
- The link field of M1 is linked to the parent field of MYVOC.
- The parent field of MYVOC is linked to the highest field of FORTH.
- The highest field of FORTH points to the LFA of MYVOC.

Now we understand the function of the dummy names, since the linked list includes both the highest and the parent field of MYVOC, these field appear in the VLIST. Therefore empty names must be added for VLIST.

When you create several definitions, switching vocabularies as you go along, the vocabularies become "interwoven". When you want to FORGET a range of definitions within a vocabulary, FORGET must be able to detect whether that range also includes definitions belonging to other vocabularies. This is the function of V-LINK. The V-LINK fields of all vocabularies are chained in the order in which they were defined.

Forward references

As a natural consequence of the tag code used by SnapFORTH, you can use a colon definition before you define it. All you have to do is entering an entry for it into the tag table using LONG or SHORT:

```
SHORT HI
: SILLY HI ;
: HI ." Hello world" ;
```

When SILLY is compiled, the SnapFORTH compiler only needs to know the tag number of HI. Once HI itself is defined, its CFA will be entered into the tag table and you are able to execute SILLY. When you try to execute SILLY before HI is defined, an error message is given:

```
ERROR: UNDEFINED silly
```

Forward referencing literals or ':C' definitions is a more complicated matter. By default, the compiler does not recognize these references. But you can force it to do so by setting the %FORW control bit:

```
SET %FORW
```

From now on, the compiler assumes every unknown word that it encounters is a forward reference. The type of a forward depends on the context of its first occurance.

If a forward reference is first encountered within a colon definition, it is assumed to be a ':C' definition.

If it is first encountered outside a colon definition (i.e. when STATE is not compiling), the compiler assumes the forward to be a literal (usually a label). Example:

```
CODE TEL ( Address --- # chars before
                                  zero )
     1 #          LDA,
     SETUP        JSR,        \ Carry always
     >START       BCC,        \ clear after
   BEGIN,         INY,        \ SETUP
 LABEL >START N )Y      LDA,
   0= UNTIL,
                  TYA,
     CPUSH        JMP,   ENDCODE
```

>START is a forward reference. When it is first encountered, STATE is not compiling (SnapFORTH assembles by executing definitions like LDA,) and therefore it is assumed to be a literal.

Forgetting forwards

SNAPFORTH keeps record of all references to a forward. When a forward is patched, SnapFORTH does not throw away its list of references. In this way it can make all references undefined again when you FORGET the forward.

Example:

```
CODE $A    FWD 3 + #      LDY,
                          ...   ENDCODE
```

The forward is patched when you define:

```
CODE $B       ....                ENDCODE
5 == FWD
```

The reference to FWD now becomes 8 # LDY. It becomes undefined again when you execute FORGET $B. Now, when you define FWD a second time, the forward reference is patched again:

```
6 == FWD
```

changes it into 9 # LDY, .

Only one forward at a time allowed

You can only use one forward at a time. Constructs like:

```
FWD1 FWD2 SWAP # LDA,
```

are confusing because the compiler does not know which forward to patch. Therefore it is forbidden to have more than one forward "active". If you attempt to use more than one forward, the message:

```
FORWARD NOT ALLOWED
```

is given.

A common cause for this error is misspelling a defining word. For example:

```
SET %FORW
XCAR BLABLA            ( Should be CVAR BLABLA )
```

The compiler does not recognize XCAR and assumes it is a forward. But now BLABLA, which has not been scanned by CVAR, is also assumed to be a forward reference and we have two subsequent "forwards"!

Implementation of forwards

The entries in the FORWARD vocabulary have the following structure:

For each reference:

LFA 9- Link to next forward reference.
LFA 7- Patch type:
 0 - 2 Bytes absolute
 1 - 1 Byte relative (BNE, etc.)
 2 - 1 Byte absolute low (#L)
 3 - 1 Byte absolute hi (#H)
LFA 6- Addresss where to patch.
LFA 4- Value which must by added when patching.

For the header only:

LFA 2- Value of HDP at patching time (internal to FORGET)
LFA 0- Header

QUIT

When an error occurs or when the CLEAR key is pressed, the routine in **(QUIT)** is executed. The default value of (QUIT) is the tag of **((QUIT))**.

((QUIT)) calls **CLEANUP**, resets the status of the compiler, prints 'ok' and then starts up the outer interpreter. CLEANUP sets STATE to executing, empties the parameter and return stacks (but it will return to its caller). If the system was in the

process of building a definition, for example when a ':' was encountered and something went wrong before the corresponding ';' was executed, or between CODE and ENDCODE, then the last incomplete definition is forgotten, including the name, code and tag (if used), as if the definition had not been typed in. The T-stack contents are not cleared until the CLEAR key is used twice. The SnapFORTH capsule itself does not use the T-stack, nor are the contents saved when the capsule is exitted with two CLEARs.

You can handle errors yourself by storing the tag of our own routine in (QUIT):

```
'X MYQUIT TO (QUIT)
```

Your routine must never return, otherwise you may return to CLEAR! Instead you must either set up your own command loop or call the SnapFORTH outer interpreter by executing ((QUIT)).

CHAPTER 5: CAPSULE GENERATION

5.1 INTRODUCTION

When you develop a SnapFORTH program, it is compiled into a dictionary file. A dictionary file does not only contain a SnapFORTH program, but also the symbol tables ("headers") for it. The exact structure of a dictionary file is discussed in Chapter 4. After you have tested your program in the dictionary file, you may wish to convert it into a stand-alone program. Two options exist to do this:

1) You can create a SnapFORTH file, which can be executed using the RUN SNAP PROGRAM option in the HHC's main menu. Just like a dictionary file, a SnapFORTH file contains a SnapFORTH program, but it contains no headers, making it more efficient with respect to space.

   There is one restriction to SnapFORTH files; unlike dictionaries and ROM capsules they are not always executed on the same location in the HHC's memory. Therefore you must not use absolute addresses in a SnapFORTH file. But, unless you use certain special constructs such as :c, absolute addresses are unlikely to occur in a SnapFORTH program.

2) You can create a ROM capsule for your program, using the HHC's EPROM burner or an Apple. We will discuss both methods in this chapter. Since ROM capsules are always located at the same address, there is no restriction in the usage of absolute addresses.

5.1.1 Run Time vs. Compile Time

There is also a restriction to which routines you can use in a stand-alone application. When you develop programs with the SnapFORTH capsule, you can use routines from:

1) The nucleus
2) Your own definitions
3) The SnapFORTH capsule

As you'll understand, the latter routines are not available when you make a stand-alone program. Therefore you must not use routines from the SnapFORTH capsule in your application program. But this should cause no problems since most of these routines are compile time routines such as ':' TABLES and COMPILE which you will probably never want to use at run time anyway.

There is one case in which you may compile these routines into your own routines; when you define a compiler extension. (But then the same arguments are applicable to that word).

So, we can distinguish two kinds of routines:

- Run time routines, which may also be executed at compile time.
- Compile time only routines.

From now on, "target" will stand for a stand-alone application, which can be either a ROM capsule or a SnapFORTH file.

5.1.2 The "700tags" Tables

When you write your own compiling routines, you usually do not want them to become part of the target, since they will not be needed at run time. SAVESNAP (the word which creates a stand-alone application) always saves the part of the dictionary from the start of your tag table or program header up to HERE. So, if you define your own compiling words before you execute TABLES, they will never be included in the target.

The addresses of those routines will then be entered into the 700tags table. Since the 700tags table is not saved in the target either, this also saves tag table space at run time!

5.1.3 %INT

The compiler control bit %INT is used to forbid compilation of "internal routines". Internal routines are routines with tag numbers between 700H and 8FFH, i.e. SnapFORTH capsule routines and user-written compiling routines.

No check is made when %INT is set. This can be done by:

```
SET %INT
```

The check is turned on again when you execute:

```
CLR %INT
```

Since %INT only disables COMPILATION of internal words, you can still use immediate words in target definitions. This would be a trivial remark for words like IF and THEN, but there are others which you would perhaps not expect to be immediate. For instance, GET-TYPE is actually a macro, which compiles (or executes) two short tags:


```
: GET-TYPE
  STATE                    \ compilation going on?
  IF COMPILE CFILE COMPILE 2+
  ELSE CFILE 2+
  THEN ; IMMEDIATE
```

5.1.4 Variables

All "TO variables" are internal. Variables which must be manipulated by '@' and the like can be used at run time too. This makes it very easy to distinguish between nucleus variables such as BASE, which can be used at any time, and compile time variables such as CURRENT and CONTEXT.

5.1.5 CREATE

CREATE is a special case. It can be used in two forms:

1) As an alternative for label. For instance:

```
CREATE TABLE 100 ALLOT
```

2) In conjunction with ;CODE or DOES> (<BUILDS is a synonym for CREATE). For example:

```
: EMITS CREATE C, DOES> C@ EMIT ;
```

In the first form, CREATE calls some run time routine in the SnapFORTH capsule. Therefore it cannot be used in this form for capsule programs. When you use the latter form, CREATE is patched by DOES> (or ;CODE) so that it will only call run time code in the nucleus.

Of course SnapFORTH does not know which form will be used when it encounters CREATE. So, you can use CREATE improperly even when %INT is set!

5.2 EXECUTION OF CAPSULES AND SnapFORTH FILES

Stand-alone programs in SnapFORTH files and ROM capsules consist of three parts:

- a header
- the tag tables
- the code

The header is used by the HHC's kernel for invoking the program. SnapFORTH files and ROM capsules each have a different header format, which will be described below. Information in the header may be: the program's name, the speed of the capsule, and some information about the tag tables used.

The latter is most important. Before a stand-alone program can be run, the locations of its tag tables must be stored at TVECTO. This is done in two phases: first, the nucleus searches the header for the address and the extension number of the so-called "primary tag table". The nucleus then sets the address and the bank number of the primary tag table in the appropriate entry of the tag table vector.

Now we are ready to execute words from the primary tag table. The first tag in the primary tag table is executed. It is called the "mother word" of the program. If your program uses more than one tag table, it must set up the other entries in the tag table vector itself.

Customarily this is done by the second tag in the primary tag table, the so-called CAPINIT word. The mother word must call the capinit word before it calls any other words.

You need not write a capinit word yourself, if you set the compiler control bit %AUTOCAP, TABLES will write the capinit word itself. In that case, your mother word must execute the word CAPINIT, which will call the routine written by TABLES.

If %AUTOCAP is set and you have more than 1 tag table, the SnapFORTH compiler will skip the second entry in the tag table. For example,if your primary tag table is the 200 tag table, the mother word will become tag number 200 and your second definition will become tag number 202. Tag number 201 will be reserved for the capinit word.

Therefore, if you want to supply your own capinit word, you must reset %AUTOCAP before defining the tag tables.

5.2.1 When Must I Write My Own Capinit Word?

The capinit word written by TABLES will only initialize the vectors to those tag tables that were defined in your application. But if your application uses tags from another capsule, it must set up the vector to that capsule itself. However, this does not force you to write your own capinit routine. You can also write a "partial capinit routine", which only sets up those vectors that were not handled by the automatic capinit routine.

Interfacing between capsules is discussed in the next chapter.

When you write your own capinit word, it can find the bank number of your application capsule in the entry of the tag table vector that has already been initialized by the nucleus.

5.2.2 The Header of a ROM Capsule

The following table shows the format of a header for a ROM capsule (all addresses are hexadecimal).

Address	Label	Contents
4000 + +	ROMADDR	A count-format string, the nucleus checks if the first letter is 'C' (for copyright).
4028-29	ROMVECT	The address of the long extrinsic tag table.
402A	ROMEXT	Tag extension for the first segment of the long extrinsic tag table (i.e. if the first tag in the table is 200H, ROMEXT is 2).
402B	CSPEED	A group of bit flags describing this ROM capsule.
	ROMID	A count format string giving the name of the application, as it is to appear in the primary menu.
40xx		The long extrinsic tag table (may be anywhere in the capsule as long as ROMVECT is properly set).
40yy		The short extrinsic tag table. TABLES allocates the short tags immediately after the longs.
40zz		Two zero bytes, indicating the end of the tag tables.

Since the HHC's software (SAVESNAP and the EPROM burner) will generate the header, you do not need to know this many details about the header if you only want to create a capsule. But if your capsule is to use routines from other capsules, its capinit word can use their headers to find the tag tables for those capsules.

5.2.3 Format of a SnapFORTH File Header

The format of a SnapFORTH file header differs from the format of a header for a ROM capsule. This is mainly because a SnapFORTH file may be executed from a RAM location different from the one it was compiled for, while a ROM capsule is always executed from the address it was compiled for.

The following table shows the structure of the header for a SnapFORTH file:

- File length (2 bytes)
- File type (1 byte). Should be 04, for "executable". (Use the symbolic constant %EXECUTE).
- A count-format string giving the file name.
- Nominal origin of tag table (2 bytes). This is the last address at which the program was executed. Used by the nucleus to test if the program must be relocated.
- Tag extension number for the primary tag table segment.
- Additional tag table segments (if any).
- Two bytes of binary zeroes, which mark the end of the tag tables (already added by TABLES).

This description is just for the sake of completeness; you need never bother about the header of a SnapFORTH file since it is automatically created by SAVESNAP.

5.2.4 SnapFORTH Files Must Be Relocatable

When the nucleus is about to execute a SnapFORTH file, it inspects the header to see whether the file is (still) loaded at its nominal origin. If not, it updates the addresses in the tag tables to reflect the change in location. This implies that colon definitions can be executed even when loaded at another location, since only their addresses in the tag tables need to be changed.

But certain other constructs cannot be used in a SnapFORTH file because they cannot be relocated. For instance, the following assembly routine uses an absolute address 'EBCDIC', and is therefore illegal in a SnapFORTH file:

```
LABEL EBCDIC ........
CODE ASCII>E    0 ,X     LDA,
                         TAY,
          EBCDIC ,Y     LDA,
          CPUT          JMP, ENDCODE
```

More generally, you are not allowed to use labels in your program.

The following routines cannot be used in a SnapFORTH file because they introduce labels:

```
:C   :P   LABEL   STRING"
```

CREATE and <BUILDS cannot be used in SnapFORTH file programs.

Notice that it is only illegal to use absolute addresses which are within the SnapFORTH file program itself. It is perfectly safe to use the vectored absolute addresses in the nucleus (as we did with "CPUT JMP,").

5.2.5 Capsules Can Be Absolute, But ...

Capsules are always executed at a fixed location. That is why you may use absolute addresses in a capsule. But again, there is a complication in the way in which you can use absolute addresses. Capsules are always located at 4000H, whereas the dictionary file is always located at another location. So, your program must run at another location than it was compiled for.

The first thing you must do to compile a program for a different location is to tell the SnapFORTH compiler the difference between both locations. The compiler variable "O" is used for this purpose. Before you define any target definitions, you must compute the difference between the location of your tag tables in the capsule and the dictionary, and assign it to "O". This computation can be made using the formula:

```
O = target - host
```

The target location of the tag tables is ROMID + (the length of the program name). Since the program name is a count format string, which uses one byte for its length, the length of the program is the number of letters plus one. For example, if your program is to be called "SAMPLE", the target location of the tag tables will be ROMID 7 +.

When you define your tables, TABLES sets the compiler variable TT.ORIGIN to the host (i.e. dictionary) location of the tag tables. So, you know the host address of the tag tables ***after*** you have executed TABLES. There is one unfortunate complication; TABLES itself wants to know whether your application is to be compiled for a different address. It uses "O" to find out ... But don't worry, TABLES is not interested in the actual value of "O", so you can assign a non-zero dummy value to "O" before you execute TABLES, and later assign the correct value to "O".

Here's an outline of what you must do to compile an application for a capsule:

```
SET %INT
```

...compiler information goes here, using 700tags...

```
1 TO 0  SET %AUTOCAP \ We are going to create
HEX 40 2 140 TABLES MYTAGS  \ a capsule
ROMID 7 + TT.ORIGIN - TO 0
                    \ Program name string will be 7 bytes
MYTAGS CLR %INT
                    \ From now on only target definitions
LONG MYMOTHER
                    \ Assign tag # 200 to mymother
```

...application program goes here...

```
: MYMOTHER CAPINIT wheretobegin ;
FORWARD VLIST FORTH
                    \ End of compilation, no forwards left?
SAVESNAP <filename>
                    \ First step to create a capsule
```

During compilation, you can execute your target colon definitions if you want to, since the tag table contains their host addresses. When the program is converted into a capsule, the tag table will be relocated.

But you must be very careful when you want to read or write absolute target addresses at compile time. When you use these symbols the compiler already uses the addresses they will have in the capsule! When you want to read or write locations which were defined by:

```
LABEL TARG .....
```

Use T>H as in:

```
: TARCOL ...  [ TARG T>H ]
    LITERAL ... ;
```

You can also convert host addresses into target addresses using "H>T". You will probably only need this in calculation where HERE is involved, since HERE leaves a host address (of course). For example:

```
HERE S" "String 1"
     S" "String 2"
     S" "String 3"
H>T CONSTANT STRINGS
```


If the named routine is a target definition, 'V gives you its target address. CFA expects a host address on the stack (of course, headers only exist in the dictionary) and leaves the target execution address for a target definition and host address for a compiling definition.

Assembler statements such as "JMP," and "JSR," expect absolute target addresses. This requirement is automatically met when the operand of these instructions is a label. More generally, you will seldom have to use "H>T" or "T>H" since the conventions used by the SnapFORTH compiler are very consistent.

5.3 SAVESNAP

SAVESNAP is used to make a SnapFORTH file from your application. All you have to do is to compile your program with the SnapFORTH capsule and execute:

```
SAVESNAP  {n:}  <filename>
```

Where <filename> is the name under which your application will appear in the RUN SNAP PROGRAM menu.

You can specify the RAM bank number n of the file in the file name. The bank number must be separated from the actual name by a colon:

`0:xyz`	Internal RAM
`1:xyz`	First RAM bank on the bus
`2:xyz`	Second RAM bank on the bus
etc.	

If <filename> does not exist, it is created. If it does exist in the specified bank, the previous contents are overwritten. If the file does exist, but its file type indicates that it is not executable, an error message is printed:

```
ERROR WRONG FILETYPE <filename>
```

If everything behaves as it should, you can then run your SnapFORTH file by entering its number in the RUN SNAP PROGRAM menu. If your program does not run, check the following points:

- Does your motherword have the first tag number in the primary long table (check with "X motherword H.)?
- Did you make a capinit word, or has your program been compiled with the %AUTOCAP control bit set?
- Did your program refer to any absolute locations within itself?
- Did your program use any words from the SnapFORTH

capsule (unless your capinit word has made an entry for the SnapFORTH capsule in the tag table vector)?

5.4 A SnapFORTH FILE EXAMPLE

Let's try to write a simple SnapFORTH program, compile it, generate an executable SnapFORTH file, and run it. First we will use the HHC's FILE SYSTEM to create a source file. Enter the FILE SYSTEM and select '1' to create a new file. Let's call it DEMO.SRC. Enter the following text:

```
SET %AUTOCAP
HEX 0 2 3 TABLES MYTAGS
MYTAGS LONG DEMO
: SHOW BEGIN CR ." KEY: "
      KEY DUP . EMIT REPEAT ;
: DEMO CAPINIT SHOW ;
```

Next enter SnapFORTH and create a new file called DEMO.SNAP. Then type:

```
LOAD DEMO.SRC
SAVESNAP DEMO
```

Next exit by pressing the CLEAR key twice and select item 4, called RUN SNAP PROGRAMS. You should see a file called DEMO appear in the menu. Select it an the display will be cleared and you will be prompted with

```
KEY: Q
```

Any key you type will be displayed first as the decimal ASCII value followed by the character itself. This is an infinite loop and can only be exited by pressing CLEAR.

Let's review what we have just done. First we created an ordinary text file with the file system that contains SnapFORTH source code. Next we LOADed that file with SnapFORTH, and saved it with the SAVESNAP command. Finally we executed the file we created with the RUN SNAP PROGRAM choice in the main menu. The executable file, DEMO, was stripped down to a bare minimum. In fact, as defined it is only 38 bytes long. SnapFORTH is indeed capable of generating extremely compact and efficient code.

5.5 CREATING A STAND-ALONE CAPSULE

You can use one of the following methods to create a stand-alone capsule:

- Save your program first with SAVESNAP and convert the resulting SnapFORTH file into a capsule with the HHC's EPROM burner. This is the easiest method, since the program header is automatically generated by the EPROM burner and the capinit word can be made by SAVESNAP if you like.
- You can create a program header yourself and send the part of the dictionary containing the tag tables and the application code to the RS232 and use the APPARAT PROM BLASTER on the Apple to make an EPROM.

5.6 USING THE HHC'S EPROM BURNER

First, you must make a SnapFORTH file from your program using SAVESNAP. Since you need not execute this file (it is only used as an intermediate file), you don't have to bother about the restrictions against using absolute addresses in your application (as long as you have assigned the proper value to "O" at compile time).

Next, you can use the HHC's EPROM burner to generate a capsule from your SnapFORTH file. Read the EPROM burner manual for further details.

This is the easiest method to create a capsule; the capinit word can be made for you by SAVESNAP, and the program header will be created by the EPROM burner. You do not have to do a thing yourself except for typing a few commands; no additional programming effort is required.

5.7 MAKING A CAPSULE YOURSELF

Capsules can also be created on the Apple, using the APPARAT PROM BLASTER. In order to do so, perform the following steps:

1) Make your own program header.
2) After the compilation, relocate the tag table.
3) Send the part of the dictionary from ROMADDR T>H to HERE to the Apple and burn the EPROM.

Steps 1) and 3) can also be done on the Apple. Relocating the tag table is very simple. Just add "O" to every entry in the table starting at TT.ORIGIN. The end of the tag table is marked by two consecutive zero bytes. (Do not add "O" to these.)

Here is a sample program header:

```
SET %INT
```

...compiler information goes here, using 700 tags...

```
HEX
 ROMADDR HERE - TO O              \ See note
 S" "Copyright Anyone Inc 1982"
              \ Start of program header (ROMADDR)
 ROMID T>H HERE - ALLOT           \ Skip to ROMID
 S" "My Program !"
      \ ROMID:appears in HHC menu
 20 2 110 TABLES MYTAB
 TT.ORIGIN H>T ROMVECT T>H !
      \ Fill in ROMVECT
 2 ROMEXT T>H C!
      \ Long tags start at 200
 PROGBIT SPEED T>H C!
 SET %FORW CLR %INT
 LONG MOTHER
```

...application program goes here...

```
: MOTHER CAPINIT wheretobegin ;
```

...now you must relocate the tag table and send your program to the Apple...

NOTE:

We have used a different calculation for "O" than in our previous example. In that example we calculated the difference between the host and target location of the TAG TABLE. But now that the program starts with the program header instead of the tag table, we must calculate the difference between the host and target location of the program header.

5.8 RELOCATING THE TAG TABLE

To relocate the tag table, simply add "O" to every entry. The tag table starts at TT.ORIGIN and the end of the table is marked by two zero bytes:

```
700TAGS
: RELOC TT.ORIGIN
  BEGIN O OVER +!          \ Start of table
        2+ DUP @ 0=        \ Relocate entry
  UNTIL ;                  \ Continue with next one
                           \ until sentinel reached
```


5.9 SENDING YOUR PROGRAM TO THE RS232C

To send bytes to an HHC peripheral you can simply ATTACH the peripheral to any of the units 2 through 8, and use the ROP operation instead of EMIT. ROP needs an ECB (Event Control Block) containing the byte to be sent. Refer to the HHC manual for more details about HHC I/O.

You do not need to allocate dictionary space for the ECB if you allocate it on the stack instead. This trick is used by WRBYTE:

```
: WRBYTE 0 SP@ 1+
      \ Allocate ECB on stack
  DUP 2 ROP DROP WAIT
      \ Write byte to peripheral
  2DROP ;                      \ Drop ECB

: SEND  [ HEX ]  46 2 ATTACH .
      \ Attach RS232C to unit
  RELOC
  ROMADDR T>H HERE BOUNDS
      \ Send prog to RS232C
  DO I C@ WRBYTE
  LOOP ;
```

Of course these words are defined with the other immediate words before you define your application.

5.9.1 Structure of a Program for an External RS232C EPROM Burner

We have now discussed all the elements of a program for an external RS232C EPROM burner. Here is a skeleton which shows how the elements are assembled to a program which you can compile and send to the EPROM burner:

```
700TAGS SET %INT
: RELOC .... ;
: WRBYTE ... ;
: SEND ..... ;
```

...other immediates here if necessary...

```
HERE == CAPSTART
```

Remember start of capsule image

...program header here...

```
SET %FORW SET %AUTOCAP CLR %INT
```

...application here, including mother word...

```
CR ." Length of capsule " HERE
CAPSTART - H.
'V FORWARD 3 + @
        \ Check if any forwards still unresolved
IFTRUE CR ." Undefined forward
     references: "
FORWARD VLIST FORTH
        \ Print them and return
IFEND
```

After this has compiled, the word SEND is available to move the image to the RS232C. This version of SEND just transmits 8-bit bytes, but it can be modified to send INTEL hex or another loader format.

The output of SEND can move the program into an Apple with an APPARAT EPROM burner card.

5.10 COMPILATION FOR RAM/ROM

Compilation into a RAM/ROM is a good way to test a program destined for capsules, without having to burn EPROMs.

The following source code contains a short routine to produce a capsule image, and move this capsule image into the RAM/ROM. After this "pseudo-burn", only the pseudo- capsule is available; the dictionary and all other files in the RAM/ROM are lost.

Therefore, the source should be kept in and LOADed from another RAM peripheral, or another system.

```
SET %INT          \ Compiler options for host
```

...first, some routines that will move the image within RAM/ROM...

```
: GO        2R> DROP
    &LBUF %LLEN CMOVE
    (CALL)  [ &LBUF , ]  ;

: BURN    TT.ORIGIN
    BEGIN 0 OVER +!
          2+ DUP @ 0=
    UNTIL DROP

    GO  [ 0 C, ]  \ Copy code below to &LBUF !

    ROMADDR T>H 800C HERE 3 PICK -
       MOVE
    CR ." "Flip switch" KEY DROP ;

CLR %INT SET %FORW        \ Options for target
```


...capsule header...

```
HERE == CAPSTART
        \ Remember start of capsule image
ROMADDR HERE - TO 0
        \ Offset for normal capsule
S" Copyright yourcompany 1982"
ROMID HERE H>T - ALLOT  \ Skip to ROMID
S" <capsulename> "

40 2 100 TABLES MYTABLES
        \ At least one long for entry!
TT.ORIGIN H>T ROMVECT T>H !
        \ Store target address of tag table in ROMVECT
2 ROMEXT T>H C!
        \ Extension of first tag table
PROGBIT CSPEED T>H C!
        \ Fast program capsule
LONG ENTRYPOINT
```

...program contents...

```
: ENTRYPOINT wheretobegin ;

CR ." Length of capsule: " HERE
    CAPSTART - H.
'V FORWARD 3 + @
        \ Unsolved forward references?
IFTRUE CR ." Undefined forward
   references: "
          FORWARD VLIST FORTH
OTHERWISE
          BURN   \ Move to start of RAM/ROM
IFEND
```

This version produces a complete capsule image with a file. When the BURN command is given, the image is moved down into actual position, destroying the file. You must immediately switch the RAM/ROM to ROM position. For this reason, you should keep the source somewhere other than in the RAM/ROM, since it will be inaccessible once the switch is flipped.

CHAPTER 6: USING EXTERNAL ROUTINES

This chapter explains how you can use routines from a ROM capsule, for example from the Scientific Calculator or from the SnapFORTH capsule itself.

To use routines from another capsule, you must do two things:

- You must tell the compiler the tag numbers of the routines you want to use.
- When your capsule is executed, its capinit word must search the header of the library capsule for the address of its tag table, and store it in the tag table vector.

6.1 TAG

TAG lets you create a header and a tag number for an "external" routine. For instance, suppose you want to use a routine named XXX from the Scientific Calculator. From the Scientific Calculator manual you know that XXX has tag number 507H. Using the same format as 'X does (i.e. reversed byte order), we can "import" XXX into our program:

```
705 TAG XXX
```

Now every time the compiler encounters the name XXX, it compiles tag number 507.

6.2 FINDING ANOTHER CAPSULE'S TAG TABLES

When your capsule (or Snap file) is about to run, it must set up the tag table vector, not only for its own tag tables but also for the library capsules used by it. Since you do not know in advance in which ROM bank the library capsule has been plugged, you must search all ROM banks for it. All ROM banks share the same address space, therefore you can not access other capsules directly. But you can use **BIGMOVE** to copy the headers of other capsules into RAM. Once the header has been copied into RAM, you can compare its name with the one you are searching for. Here's a routine that searches all ROM banks for the Scientific Calculator. It does so by copying the capsule headers into a scratch area, and comparing their ROMIDs with that of the Scientific Calculator. It leaves a flag and (when the flag is true) a bank number:

```
STRING" "SCIENTIFIC CALCULATOR" HISNAME
HISNAME T>H C@ 1+ == L#NAME

AREA HEADER
     L#NAME STRING CAPNAME \Notice that fields in
     CVAR CAPSPEED         \area are allocated
     CVAR CAPEXT           \in reverse order.
     VAR CAPVECT
ENDAREA
L#NAME 4 + == L#HDR        \Length of header

: SEARCH ( ---  [bank#]  bool )
HEADER ?ENOUGH-ROOM FALSE
80H 0
DO I FLIP ROMVECT CAPVECT L#HDR BIGMOVE
       \Get header
    CAPNAME COUNT HISNAME COUNT S=
       \Compare name
    IF DROP LEAVE I TRUE
       \Found it!
    THEN
LOOP ;
```

BIGMOVE moves a range of memory between two bank switched areas of memory, one of which is a ROM bank and the other is a RAM bank. Its linkage is:

```
BIGMOVE ( Bank Fromadr Toadr Len -- )
```

Where

Bank	High byte is ID of ROM bank, low byte is ID of RAM bank.
Fromadr	is the address of the source
Toadr	is the address of the destination
Len	is the length to move

Once we have found the bank number of the library capsule, we must initialize its tag table vector entries. This can be done by initializing its primary tag table entry ourselves, and then calling the capinit word of the libary capsule:

```
: LINKLIB  [ HEX ]
    SEARCH
    NOTIF ." Libary capsule not found "
        CLEAR
    THEN
    80 + CAPEXT C@ 3 * TVECT0 +
        \Init address
     CAPVECT @ OVER ! 2 + C!
        \Init bank #
    CAPEXT C@ 100 OR EXECUTE ;
        \Execute capinit
```

Before your program calls any library routines, it must call LINKLIB.

CHAPTER 7: MANAGING DATA IN THE SnapFORTH LANGUAGE

7.1 CONSTANTS

7.1.1 Symbolic Constants

Instead of FORTH's conventional defining word CONSTANT, SnapFORTH has a word called '= =', which defines a **symbolic constant**.

A symbolic constant does not compile a word that appears in the object code; rather, it causes the compiler to compile any reference to the defined symbol as the symbol's value.

Here is an example of '= =' in use:

```
64 == BFR.LEN
```

This defines a symbolic constant named BFR.LEN whose value is 64. After the compiler has processed this definition, the following two code fragments will compile to the same sequence of tags:

```
BFR 64 ERASE
BFR BFR.LEN ERASE
```

7.1.2 Colon Definitions As Constants

You can get effect more similar to that of a conventional FORTH constant by creating a colon definition that stacks a constant value, like this:

```
: BFR.LEN 64 ;
```

or even like this:

```
: 64 64 ;
```

You can also use the traditional defining word CONSTANT:

```
64 CONSTANT BFR.LEN
```

but this takes more space than a colon definition.

If you give such a colon definition a short tag it may reduce your program's size, because while a ***definition*** of a symbolic constant consumes no memory at run time, a ***reference*** to a symbolic constant consumes slightly more memory than a reference to a colon definition.

	definition consumes	reference consumes
==	0 bytes	2 bytes (0<= k <= 1023)
CONSTANT (short tag)	5 bytes	1 byte
':'	4 (0<= k <=1023) 5 (other values)	1 byte

When the definition of a CONSTANT or a colon definition is followed by a colon definition, the compiler can perform some optimization. In that case the table changes to:

CONSTANT (short tag)	5 bytes	1 byte
':'	3 (0<= k <=1023)	1 byte
	4 (other values)	1 byte

If you refer to a value several times, using a short- tag colon definition will make your program slightly smaller (but slightly slower). If you refer to the value only once or twice, a symbolic constant will be both more compact and faster.

7.1.3 Set.Constant

SET.CONSTANT is a "defining word" that can be used to notify the SnapFORTH compiler that a certain word will stack a certain constant value at run time, and should be compiled from references to that constant.

SET.CONSTANT lets you make your code more compact by taking advantage of tags which stack constant values instead of compiling more space-consuming literal values.

For example, suppose you defined the following code:

```
1024 CONSTANT 2^10
1024 SET.CONSTANT 2^10
```

The first line is an ordinary definition of a constant whose name is 2^10, and whose value is 1024.

The second line informs the SnapFORTH compiler that the word 2^10 will stack the value 1024 at run time. After executing this line, the compiler will compile any reference to the literal value 1024 into the tag for the word 2^10.

7.2 ALLOCATING RAM, PART I: THE TEMPORARY STORAGE AREA (TSA)

You allocate variables for an application program by creating a **temporary storage area** (TSA).

A TSA is an area in RAM that is allocated at run time from the **temporary stack**. A TSA is allocated dynamically at run time. This avoids the problem that would occur if SnapFORTH attempted to allocate working storage in the conventional FORTH fashion: that every application would have its own exclusive area in RAM, and soon all the RAM would be used up.

7.2.1 How To Use a TSA

To use a TSA, you must (1) define it at compile time, and (2) allocate it at run time.

You define a TSA at compile time with the AREA word. AREA is followed by another word which becomes the name of a word which AREA defines; you may execute this word at run time to allocate the TSA. AREA is followed by variable definitions, and then ENDAREA. (See the example below.) You must compile this sequence before any code which allocates the TSA or refers to its variables.

You allocate a TSA at run time by using the TSA's allocation word (the name that followed the AREA word) in your program. The allocation word returns a boolean: TRUE if it succeeded in allocating the TSA, FALSE if there was not enough free RAM for it to do so.

7.2.2 An Example

This example shows how you could define and allocate a TSA.

```
AREA TSAWURD              \ Declare TSA for WURD.
  VAR WURD.TEMPC          \ Ctrl word holding area
  VAR WURD.PTR            \ Char PTR
ENDAREA
  :WURD    ( ... )
  TSAWURD NOTIF           \ Alloc TSA & test success.
  ." NO SPACE!" ENDIF
  ...
```

7.2.3 Types Of TSA Variables

Here are the words that may be used to allocate RAM in a TSA.

CVAR ccc

Defines a one-byte variable named "ccc".

n CVECTOR ccc

Defines an array named "ccc" containing "n" byte elements.

DVAR ccc

Defines a two-word variable named "ccc".

FVAR ccc

Defines a floating point (8 byte) variable named "ccc".

n STRING ccc

Defines a character string variable with room for "n" characters, named "ccc".

VAR ccc

Defines a one-word variable named "ccc".

n VECTOR ccc

Defines an array named "ccc" containing "n" word elements.

7.2.4 Notes On Using the TSA

You cannot define a TSA more than 250 bytes long. If you try, the SnapFORTH compiler will issue an error message.

If you need more than 250 bytes of space, use GRAB (described below) to get it before you allocate your TSA, or use the <BUILDS DOES> to redefine AREA, ENDAREA, etc.

If the allocation word returns TRUE, it has allocated the TSA from the temporary stack. The TSA will remain allocated until the user presses the CLEAR key, or until the program executes the LETGO word (see glossary).

When the TSA is allocated, its contents are NOT initialized. This means that if a particular variable should have an initial value (e.g., zero), you should initialize it after allocating the TSA.

The variables in a TSA are allocated in reverse order; i.e., the first-defined variable has the highest address, and the last-defined variable has the lowest address. (This makes sense when you recall that the temporary stack grows down from the top of the temporary stack space.)

It is good practice to initialize the TSA one variable (or one vector) at a time. Do not calculate the total length of the TSA and do a one-shot initialization with a word like FILL. On the simulator, one-shot initialization is likely to be disastrous, because it will wipe out the FORTH headers that are created by each defining word!

At compile time, the storage words defined in the TSA are global. It is your responsibility to ensure that the TSA is allocated before any of the variables in it are referenced.

7.2.5 Releasing TSAs

When a TSA is allocated, it "hides" all previously allocated TSAs. You can think of TSAs as entries that are pushed onto the temporary stack; only the top entry is available at any time.

Because of this, your program must release TSAs if it needs more than one TSA.

You release a TSA with the LETGO word. LETGO takes no parameters from the stack, and leaves none:

```
LETGO
```

LETGO releases the TSA most recently allocated and not yet released. In other words, it acts like a DROP that applies to the temporary stack, dropping one whole TSA at a time.

7.2.6 About the Size of the Temporary Stack

The temporary stack is allocated down from the top of intrinsic RAM. Virtual file space is allocated up from (approximately) the middle of page 3. Each of these entities may grow until the two meet. Thus, the amount of space available for the temporary stack is a function of the amount of intrinsic RAM that is provided, and of the amount of intrinsic RAM being used for virtual files.

The minimum configuration of the HHC has 2K of intrinsic RAM, so at least 1.1K of RAM will be available for the temporary stack and virtual files. The file system has an overhead of only about 10 bytes in intrinsic RAM, so the maximum amount of temporary stack space an application can have on a minimum configuration HHC is also about 1.1K. If an application is to be runnable on a minimum configuration HHC (highly desirable in most cases), it should use no more intrinsic RAM space than this.

Note that most peripheral devices allocate some work space from the temporary stack; if an application is to be used with peripherals, it should make allowances for that.

Caution: always check the boolean value returned by the TSA word to be sure the TSA was successfully allocated. It it was not, give the user a meaningful error message and stop. Do not let the program try to run; it will not work.

Do not confuse the temporary stack with the regular SnapFORTH stack that most SnapFORTH words operate on. The latter stack is simply called "the stack," or when a more explicit term is needed, the "PARAMETER STACK." It is allocated in the 6502's page 0, not page 3.

7.3 ALLOCATING RAM, PART II: DYNAMIC ALLOCATION

A TSA cannot meet all of a program's possible needs for RAM. Sometimes you cannot say how much RAM a program will need at the time you compile it. This might be because the amount of space the program will need is highly dependent on the data it is given, or because the program can use as much RAM as it can get. (A sort program is an example of the latter case.)

When you cannot predict your program's space requirements at compile time, you must allocate RAM dynamically at run time.

You can allocate RAM dynamically from the temporary stack with the word GRAB. The linkage to GRAB is:

```
n GRAB  [adr]  b
```

"n" is the amount of RAM you want to allocate. It too is limited to 255 bytes. If you need more space, you will need to execute multiple GRABs. If GRAB allocates this amount of RAM successfully, it returns "adr", the address of the RAM, and TRUE. If GRAB does not allocate the RAM successfully, it returns FALSE.

7.3.1 Is There Enough Room?

You can tell whether a certain amount of RAM is available without trying allocate it by using the word ?ROOM. The linkage to ?ROOM is:

```
n ?ROOM b
```

"n" is a number of bytes of RAM. "b" is TRUE if this amount of RAM is available, and FALSE if it is not.

7.3.2 How Much Room Is There, Anyway?

Suppose you need to know exactly how much RAM is available? You can find out with the AVAIL word. The linkage to AVAIL is:

```
AVAIL n
```

"n" is the amount of RAM that is available, in bytes. Note: to guarantee that AVAIL returns the number of bytes in intrinsic RAM, as opposed to extrinsic RAM, execute;

```
0 &EXTRINSIC C!
```

prior to executing AVAIL or ?ROOM.

Suppose we want to GRAB large amounts of memory. We could define the following:

```
:BIGGRAB  (n ---  [A]  b )
  DUP ?ROOM IF
     BEGIN DUP 255 > WHILE
        255 GRAB 2DROP 255 - REPEAT GRAB
  ELSE FALSIFY THEN ;
```

For example, suppose you wanted to allocate as much intrinsic RAM as you could, leaving 500 bytes free for virtual files. You could do so like this:

```
501 ?ROOM NOTIF ." NO ROOM" KEY
   CLEAR ENDIF
AVAIL 500 - BIGGRAB DROP
```

Recall that intrinsic RAM is shared by the temporary stack and the virtual file system, so adding data to a virtual file that is in the intrinsic file space will decrease the amount of space available for your TSA.

7.4 ALLOCATING SPACE IN ROM

The most common and simplest way to allocate space in ROM is to do so implicitly with colon definitions and code definitions.

Sometimes it is necessary to allocate ROM space explicitly. For example, you may have to build a table or some other data structure that your program will reference. You can do this with standard FORTH dictionary management words, such as ',', 'ALLOT', and 'HERE'.

The word LABEL creates a symbol whose value is the current value of the "target dictionary pointer". It is useful for giving a name to an area in ROM:

```
LABEL EXTAB
HEX 00 C, 4A C, 25 C, ...
```

A reference to EXTAB in a colon definition will stack the address of the first byte in the table at run time, if you want to use this address at compile time, it must be converted by T>H.

An alternate way to label a table is to stack HERE before compiling the table, and assign it to a constant when done:

```
HERE
00 C, 4A C, 25 C, ...
33 C, 21 C,
H>T == EXTAB
```

Note that you ***cannot*** use dictionary management words like HERE in colon definitions. Since your compiled code will be in ROM at run time, the concept of a run-time dictionary pointer is meaningless!

7.5 RESTARTING A PROGRAM

7.5.1 The CLEAR Key

The CLEAR key is the ultimate corrector of mistakes on the HHC. It can reset the system to its **ground state**, the primal state from which all other things come. There is little you (or a user) can do to the HHC's software that cannot be fixed by pressing CLEAR.

Pressing the CLEAR key once makes the system perform a SOFT CLEAR. A soft clear consists of the following functions:

1. Empties the parameter stack, loop stack, and return stack.
2. Resets most aspects of the HHC's status. Cancels all timers except the clock controller; clears LCD and most blips; re-initializes certain RAM locations used by nucleus.
3. The application program zero page space (&APS) is cleared.
4. Restarts the current application by executing the tag in SOFTAG. If SOFTAG is empty, the HHC does a hard clear.
5. CFILE is cleared; that is, no file remains opened.

Pressing the CLEAR key a second consecutive time makes the system perform a HARD CLEAR. In a hard clear, the system performs the following steps:

1. Empties the parameter stack, loop stack, and return stack (same as for soft clear).

2. Resets most aspects of the HHC's status. Cancels all timers except the clock controller's; clears LCD and most blips; re-initializes certain RAM locations used by nucleus (same as for soft clear).
3. Empties the temporary stack.
4. Resets, unattaches and unbinds all peripherals; re-initializes the EMIT vector and KEY vector; releases all RAM reserved as workspace for the peripheral I/O drivers.
5. Zeroes SOFTAG and places the machine in the ground state.

7.5.2 Changing the Restart Word

The HHC nucleus contains several variables that determine how the HHC will respond to a soft clear. These variables are:

SOFTROM: the bank ID of the ROM bank to be switched into the capsule ROM address space when a soft clear occurs (one byte).

SOFTCTL: the bank ID of the ROM bank to be switched into the control ROM address space when a soft clear occurs (one byte).

SOFTVECT: the address of the primary tag table to be placed in the tag table vector when a soft clear occurs (2 bytes).

SOFTAG: The value of the tag to be executed when a soft clear occurs (2 bytes). The tag table vector cor- responding to the extent byte of SOFTAG is plugged by SOFTVECT. NOTE!! This means SOFTAG may ***never*** be a short tag.

When the HHC enters an application, it sets these words to restart the application from the beginning. You can change these words to take some other action for a soft clear if you wish to do so.

There are two basic ways to change the restart words. You can store into them directly, or you can do a FLEE. When you do a FLEE, the nucleus sets SOFTROM, SOFTCTL, and SOFTVECT to the current capsule ROM, the current control ROM, and the current primary tag table. It sets SOFTAG to the tag you are FLEEing to.

Most often you will need to reset only SOFTAG. Here are examples of how to reset SOFTAG with and without FLEEing to the new restart word:

```
'X restart-word FLEE          \ FLEEs to it

'X restart-word SOFTAG !      \ just sets it
```

If an application presents the user with various modes of operation, you may want CLEAR to restart the current mode, rather than restart the whole application. You can accomplish this conveniently by entering the submode by FLEEing to an appropriate word, and thus changing the value of SOFTAG.

7.5.3 Software Simulation of the CLEAR Key

You can get the effect of a soft clear in your program by executing the word SOFT.CLR; or of a hard clear by executing HARD.CLR. You can get the exact effect of pressing the CLEAR key (i.e. a soft or a hard clear) by executing CLEAR.

7.5.4 Inhibiting Hard Clears

In some applications it is necessary to inhibit the HHC from doing a hard clear when the CLEAR key is pressed twice in a row.

The HHC distinguishes a hard from a soft clear by the value of the one-byte variable SOFTFLAG. The CLEAR key sets SOFTFLAG to 0; any other keystroke sets it to 5AH. If the value of SOFTFLAG is anything other than 5AH when a clear occurs, it is presumably the second clear in a row, with no other keystrokes in between; the HHC thus does a hard clear. If the value of SOFTFLAG is 5AH when a clear occurs, the HHC does a soft clear.

Thus, you can inhibit hard clears simply by setting SOFTFLAG to 5AH at the very beginning of your restart word.

Conversely, you can force the next clear to be a hard clear by setting SOFTFLAG to 0. (But remember that the next keystroke will reset it to 5AH.)

Reset SOFTFLAG with caution; if there is a bug in your program, you or a user will be able to break out of it only by taking some extreme measure such as removing the ROM capsule containing the program.

7.5.5 Ending a Program

There are three ways you can end an application program.

The first way is place the main processing part of the program in an endless loop, that is, not to end the program at all. This is a logical choice for an endless-task program such as a calculator.

The second way is to let the mother-tag word finish executing and return control to the system. This returns the HHC to the state it was in before the mother-tag word was executed. For example, if the mother-tag word was executed from the primary menu, returning control to the system returns the system to the mother-tag menu. In this case the system performs a hard clear.

The third way is to execute the word CLEAR, SOFT.CLR, or HARD.CLR.

CHAPTER 8: CONTROL FLOW IN THE SnapFORTH LANGUAGE

8.1 TRANSFER OF CONTROL IN SnapFORTH

SnapFORTH's control structures (IF/ELSE/ENDIF, BEGIN/WHILE, etc.) have some unusual characteristics because of SnapFORTH's use of tags. In addition, SnapFORTH supports several useful control structures, such as CASE, that do not appear in most conventional implementations of FORTH.

8.2 TAG STRUCTURE AND TRANSFER OF CONTROL

Since SnapFORTH code is very compact, and the scope of a control structure (***i.e.*** the distance from a transfer-of-control instruction to its target) is usually less than 256 bytes, SnapFORTH saves space by using one- byte unsigned displacements, rather than 2-byte signed displacements, in most of its control structures.

For example, consider this simple IF/THEN/ELSE structure:

```
IF DUP ELSE DROP ENDIF   COUNT
```

In most implementations of FORTH, this structure would compile to code something like the following:

address	**contents**	**comment**
1000H	0BRANCH	Branch if top of stack is 0.
1002H	0008H	Displacement of branch target.
1004H	DUP	
1006H	BRANCH	Unconditional branch.
1008H	0004H	Displacement of branch target.
100AH	DROP	Target of 0BRANCH.
100CH	COUNT	Target of BRANCH.

In SnapFORTH, the same code compiles to the following:

address	**contents**	**comment**
1000H	(IF)	Branch if top of stack is 0.
1001H	04H	Displacement of branch target.
1002H	DUP	
1003H	(ELSE)	Unconditional branch.
1004H	02H	Displacement of branch target.
1005H	DROP	Target of 0BRANCH.
1006H	COUNT	Target of BRANCH.

Since the object-time word (IF) takes a one-byte ***unsigned*** displacement, it can only jump forward. The word (UNTIL) also takes a one-byte unsigned displacement, but jumps backward; it is used to implement the BEGIN/UNTIL structure. For example, the following source code:

```
BEGIN 2+ 2DUP @ = UNTIL   COUNT
```

would compile to code something like the following in most implementations of FORTH:

address	contents	comment
1000H	2+	Target of 0BRANCH.
1002H	2DUP	
1004H	@	
1006H	=	
1008H	0BRANCH	Compiled from UNTIL.
100AH	FFF6H	FFF6H = = -000AH.
100CH	COUNT	

In SnapFORTH, the same code compiles to the following:

address	contents	comment
1000H	2+	Target of (UNTIL)
1001H	2DUP	
1002H	@	
1003H	=	
1004H	(UNTIL)	Compiled from UNTIL.
1005H	06H	Displacement back to 2+.
1006H	COUNT	

The following table shows the object-time words and displacements compiled from the common control structures in SnapFORTH:

source word	object word	followed by
IF	(IF)	one-byte unsigned forward displacement
ELSE	(ELSE)	one-byte unsigned forward displacement
ENDIF		(does not produce object code)
BEGIN		(does not produce object code)
UNTIL	(UNTIL)	one-byte unsigned backward displacement
BEGIN		(does not produce object code)
WHILE	(IF)	one-byte unsigned forward displacement
REPEAT	(AGAIN)	one-byte unsigned backward displacement
BEGIN		(does not produce object code)
AGAIN	(AGAIN)	one-byte unsigned backward displacement
DO	(DO)	one-byte tag
LOOP	(LOOP)	one-byte unsigned backward displacement
DO	(DO)	one-byte tag
+LOOP	(+LOOP)	one-byte unsigned backward displacement

8.3 SOME ADDITIONAL CONTROL STRUCTURES

SnapFORTH has several additional control structures which are useful for writing compact code. These structures are listed and described below. The individual words are described in detail in the glossary.

<IF . . . ELSE . . . ENDIF

<IF compares the top two items on the stack, and executes the following code if the second one is less than the top. <IF is equivalent to '< IF'. It compiles to (<IF), followed by a one-byte unsigned forward displacement.

=IF . . . ELSE . . . ENDIF

<IF compares the top two items on the stack, and executes the following code if they are equal. =IF is equivalent to '= IF'. It compiles to (=IF), followed by a one-byte unsigned forward displacement.

?DO . . . LOOP or ?DO . . . +LOOP

?DO is equivalent to DO, except that the code within the loop is not executed if the ending condition is satisfied before the first loop. (DO always executes the code within the loop at least once.) It compiles to (?DO), followed by a one-byte unsigned forward displacement.

CASE. . .ELSE. . .ENDIF

Defines a CASE clause, which executes one of several pieces of code depending on the values of the top two words on the stack.

Here is an example of CASE in use:

```
:.DIGIT   ( N --- )
      0 CASE   FOR-ZEROS  ELSE
      1 CASE   FOR-ONES   ELSE
      2 CASE   FOR-TWOS   ELSE
      DROP FOR-OTHERS
      ENDIF ENDIF ENDIF   ;
```

CASE compares the two top words on the stack (N and 0, for the first CASE above). If they are equal, both are dropped, and control passes to the next word after CASE. Then control passes from the following ELSE to the code after the last ENDIF in the construction.

If the two top words on the stack are not equal, the top word (0, in our example) is dropped and execution skips past the matching ELSE or ENDIF (to "1 CASE. . ." in our example).

In this example, FOR-ZEROS is executed if N=0; FOR-ONES is executed if N=1; FOR-TWOS is executed if N=2; and FOR-OTHERS is executed if N is anything else.

CASE compiles to (CASE) followed by a one-byte unsigned forward displacement to the code following the next ELSE or the ENDIFs.

For a more convenient but less compact and efficient CASE-like structure, use IF.ITS.

DOCASE ... ENDCASE

Note that the case construction above required one ENDIF per CASE. This is rather inconvenient when many cases are involved. When you enclose a number of nested IF or CASE statements with DOCASE and ENDCASE, all necessay ENDIFs will be generated by ENDCASE, thus making your program more readable:

```
:.DIGIT  ( N --- )
    DOCASE
   0 CASE   FOR-ZEROS  ELSE
   1 CASE   FOR-ONES   ELSE
   2 CASE   FOR-TWOS   ELSE
   DROP FOR-OTHERS
    ENDCASE ;
```

Notice that DOCASE ... ENDCASE is not restricted to CASE or IF; you may freely intermingle all control-structures requiring an ENDIF. Example:


```
: XYZ ( N -- )
  DOCASE
    DUP 0 <IF DROP code ( <0)    ELSE
    DUP 4 <IF DROP code (0-3)    ELSE
    5 CASE  code for 5          ELSE
    7 =IF   code for 7          ELSE
      code for 6 and N>7
  ENDCASE;
```

Also, DOCASE and ENDCASE can be used to balance a number of IF statements without ELSEs (but then your program will have a different meaning of course).

IF.ITS. . .ENDIF

IF.ITS is a CASE-like word used to make a choice based on a numeric value. It is useful where clarity or compactness is more important than efficiency, or the possible values are scattered. It is most often used to execute code based on a menu selection.

IF.ITS expects one numeric item on the stack; it does ***not*** pop the item, but leaves it unchanged. IF.ITS is used like this:

```
X C@        Value to choose with

IF.ITS 1 code-for-choice-1 ENDIF
IF.ITS 2 code-for-choice-2 ENDIF
IF.ITS 3 code-for-choice-3 ENDIF
   :
   :
IF.ITS n choice-n ENDIF
DROP
```

The word IF.ITS must be followed by a literal or a symbolic literal in the range 0-255. If N is equal to the constant, IF.ITS executes the code between the constant and the ENDIF; if not, IF.ITS jumps to the code following the next ELSE or ENDIF.

IF.ITS compiles to (IF.ITS), followed by a one-byte value for the number that came after IF.ITS in the source code, followed by a one-byte unsigned forward displacement.

JUMP-TAB

Defines a "jump table" that can be used to execute one of several words depending on the value of a word on the stack.

JUMP-TAB is used like this:

```
4 JUMP-TAB FOUR.WORDS   WORD1
      WORD2 WORD3 WORD4
```

This example defines a jump table with 4 entries, named FOUR.WORDS. The entries in the jump table are four words (defined elsewhere, with ***tags***) named WORD1, WORD2, WORD3, and WORD4.

A jump table defined by JUMP-TAB is used like this:

```
: JUMPING.LIZARDS ( ---- )
      .
      .
  JUMP.TO.ME @ FOUR.WORDS
      .
      .
  ;
```

In this example, JUMP-TO-ME is presumably a variable containing a jump table index between 0 and 3. If the index is 0, the illustrated line of code executes WORD1; if the index is 1, the code executes WORD2; and so forth.

Caution! A jump table word defined by JUMP-TAB does not do its own range checking. Executing a jump table word with an invalid index produces unpredictable results. You can conveniently check the range before executing a jump table word by using ?RANGE.

NOTIF . . . ENDIF

NOTIF is equivalent to IF except that it executes the following code if the top item on the stack is false, rather than if it is true. NOTIF saves you a byte of code in the following common situation:

```
0= IF . . . ENDIF
```

which can be recoded:

```
NOTIF . . . ENDIF
```

NOTIF may be used with ELSE, although the resulting code is no more compact than code that uses IF, and is usually more obscure. Of course NOTIF can also be part of a DOCASE .. ENDCASE clause, in which case the corresponding ENDIF (or THEN) will be generated by ENDCASE.

8.4 LONG CONTROL STRUCTURES

The SnapFORTH compiler issues an error message if you try to use a "normal" SnapFORTH control structure with a scope greater than 255 bytes. For those rare situations where a larger scope is necessary, SnapFORTH provides several control structures in a "long" form that uses a two-byte displacement.

To use the long form of a control structure, just substitute the appropriate set of words below for the corresponding words above. Be sure to change all the words in a given structure to the long form!

source word	object word	followed by
IF.L	?JUMP	two-byte signed displacement
ELSE.L	JUMP	two-byte signed displacement
THEN		(does not produce object code)
BEGIN		(does not produce object code)
WHILE.L	?JUMP	two-byte signed displacement
REPEAT	JUMP	two-byte signed displacement

8.4.1 The WHILE Statement

SnapFORTH knows several powerful variants of the WHILE clause. The classical FORTH WHILE and UNTIL statements are:

BEGIN condition WHILE statements REPEAT

or

BEGIN statements REPEAT

or

BEGIN statements condition UNTIL

SnapFORTH allows you to use even more powerful control-structures like:

BEGIN condition WHILE condition UNTIL

or

BEGIN condition WHILE condition WHILE statements REPEAT

or

BEGIN condition WHILE condition WHILE condition UNTIL etc.

Also, several variants of WHILE exist with different stack behavior, such as:

n1 n2 CASEWHILE	Like CASE; when n1 equals n2, both are dropped and execution continues after CASEWHILE. Otherwise n2 is dropped and execution transfers to the corresponding REPEAT or UNTIL.

n1 n2 =WHILE	Like =IF, drops n1 and n2 and continues if both are equal, otherwise execution goes to the corresponding REPEAT or UNTIL.
n1 n2 <WHILE	Like <IF
b NOTWHILE	Like NOTIF, drop b and continues after NOTWHILE when b is false, otherwise skips to the corresponding REPEAT or UNTIL.
n1 WHILE.ITS	Like IF.ITS, example:

```
BEGIN dosomething
      KEY L>U
WHILE.ITS &Y
      something
REPEAT
```

THEN and AGAIN

THEN and AGAIN are synonyms for ENDIF and REPEAT. Some FORTH dialects require that a BEGIN clause without a WHILE in it should be ended with AGAIN instead of REPEAT. However, in SnapFORTH AGAIN and REPEAT are equivalent.

CHAPTER 9: <BUILDS DOES>

9.1 INTRODUCTION TO <BUILDS DOES>

These are probably the most powerful words in FORTH. <BUILDS and DOES> are compiler extending routines, which can be used to define new data structures, new routines, etc.

Generally, a FORTH object consists of two parts:

- Some data
- A routine to manipulate the data when the object is executed

For example, the two parts of a colon definition are:

- A routine body (the data)
- A call to the SnapFORTH interpreter which will interpret the body when the colon definition is executed

The two parts of a CONSTANT are:

- The value of a constant
- A call to the code that will push the value on the stack when the CONSTANT is executed

You can define define CONSTANT as:

```
: CONSTANT
    <BUILDS ,
      DOES> @ ;
```

It works this way; when you execute CONSTANT to define a new object:

```
5 CONSTANT FIVE
```

<BUILDS will create the header 'FIVE', the code between <BUILDS and DOES> is executed to compile the data of the new routine. In our example, ',' is executed and compiles the number 5 into the dictionary.

The code after DOES> ('@') is not yet executed when we define FIVE. Instead, its execution is "delayed" until FIVE itself is executed. And now for the trick: When FIVE is eventually executed, it will put the address of whatever has been compiled by the <BUILDS part of CONSTANT on the stack, and execute the part of CONSTANT after DOES>. Since the address of the 5 which has been compiled by the <BUILDS part is now on the stack, all that the DOES> part has to do is execute '@' and the value 5 itself will be on the stack!

More generally, when you define a new object with <BUILDS DOES>, the DOES> part will eventually be executed with the

value of the dictionary pointer at the time <BUILDS was executed on the stack. The DOES> part uses this address to retrieve its parameters. When you define several objects of the same class, for instance

```
3 CONSTANT THREE
4 CONSTANT FOUR
```

each of these objects will have a different "parameter field", but everyone of them will execute the same code (the DOES> part of CONSTANT) with the address of its particular parameter field on the stack.

This is very useful because we are now able to define classes of objects which all behave about the same way, but each one will have its own parameters. Here is another example; we will define a class of objects, each of which will print a particular character:

```
: EMITS
  <BUILDS C,     \Compile character in param field
  DOES> C@       \Fetch character from param field
       EMIT ;    \and print it

HEX 0D EMITS CR       \CR will print 'carriage return'
  0A EMITS LF         \LF will print 'line feed'
  BL EMITS SPACE      \SPACE will print ' '
  &X EMITS "X         \"X will print 'X'
```

When you executed:

```
0D EMITS CR
```

<BUILDS created the header 'CR', then the 0D was compiled into the dictionary by 'C,'. When you execute 'CR', it calls the DOES> part of EMITS, with the address of its parameter field on the stack. The DOES> part will retrieve the 0D stored in the parameter field, and print it. When you execute "X the same thing will happen, but since the parameter field of "X contains &X, the DOES> part of EMITS will now print 'X' instead of 0D.

Our final example is similar, but instead of storing a single character at <BUILDS time and printing it at DOES> time, it stores and prints an entire string:

```
: CONVERSATION
    <BUILDS S"
      DOES> COUNT TYPE ;

CONVERSATION HELLO "Hi there"
CONVERSATION HOWAREYOU? "I'm fine"
```


9.2 IMPLEMENTATION OF <BUILDS DOES>

As we've discussed, an object created by <BUILDS will call the corresponding DOES> part with the address of its parameter field on the stack. The implementation is as follows:

DOES> is an immediate word, when CONSTANT was defined, DOES> compiled two things into the body of CONSTANT:

- The tag of a routine named (DOES>)
- A JSR to a nucleus label: DODOES

When CONSTANT is executed at a later stage, and execution reaches <BUILDS, <BUILDS not only creates a header for the new object, but it also compiles a JSR at its CFA to an address which will later be specified by (DOES>). When execution of the <BUILDS part reaches (DOES>), (DOES>) fills in the address of that JSR and terminates the execution of the <BUILDS part.

So, when the code of CONSTANT looks like:

Address	Code	
1000H	BRK	Invoke SnapFORTH interpreter
1001H	<BUILDS	
1003H	,	
1005H	(DOES>)	
1007H	DODOES JSR,	
100AH	@	
100BH	EXIT	

FIVE will be compiled into:

1100H	1007H JSR,	Call DOES> part of CONSTANT
1103H	5	Parameter field

and Three will be compiled into:

1200H	1007H JSR,
1203H	3

(All addresses are arbitrary).

When FIVE is executed, the JSR to 1005H will leave a return address on the machine stack. This return address points to the parameter field of FIVE (minus one, due to the ideosyncracies of the 6502). Next, a JSR to DODOES is made.

Now DODOES has two addresses on the machine stack; the address of the parameter field and the address of the DOES> part (also minus one). DODOES pushes the address of the parameter field on the parameter stack (after incrementing it

by one) and uses the second address to invoke the SnapFORTH interpreter, which will then execute the DOES> part (starting at 100AH).

Note that though the implementation of colon definitions may be different (they start with a BRK instead of a JSR), they are very much like <BUILDS DOES> words at the conceptual level; the same routine (the SnapFORTH inner interpreter) is executed for every colon definition, but each time with different parameters (i.e. the body of the definition).

9.3 TAG NUMBERS FOR DEFINING WORDS

A <BUILDS DOES> word has two parts:

- A compile time part (i.e. the <BUILDS part)
- A run time part (i.e. the DOES> part)

The run time part is always invoked via a JSR, and therefore does not require a tag. The compile time part will obviously only be executed at compile time only, and does therefore not need to have a tag in the run time tag table.

Since defining words will often only be called outside a colon definition, you may even be tempted to make them :C definitions; having no tag at all.

But beware! :C definitions can only be executed in immediate mode when the compiler variable "O" is zero! If not (i.e. when you are compiling a stand alone application) the wrong address is called!

But of course you can always give your defining words tags in the 700's. In that case they will not consume tag table space when your application capsule is executed, but still can be executed at compile time.

CHAPTER 10: LOADING

This chapter discusses how the SnapFORTH outer interpreter works, and how you can supply your own input routines to the outer interpreter if you want to load from a non-standard device.

The outer interpreter is called INTERPRET. INTERPRET scans the input stream for words separated by blanks and executes or compiles those words depending on STATE. When SnapFORTH is interpreting STATE has the value 0, when it is compiling STATE has the value C0H.

Since INTERPRET is a text interpreter, it obviously needs some text to interpret. The dictionary variable TIB (Terminal Input Buffer) points to the text that INTERPRET must interpret. The last byte of TIB must be a zero. Another dictionary variable, IN, is used as an offset in TIB, it indicates how much text has already been interpreted. When INTERPRET wants to scan a word from TIB, it calls -WORD, which saves the current value of IN in the dictionary variable LASTIN and scans for a word from TIB, advancing IN. The word is stored at HERE as a count format string. INTERPRET calls FIND to search for the word in the dictionary. Before searching, FIND converts the word to upper case.

10.1 CHANGING IN

As said before; TIB and IN are variables. This offers some interesting possibilities. For instance, what would happen if you assigned a new value to TIB or IN?

IN points to the character position where INTERPRET should continue interpreting after the execution of the current word has finished. Suppose you have typed the following line:

```
." "Hello " 0 TO IN<enter>
```

When this line is about to be interpreted, the situation is as follows:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:^
```

(We will depict the current position of IN by "^", in this case the value of IN is 0.) INTERPRET will read the first word (.") from TIB and advance IN:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:   ^
```

Next, INTERPRET searches for the tag of ." and executes it (by calling the routine EXECUTE).

." Scans the string that it must print. After that, TIB and IN look like:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:             ^
```

After "Hello" is printed, ." returns to whatever has called it. Since INTERPRET happens to be the caller, interpretation continues; the next word (0) is read, and the situation is now as follows:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:               ^
```

The number zero is put on the stack, and here we go again:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:                  ^
```

TO is interpreted and executed, changing %VAR. Now let's do the trick, IN is interpreted, at this moment TIB and IN have the following values:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:                    ^
```

But when IN is executed, it is set to the value 0, and we suddenly face the following situation:

```
TIB: ." "Hello " 0 TO IN<null>
 IN:^
```

This is exactly the situation we started with! So the whole process will continue for ever and ever, unless someone stops it.

This is a nice little trick, suppose for instance that you have redefined "XXX" several times. You can forget them all using:

```
FORGET XXX 0 TO IN
```

This will stop when no more XXXs can be found (assuming that XXX was not protected by FENCE).

IN can only be used as on offset within the current line, even when you are loading from a file. Therefore, the following is illegal:

```
IN ( save IN ) ....... <cr>
.... TO IN
```

This is different from other FORTH systems, such as the HHC's "host operating system", which uses IN as a offset within the current file rather then the current line. The reason for this is that SnapFORTH copies one line at a time from the source file to TIB.

10.2 CHANGING TIB

When you enter a new dictionary, TIB points to a buffer specially reserved for this purpose. But TIB can also be assigned other values. We'll discuss an interesting application of this, you can follow hands on.

First, reserve some working space by typing

```
CREATE MACROBUF 80 ALLOT
```

Next, you must read some text into MACROBUF, for example by typing

```
MACROBUF 80 EXPCT <enter> ." Hello
    world" <enter>
```

Notice that EXPCT converts the <cr> to the zero terminator needed for INTERPRET. You will also need the following definition:

```
: EXECMACRO
  TIB IN LASTIN
  MACROBUF TO TIB 0 TO IN 0 TO LASTIN
  INTERPRET
  TO LASTIN TO IN TO TIB ;
```

Because TIB and IN point at MACROBUF when EXECMACRO calls INTERPRET, the words ' ." Hello world" ' will be executed every time you type

```
EXECMACRO
```

The default location of TIB is &LBUF. When your application wants to use &LBUF itself, you can set TIB to another location while compiling and testing your program with the SnapFORTH capsule.

10.3 LOADING FROM FILES

Four words allow you to load source code:

```
LOAD<name>

LOAD"    <string> <name>

DEVLOAD

DEVLOAD" <string>
```

LOAD and LOAD" read from a file in the HHC. DEVLOAD and DEVLOAD" read from the input of an HHC peripheral, such as the RS232C.

You can, and often should, use another computer or word processor to edit and store FORTH source code, and then use

the RS232C port on the computer or a "communications option" on the word processor to send the source through the HHC RS232C for compilation by DEVLOAD.

LOAD and LOAD" make sure that the file you ask to load is either a text file of the type created by the built-in File System editor, or a compressed text file, of the type created by the Portawriter word processor capsule. If not, the message

```
ERROR: WRONG TYPE <name>
```

appears.

If the file is not present in the current or specified RAM bank, the message

```
ERROR: FILE NOT FOUND <name>
```

appears.

Without changing banks with the I/O key you can specify a bank with a single digit and colon in front of the file name. For example:

```
LOAD 2:MYFILE
```

(This also works for EDIT).

Unless the value of the BEEPER control bit has been reset, each line loaded causes a click. If the clicking stops, either LOADing is complete, or what you loaded caused a ERROR or crash.

LOAD" <searchstring> and DEVLOAD" <searchstring> begin scanning the input, ignoring everything until the searchstring is encountered at the start (not the middle) of an input line. The searchstring must be exactly equivalent to what is in the file. For this purpose, upper case and lower case are not equivalent. The contents of the searchstring and the rest of the file is interpreted until the end of the file.

When LOADing a large file stops at an ERROR, you can FORGET a previous definition that began at the start of a line, and begin loading from the same point with LOAD". This can be much faster than reloading the whole file.

For example, if you were DEVLOADing the following source in the middle of your program:

```
: GO    2R> DROP
    TIB %LLEN CMOVE
    <CALL>  [ TIB , ]  ;
: BURN  TT.ORIGIN
    BEGIN O OVER +! 2+
        DUP @ 0=
    UNTIL DROP
    GO  [ 0 C, ]
    ROMADDR T>H 8000 HERE T>T
        ROMADDR - MOVE
    CR ." Flip switch " DROP ;
```

and got an error on T>T in the next to the last line, you would use VLIST to see that the last successful definition was "GO", edit the error out of the source, and begin loading again with

```
DEVLOAD" ": BURN" <sourcefilename>
   <ENTER>
```

Here FORGET was not needed since the previous definition began at the start of a line.

10.4 END OF FILE CONDITION

By inserting any control character other than CR, LF or TAB at the end of file, you can signal SnapFORTH to stop loading.

Otherwise, you can either say EXIT in your source file (outside a colon definition of course) or press the CLEAR key at the end of the file.

10.5 IMPLEMENTATION OF LOAD

LOADing is controlled by four variables:

L1 - Tag of the routine that is called when WORD needs a new line.

L2 - Parameter/scratch-variable for L1.

L3 - "

L4 - "

L1 must read a source line (if possible) and leaves a flag on the stack. The source line must be read at TIB. It must not exceed 80 decimal characters (for there is no more room at TIB). The flag is true when another line was available and false when there are no more lines. In the latter case, WORD returns the <null> word (length byte is 1, first letter is ASCII value 0).

The executive routine for LOAD, LOAD" and the like is called (LOAD). (LOAD) gets five parameters on the stack:

1: Skipflag for searchstring at &SPAT+1 (count format)
2: New value for L1
3: New value for L2
4: New value for L3
5: New value for L4

(LOAD) saves the old values of L1, L2, L3 and L4. When <null> is eventually executed, it performs an exit, which makes the system return to (LOAD). (LOAD) thereupon restores L1 upto L4 to their old values.

If you want to write your own load routine, all you need to do is make your own L1 routine and pass it to (LOAD). You can use the other load parameters as you like. For instance, LOAD puts the bank# of the file in L2, its current address in L3 and the last address of the file in L4.

10.6 CHECKING THE LINE LENGTH

As said before, the length of a source line should not exceed 80 characters because there is no more room at TIB. But perhaps you have already noticed that you can enter lines of up to 99 characters from the keyboard. This is allowed because &SPAT (where the search pattern is stored) is allocated immediately after TIB. The search pattern is not used when loading from the keyboard. Therefore &SPAT can also be used as an input buffer while reading from the keyboard.

The HHC supports two kinds of text files:

- Built in File System Editor files
- Portawriter files

Both have different file formats:

Lines in a File System Editor file are implemented as a count byte followed by a number of characters. When loading from such a file, SnapFORTH checks if the value in the count byte is greater than 80. In that case the excess characters on the line are ignored.

Portawriter lines are separated by carriage returns (ASCII code 0DH). Consecutive blanks are stored in a clever way. Bit 7 (the 80' bit) of a byte indicates if that byte represents a number of spaces. Thus 81H represents one space, 82H means 2 spaces etc. 80H stands for 100H blanks, which is thus illegal.

EXPND is used to expand a line to TIB. When the length of the line exceeds 80 characters, EXPND gives the error message:

```
ERROR: LINE TOO LONG
```

and compilation stops. However, this seldom occurs since Portawriter's line length defaults to 80.

Note that EXPND is also used to scan a word from TIB to HERE. Therefore it is possible that you get this error message when you type in a long word/string from the keyboard.

When DEVLOADing from a device, excess characters are ignored if a line is longer than 80 characters.

CHAPTER 11: THE HHC KEYBOARD

The keyboard is a QWERTY design. Some changes have been made to allow for the keyboard's small size, and the fact that it is not intended for touch typing. For example, the "space bar" is only three keys wide, and the ENTER (carriage return) key is near the lower right corner.

The keyboard does not have a "control" key, but does have two "shift" keys. One of these, named **SHIFT**, controls alphabetic case; pressing it gives upper case letter, and releasing it gives lower case letters. The other key, named **2nd SFT**, gives special symbols, which are separately marked on the keyboard.

11.1 HARDWARE ARCHITECTURE

Each time a key is pressed it generates an interrupt request which is serviced by a machine code routine in the nucleus. This routine accumulates keystrokes in a queue from which words such as KEY or RIP may fetch them, first-in-first-out.

11.1.1 Hidden and Immediate Keys

At the hardware level, all keyboard keys generate input by placing a one-bit-on code in one of 8 fixed hardware locations.

The nucleus translates some of these codes to ASCII and places them in the keyboard queue, from which words such as KEY may take them. The nucleus acts on other codes (such as LOCK and I/O) directly; these codes never make it to the buffer. The keys that generate such codes are called **hidden keys**.

Some of the hidden keys (such as LOCK) have an immediate effect on the HHC, ***i.e.*** their effects do not filter through the keyboard buffer. The keys that generate such codes are called **immediate keys**.

11.1.2 Special Keys On the Keyboard

The following table shows all the keys on the HHC keyboard that have special functions, including all the hidden and immediate keys.

Key	Function
ON,OFF (H,I)	Turn the HHC on and off. Note that these keys do not turn the power on and off in the conventional sense; they trigger software operations that turn CPU power on and off. The HHC operating system can (and often does) turn itself on and off independently of these keys. RAM is powered continuously, so that the HHC "remembers" the current time, the contents of virtual files, the clock controller's appointments, etc.
CLEAR (H,I)	Causes a soft clear when pressed once, or a hard clear when pressed twice.
SHIFT (H,I)	Case-shifts the next keystroke entered.
2nd SFT (H,I)	Second-shifts the next keystroke entered.
LOCK (H,I)	When followed by SHIFT or 2nd SFT, "locks" the keyboard into shifted or second-shifted mode. The keyboard remains in that mode until SHIFT or 2nd SFT is pressed again.
I/O (H)	Suspends the program that is running and presents a menu through which the user may turn I/O devices on and off. When the user leaves the I/O menu, the program continues running.
STP/SPD (H)	Suspends the program that is running and allows the user to press a numeric key to change the scrolling rate of the display and resume execution. (This action stores a value in a system variable, SPEED.)
HELP (H)	Interrupts the program that is running, if any; waits for the next keystroke, interprets that keystroke as a command, and displays an explanation of the command. (Does not execute the command.) HELP is totally transparent to the application program that is running.

	Pressing HELP twice in a row performs a soft clear.
ENTER	Enters the "carriage return" character.
▲ ▼ ◀ ▶	Customarily used for cursor control, but may be used by a program for any purpose.
DELETE	Used by EXPECT and the file system editor to delete characters. Other programs may use this key for any purpose.
INSERT	Used by EXPECT and the file system editor to insert characters. Other programs may use this key for any purpose.
SEARCH	Used by the file system editor to search for a character string. Other programs may use this key for any purpose.
ROTATE	Used by EXPECT and the system file editor to control rotation of the display. Other programs may use this key for any purpose.
C1 C2 C3 C4	May be used by applications for any purpose. Intended for use as application-specific function keys.
f1 f2 f3	Function keys. Insert a user-defined string of up to 15 characters in the keyboard input stream.

11.1.3 "Unhiding" Hidden Keys

Some of the hidden keys can be "unhidden" by setting control bits in the HHC's nucleus. This makes the keys deposit bytes in the keyboard buffer, just like ordinary keys.

To unhide the SHIFT, 2nd SFT, and LOCK keys, turn on the bit masked by PASSRAW in the byte FLAG2:

```
FLAG2 PASSRAW SET.BITS
```

If you want to unhide the function keys (f1, f2 and f3), you must set both the PASSRAW and the FUNBIT in FLAG2:

```
FLAG2 FUNBIT PASSRAW OR SET.BITS
```

Key	Function
ON,OFF (H,I)	Turn the HHC on and off. Note that these keys do not turn the power on and off in the conventional sense; they trigger software operations that turn CPU power on and off. The HHC operating system can (and often does) turn itself on and off independently of these keys. RAM is powered continuously, so that the HHC "remembers" the current time, the contents of virtual files, the clock controller's appointments, etc.
CLEAR (H,I)	Causes a soft clear when pressed once, or a hard clear when pressed twice.
SHIFT (H,I)	Case-shifts the next keystroke entered.
2nd SFT (H,I)	Second-shifts the next keystroke entered.
LOCK (H,I)	When followed by SHIFT or 2nd SFT, "locks" the keyboard into shifted or second-shifted mode. The keyboard remains in that mode until SHIFT or 2nd SFT is pressed again.
I/O (H)	Suspends the program that is running and presents a menu through which the user may turn I/O devices on and off. When the user leaves the I/O menu, the program continues running.
STP/SPD (H)	Suspends the program that is running and allows the user to press a numeric key to change the scrolling rate of the display and resume execution. (This action stores a value in a system variable, SPEED.)
HELP (H)	Interrupts the program that is running, if any; waits for the next keystroke, interprets that keystroke as a command, and displays an explanation of the command. (Does not execute the command.) HELP is totally transparent to the application program that is running. Pressing HELP twice in a row performs a soft clear.
ENTER	Enters the "carriage return" character.
↑ ↓ ← →	Customarily used for cursor control, but may be used by a program for any purpose.
DELETE	Used by EXPECT and the file system editor to delete characters. Other programs may use this key for any purpose.
INSERT	Used by EXPECT and the file system editor to insert characters. Other programs may use this key for any purpose.
SEARCH	Used by the file system editor to search for a character string. Other programs may use this key for any purpose.
ROTATE	Used by EXPECT and the system file editor to control rotation of the display. Other programs may use this key for any purpose.
C1 C2 C3 C4	May be used by applications for any purpose. Intended for use as application-specific function keys.
f1 f2 f3	Function keys. Insert a user-defined string of up to 15 characters in the keyboard input stream.

11.1.3 "Unhiding" Hidden Keys

Some of the hidden keys can be "unhidden" by setting control bits in the HHC's nucleus. This makes the keys deposit bytes in the keyboard buffer, just like ordinary keys.

To unhide the SHIFT, 2nd SFT, and LOCK keys, turn on the bit masked by PASSRAW in the byte FLAG2:

```
FLAG2 PASSRAW SET.BITS
```

If you want to unhide the function keys (f1, f2 and f3), you must set both the PASSRAW and the FUNBIT in FLAG2:

```
FLAG2 FUNBIT PASSRAW OR SET.BITS
```

11.1.4 Keyboard Codes

The codes generated by the keyboard (with its standard translation table) are roughly a superset of ASCII. All the ASCII printable characters that appear in the HHC's character set have the standard ASCII codes.

11.2 PROGRAM-DEFINED KEYBOARD TRANSLATION TABLES

An application program can define its own keyboard translation table; this allows it to define the binary codes that the keyboard keys will generate. This is useful when the logic of the application requires that the keyboard be made to operate in some non-standard way (e.g., like a terminal keyboard, with a CONTROL key). The same result could be achieved by programming character translation into the application, but a keyboard translation table is more efficient and uses less memory where full-keyboard translation is required.

You use a keyboard translation table by (1) coding the table at compile time, and (2) storing the table's address in a nucleus variable at run time.

11.2.1 Coding a Keyboard Translation Table

A keyboard translation table consists of 192 one- byte entries. Each entry translates one keystroke from **a raw keyboard code** between 00H and 0BFH to a code which is placed in the keyboard buffer.

Each key on the keyboard generates a raw keyboard code between 00H and 3FH. The raw keyboard codes generated by each key on the keyboard are shown in the diagram at the end of this section. SHIFT adds 40H to each raw keyboard code and 2nd SFT adds 80H, yielding the full range of 00H TO 0BFH.

Each time the user enters a keystroke, the raw keyboard code is used as a displacement into the keyboard translation table. The contents of the table entry (i.e., the translated code) is placed in the keyboard buffer.

For example, suppose the user presses the key in the lower left corner of the keyboard (the 'C1' key). This key generates a raw keyboard code of 31H. The HHC places the contents of entry 31H from the current keyboard translation table (the first entry in the table being "number 0") in the keyboard buffer. If the user presses "SHIFT C1," the raw keyboard code is 71H, and the HHC places the contents of table entry 71H in the buffer.

11.2.2 Adopting a New Keyboard Translation Table

A keyboard translation table may not be located in an application capsule, since the nucleus uses it at a time when the application capsule is bank- switched out of the address space. The table may be located in a control ROM (which is convenient for an application program in a control ROM) or in non-switchable RAM.

The easiest way to use a keyboard translation table with a program in an application capsule is to allocate the table on the temporary stack with GRAB, then allocate the TSA above it. (Recall that a TSA may not be more than 250 bytes long; if you made the keyboard translation table part of the TSA, you would have little space left for other data.)

You may not put a keyboard translation table in swappable RAM, *i.e.* in a Programmable Memory Peripheral.

To adopt a new keyboard translation table, store the table's address in the system variable KBVECT:

```
YOUR-KEYBOARD-TABLE KBVECT !
```

If you want to be able to restore the original keyboard translation table, save its address before adopting your own table. The word SAVE is useful for this purpose. Later you can re-adopt the original table simply by storing its address back in KBVECT (*e.g.* with RESTORE).

The keyboard translation table pointer is reset to its original value by a CLEAR.

<table>
<tr><td>-01-
! 21
! 21
1 31</td><td>-02-
“ 22
“ 22
2 32</td><td>-03-
23
23
3 33</td><td>-04-
$ 24
$ 24
4 34</td><td>-05-
% 25
% 25
5 35</td><td>-06-
& 26
& 26
6 36</td><td>-07-
' 27
' 27
7 37</td><td>-08-
(28
(28
8 38</td><td>-09-
) 29
) 29
9 39</td><td>-0A-
_ 5F
_ 5F
0 30</td><td>-0B-
HELP
14</td><td>-0C-
▲
80</td><td>-0D-
I/O
0B</td></tr>
<tr><td>-11-
% 25
Q 51
q 71</td><td>-12-

W 57
w 77</td><td>-13-

E 45
e 65</td><td>-14-

R 52
r 72</td><td>-15-

T 54
t 74</td><td>-16-
+ 2B
Y 59
y 79</td><td>-17-
- 2D
U 55
u 75</td><td>-18-
× 92
I 49
i 69</td><td>-19-
÷ 91
O 4F
o 6F</td><td>-1A-
= 3D
P 50
p 70</td><td>-1B-
◀
81</td><td>-1C-
STP/
SPD
0E</td><td>-1D-
▶
82</td></tr>
<tr><td>-21-

A 41
a 61</td><td>-22-
\5C
S 53
s 73</td><td>-23-
{ 7B
D 44
d 64</td><td>-24-
} 7D
F 46
f 66</td><td>-25-
[5B
G 47
g 67</td><td>-26-
] 5D
H 48
h 68</td><td>-27-
< 3C
J 4A
j 6A</td><td>-28-
> 3E
K 4B
k 6B</td><td>-29-

L 4C
l 6C</td><td>-2A-
f1
15</td><td>-2B-
SEARCH
89</td><td>-2C-
▼
83</td><td>-2D-
ROTATE
07</td></tr>
<tr><td>-39-

Z 5A
z 7A</td><td>-3A-
^ 5E
X 58
x 78</td><td>-3B-
~7E
C 43
c 63</td><td>-3C-
` 60
V 56
v 76</td><td>-30-
@ 40
B 42
b 62</td><td>-20-
| 7C
N 4E
n 6E</td><td>-10-
* 2A
M 4D
m 6D</td><td>-33-
; 3B
, 2C
, 2C</td><td>-0E-
: 3A
. 2e
. 2E</td><td>-1E-
f2
16</td><td>-2E-
INSERT
84</td><td>-3E-
DELETE
85</td><td>-38-
LOCK
86</td></tr>
<tr><td>-31-
C1
ä8B</td><td>-32-
C2
ö8C</td><td>-36-
C3
ü8D</td><td>-37-
C4
ñ8E</td><td>-34-
/ 60

? 3F</td><td colspan="3">-35-
SPACE
20</td><td>-0F-
2ND
SFT
8F</td><td>-1F-
f3
17</td><td colspan="2">-3F-
ENTER
0D</td><td>-2F-
SHIFT
8A</td></tr>
</table>

FIRST LINE: raw keyboard code (unshifted).
SECOND LINE: second-shifted character & ASCII code.
THIRD LINE: shifted character & ASCII code.
FOURTH LINE: unshifted ASCII character & code.

For keys where shift and second-shift are not significant, the second line shows the ASCII code or key function and the third line shows the input value's character code.

11.2.3 The HHC Character Set

The character set used by the HHC is listed in order of the numbers that represent the characters in ASCII notation.

Control Characters

The following table lists characters #0 through #31. These characters are control characters, rather than graphic characters; that is, their customary function is to perform a control function on an output device, rather than to display a symbol like 'A' or '?'.

The meanings of the columns in the table are:

- **hex value:** the character's hexadecimal value.
- **Numeric value:** the number used to represent this character in the HHC's memory. If NV is the numeric value of the character, then CHR$I(NV) is the character.
- **HHC name:** the name, or description, used to identify this character in the HHC system.
- **ASCII name:** the name or description used to identify this character in "pure" ASCII.
- **HHC key:** key on the HHC keyboard that inputs this character.
- **HHC graphic:** the graphic symbol that represents this character when the character is displayed on the LCD. For characters that have a control function, you must use the "ESCDC" escape control sequence to display the character.
- **LCD action:** control function performed by this character when it is sent to the LCD. If this column is empty, the character has no control function; it displays the graphic symbol listed under "HHC graphic".

hex value	numeric value	HHC name	ASCII name
00	0		NUL, Null
01	1		SOH, Start of heading
02	2		STX, Start of text
03	3		ETX, End of text
04	4		EOT, End of transmission
05	5		ENQ, Enquiry
06	6		ACK, Acknowledge
07	7	Rotate; Bell	BEL, Bell
08	8	Backspace	BS, Backspace
09	9		HT, Horizontal tab
0A	10	Line feed	LF, Line feed
0B	11	I/O	VT, Vertical tab
0C	12	Form feed	FF, Form feed
0D	13	Enter	CR, Carriage return
0E	14	Stop/Speed	SO, Shift out
0F	15		SI, Shift in
10	16		DLE, Data link escape
11	17		DC1, Device control 1, XON
12	18		DC2, Device control 2
13	19		DC3, Device control 3, XOFF
14	20	Help	DC4, Device control 4
15	21	f1	NAK, Negative acknowledge
16	22	f2	SYN, Synchronous idle
17	23	f3	ETB, End of transmission block
18	24		CAN, Cancel
19	25		EM, End of medium
1A	26		SUB, Substitute
1B	27	Escape	ESC, Escape
1C	28		FS, File separator
1D	29		GS, Group separator
1E	30		RS, Record separator
1F	31		US, Unit separator

hex value	numeric value	HHC key	HHC graphic	LCD action
00	0		@	
01	1		A	
02	2		B	
03	3		C	
04	4		D	
05	5		E	
06	6		F	
07	7	ROTATE	G	"beep"
08	8		H	Backspace cursor; erase character under cursor after backspace.
09	9		I	
0A	10		J	
0B	11	I/O	K	
0C	12		L	
0D	13	ENTER	M	Erase LCD; move cursor to left edge.
0E	14	STOP/ SPEED	N	
0F	15		O	
10	16		P	
11	17		Q	
12	18		R	
13	19		S	
14	20	HELP	T	
15	21	f1	U	
16	22	f2	V	
17	23	f3	W	
18	24		X	
19	25		Y	
1A	26		Z	
1B	27		[	Begins an escape control sequence.
1C	28		\	
1D	29		]	
1E	30		^	
1F	31		_	

Displayable Characters

The HHC's use of characters #32 through #126 conforms exactly to the ASCII standard.

hex value	numeric value	HHC key	name
20	32	space	
21	33	!	exclamation mark
22	34	"	quotation mark
23	35	#	pound sign
24	36	$	dollar sign
25	37	%	percent sign
26	38	&	ampersand
27	39	'	apostrophe
28	40	(	left parenthesis
29	41	)	right parenthesis
2A	42	*	asterisk, star, or 'times' sign
2B	43	+	plus sign
2C	44	,	comma
2D	45	-	hyphen, dash, or minus sign
2E	46	.	period
2F	47	/	slash
30	48	0	
31	49	1	
32	50	2	
33	51	3	
34	52	4	
35	53	5	
36	54	6	
37	55	7	
38	56	8	
39	57	9	
3A	58	:	colon
3B	59	;	semicolon
3C	60	<	left angle bracket or 'less than' sign
3D	61	=	equal sign
3E	62	>	right angle bracket or 'greater than' sign
3F	63	?	question mark
40	64	@	
41	65	A	
42	66	B	
43	67	C	
44	68	D	
45	69	E	
46	70	F	
47	71	G	
48	72	H	
49	73	I	
4A	74	J	
4B	75	K	
4C	76	L	
4D	77	M	
4E	78	N	
4F	79	O	
50	80	P	
51	81	Q	
52	82	R	
53	83	S	
54	84	T	
55	85	U	
56	86	V	
57	87	W	
58	88	X	
59	89	Y	
5A	90	Z	
5B	91	[	left bracket
5C	92	\	backslash
5D	93	]	right bracket
5E	94	^	caret
5F	95	_	underscore
60	96	`	accent grave
61	97	a	
62	98	b	
63	99	c	
64	100	d	
65	101	e	
66	102	f	
67	103	g	
68	104	h	
69	105	i	
6A	106	j	
6B	107	k	
6C	108	l	
6D	109	m	

6E	110	n	
6F	111	o	
70	112	p	
71	113	q	
72	114	r	
73	115	s	
74	116	t	
75	117	u	
76	118	v	
77	119	w	
78	120	x	
79	121	y	
7A	122	z	
7B	123	{	left brace
7C	124	|	vertical bar
7D	125	}	right brace
7E	126	~	tilde

Additional Characters

The HHC uses characters from #127 up as displayable characters and control characters.

In "pure" ASCII, character #127 represents the control character "delete". Characters above #127 are undefined.

hex value	numeric value	HHC name	HHC key	HHC graphic	LCD action
7F	127	"insert" cursor		▒	
80	128	up arrow	▲	↑	
81	129	left arrow	◀	←	Backspaces cursor; does not disturb character under cursor after move.
82	130	right arrow	▶	→	Advances cursor; does not disturb character that was under cursor before move.
83	131	down arrow	▼	↓	
84	132	"AM" symbol	INSERT	A	
85	133	"PM" symbol	DELETE	P	
86	134	superscript M		M	
87	135	division sign		÷	
88	136	"times" sign		×	
89	137	block cursor	SEARCH	█	
8A	138	"delete" cursor		▯	
8B	139	"a" umlaut	C1	ä	Except when read by GET, causes a "break" in execution of the current program.
8C	140	"o" umlaut	C2	ö	
8D	141	"u" umlaut	C3	ü	
8E	142	"n" tilde	C4	ñ	

Value	Key	Action
7	Rotate	"beep"
B[1]	I/O	inverse K
D	Enter	Erase LCD; cursor to col. 0.
E[1]	Stop/Speed[1]	inverse-image N
14[1]	Help	inverse T
15	f1[1]	inverse U
16	f2[1]	inverse V
17	f3[1]	inverse W

[1] - normally an immediate key.

11.3 EXPECT: KEYBOARD INPUT

The EXPECT word is a convenient means for reading a line from the keyboard. It is similar in function to the EXPECT word in FORTH, but has a more elaborate linkage, and can edit a line of text as well as accept one as input.

You can use EXPECT like this:

```
a l i c e m EXPECT a l' k m'
```

where:

`a`

Address of the buffer where EXPECT is to deposit the line it reads. The operating system provides a buffer named &LBUF which you may use for this purpose.

`l`

Length of the buffer indicated by 'a'. The maximum allowed length is 256. (The length of &LBUF is 80.)

EXPECT will deposit up to 'l' characters in the buffer. Unlike some FORTH implementations of EXPECT, it will ***not*** deposit a NULL after the last character.

`i`

Length of initial text in buffer. If i=0, EXPECT reads a completely new line. If i>0, EXPECT considers the buffer to contain 'i' characters, and displays that many characters on the LCD before beginning to read input.

`c`

Initial position of cursor.

If c=0, EXPECT scrolls the cursor through the line from the beginning to the end, stopping when it reaches the end or when the user presses any key.

If 0<c<i, the cursor is placed in position (c-1), origin 0. For example, if c=3, the cursor is placed in position 2 (over the third character in the buffer).

If c>=i, the cursor is placed just after the last character in the buffer.

If c>26, EXPECT cannot position the cursor without scrolling the beginning of the line off the left edge of the LCD. In this case it scrolls the line as much as necessary to bring position (c-1) under the cursor. It positions the cursor in the right half of the LCD (if (c-1) is near the end of the text) or in the approximate center of the LCD (if not).

`e`

Tag of a word which detects the end-of-line (EOL) character.

EXPECT calls the EOL word each time a character is read from the keyboard. EXPECT finishes editing the line when the EOL word returns a true flag.

The linkage to the EOL word is:

(C --- B)

where C is a character that has been read from the keyboard, and B is TRUE iff the character is an EOL character.

You can use the file system editor's EOL word if you want to. Its name is EOL. It accepts 4 characters as end-of-line characters: ENTER, ↑, ↓, and SEARCH.

The initial editing mode EXPECT should operate in. The following symbolic constants represent the possible modes:

%OVR — Overstrike. This is EXPECT's "normal" mode.

Each input character replaces the character under the cursor, and causes the cursor to move right.

If the cursor is at end-of-line, each input character is appended to the line.

The left- and right-cursor keys move the cursor; but an attempt to move the cursor past the beginning or end of the text will not succeed, and will make the HHC beep.

%INSERT — Insert. Each input character is inserted just before the character under the cursor. All following

characters are bumped one position to the right. EXPECT returns to overstrike mode after receiving one character.

The normal cursor is replaced by an "insert" cursor, which looks like a filled-in rectangle with alternate dots on and off.

If the cursor is at end-of-line, each input character is appended to the line.

%DELETE	Delete. The "cursor left" key deletes the character under the cursor and moves the cursor to the left, over the previous character; the "cursor right" key deletes the character under the cursor and leaves the cursor where it was, which is now the next character. Any other key is considered an error, and causes EXPECT to beep. The normal cursor is replaced by a "delete" cursor, which looks like an outline of a rectangle, with the interior clear. EXPECT returns to overstrike mode after receiving one character.
%LI	Lock-insert mode. Identical to insert mode except that EXPECT does not return to overstrike mode after receiving one character.
%LD	Lock-delete mode. Identical to delete mode except that EXPECT does not return to overstrike mode after receiving one character.

l'

Length of the text in the buffer when EXPECT finished processing.

k

The character which caused the EOL word to return TRUE.

m'

Editing mode that applied to the last character received from the keyboard.

11.3.1 EXPECT Modes

Note that the "mode" parameter in a call to EXPECT sets only the INITIAL mode of EXPECT, that is, the mode that applies to the first keystroke processed. Each keystroke may change the mode.

As was explained above, "overstrike" is EXPECT's "normal" mode of operation. "Insert and delete" modes each revert to "overstrike" after processing one character. "Insert-lock" and "delete-lock" modes do not.

The user can change EXPECT's mode from the keyboard as follows:

The INSERT and DELETE keys put EXPECT in "insert" and "delete" modes, respectively, for one keystroke. The key sequences "LOCK INSERT" and "LOCK DELETE" put EXPECT in "LOCK INSERT" and "LOCK DELET" modes.

When EXPECT is in "LOCK INSERT" mode, pressing INSERT again will return it to "overstrike." When EXPECT is in "LOCK DELETE" mode, pressing DELETE again will return it to "overstrike."

When EXPECT is in "LOCK INSERT" mode, pressing DELETE will place it in "delete" mode for one character, after which it will revert to "OVERSTRIKE." When EXPECT is in "LOCK DELETE" mode, pressing INSERT will place it in "insert" mode for one character, after which it will revert to "OVERSTRIKE."

11.3.2 EXPCT

The SnapFORTH Capsule also has a routine EXPCT (note the spelling), which behaves like the classical FORTH EXPECT. Its parameters are

```
( Address Length --- )
```

However, EXPCT is a routine in the SnapFORTH capsule itself and not a nucleus routine, therefore it can only be used by dictionary programs, not by stand-alone ROM capsule programs.

11.3.3 Example 1

Here is a simple example of how EXPECT may be set up and used.

```
: EOLWD    ( C ---- B )
   &CR = ;
```

(P true if C is a carriage return.)

```
: CALL-EXPECT ( ---- ADR LEN )
   &LBUF %LLEN        \A/L buffer
   0 0      \INIT INITC (null initial text); EOL, MODE
   'X EOLWD %OVR
   EXPECT
   2DROP      \Drop EOLCHAR & MODE'

;
```

(Call EXPECT & return adr/len of data read.)

This example contains two colon definitions. The first is an EOL word that will be used by EXPECT. The second calls EXPECT and returns the address and length of a line of text.

'X EOLWD compiles to a literal which, when executed at run time, stacks the tag of EOLWD.

11.3.4 Example 2: Editing Text

Suppose we wanted to modify this code to perform a slightly different function. We have a piece of text that we want to place in the buffer before EXPECT is called; we want EXPECT to display this text and position the cursor at the end of it, then allow the user to back up the cursor in order to edit the text.

We could accomplish this by redefining CALL-EXPECT like this:


```
: CALL-EXPECT ( ---- ADR LEN )
            \Move initial message into buffer.
   MSGPTR COUNT    \A/L of message.
   &LBUF SWAP MOVE  \Move message to buffer.
   ..        \Call EXPECT.
   &LBUF %LLEN     \A/L of buffer,
   MSGPTR C@ 0     \INIT, INITC,
   'X EOLWD %OVR   \EOL, MODE;
   EXPECT
   2DROP    \Drop EOLCHAR & MODE'
         .
         .
         .
;
```

(Call EXPECT & return adr/len of data read.)

11.3.5 Example 3: Changing the End-of-Line Character

Suppose we wanted to treat blank, as well as ENTER, as an end-of-line character? We could do so by redefining EOLWD like this:

```
: EOLWD    ( C ---- B )
      DUP &CR = SWAP BL = OR ;
```

11.4 CHARACTER INPUT: KEY AND ?KEY

When you need to do input a character at a time, you can use KEY just as you would in FORTH.

The word ?KEY determines whether a keystroke is waiting in the keyboard buffer.

11.4.1 Note on Battery-Conserving Code

It is not good practice to write SNAP code that loops indefinitely while testing for a condition. The place where this sort of code comes up most often is in testing for a keystroke; for example:

```
BEGIN
     REPETITIVE-TASK
?KEY UNTIL
...
```

This sort of loop is bad because it keeps the HHC'S CPU on continuously while the HHC is not being used. This drains the HHC's batteries. In the worst case, it can force the user to recharge the batteries every few hours of use.

As an alternative to this sort of code, you can use the word NAP to put the HHC "to sleep" for a specific length of time, or until a keystroke becomes available, whichever occurs first.

The linkage to NAP is:

```
sec ticks NAP  [c]  b
```

"sec" and "ticks" are the number of seconds and ticks (256ths of a second) the HHC is to nap.

If a keystroke becomes available before the specified interval expires, the nap ends immediately. "c", the value of the keystroke, is stacked and "b" is TRUE.

If no keystroke becomes available before the specified interval expires, the nap ends at the end of the interval. "c" is not stacked; "b" is FALSE.

Here is an example of code that uses NAP to perform a repetitive task approximately every 1/10 second until a keystroke becomes available:

```
BEGIN
  REPETITIVE-TASK
  0 27 NAP
UNTIL        \ At exit, character is on stack.
```

11.5 SIMULATING KEYBOARD INPUT FROM A PROGRAM

The HHC stores keystrokes that it has not yet processed in a KEYBOARD BUFFER. You can look ahead at the contents of this buffer before you do a KEY, and you can put characters into the buffer, so that your program, in effect, is pressing keys on the keyboard.

11.5.1 Structure Of the Keyboard Buffer

The keyboard buffer is addressed by the symbol KQ; it is 8 bytes long. The HHC places the first character typed in the 7th byte, the second in the 6th byte, etc. After the 8th character typed has been placed in the 0th byte, the 9th character typed is placed in the 7th byte (assuming the 1st character typed has been read by the program), and so on.

The HHC maintains two pointers to the keyboard buffer. The "store" pointer contains the offset (0 to 7) of the byte where the next character typed on the keyboard will be stored; it is one byte long, and is addressed by the symbol KQI. The "fetch" pointer contains the offset of the byte where the next character read by the program will come from; it is one byte long, and is addressed by the symbol KQO.

For example, suppose you have just entered your program and nothing has been typed yet. (This is an oversimplification, since something had to have been typed for you to have gotten to your program from the primary menu. The discussion of the proceess is accurate, however.) The keyboard buffer and its pointers look like this:

```
- - - - - - - -  KQ
              ↑  KQI
              ↑  KQO
```

Now suppose you type in 3 characters, 'ABC'. Your program does a KEY, so that you have input 3 characters, and your program has read one. Now the buffer looks like this:

```
- - - - - - - -  KQ
          C B A
        ↑        KQI
              ↑  KQO
```

You continue typing in the alphabet, and your program continues reading characters. At some later time when you have typed the alphabet through J and your program has read it through E, the buffer looks like this:

```
- - - - - - - -  KQ
H G F E D C J I
            ↑    KQI
      ↑          KQO
```

11.5.2 The Pushkey Buffer

The HHC has a second buffer, called a **pushkey buffer**, which it uses to hold characters that are "pushed" back onto the input stream by a program.

Whenever your program does a KEY, the HHC returns any characters that are in the pushkey buffer before going to the keyboard buffer. Thus, any characters you store in the pushkey buffer will be read ***before*** characters typed in through the keyboard, even if the keyboard characters go into their buffer first.

The pushkey buffer is addressed by the symbol PKQ; it is 4 characters long. It is used as a LIFO queue (the last character put in is the first taken out). The "bottom" of the buffer, where the first character is pushed, is at PKQ; the "top," where the last character may be pushed, is at PKQ+3.

A one-byte counter at PKCT indicates the number of characters already in the buffer. Its value may be 0 to 4. A value of 0 means the pushkey buffer is empty; 4 means the pushkey buffer is full, and there is no more space for characters to be pushed onto it.

For example, if you push a '2', then a 'G' onto the buffer, the buffer looks like this:

```
_ _ _ _ PKQ
2 G
        PKCT = 2
```

The next KEY will get the 'G'; the one after that will get the '2':

```
_ _ _ _ PKQ
2
        PKCT = 1

_ _ _ _ PKQ
        PKCT = 0 (empty)
```

11.5.3 Operations On the Buffers

To inspect the keyboard buffer, compare the values of KQI and KQO. If they are equal, the keyboard buffer is empty. If not, use them to extract the contents of the keyboard buffer.

To inspect the pushkey buffer, examine the pushkey pointer. If it is 0, the pushkey buffer is empty. If it is not zero, use it to extract the contents of the pushkey buffer.

To store a character at the top of the keyboard buffer, so that it will be the next character to come out: first, check to make sure that the buffer is not full. Then move the value of KQO backwards one location, and store the character in the location that KQO now points to. (Remember to wrap KQO around to the beginning of the buffer if moving it backwards moves it past the end.)

Do ***not*** try to put a character into the "bottom" of the keyboard buffer by storing into the location indicated by KQI and then advancing KQI. If you do this, there is always a risk that the device driver for the keyboard will get an interrupt and store a character in the buffer between the time you fetch KQI and the time you store it back. If that happens, your character will overlay the one from the keyboard.

To put a character into the pushkey buffer, use the PUSHKEY word. PUSHKEY pops a character off the stack, and leaves nothing. It gives no warning when the pushkey buffer overflows.

11.5.4 "Typing" a Function Key

FUNCT is the name given to location 208H and FUNY is location 209H. Unfortunately SnapFORTH does not have symbols defined for these locations.

You can "type" a function key by storing it in the keyboard buffer, but the preferred way to do it is to store the length of the desired key's definition in the byte at FUNCT, and the offset of the definition (1, 17, or 33) in the byte at FUNY.

When you "type" a function key in this way, your program will read it before any of the characters in the keyboard buffer or the pushkey buffer.

You cannot "type" a function key by storing it into the pushkey buffer. If you try, your program will read the ASCII code that represents the function key (15H, 16H, or 17H) instead of the function key's current definition.

11.6 KEYBOARD I/O WORDS

11.6.1 Symbolic Constants For Character Codes

symbol	key	comments
&BL	space	***See also*** the word '20H'.
&BSP		Represents ASCII "backspace," not represented on the HHC keyboard.
&C1	C1	
&C2	C2	
&C3	C3	
&C4	C4	
&CR	ENTER	
&ESC		Represents ASCII "escape," not represented on the HHC keyboard.
&HLP	HELP	
&IO	I/O	
&LF		Represents ASCII "line feed," not represented on the HHC keyboard.
&STP	STP/SPD	

11.6.2 Keyboard Words

'KEY	(--- A)
(KEY)	(--- C)
?KEY	(--- B)
KBVECT	(--- A)
KEY	(--- C)
NAP	(TICKS SEC --- [C] B)
PASSRAW	(--- FL)
PUSHKEY	(C ---)

11.6.3 EXPECT Words

%DELETE	(--- N)
%INSERT	(--- N)
%LD	(--- N)
%LI	(--- N)
%OVR	(--- N)
&LBUF	(--- A)
EOL	(C --- B)
EXPECT	(A L I C E M --- A L' K M')
LOCKED?	(--- FL)

CHAPTER 12: THE LCD DISPLAY AND BEEPER

12.1 THE LCD

The HHC's primary output device is the liquid crystal display (LCD) on its front panel.

The LCD is a matrix of dots, 159 wide by 8 high, which may be turned on (black) and off (clear) in any combination. A high-level set of I/O words for the LCD allows you to do ASCII I/O. In this context the LCD consists of 26 character positions, each 5 columns wide and followed by one empty column, with 3 extra columns of dots at the right end.

The dot columns are addressed left to right as #0 through #158. The character positions are addressed left to right as #0 through #25.

The LCD has a **cursor** which is normally displayed as a solid black rectangle flashing in a character position. The cursor may be turned on and off by the program, using the words START.CURSOR and STOP.CURSOR. It may be positioned on the display with POSN. Its graphic representation may be redefined (see below).

The LCD's character set is roughly a superset of ASCII. It is described in detail in the previous chapter (see section 11.2.3). The following are some terms that will be used throughout this chapter:

- VALUE: the binary value that represents this character.
- ASCII NAME: the name or description used to identify this character in "pure" ASCII.
- HHC KEY: key on the HHC keyboard that inputs this character.
- HHC GRAPHIC: the graphic symbol that represents this character when the character is displayed on the LCD. For characters that have a control function, you must use the "ESCDC" escape control sequence to display the character.
- LCD ACTION: control function performed by this character when it is sent to the LCD. If this column is empty, the character has no control function; it displays the graphic symbol listed under "HHC graphic."

12.1.1 The LCD Character Set: Special Characters

These characters do not normally display their "HHC graphic" when sent to the LCD.

Val	Symbol	Normal action when EMITted
7	inverse G	"beep"
8	inverse H	Backspace, then erase char. under cursor.
D	inverse M	Erase LCD; move cursor to left edge.
1B	inverse [	Begins an escape control sequence.
81	◂	Backspaces cursor; does not disturb character under cursor after move.
82	▸	Advances cursor; does not disturb character that was under cursor before move.

The symbols for these special characters may be displayed by writing them with SOFT.EMIT instead of EMIT, or by EMITting the escape control sequence "&ESC ESCDC val."

12.2 I/O WORDS

You may approach display I/O on several levels. We will describe each level, beginning with the highest.

12.2.1 EMIT

You may use EMIT to display characters on the LCD as though it were a little dumb video display with horizontal scrolling.

LCD Control Characters

The LCD treats a number of characters as control characters rather than data. Examples are 08H (ASCII "backspace"), which backspaces the cursor and erases a character, and 07H (ASCII "bell"), which emits a beep on the HHC's speaker.

A complete list of LCD control characters may be found in the previous chapter in the section describing the HHC character set.

If you want to display control characters on the LCD as data for diagnostic purposes or to get a special effect, you can do so with the word SOFT.EMIT. SOFT.EMIT functions exactly the same as EMIT, except that it treats control characters as data, including the cursor control codes beginning at 80H and escape control sequences beginning with 1BH (see below). The symbols that SOFT.EMIT uses to represent control characters are shown in the previous chapter.

The Control Byte

The HHC's EMIT allows you a number of unusual controls over the way each character is displayed. You can display a character in inverse video; with flash (blink); from an alternate, program-defined character set; with a floating accent mark; or in any combination of these modes.

You control the display mode of a character by turning on bits in the high-order byte of the parameter that you pass to EMIT (which contains the character to be displayed in the low-order byte). The functions of the bits in this byte are:

Bit	Symbolic constant	Function
80H	CNEG	1-> reverse video, 0-> standard.
40H	CFLSH	1-> flash character, 0-> no flash.
20H	CQMK	1-> flash character alternating with '?'s, 0-> no flash.
10H	CALT	1-> use alternate character set, 0-> use standard set.
F-0	- -	value determines what floating accent is superimposed on the character. See details below.

For example, you can display a character in inverse video like this:

```
... C@        \The character.
CNEG FLIP     \Inverse video flag.
OR EMIT       \Combine & output.
```

The value of the low-order nybble of the high-order byte represents a **floating accent** which will be superimposed on the character as it is displayed.

The image of a floating accent is obtained from a table which your application program must define. Details on how to define a floating accent table are given in a later section.

Escape Control Sequences

The LCD also recognizes a number of three-byte **escape**

control sequences beginning with the value 1BH (ASCII "escape"). These sequences can do things like shift all characters written to the LCD into inverse-image until further notice.

Escape control sequences are described in the chapter on "Peripheral I/O and Timer Services," since they are implemented consistently for all peripherals as well as for the LCD.

Clearing the LCD: CR

The CR emits a "carriage return" (0DH, ENTER on the HHC keyboard). This is the customary, way of clearing the LCD, which will have the desired effect when LCD output is directed to another device. (That usually means starting a new output line.)

The effect of CR is actually rather complex. It does not always cause an immediate clear; often it sets a flag that causes a clear the next time a character is EMIT'ed. This allows a program to write a line and end it with a CR without making the line disappear from the LCD as soon as it is displayed.

The behavior of CR is described in detail in the glossary at the end of this manual.

To do clear the LCD immediately, use FAST.CR instead of CR.

12.2.2 "Random Access" Output To the LCD

Using POSN

You can move the cursor to any character position on the LCD without disturbing the data that is displayed by executing the word POSN:

```
n POSN
```

POSN moves the cursor to character position "n." It functions by EMITing a string of control characters to move the cursor left or right to the desired position. SMART- POSN has the same effect as POSN, but it works with peripherals like the video as well as the LCD.

Storing Data Directly Into the LCD Buffers

You can update the LCD by storing data directly into its buffers. This is a lower level technique than using SMART-POSN and EMIT, and allows you to update the LCD more efficiently.

The HHC forms an image on the LCD by displaying the contents of two buffers, each of which contains one byte per character position. You can place a character in any position (in effect, "random access" output) by storing data directly into these buffers.

The two buffers are the **character display buffer**, which contains ASCII characters, and the **character control buffer**, which contains control bytes.

You can store data in the character display buffer with the aid of the word CHARBUF:

```
n CHARBUF adr
```

CHARBUF adds the offset "n" to the location of the display buffer, which is DBUF. Thus its definition is:

```
: CHARBUF DBUF + ;
```

"n" is a character position (0 to 26); "adr" is the address of the corresponding byte in the character display buffer.

Similarly, you can store data in the character control buffer with the aid of the word DBUF1:

```
n DBUF1 + adr
```

"n" is a character position; "adr" is the address of the corresponding byte in the character control buffer.

After modifying the contents of the buffers, you must execute UPDISP to make the HHC update the LCD. UPDISP takes one parameter, the position of the first column to be updated.

For example, to display an inverse video character with an umlaut in position 18 of the LCD:

```
 ... C@              \The character
18 CHARBUF C!        \Store it in character buffer
   3 CNEG OR         \The control byte
18 DBUF1 + C!        \Store it in control buffer
0 UPDISP             \Update display
```

Note that you can find out what is on the LCD by ***fetching*** from the addresses given by CHARBUF and CTRLBUF. For example, to capitalize the character in position #5:

```
5 CHARBUF C@         \Push character
L>U                  \Force upper case
5 CHARBUF C!         \Store it back
5 UPDISP             \Update display
```

12.2.3 Graphics Output

At the dot level, the image on the LCD is determined by the contents of the LCD display buffer. This is a memory- mapped area 159 bytes long, each byte contrólling the contents of one dot column in the LCD. The buffer is addressed by the word DSPLY.

The low-addressed byte of the display buffer represents the leftmost dot column in the LCD; the second high- addressed byte represents the rightmost column; and the highest addressed byte (DSPLY + 159) represents the eight blips on the bottom of the LCD. Within each byte, each bit represents one dot; the highest-order bit represents the bottom dot and the lowest-order bit represents the top dot. In each bit, a 1 makes the dot dark, and a 0 makes the dot clear.

You can manipulate the LCD buffer, and thus the dots, with the word DMOVE. DMOVE works like MOVE, but performs special processing necessary to move information into and out of the LCD display buffer. Its linkage is:

```
from to len DMOVE
```

"from" is the address of an area containing one or more bytes of data to be moved to the LCD buffer; "to" is the address in the buffer the data is to be moved to; "len" is the number of bytes to be moved.

For example, to update dot columns 79-158 (approximately the right half of the LCD) from a location addressed by RHALF, you could execute the following code:

```
RHALF   DSPLY 79 +   80   DMOVE
```

You need not execute UPDISP to update the display.

Note that you can use DMOVE to read the contents of the LCD, as well as to write it. Simply specify a "from" address that is in the LCD buffer, and a "to" address that is elsewhere.

Caution: before starting to use DMOVE, clear the LCD by executing the following code:

```
LCD.CR          \ Clears LCD at 'DMOVE' level.
STOP.CURSOR     \ Makes cursor invisible.
26 BUFPOSN C! \ Moves cursor to char POSN 26.
```

Avoid writing dots into columns 156-158 of the display; these dots are in the visible half of character position 26, where the cursor is, and a bug in the nucleus prevents dots in the cursor's position from being displayed properly.

When you are ready to begin writing characters to the LCD again (e.g., with EMIT), execute the following code:

```
LCD.CR          \ Clears LCD at 'DMOVE' level.
START.CURSOR    \ Makes cursor visible again.
CR              \ Moves cursor back to left edge.
```

12.2.4 Program-defined Blips

When you are using character-oriented words such as EMIT to write to the LCD, neither the cursor, nor any character, will ever appear in dot columns 156-158. In this situation you can use DMOVE to put your own, program-defined "blips" in these columns.

If you use DMOVE to create your own blips, you must be careful to refresh the blips whenever EMIT causes the LCD to rotate. Although columns 156-158 never contain characters, they DO rotate along with the rest of the display.

When the display rotates left (the usual case), clear columns 156-158 first, so that the blips will not rotate into columns 150-152. After the rotation, restore the blips to columns 156-158 (which will initially be empty).

When the display rotates right the blips need not be cleared, since they will rotate off the LCD. Simply restore them after rotation.

Caution: DMOVE is not compatible with the higher (character-oriented) levels of LCD words when used to modify columns #0 to #152; that is, in all cases except when used to display user-defined blips. Do not try to mix DMOVE with character I/O words. Clear the LCD immediately before switching from character-oriented I/O to DMOVE, or vice versa, as described above.

Caution: EXPECT uses columns 156-158 to indicate a line of text that overflows the LCD. Avoid using these columns in a way that might conflict with EXPECT.

12.3 ALTERNATE CHARACTER SETS

The HHC allows you to define your own character set for displaying characters on the LCD. To define your own character set, you must (1) create a character translation table containing a dot-pattern definition for each displayable character, and (2) inform the HHC of its location.

12.3.1 Defining a Character Translation Table

The HHC displays each character in a 5x8 matrix on the LCD. A character set table consists of up to 224 5-byte entries corresponding to the values 20H through 0FFH. The values 0 through 1FH are EMITted as the inverted graphics of 40H-5FH. The entries define the dot patterns to be displayed for each character.

Each entry consists of one byte for each column of dots that makes up the character being defined. The bytes, from first to last, represent the dot columns from left to right.

Each byte consists of one bit for each dot that makes up the column being defined. The bits, from high-order to low-order, represent the dots from bottom to top.

For example, consider the standard character set used in the simulator as of this writing. Entry #20H, representing "space," is all zero. The following 5-byte entry, representing '!', is

00 DE DE 00 00 , producing:

```
.....
.00..
.00..
.00..
.00..
.....
.00..
.00..
```

The entry representing '/' is

40 20 10 08 04 , producing:

```
.....
.....
....0
...0.
..0..
.0...
0....
.....
```

The entry representing 'B' is

82 FE 92 92 6C , producing:

```
.....
0000.
.0..0
.0..0
.000.
.0..0
.0..0
0000.
```

Notice that the top row of dots is not used for ordinary characters in the standard character set; it is reserved for accent signs.

The standard baseline (the line the letters appear to rest on) is at the bottom row of dots.

The standard X-height (the height of a lower case letter) is 5 rows of dots.

A character translation table with a nonstandard domain: if binary values through 0FFH are not going to be displayed, the table need not be a full 224 characters long. The programmer bears the risk that his code will send an unexpected high value to the LCD, and display random data beyond the end of the table.

A Convenient help word: the following word is a useful help for defining an entry in a character set table and accent table:

```
: CHAR  0 0 0 0 0   \Empty pattern
   -WORD  \ Skip comment word
   8 0 DO -WORD HERE 6 + HERE 1+
              \ Scan one row
         DO 5 ROLL
              I C@ &0 = FLIP OR 2/
              \ Add bit to pattern
         LOOP
   LOOP
         \ Store the pattern in the dictionary:
      5 0
      DO 5 I - ROLL
      C,
      LOOP ;
```

It is used like this:

```
CHAR B
    .....
    0000.
    .0..0
    .0..0
    .000.
    .0..0
    .0..0
    0000.
```

producing 82 FE 92 92 6C

Notice that the letter 'B' after CHAR is just a comment; it is skipped by the first -WORD. The character table should be loaded from a file, when entered from the keyboard, - WORD will not read past the end of line.

Of course, you may enter the character definition from the keyboard if you type everything on one line:

```
CHAR B ..... 0000. .0..0 .0..0
      .000. .0..0 .0..0 0000.
```

12.3.2 Informing the HHC of the Table's Address

A character translation table, like a keyboard translation table, may not be located in an application capsule. All the considerations presented for keyboard translation tables in the chapter on "The HHC Keyboard" apply to character translation tables as well.

The HHC has a "standard" character translation table and an "alternate" table. Which table is used to display a given character depends on the value of the CALT bit in the high-order byte EMITted with the character.

Normally the standard and alternate character translation tables both implement the HHC's standard character set. You can change either or both to point to your own table(s).

To adopt a new standard character translation table, store the table's address at CHVECT1:

```
YOUR-CHARACTER-TABLE CHVECT1 !
```

Similarly, to adopt a new alternate character translation table, store its address at CHVECT2.

If you want to restore the original character translation table, save its address before adopting your own, and restore it later.

When you change the character translation table, your change will become visible as soon as the LCD is modified (e.g., by an EMIT that causes it to rotate or by UPDISP).

Both character translation tables are restored to their original setting by a CLEAR.

12.3.3 Defining a Floating Accent Translation Table

Just as you can define your own translation table for the HHC's character set, you can define a translation table for the floating accents.

The accent translation table has the same structure as the character translation table, except that it has a maximum of 16 entries. Each entry defines a character that will be

superimposed on a text character when that text character is displayed with the appropriate accent bits turned on in the high-order byte.

Although there is no default accent translation table, there is a set of standard meanings for the accent- character bit values:

0 no accent
1 accent acute
2 accent grave
3 umlaut

4 circumflex
5 inverted circumflex
6 tilde
7 bar

Informing the HHC Of the Accent Translation Table's Address

An accent translation table, like a character translation table or a keyboard translation table, may not be located in an application capsule.

To make the HHC use your accent translation table, store its address at ACVECT:

```
YOUR-ACCENT-TABLE ACVECT !
```

There is no way to designate different accent tables for the standard and alternate character sets.

The accent translation table is reset to its original, undefined state by a CLEAR.

12.3.4 Defining an Alternate Cursor Pattern

The HHC's standard cursor pattern is a 5 by 8 block of dots that are all black. The cursor blinks to permit the user to see the character underneath it.

You can create an alternate pattern for the cursor by defining a 5-byte area in the same format as an entry in an alternate character set table.

To instruct the HHC to use an alternate cursor pattern, store the pattern definition's address in the location ^CURSOR:

```
YOUR-CURSOR-DEFINITION ^CURSOR !
```

The alternate cursor pattern will become visible the next time a high-level I/O operation is done on the LCD.

NOTE: EXPECT resets the cursor pattern, depending on the value of its 'm' parameter.

12.4 ROTATION MODE

You can change the way in which the LCD responds to overflow (EMITting a character when the cursor is past the last character position) by storing a value into the byte at (ROTMODE).

Meaningful values you can store at (ROTMODE) are:

0: "Fill mode." After the LCD is full, the HHC erases everything and starts filling the LCD again, left to right.

1: "Rotate mode." after the LCD is full, the HHC shifts characters off the left end to make room for more characters on the right. (This is the HHC's normal rotation mode.)

2: "Rotate and fill mode." Same as rotate mode except that carriage returns are ignored. The HHC never clears the LCD.

Note that rotation mode is ***not*** reset by a CLEAR.

12.5 THE STOP/SPEED KEY

The STOP/SPEED key works by storing a value into a byte addressed by the symbol 'SPEED'. You can change the speed of the HHC's menu rotation, LCD rotation, and keyboard auto-repeat by storing a different value in this byte.

The meaning of the byte at SPEED is:

Value	STOP/SPEED setting
10	1 (slowest setting)
9	2
8	3
7	4
6	5
5	6
4	7
3	8
2	9
1	0 (fastest setting)
0	faster than fastest STOP/SPEED setting

You can make the HHC ignore the value of SPEED, and display data with no delay, by turning on the bit represented by the symbol RUNBIT in FLAG2:

```
FLAG2 RUNBIT SET.BITS
```


12.6 SLAVING PERIPHERALS TO THE LCD

12.6.1 FLAME-ON

In applications which do not make explicit use of peripherals, it is a common practice to attach any peripherals that are connected to the HHC and slave them to the LCD. Then output that is sent to the LCD will be sent to each peripheral as well.

The word FLAME.ON performs this attach-and-slave service. It is generally executed during the initialization of an application. When FLAME.ON is executed it scans the HHC bus for one peripheral of each type that is connected, attaches it, and slaves it to the LCD.

FLAME.ON will attach a device that is physically connected to the HHC whether or not the device has been turned on through the I/O menu. Until a device is turned on, however, it will not actually be slaved to the LCD. If a device is connected to the HHC when FLAME.ON is executed, but is not turned on until later, it will be slaved to the LCD when it is turned on; if it is later turned off, it will be un-slaved again. (But the reverse is not true; you cannot slave a device to the LCD by connecting the device after an application has been initialized. Indeed, if you connect a device to the HHC at any time, the HHC operating system will do a hard clear!)

12.6.2 Vectored I/O; (EMIT)

A different technique makes it convenient for you to slave a specific device to the LCD. You can slave and un-slave this device during execution if you wish to do so.

The technique makes use of the words (EMIT) and 'EMIT. (EMIT) is the HHC's "inner EMIT" word, which performs all the functions normally associated with EMIT. 'EMIT is the "EMIT vector variable, which normally contains the tag of (EMIT).

By placing your own tag inside the 'EMIT variable, you have the ability to not only redirect the output to different devices, but also transform the output any way you wish. For example, suppose you wanted to change the output to the LCD so that all characters are displayed in UPPER CASE. You could use the following definitions:

```
:MY-EMIT (char --)
    L>U (EMIT) ;
:TURN-ON-MY-EMIT
    'X MY-EMIT 'EMIT ! ;
:TURN-OFF-MY-EMIT
    'X (EMIT) 'EMIT ! ;
```

TURN-ON-MY-EMIT stores the tag of MY-EMIT at 'EMIT. TURN- OFF-MY-EMIT stores the tag of (EMIT) at 'EMIT.

While 'EMIT contains the tag of MY-EMIT, any call to EMIT will result in a call to MY-EMIT rather than (EMIT).

The convenient thing about vectored I/O is that adding it to an existing program is completely "clean". No matter how many places EMIT appears in the program, revectoring changes EMIT's function globally. At the same time, redirected I/O can be turned on or off at run time by executing a single word.

12.7 BLIPS

Blips are the eight stationary indicator dots below the text line on the LCD display. There is a fixed set of blips with fixed meanings.

The blips may be turned on and off by changing the HHC's **blip mask** with the words ANDBLIP and ORBLIP.

The following words are symbolic constants which should be used to control the bits in the blip mask. On the LCD, the lowest-order bit is represented on the left, and the highest-order bit on the right.

Word	Value (hex)	Meaning
BATBLIP	01	Battery is low.
SHFTBLIP	02	Keyboard in SHIFT (or SHIFT LOCK) state.
LOCKBLIP	04	Keyboard in LOCK state.
2SHFTBLIP	08	Keyboard in SECOND SHIFT state.
RECBLIP	10	EXPECT is in delete mode.
MEMBLIP	20	EXPECT is in insert mode.
ALRMBLIP	40	An alarm is "ringing."
RUNBLIP	80	Carrier detect on modem.

It is not good practice to use most of the blips, since the HHC's intrinsic software will continue to use them as though it had exclusive control. Thus the blips will not be accurate reflections of either their standard meanings or your non-standard ones. This caution applies with somewhat less force to SECBLIP and MEMBLIP, which are used only by EXPECT, and thus are used only by application programs. RUNBLIP is comparatively available as well.

Note: there are three columns of dots at the right edge of the LCD matrix that are not used to display characters; they may be used to display additional, user-defined "blips." See the description of LCD graphics in the previous section for details.

12.8 BEEPS AND SQUEAKS

The HHC has a small built-in speaker which it may use to make various kinds of noise.

A **beep** is a noise of fixed pitch and duration which is analogous to the bell on a conventional terminal. You can create a beep by executing the word BEEP, or by sending a 07H (control-G) to the LCD.

A **squeak** is a noise of controllable pitch and duration. It is made by the I/O word SQUEAK. (See details in glossary.)

12.9 DISPLAY AND OUTPUT WORDS

12.9.1 LCD Words

EMIT	(C ---)
EMIT.ESC	(DATACHAR CTRLCHAR ---)
'EMIT	(--- A)
(EMIT)	(C ---)
(FREEZE)	(--- A)
(ROTMODE)	(--- A)
ACVECT	(--- A)
BEEP	(---)
BUFPOSN	(--- A)
CALT	(--- N)
CFLSH	(--- MASK)

CHARBUF	(N --- A)
CHVECT1	(--- A)
CHVECT2	(--- A)
CNEG	(--- N)
CQMK	(--- N)
CR	(---)
DBUF	(--- A)
DMOVE	(FROM TO LEN ---)
DSPLY	(--- A)
FAST.CR	(---)
LCD.CR	(---)
RUNBIT	(--- FL)
SET.FLSH	(---)
SET.INV	(---)
SOFT.EMIT	(C ---)
SPEED	(--- A)
SQUEAK	(PITCH TIME ---)
START.CURSOR	(---)
STOP.CURSOR	(---)
UNSET.FLSH	(---)
UNSET.INV	(---)
UPDISP	(N ---)
^CURSOR	(--- A)

12.9.2 Formatted Output Words

.	(N ---)
."	(---)
.R	(N1 N2 ---)
D.R	(DN N ---)
F.	(FP NDIGITS --- N)
FLAME.ON	(---)
SMART-POSN	(N ---)
SPACE	(---)
SPACES	(N ---)
TYPE	(A L ---)
TYPEDROP	(X A L ---)

12.9.3 Blip Words

2SHFTBLIP	(--- MASK)
ALRMBLIP	(--- N)
ANDBLIP	(N ---)
BATBLIP	(--- MASK)
LOCKBLIP	(--- MASK)
MEMBLIP	(--- MASK)
ORBLIP	(N ---)
RECBLIP	(--- MASK)
RUNBLIP	(--- MASK)
SHFTBLIP	(--- MASK)

12.9.4 ASCII Constants

&CR	(--- C)
&ESC	(--- C)

CHAPTER 13: THE VIRTUAL FILE SYSTEM

The **virtual file system** is a part of the nucleus that stores data in RAM. From a software point of view, it is a random access file system; thus the name, "virtual file system."

The virtual file system is a convenient way of storing small amounts of data, including executable programs. Because of the HHC's low-drain CMOS RAM, it can preserve data when the HHC is "turned off." It is available to every HHC user without the purchase of any peripherals, although a minimal HHC can offer only about 1.1K of storage.

13.1 VIRTUAL FILE CONCEPTS

The virtual file system can store any number of files. A file may contain up to 16K characters (counting storage-format overhead) and an unlimited number of records, subject to the limit of the amount of memory that is available. A record may be any length from 0 to 255 characters. (The 16K limit is imposed by the size of a Programmable Memory Peripheral, not by the file system's data structure.)

Each file is identified externally by a name, which may be from 1 to 255 characters long. The name may consist of any combination of characters. (Applications should impose reasonable requirements for a well-formed file name.)

Each file begins with a **header** which contains the file name and various housekeeping information such as the length of the file. This is followed by the data held in the file.

All files are random-access. If records are defined, they are numbered sequentially with origin 0. Any record may be read or written by number.

Since files are stored contiguously in RAM, writing to a file generally causes other files (and other records in the same file) to move up or down in RAM. Because of this, you should never assume that a record is in the same place from one read or write to the next. Always do a new I/O operation to get the record's current address. For the same reason, never use an absolute address as a pointer to data in a file (e.g., from an index elsewhere in the file); use a record number, or some other invariant record identifier, and an offset from the beginning of the record to the data.

13.2 FILE TYPES

Virtual files may have certain **file types** which describe their contents in various ways. A file may have any combination of types, or it may have none.

A file with the **invisible** type is not shown on the file system editor's menu. The HHC software may use it, since files are identified by name in calls to the virtual file handler (see OPEN, below), but the user cannot get at the file, since it has no menu number. The CLOCK/CONTROLLER's calendar file is one example of an invisible file.

The second type a file can have is the **executable** type. An executable file contains machine code and/or SNAP code. It is not divided into records.

The third type a file can have is the **text** type. A text file is divided into records (lines), each of which is 0- 255 characters long. (Details of file format are given below.) The lines are generally expected to consist of ASCII text. The file system editor permits the user to edit a text file; it does not permit him to edit any other kind of file, except for record #-1 (which contains the file name).

Applications may define other file types for their own use. Examples are MICROSOFT BASIC files and the capsule image files produced by the BURN function of SnapBASIC. File types are used as quick identifiers for determining a file's relevance to an application program (see GET-TYPE).

13.3 VIRTUAL FILE OPERATIONS

The HHC allows I/O to only one virtual file at a time. At any time, the file that virtual I/O words will access is called the **current file.**

Since all file I/O is random-access and is very fast, this is a detail of the linkage rather than a restriction.

13.3.1 OPEN: Opening an Existing File

To select an existing file as the current file, execute OPEN. The linkage is:

```
adr len OPEN b
```

"adr len" are the address and length of a file name. "b" is TRUE if the file was opened successfully, and FALSE if not.

OPEN fails when the requested file does not exist.

The current file does not remain open when CLEAR is pressed.

13.3.2 MAKE: Creating and Opening a New File

You cannot select a new (previously non-existent) file with OPEN. For that purpose you must use MAKE. The linkage to MAKE is:

```
adr len MAKE b
```

"adr len" are the address and length of a file name. "b" is TRUE if the file was created and selected successfully, and FALSE if not.

Note that there is no harm in OPENing a file immediately after MAKEing it. Thus, it is reasonable to MAKE a file in one part of a program, and OPEN WRITE WRITE WRITE ... in another part, without bothering to test for a brand new file that need not be OPENed.

Note: the HHC gives no protection against MAKEing multiple files with the same name. MAKE fails only when there is not enough room in the file space to create an empty file with the specified name. Therefore, if the existence of a file is in doubt, you should always try to OPEN it before MAKEing it.

Note: Files created by MAKE do not yet have a "file type". Therefore, if you want to create a text file for instance, you must execute:

```
%TEXT GET-TYPE C!
```

after you have made the file.

13.3.3 READ: Reading a Text File

You fetch a record from the current file with READ. The linkage is:

```
adr len recnum READ len' 1      success
adr len recnum READ 0           failure
```

"adr len" are the address and length of a buffer where READ is to deposit the record. "recnum" is the number of the record being requested.

If READ is successful, "len'" is the length of record as it was deposited in the buffer. If the record is no longer than "len," then the whole record is moved to "adr," and "len'" is the length of the record. If the record is longer than "len," then it is

truncated to length "len," and "len'" equals "len." You get no warning that the record has been truncated.

READ fails if record "recnum" does not exist.

13.3.4 WRITE: Replacing an Existing Record In a Text File

You replace an existing record in the current file with WRITE. The linkage is:

```
adr len recnum WRITE b
```

"adr len" are the address and length of the record to be written. (***Note***: it may be any length up to 255 characters; it may be longer than the record it replaces.) "recnum" is the number of the record to be replaced.

If WRITE is successful, "b" is TRUE ; if not, "b" is FALSE.

WRITE fails if record "recnum" does not exist, or if the new record is longer than the old one and there is not enough room in the virtual file space to hold it.

Note: If recnum is less than -1, it is treated by READ and write as if it is -1.

13.3.5 INSERT: Inserting a New Record In a Text File

You insert a new record in the current file with INSERT. The linkage is:

```
adr len recnum INSERT b
```

"adr len" are the address and length of the record to be inserted. It may be any length up to 255 characters. "recnum" is the number of the record ***before*** which the new record should be inserted.

The inserted record becomes record number "recnum;" the record that was previously "recnum" becomes "recnum + 1," and so forth.

If record "recnum" does not exist, INSERT inserts the new record at the end of the file when recnum is positive. If recnum is less than -1, the new record is inserted after record -1.

INSERT fails if there in not enough room in the virtual file space to insert the record.

If INSERT is successful, "b" is TRUE; if not, "b" is FALSE.

13.3.6 REC-CNT and INSERT: Appending a New Record To a Text File

The word REC-CNT returns the number of records in the current file:

```
REC-CNT n
```

If the current file is not a text file, "n" will be set to 1.

The best way to insert a record at the end of a file is to use the number of records (which is 1 greater than the record number of the last record) as "recnum:"

```
adr len REC-CNT INSERT b
```

Recall that any non-existant record number will add a record at the end of a file. Using REC-CNT guarantees a non-existant record number no matter what the size of the file is, and also makes the code's intention clear.

13.3.7 DELETE: Deleting a Record From a Text File

You delete a record from the current file with DELETE. The linkage is:

```
recnum DELETE b
```

"recnum" is the number of the record to be deleted. The following record (formerly number "recnum + 1") becomes number "recnum," the former "recnum + 2" becomes "recnum + 1;" and so forth.

If DELETE is successful, "b" is TRUE; if not, "b" is FALSE.

DELETE fails if record "recnum" does not exist.

13.3.8 REVISE: Allocating Space in a File

REVISE is used to allocate space in a file. The format of REVISE is:

```
adr len REVISE b
```

"adr" is the address where the data is to go. "len" is amount of data you want to make space for, in bytes. "b" indicates success or failure of the operation. REVISE fails when there is not enough space to allocate the space.

If REVISE succeeds, you can place data in the file with MOVE or any other appropriate method.

CAUTION: REVISE should not be used to insert or delete data within a text file record since it does not modify the record's length count.

13.3.9 More REVISE: Adding and Deleting Space In a Non-Text File

You can use REVISE to expand a non-text file. This is because the actual function of REVISE is to open up a "hole" in a file by moving everything after the "hole" a specified number of bytes upward and updating all affected pointers.

To use REVISE in this manner, you only need to change the value of "adr" so that it points at the spot where you want to open up a hole.

To allocate space in a new file (which initially contains zero bytes of data), use the ceiling of the file header as "adr." (See the file format description below.)

To delete part of a non-text file, simply give REVISE a negative value for "len." It will delete "len" bytes from the file, beginning at "adr."

Caution: REVISE does no validity checking on the values of "adr" and "len." If you give it invalid values, it may destroy the structure of the file space, or even destroy data that is outside the file space.

Caution: bear in mind that the location of one file can change when another file is made longer or shorter. If you are working with several files, ***always*** open each file with OPEN each time you manipulate it. Do not assume that an address obtained the last time it was open is still valid.

13.3.10 CFILE and DELETE-FILE: Deleting a File

You delete a file with DELETE-FILE. The linkage is:

```
adr DELETE-FILE
```

"adr" is the address of the first byte of the file.

You can fetch the address of the current file with CFILE:

```
CFILE adr
```

Thus, to delete a file, you should (1) open it (making it the current file), (2) execute CFILE to get its address, AND (3) execute DELETE-FILE to delete it.

13.3.11 Renaming a File

The file system stores the name of each file before the file's data. The format of the file name is the same as the format of a record. In fact, SNAP treats the file name as though it were a record with record number -1. You can change the name of the current file by changing the contents of "record -1," from the file system editor or from a SNAP program.

For example, here is some SNAP code that changes the name of the current file to the value of a string named ITS-NEW- NAME:

```
ITS-NEW-NAME COUNT -1 WRITE
```

This technique works even with non-text files, which have no true records.

13.3.12 GET-TYPE: Testing and Setting File Type

You can test or set the type of the current file by fetching or storing a byte at the address given by GET-TYPE. The linkage is:

```
GET-TYPE adr
```

The byte stored at "adr" consists of bit flags for the various file attributes you can set. You can set any combination of attributes, although some, like "executable" with "text", are not meaningful. The bit flags are represented by the following symbolic constants:

%BASIC

For a Microsoft Basic program file. (Basic source programs are stored in tokenized form rather than as pure ASCII text.) Note: this symbolic constant is not defined in the SnapFORTH capsule. If you need it, you must define it yourself. Give it the value 10H.

%EXECUTE

For "executable." The file contains and executable SNAP program.

%TEXT

For "text." The file contains text. This is the only type of file that the file system editor will operate on (except for the renaming operation; see below). Notice that %TEXT and %EXECUTE are logically exclusive.

%INVISIBLE

Files having this attribute are not shown in the file system menu.

For example, to make the current file invisible, execute:

```
%INVISIBLE GET-TYPE C!
```

You can OR these symbols together to set any combination of types that makes sense:

```
%INVISIBLE %EXECUTE OR GET-TYPE C!
```

You can OR them into the value of GET-TYPE to add a new type to those already set:

```
GET-TYPE C@   %TEXT OR   GET-TYPE C!
```

Executing '0 GET-TYPE C!' resets ALL the types; that is, it makes the current file non-invisible, non-temporary, non-executable, and non-text.

13.3.13 LOOKUP: Look Up the Nth File

You can get the name of the n+1'th file in the current file space with LOOKUP. The linkage is:

```
n LOOKUP  [adr]  b
```

where "n" is the sequence number of the desired file. If such a file exists, "adr" is the address of the file, and "b" is TRUE. If no such file exists, "adr" is absent and "b" is FALSE. The first file is file number 0.

13.4 THE FILE SYSTEM EDITOR

An application program can call the file system editor by executing EDIT-FILE. This lets the user of your application edit a file without leaving the application, returning to the application when done.

The linkage to EDIT-FILE is fully described in the glossary.

Note that when you call EDIT-FILE you can specify your own end-of-line and end-of-edit characters. This lets you modify the behavior of EDIT-FILE to fit situations where ENTER as an end-of-line character would not make sense.

13.5 THE VIRTUAL FILE SPACE

There are two virtual file spaces: intrinsic and extrinsic.

The **intrinsic virtual file space** is located between approximately 380H and 1FFFH. The address of the end of the file space may by 7FFH, 0FFFH, 17FFH or 1FFFH depending on the model of the HHC.

Virtual files grow up from the bottom of the intrinsic virtual file space; the temporary stack, which shares the space, grows down from the top. When the two meet, both are considered full. Thus the amount of space available for virtual files is a function of (1) the amount of intrinsic RAM the HHC has, and (2) the amount of RAM consumed by the temporary stack at any moment.

Typically the temporary stack takes about 0.25K. This leaves about a minimum of 1.25K for virtual files. Additional RAM does nothing to increase software's demands on the temporary stack, so in most cases it will increase the space available for virtual files by the amount of RAM added.

The **extrinsic virtual file space** is whatever space is available in a Programmable Memory Peripheral. This space is not shared by any other part of the system, and so its size is equal to the size of the Programmable Memory Peripheral.

Note that more than one Programmable Memory Peripheral may be plugged in to the HHC at a time, but only one may be used at a time. This is because all of the Programmable Memory Peripherals must be bank-switched into the same address space. This means that it is rather difficult to use a Programmable Memory Peripheral as ordinary RAM. (Such use is not recommended, since it interferes with the file system—and vice versa.)

13.5.1 Structure of Virtual File Space

In general, it is good practice to manipulate virtual files only through the words that the file system supplies for that purpose. In some cases, however, applications must examine or even modify the virtual file space directly. The following information on the structure of the virtual file space is for use in those cases.

The intrinsic and extrinsic virtual file spaces each begin with a header in the following format:

bytes	*contents*
0-3	Hexadecimal A5A5A5A5. The file system checks these bytes frequently; if they contain any other value, the file system assumes that the HHC has just been cold-started, and initializes the file space. ***This deletes all files.***
4-5	Total length of file space (counting from the first 'A5H' to the last byte available for files). At present this field is maintained and used only in the extrinsic file space, but it is present in the intrinsic file space too, and may be used in the future.
6-7	Amount of file space used, ***i.e.***, offset of the first free byte from the first 'A5H'.
8-9	Number of files in the file space.
10+	Virtual files.

Virtual files are stored contiguously in the order in which they were created. Inside the file system, the first file is identified as #0, the second as #1, etc. (Externally, files are identified only by name.)

Format of a Text File

bytes	*contents*
0-1	Length of this file in bytes (including this field). Words which refer to "the address of a file" refer to the address of this field.
2	Type code (set by GET-TYPE C!).
3	Length of file name (not including length byte itself).
4-xx	File name (1 to 255 characters long).
xx+1+	Records. Each record consists of a length byte giving the length of the record (not including the length byte itself), followed by the data. Note that the first data record is record #0. The file name may be treated as "record -1." You can rename a file by changing the contents of record -1—even for a non-text file, which has no true records.

Format of a Non-Text File

A non-text file contains data, machine code and/or SNAP code. It has no record structure after record #-1 (the file name); the rest of the file's contents are totally arbitrary.

bytes	*contents*
0-1	Length of this file in bytes (including this field).
2	Type code (set wih GET-TYPE).
3	Length of file name.
4-xx	File name (1 to 255 characters long). Note that executable and non-executable files have the same structure up to this point.
xx+1+	Data.

For an executable file, the format is very similar to that of a ROM capsule, except that there is no header containing address of tag table, copyright notice, etc.

The first item in the format is the extrinsic tag table for this file. Its length is arbitrary (up to the maximum imposed by the design of the tag concept). The first entry in the table ***must*** be file's entry point; the nucleus executes the program in the file by FLEEing to that tag. Following entries define the subordinate words in the file.

Following the tag table is the SNAP code defining the program. The SNAP code can drop into low-level code by using the technique described in the chapter "General Technical Information." ***But note*** that since a file never has a fixed location in the address space, low level code in an executable file may not use absolute addressing to refer to itself.

Non-executable files can have any format whatsoever. It is completely up to the user.

13.6 VIRTUAL FILE SYSTEM WORDS

%BASIC	(--- FL)
%EXECUTE	(--- N)
%INVISIBLE	(--- N)
%TEXT	(--- N)
&CFILE	(--- A)
&EXTRINSIC	(--- A)
'FILEORG	(--- N)
(FILEORG)	(--- A)
?ENOUGH-ROOM	(B ---)
?FILE	(A L --- [A'] B)
?ROOM	(N --- B)
?SF	(A --- B)
AVAIL	(--- N)
CFILE	(--- A)
DELETE	(N --- B)
DELETE-FILE	(A ---)
FILELEN	(--- LEN)
FILEORG	(--- A)
FSPACE	(--- A)
GET-TYPE	(--- A)
LOOKUP	(N --- [A] B)
MAKE	(A L --- B)
NFILES	(--- A)
OPEN	(A L --- B)
OPEN-FILE	(N ---)

READ	(A L N --- [L'] B)
REC-CNT	(--- N)
REVISE	(A N --- B)
SNAP-FILE	(---)
TP	(--- ADR)
USED	(--- A)

CHAPTER 14: PERIPHERAL I/O AND TIMER SERVICES

14.1 EVENT CONTROL BLOCKS

14.1.1 Introducing Asynchronous Processing

Many I/O operations are inherently time-dependent. After starting an operation, a computer must wait until the operation is complete before it can start another, related operation.

Simply halting the computer while an I/O operation is in progress is a simple way to deal with time dependency. Most current microcomputer systems work this way; but the technique is inefficient, and it often gives the computer a maddening tendency to miss keystrokes, etc., while waiting for an I/O operation on a peripheral. It also keeps the CPU running in a loop while it waits for the I/O operation to complete. This is insignificant in a machine that runs on line power, but is undesirable on the HHC, which tries to conserve power by turning the CPU off whenever possible.

The HHC uses a more sophisticated technique. While the it is waiting for an I/O operation to complete, it continues processing the application program that started the operation. The application program may even initiate more I/O operations on other devices. When an I/O operation completes, it generates an **interrupt**, that is, a hardware signal that causes the HHC to suspend what it is doing and give the I/O operation whatever attention it needs. (This is called **servicing the interrupt**.) Then the HHC resumes processing the application program. The interrupt is transparent to the application program, and that program's author never needs to be concerned with it.

There is no way to predict what part of an application will be executing when an interrupt occurs. Thus I/O operations and application-program execution proceed in parallel, but cannot be synchronized. This arrangement is called **asynchronous processing**; the I/O operation (or any other time-dependent function that is handled the same way) is called an **asynchronous process**.

If the idea of asynchronous processing seems confusing, think of a dot moving along a path between two points. The progress of the dot represents the execution of an application program, moving from its initiation to its conclusion.

At some point the dot splits in two, and one part proceeds on its course while the other goes off in another direction. This represents the application program starting an asynchronous process (e.g., an I/O operation) and then continuing to execute while the I/O operation is under way.

At some later point the dots combine again, and proceed toward their goal as a single unit. This represents the conclusion of the I/O operation.

14.1.2. The Event Control Block: Keeping Processes In Step

Although I/O on the HHC is asynchronous, there has to be some way to keep different processes in step. For example, suppose your program starts an input operation. Sooner or later it reaches a point where it needs the information that the input operation was meant to procure. At that point, the program must be able to tell whether the input operation is complete, and if not, wait until it completes.

The mechanism that the HHC uses to keep processes in step is the **event control block** (ECB).

When an application program wants to do I/O, it executes an appropriate SNAP word to start the I/O operation, and it passes an ECB to the HHC's operating system as a parameter of that word. The ECB often contains information about the operation to be performed; in an output operation, for example, it contains a character to be output. While the I/O operation is under way, the operating system keeps track of the ECB's associated with it; we say that the ECB is **queued**.

The operating system starts the requested operation and turns off a **wait bit** (also called a **traffic bit**) in the ECB. Then it returns control to the application program. The program can continue processing up to the point where it must wait for the requested operation to finish. Then it executes the WAIT word, passing the same ECB to the operating system as a parameter.

If the requested operation completes before the application program does its WAIT, it causes an interrupt; the HHC services the interrupt and turns on the wait bit in the associated ECB. (We say that it **posts** the ECB). Then, when the application does its WAIT, the system finds the wait bit on, and so knows that the operation being waited on is already complete; it returns control to the application program immediately.

On the other hand, if the requested operation does not complete before the application program does its WAIT, the WAIT causes the HHC to "go to sleep." Whenever an interrupt occurs, the HHC wakes up, services the interrupt, and checks to see if any process (e.g., the application program) is WAITing on that ECB. If so, it returns control to the process in question; if not, it goes back to sleep.

In this way, the application program that does a WAIT on an ECB is guaranteed that when it gets control back, the operation associated with the ECB will be complete—even though the application has not checked whether the operation is complete already, and has no way of knowing when it will be complete if it presently is not.

14.1.3 Other Uses For ECBs

So far we have discussed ECBs largely in the context of I/O operations. This is the main use for them on the HHC. ECBs can have other functions, though; they are useful wherever one process needs to wait for an asynchronous EVENT to occur. The completion of an I/O operation is one kind of event.

The HHC also uses ECBs to provide timer services. For example, if an application program wants to "sleep" for a specified period of time, it can execute a SET.DELAY or SET.DELAY.LONG, with an ECB containing the desired length of sleep as a parameter. Then it can execute a WAIT on the ECB. It will go to sleep until the specified interval has elapsed. At the end of the interval an event will occur, the operating system will post the ECB, and the application will wake up.

14.1.4 More About ECBs

An ECB has three main parts. They are:

1. The wait bit. This is the first (80's) bit of the first byte.
2. The **return code** field. This comprises the remaining 7 bits of the first byte. When the operating system posts the ECB, it places a code in this field indicating how the operation completed. A zero generally indicates normal completion, and other values indicate various types of errors. More details are given in a table below.
3. Other data. The operation associated with the ECB determines the number of bytes needed, and their functions.

The format of an ECB is:

byte.bit	contents
0.0	Wait bit. 0 => waiting; 1 => event has occurred.
0.1-0.7	Return code. Indicates manner in which the queued operation ended. Codes are listed below.
1+	Operation-dependent information. Details are given in the discussion of each operation.

return code (hex)	meaning
00	Successful operation.
01	Invalid logical device number (not 00-0FH) (I/O only).
02	Logical device number not assigned (I/O only).
03	Invalid device type (I/O only).
04	Device control ROM absent, or its contents are invalid (I/O only).
05	Prior operation still pending for device (I/O only).
06	RAM needed to perform operation is not available.
07-7FH	Additional codes, where defined, are device- specific. See information on individual devices later in this chapter.

14.1.5 Using ECBs

To use an ECB, you first must allocate and initialize it.

An ECB may be allocated in any way, and may be anywhere in intrinsic RAM. An ECB should NOT be located in a Programmable Memory Peripheral, which might be bank-selected out of the address space when the ECB is needed.

We recommend that you not try to share space between ECBs and other program elements; if you make errors in keeping the uses separate, they are likely to be hard to debug.

Also, do not use the same ECB for two or more types of operations, even on the same device. If two operations should happen concurrently on the same ECB, the results are unpredictable (and generally unpleasant).

When you queue an ECB, the system automatically zeroes the wait bit. Thus you need not zero this bit explicitly. (But if the event has already occurred when the ECB is queued, the system will turn the wait bit ***on***! This is logically impossible in I/O operations, but can happen with some other types of asynchronous operations, such as setting timers.)

Execute a WAIT to make your application wait until the event occurs and the ECB is posted; or execute a WAITM to make your application wait until any one of two or more events occurs and the corresponding ECB is posted.

When your program gets back control after a WAIT, it generally should check the return code to determine whether the operation it was WAITing for was terminated normally, and should take appropriate action if not.

Caution: if you accidentally wait on an ECB that was never queued, or on an erroneous location that looks like an unposted ECB, your program will go into an endless wait. The only way to recover will be to press CLEAR.

14.2 INTRODUCTION TO PERIPHERAL I/O

Peripherals may be plugged into the HHC through its external bus socket, or through a I/O adaptor plugged into the bus socket. (The I/O adaptor is technically a peripheral device itself.)

Detailed information about the hardware and software characteristics of each peripheral is given after the next few sections, which describe device-independent aspects of software.

14.2.1 Software: Architecture

Peripheral devices are accessed through the peripheral interface, which provides a consistent set of high level words for opening and closing, input and output, and control.

The high level words call lower-level device routines in standardized locations in ROM. Physically, each device control ROM is part of the device itself. Logically, each ROM is bank-switched into the HHC's address space when the device is used.

14.2.2 Software: In General

The HHC uses three kinds of data items to control I/O operations: hardware device codes, logical unit numbers, and ECBs.

A **hardware device code** is a one-byte value which an application program may use to request access to a particular type of device. Each type of device (or class of types of devices) that can be attached to the HHC has a unique hardware device code.

A **logical unit number** is a number from 0 to 15 which is assigned to a specific device for the time that an application program uses that device. The program uses the logical unit number to identify the device in I/O operations.

An ECB (event control block) is an area of storage that is used to pass information between a program and the device routines that control a device. It is described in more detail above.

ECB's and Asynchronous I/O

Most I/O operations are only ***initiated*** by an operation such as RIP (Request Input, which reads a character) or ROP (Request Output, which writes a character). Once initiated, control returns to the word after the RIP or ROP or whatever. The program can continue executing, if that is logically possible, while the I/O operation is going forward. We say that the I/O operation is **asynchronous** because its progress is not synchronized with the execution of the program that initiated it.

When the I/O operation is complete, the I/O hardware generates an interrupt. This causes the HHC software to suspend execution of your program, process the interrupt, and then allow your program to continue executing as though nothing had happened.

In this context, "processing the interrupt" means that the system determines what I/O operation has completed, and how it has completed (normally, with end-of-file, with an error, etc). It puts an appropriate return code in the operation's ECB and posts the ECB.

When your program needs the results of the I/O operation it has initiated, it must execute a WAIT on the ECB. WAIT checks whether the ECB has been posted yet. If not, WAIT halts execution of your program until the ECB is posted. The linkage to WAIT is:

```
ecb WAIT
```


"ecb" is the address of the ECB you used to initiate the operation.

You determine the outcome of the operation by inspecting the ECB return code. If it is zero, the operation succeeded. If not, the operation failed; the value of the code identifies the reason.

Steps In Doing I/O

To do I/O on a peripheral device, you must (1) connect the peripheral to the HHC's bus and turn it on with the I/O menu, (2) use ATTACH to associate a device with a logical unit number; (3) in some cases, use ROPEN to prepare the device for I/O; (4) use RIP to do input or ROP to do output; and, when done, (5) in some cases, use RCLOSE to close the device.

14.3 PERIPHERAL I/O OPERATIONS

14.3.1 The ATTACH Operation

Your program must ATTACH a peripheral device before using it. This means the program must tell the HHC that it needs a certain type of device, and wants to associate the device with a certain logical unit number.

You attach a device with the ATTACH word. Its linkage is:

```
devcode lun ATTACH b
```

"devcode" is the device type code of the desired device. Valid codes and their meanings are listed under "Hardware Device Codes," below, and may also be found with the table of characteristics of each device.

"lun" is the logical unit number to be attached to this device. Valid numbers are 0 through 15.

"b" is a boolean that is TRUE if the ATTACH operation has succeeded, and FALSE if it has not.

You do ***not*** have to do a WAIT after ATTACH. (Since ATTACH uses no ECB, there is no way you could do one!)

Note: logical unit #0 is the system input unit, normally attached to the keyboard. Unit #1 is the system output unit, normally attached to the LCD. These two logical unit numbers need not be attached before use, although they may be reattached to different devices. Logical units #2 through #15 have no "normal" attachments.

When CLEAR is pressed, logical units #0 and #1 are reset to the keyboard and LCD, respectively. Logical units #2 through #7 are reset to "undefined."

14.3.2 The ATTACHX Operation

If your program needs to attach two devices of the same type, use ATTACHX instead of ATTACH. ATTACHX will not unattach a device that is already attached. If it cannot find a nonattached device, it fails.

If a certain device is already attached to a LUN and you do another ATTACH with the same device type code, ATTACHX attaches the device to the new LUN without detaching it from the old one. This sort of behavior is generally undesirable. For an output device, it causes output from the two LUNs to be mixed together. For an input device, it may cause the new LUN to take the device away from the old one, or it may have no effect.

14.3.3 RIP: Request Input

Once a device is attached, you may initiate a read operation (if reading makes sense for the device) with RIP (Request InPut). Each time you use RIP, you initiate an operation which will read one character from a device.

The linkage to RIP is:

```
ecb lun RIP b
```

"ecb" is the address of the ECB to use for this input operation. The ECB must be at least 2 bytes long.

"lun" is the logical unit number you wish to read from.

"b" is a boolean that is TRUE if the RIP has succeeded, and FALSE if it has not. If the RIP has not succeeded, the ECB return code indicates the cause of the failure.

If RIP succeeds, do a WAIT on the ECB before the input character is needed. If WAIT gives a return code of zero, indicating success, byte 1 of the ECB contains the character read.

14.3.4 ROP: Request Output

Once a device has been attached, you may request a write operation (if writing makes sense) with the ROP (Request OutPut) word. Each time you use ROP, you initiate an operation to write one character to a device.

The linkage to ROP is:

```
ecb lun ROP b
```

"ecb" is the address of an ECB to use for this output operation. The ECB must be at least 2 bytes long. Byte 1 of the ECB should contain the character to be written.

"lun" is the logical unit number you wish to write on.

"b" is a boolean that is TRUE if the ROP has succeeded, and FALSE if it has not. If the ROP has not succeeded, the ECB return code indicates the reason.

If ROP succeeds, do a WAIT on the ECB before initiating the next ROP on the same device.

14.3.5 Relation of RIP and ROP to KEY and EMIT

We have stated that RIP reads from the system input device (normally the keyboard), and the ROP writes to the system output device (normally the LCD). The KEY word also reads from the system input device, and the EMIT word writes to the system output device. What is the relationship between these two sets of words?

The relationship is this: KEY does a RIP with logical unit #0, and EMIT does a ROP with logical unit #1. If these logical units are reassigned, KEY and EMIT will send their output to different devices, just as RIP and ROP will.

The only important difference between KEY/EMIT and RIP/ROP is that KEY and EMIT belong to a larger group of words which do I/O only on logical units #0 and #1, but which support a greater range of operations on those devices than RIP and ROP do. For example, the LOCK-KEY word is similar to KEY, but will return the same character over and over while the LOCK key is in effect.

Outside the context of peripheral I/O, we say that words like KEY read from the keyboard, and words like EMIT write Z to the LCD. #0, which is normally the keyboard, and words like EMIT write to logical unit #1, which is normally the LCD. Keep this in mind as you deal with those words!

14.3.6 Opening and Closing Devices

Opening and closing are two I/O operations which you must perform for some devices.

Opening means preparing the device for I/O, after attaching it but before doing the first RIP or ROP.

Closing means "cleaning up" after the last ROP, *e.g.*, writing out the last record to a record-oriented device, etc.

ROPEN and RCLOSE

A device may be opened and closed with the ROPEN and RCLOSE words, respectively:

```
ecb lun ROPEN b
ecb lun RCLOSE b
```

"ecb" is the address of an ECB to use for this open or close operation. The ECB must be at least one byte long.

"lun" is the logical unit number you wish to open or close.

"b" is a boolean that is TRUE if the open or close has been initiated successfully, and FALSE if it has not. If the operation has not been initiated successfully, the ECB return code indicates the cause of the failure.

Use a WAIT after ROPEN or RCLOSE to halt program execution until the open or close operation is complete.

Functions of ROPEN and RCLOSE

The functions of ROPEN and RCLOSE differ for different devices. They are described in the sections detailing the characteristics of each device, below.

For most HHC peripherals, ROPEN merely allocates workspace from the temporary stack. Any device routine for an HHC peripheral performs this minimal ROPEN automatically when the device is first used (e.g., by RIP or ROP). Thus, an explicit ROPEN is generally not necessary. You may need to do one in some specific situations, e.g., when you want to force the HHC to open the device sooner than it normally would.

Similarly, an explicit RCLOSE is generally not necessary; its functions are performed automatically by CLEAR.

When you write I/O routines that are intended to be device independent, you should not use ROPEN or RCLOSE. Use these words only in device-specific routines, when you are addressing a device that requires opening and/or closing in a particular situation.

14.3.7 RCTL: Control and Status Operations

You can send control information to a device, or read status information from a device, with the RCTL (Request ConTroL) word. Its linkage is:


```
ecb lun RCTL b
```

"ecb" is the address of an ECB to use for this control operation. The length of the ECB is device dependent.

"lun" is the logical unit number you wish to operate on.

"b" is a boolean that is TRUE if the RCTL was successful, and FALSE if not. If the operation was not successful, the ECB return code indicates the cause of the failure.

If the operation is a "read status" type of operation, and completed successfully, bytes from 1 through the end of the ECB contain the information requested.

RCTL should be followed by WAIT.

14.3.8 A Note on Multiple Waits

The HHC can wait on a list of ECBs, as well as on a single ECB. It does this with the WAITM (WAIT Multiple) word. WAITM waits until any one of a list of ECBs is posted, and indicates which ECB was posted.

WAITM is useful for programs which use several devices, and must be able to attend to an asynchronous operation on any of them. For example, consider a modem program, which must be able to read a character from the keyboard and write it to a modem, or read a character from the modem and write it to the LCD. A logical way to set up such a program would be to do a RIP on the keyboard and another RIP on the modem, then do a WAITM on both ECBs. Whichever input device delivered a character first would end the wait and get attention.

14.3.9 The System Device Table (SDT)

The HHC keeps track of LUN attachments through the **system device table** (SDT). The SDT's address is given by the symbol 'SDT'.

The SDT is 16 bytes long. Byte 0 represents LUN #0, byte 1 represents LUN #1, and so forth.

Each byte contains a relative pointer to a six byte control block which describes the device that the LUN is attached to. This block is called the Hardware Device Table, or **HDT**. A pointer of 0 represents the keyboard; 6 represents the LCD; 0CH, 12H, 18H,. . . represent peripherals. A pointer of 0FFH represents an unattached LUN.

You can "unattach" a LUN by storing 0FFH in its SDT entry. Then you can use ATTACH to attach another device to the LUN.

14.4 PARAMETER VALUES

14.4.1 Hardware Device Codes

A hardware device code is a one-byte value. Each valid value defines a type of device or a class of devices that may be attached to the HHC.

Device codes defined for the HHC are:

input code	output code	device
	41H	LCD
81H		keyboard
82H	42H	modem
	43H	video
	44H	micro printer
85H	45H	reserved for expansion
86H	46H	serial interface adaptor
	47H	speech synthesizer (not yet avail.)
88H-8FH	48H-4FH	reserved for expansion

The six low-order bits are used to distinguish device types. The 80's and 40's bit are used to indicate whether on, the device may be used for input; if the 40's bit is on, the device may be used for output.

14.4.2 Logical Unit Numbers

Valid logical unit numbers are 0 through 15.

#0 is the system input device, defaulting to the keyboard.

#1 is the system output device, defaulting to the LCD.

#2 through #15 have no inherent functions, and default to "unassigned."

14.4.3 ECB Return Codes

Following are the codes which are returned in bits 0.1-0.7 of an ECB which is posted after an I/O operation:

Code	Meaning
0	Successful operation.
1	Logical device number invalid (not 0-15).
2	Logical device number not assigned.
3	Invalid operation.
4	Device control ROM absent, or contents invalid.
5	Operation already pending on this device.
6	No RAM workspace available for device.
7-7FH	Device dependent codes; see individual device descriptions.

14.5 OUTLINE OF PERIPHERAL I/O WORDS

Following is a table which summarizes the words that relate to peripheral I/O and their linkages.

linkage: input WORD output	Minimum ECB length; contents after 1st byte
devcode lun ATTACH b	No ECB; no wait used.
devcode lun ATTACHX b	No ECB; no wait used.
ecb lun ROPEN b	1
ecb lun RIP b	2; 1 = character read.
ecb lun ROP b	2; 1 = character to write.
ecb lun RCTL b	Length varies. Byte 1 = control code indicating operation to perform. For a "read status" type operation, bytes 2,3,4... = status information returned by the operation.
ecb lun RCLOSE b	1

14.6 CONTROL CHARACTERS

Control characters may be included in a character string sent to an output device (e.g., the TV adapter) to perform control functions such as cursor movement.

Not all control characters have meaning for all devices. If a control character is sent to a device that does not support it, the character will generally be ignored.

14.6.1 ASCII Control Characters

symbolic constant	hex	meaning
&BSP	08	backspace
&LF	0A	line feed
	0C	form feed
&CR	0D	carriage return
&ESC	1B	escape (starts escape control sequences, below)
&U	80	cursor up
&<	81	cursor left
&>	82	cursor right
&D	83	cursor down

14.6.2 Escape Control Sequences

Escape control sequences provide an extended range of control characters. Each sequence is 3 bytes long:

byte	meaning
0	1B, escape, begins escape control sequence. Use the symbolic equate &ESC to represent this code.
1	opcode.
2	data; meaning depends on opcode.

Each control sequence's opcode is represented by a symbolic equate. These equates and their meanings are given in the following mini-glossary, and in the main glossary. The data byte is ignored in each case except where a use for it is explicitly mentioned.

Note that not all escape control sequences are supported by all devices. Each device ignores escape control sequences that it does not support.

ESCCC
: Set Control Character Mode. If the data byte is TRUE, subsequent nonexecutable control characters sent to the device which the device does not support will be displayed. If FALSE, subsequent control characters sent to the device which the device does not support not be displayed.

ESCDC
: Display Character Absolute. The next data character will be displayed, even

if it is a control character that would normally be executed. For example, if the next character is 0DH (carriage return) and it would normally cause a physical end-of-line on the device, it causes no carriage end-of-line, but is written as an inverse M.

ESCDR
: Delete Right. The character under the cursor is deleted; following characters on the line are moved left. The data byte is used as a fill character at the end of the line.

ESCFL
: Flush I/O Buffer. Characters in the device's I/O buffer are written (if being output) or dumped (if being input), emptying the buffer. (This operation is generally applied only to output devices that write a line of data at a time. Writing a line would normally be triggered by a CR character.)

ESCHM
: Home Cursor. This normally returns the cursor to the upper left corner of a device with a two-dimensional page or display.

ESCIR
: Insert Right. The data byte is displayed at the cursor. The character under the cursor, and all following characters on the line, are pushed to the right.

ESCSF
: Set Flash Mode. Subsequent output will be displayed in flashing characters. Reverses the effect of ESCUF.

ESCSI
: Set Inverse Mode. Subsequent output is displayed in inverse. Reverses the effect of ESCUI.

ESCUF
: Set Unflash Mode. All subsequent output is displayed in non-flashing characters, until the mode is changed by ESCSF. Reverses the effect of ESCSF.

ESCUI
: Set Uninverse Mode. Subsequent output will be displayed normally (non-inverse). Reverses the effect of ESCSI.

ESCUN	"LCD unescape." On the LCD, the data byte is displayed (as in ESCDC). On all other devices, it is ignored.
ESCWB	Set Word Break. The data byte defines the word break character that can trigger automatic word wrap. (This means that when an output line becomes longer than a device's line length, the device automatically starts a new line and moves the last word of the old line to the start of the new line so that the word will not straddle a line break.) The word break character is initially blank (20H) in all cases. Setting the word break character to 0FFH turns automatic word breaking off.

14.6.3 The Keyboard

The keyboard is not actually a peripheral. Since it has no entry in the device type table, it cannot be attached to a logical unit number as other peripherals can. It can be read by LUN #0 (its standard assignment) with RIP and ROP, however.

14.6.4 The LCD

The LCD is not strictly a peripheral, but it may be used in many of the same ways. It obeys many of the standard control codes and escape control sequences. It may be used as a peripheral by writing to logical unit number 1 (which is attached to the LCD by default).

Technical Information

Device code: 41H

ECB length: 2 bytes

Device-specific ECB return codes: none. LCD I/O is performed synchronously; I/O operations always succeed, and post the ECB immediately.

I/O operations:

ROPEN	Unneeded.
RIP	Not applicable.
ROP	Outputs the character stored in byte 1 (origin-0) of the ECB.
RCTL	Not applicable.
RCLOSE	Unneeded.

Control characters used:

07	Makes the HHC beep.
08	Destuctive backpace.
0D	Carriage return; clears display.
1B	Begin escape control sequence.
80	Cursor Up. Displays as '▲'.
81	Cursor Left (non-destructive backspace).
82	Cursor Right (non-destructive space).
83	Cursor Down. Displays as '▼'.

Escape control sequences: the LCD supports all escape control sequences except ESCWB. Note that the data byte of ESCUN is displayed on the LCD (as in ESCDC); it is ignored on all other devices.

14.6.5 The Micro Printer

General Information

The Micro Printer is a the thermal dot matrix printer. It prints 15-character lines in a 1" column on rolls of paper 1.4" wide, at 7 lines/inch. The printer's speed is 50 characters / second. It is powered by 4 "AA" batteries, or the bus.

The Micro Printer prints two lines at a time. It uses a 30-character buffer, which enables it to print a line of data long enough to fill the LCD, with a little bit left over for word wrapping.

The Micro Printer allocates about 50 bytes of working space at the tip of the temporary stack. This space contains the two-line buffer and some data that the printer's device routine uses.

The Micro Printer uses the same standard character set as the LCD. No alternate character sets or graphics are available.

Technical Information

Device code: 44H

ECB length: 2 bytes

Device-specific ECB return codes: none

I/O operations:

ROPEN	Unneeded. If used, allocates the 50-byte workspace that the Micro Printer device routine needs from the temporary stack.
RIP	Not applicable.
ROP	Outputs the character stored in byte 1 (origin- 0) of the ECB.
RCTL	Not applicable.
RCLOSE	Unneeded (unless last line of output ends without a CR). If the printer buffer is not empty, flushes the buffer and simulates a CR.

Control characters used:

- 07 Bell; ignored.
- 08 Backspace. Moves the printer buffer pointer left one character and deletes the last character in the buffer.
- 0A Line feed. The printer treats this code as a no-op. (Carriage return causes an automatic line feed.
- 0C Form feed. Writes and empties the printer buffer and advances the paper 4 lines.
- 0D Carriage return. Makes the printer print an accumulated line of output and do a line feed. (Note that the line feed character is NOT supported.) After the printer has printed the contents of the buffer, the buffer is reset to all-blanks.
- 1B Escape. Marks the beginning of an escape control sequence.
- 80 Cursor Up. Displays as '▲'.
- 81 Cursor Left. Moves the printer buffer pointer left one character. The last character in the buffer is not deleted, but will be overwritten by the next output character.

- 82 Cursor Right. Moves the printer buffer pointer right one character without changing the contents of the buffer.
- 83 Cursor Down. Displays as '▼'.

Escape control sequences: ESCCC, ESCDC, ESCDR, ESCFL, ESCHM, ESCIR, ESCUN, ESCWB.

14.6.6 Video Interface

General Information

The video interface operates a black and white or color television set via an RF modulator, or operates a black and white or color monitor directly. It has 1.5K of internal RAM, which enables it to maintain two alternate screen images, software selectable. The interface draws power from an AC adaptor, not from the HHC bus.

There is a 96-character set including ASCII, Katakana, & graphics characters in 7x9 dot matrix. Lower case characters have true descenders.

The text mode screen format is 16 lines x 32 characters. Text colors are green (initial) or orange (alternate) on black ground. Inverse video displays black on orange or green ground; characters may be flashing or steady.

There are several graphics modes offering different resolutions and capabilities.

The video interface is described in detail in a separate document.

Technical Information

Device code: 43H

For other technical information, see separate document.

14.6.7 Modem

General Information

The modem (actually an acoustic coupler) supports asynchronous communications at 110 and 300 baud (software selectable). It has the following software selectable options: originate/answer, with 5 to 8 data bits, 1 or 2 stop bits, and even/odd/no parity.

Technical Information

Note: Unless stated otherwise, the following information applies to the Telecomputing 2 software.

Device code: input 82H; output 42H

ECB length: 4 bytes for RCTL; 2 bytes for input & output

Device-specific ECB return codes:

07	Input buffer is within 5 characters of overflowing. (Occurs on input operations only). Usual cause: the host is ignoring XOFF. (Telecomputing 1 only.)
08-0E	Interpreted bitwise. If the return code is expressed as 0000 1pfv, p=0 means a parity error occurred. f=0 means a framing error occurred. v=0 means an overrun error occurred.
0F	Input buffer is overflowing.
10	Output buffer overflow.

I/O operations:

ROPEN	Initializes buffers.
RIP	Inputs a character and stores it in byte 1 of the ECB.
ROP	Outputs the character stored in byte 1 of the ECB.
RCTL	Used to perform the following operations: -initializing modem for I/O. -changing status of modem. -querying status of modem. Byte 1 of the ECB determines what operation will be performed. The meaning of bytes 2 & 3 varies from one operation to another. (See details below.)
RCLOSE	No function.

Control characters used:

11H XON (resume-transmission signal in standard XON/XOFF protocol).

Through RCTL, you can specify whether the modem is to observe XON/XOFF protocol for receiving data and/or for transmitting data.

When the modem observes the protocol for reception and/or transmission, it handles the protocol entirely in the device driver; thus the protocol is transparent to the application program.

When the modem ignores the protocol for transmission, it becomes capable of receiving binary codes 11H and 13H as data. The system the HHC is communicating with must be capable of receiving data continuously, since the HHC will treat an XOFF as a data byte.

When the modem ignores the protocol for reception, the system it is communicating with may safely assume that when it receives binary codes 11H and 13H, they are data, not protocol signals. Without a protocol, the HHC must be capable of receiving data continuously, since the device driver will not send an XOFF when the modem's buffer is nearly full.

13H XOFF (cease-transmission signal in standard XON/XOFF protocol). See XON for details.

Escape control sequences: none

Device Initialization

The modem must be initialized before you can begin doing I/O on it. Initialization sets hardware options such as data rate, parity, etc., to values compatible with the host system.

A routine in the device control ROM initializes the modem automatically when the modem is turned on with the I/O key, when the user presses ON, and when a hard clear occurs.

The modem is initialized from a file in the internal RAM of the HHC. This file is created by the telecomputing program when the user configures the program to talk to a particular host. You can re-initialize the modem, if you wish to do so, with an initialization call to RCTL.

Initialization Call

Here is the ECB format for a "device hardware initialization" call to RCTL:

byte	*bit*	*Meaning for call to RCTL*
0		Traffic bit & return code (standard ECB format).
1		Function code; must be set to 1.
2	80	Ignored.
	40	1->use breaktone; 0-> don't.
	20	Data rate. 1-> 300 baud; 0-> 110 baud.
	10-08	Character length in bits. 00-> 5 bits; 01-> 6 bits; 10-> 7 bits; 11-> 8 bits.
	04	1-> suppress parity check; 0->don't.
	02	1-> even parity; 0-> odd parity.
	01	Number of stop bits 1-> 2 bits; 0-> 1 bit.
3	80	0-> reset read; 1-> don't. In normal use, should always be set to 1.
	40	1-> enable input IRQ; 0-> disable. (Must be set to 1 if modem is to detect subsequent input.)
	20	1-> enable input IRQ; 0-> disable.
	10	1-> enable modem receiver (also turns on carrier signal); 0-> disable.
	08	1-> enable modem transmitter; 0-> disable.
	04	1-> set originate mode; 0-> set answer mode.
	02	1-> echo suppressor; 0-> normal.
	01	ignored.

Set XON/XOFF Status

"Set XON/XOFF status" is an optional RCTL operation. Its purpose is to disable the XON/XOFF protocol for reception and/or transmission. A device initialization call always enables the protocol going both ways; if you do not perform a "set device status" operation after a "device initialization" operation, the protocol will simply stay enabled.

Here is the ECB format for a "set XON/XOFF status" call to RCTL:

byte	*bit*	*Meaning for call to RCTL*
0		Traffic bit & return code (standard ECB format).
1		Function code; must be set to 2.
2	80	1-> enable protocol for transmission; 0-> disable.
	40	1-> enable protocol for reception; 0-> disable.
	20-01	Used internally; should always be set to 0.

Get Device Status

"Get device status" returns information about the current hardware status of the modem in the ECB.

Here is the ECB format for a "get device status" call to RCTL:

byte	*bit*	*Meaning for call to RCTL*
0		Traffic bit & return code (standard ECB format). Return code is always 0.
1		Input to call: function code. Must be set to 3. Return from call: protocol status. Indicates whether XON/XOFF protocol is being observed for input and/or output. Bit assignments are the same as for ECB byte 2 of the "set XON/XOFF status" function.
2	80-20	Ignored.
	10	1-> no carrier detected. 0-> carrier detected.
	08	1-> last read's parity OK (or not being checked); 0-> parity error.
	04	1-> last read's framing OK; 0-> framing error.
	02	1-> last read no overrun error; 0-> overrun error.
	01	Ignored.

14.6.8 Serial Interface Adaptor (RS-232 Interface)

General Information

The Serial Interface Adaptor supports full RS-232 protocol for input and output. It is switch-selectable for all standard data rates up to 9,600 baud. Characteristics such as character length and parity may be set through software. The modem's device control ROM has built-in hardware support for XON/XOFF and ETX/ACK protocols.

Technical Information

Device code: input 86H; output 46H

ECB length: 4 bytes for RCTL; 2 bytes for input & output

Device-specific ECB return codes:

07	Input buffer is within 5 characters of overflowing. (Occurs on input operations only). Usual cause: the remote system or device is ignoring transmission protocol.
08-0E	Interpreted bitwise. If the return code is expressed as 1pfv, p=0 means a parity error occured. f=0 means a framing error occured. v=0 means an overrun error occured.
0F	Input buffer overflow.
10	Output buffer overflow.

I/O operations:

ROPEN	Initializes buffers.
RIP	Inputs a character and stores it in byte 1 of the ECB.
ROP	Outputs the character stored in byte 1 of the ECB.
RCTL	Used to initialize the modem for I/O, and to modify and query its status. Byte 1 of the ECB determines what operation will be performed. The meaning of bytes 2 & 3 varies from one operation to another. (See details below.)
RCLOSE	No function.

Control characters used:

11H	XON. Same function as in the modem when XON/XOFF protocol is used. Otherwise treated as a data character. See the description of modem control characters for details.
13H	XOFF. Cease-transmission signal in standard XON/XOFF protocol. See description of modem control characters for details.
03H	ETX. Used in ETX/ACK protocol to mark the end of a transmission. In ETX/ACK protocol, the HHC sends a message known to be short enough for the receiver to accept and process without losing characters, and follows the message with an ETX code. When the device has processed the entire message and is ready for another one, it sends an ACK code. This signals the HHC that it may send another message.
06H	ACK. Used in ETX/ACK protocol to acknowledge a transmission. See 03H, ETX, for details.

Escape control sequences: none

Device Initialization

The Serial Interface Adaptor must be initialized before you can begin doing I/O on it. Initialization sets hardware options such as word length, parity, etc., to values compatible with the peripheral connected to the interface.

A routine in the device control ROM initializes the adaptor automatically when it is turned on with the I/O key.

Unlike the modem initialization file, the Serial Interface Adaptor initialization file can contain a separate set of initialization data for the bus socket on the HHC (slot #0) and for each socket in the I/O adaptor (slots #1 through #6). Thus, the user can change the interface's default configuration just by plugging it into a different slot.

There are four types of calls to RCTL for the modem:

1. Hardware initialization: sets interface characteristics such as number of data bits and parity.
2. POKE: sets a value in the interface's RAM workspace.

3. PEEK: retrieves a value from the interface's RAM workspace.
4. STATUS: retrieves data about the interface's current hardware state (e.g., carrier detect).

Initialization

Here is the ECB format for an "initialization" call to RCTL:

byte	*bit*	*Meaning for call to RCTL*
0	–	Unused.
1	–	Function code (=1).
2	80	1->device power on, 0->off.
	40	1->use breaktone; 0-> don't.
	20	unused.
	10-08	Character length in bits. 00-> 5 bits; 01-> 6 bits; 10-> 7 bits; 11-> 8 bits.
	04	1->suppress parity check; 0->don't.
	02	1->even parity; 0->odd.
	01	Number of stop bits; 1-> 2 bits; 0-> 1 bit.
3	80	0->reset read; 1->don't. In normal use, should always be set to 1.
	40	1-> enable input IRQ; 0-> disable. (Must be set to 1 if serial interface is to detect subsequent input.)
	20	1-> enable output IRQ; 0-> disable.
	10	1-> DTR signal on; 0-> off.
	08-01	ignored.

byte	*bit*	*Meaning for return from RCTL*
0	–	Traffic bit & return code.
1-3	–	Unchanged.

POKE Call

byte	*bit*	*Meaning for call to RCTL*
0	–	Unused.
1	–	Function code (=2).
2	–	Offset of field to be POKEd. See the description of PEEK for useful offsets.
3	–	Value to POKE.

byte	*bit*	*Meaning for return from RCTL*
0	–	Traffic bit & return code.
1-3	–	Unchanged.

PEEK Call

byte	*bit*	*Meaning for call to RCTL*
0	–	Unused.
1	–	Function code (=3).
2	–	Offset of field to be PEEKed at. See table of useful values below.
3	–	Unused.

byte	*bit*	*Meaning for return from RCTL*
0	–	Traffic bit & return code.
1-2	–	PEEKed value.
3	–	Unchanged.

Useful Work Area Offsets With PEEK and POKE

Peeks and pokes can be tricky, since many fields in the PCA are maintained by asynchronous routines. Only fields that are safe to peek are listed below. Fields that are also safe to poke are noted explicitly.

offset	meaning
2	Number of characters in the input buffer.
5	Number of output characters pending.
62H	Software option byte. Safe to poke. The meanings of the bits are: 80:1-> enable XON/XOFF for transmission; 0-> disable. 40:1-> enable XON/XOFF for reception; 0-> disable. 20:1-> enable ETX/ACK. 10:1-> auto release from RXOFF state. 08:unused. 04:1-> Auto line feed; when CR is written to the interface, the interface transmits CR/LF to the peripheral. 02:1-> Let escape control sequences go through the interface; 0-> filter them out. 01:1-> zero first data bit in each byte; 0-> transmit it as-is.
63H	ETX message length. Safe to poke.
64H	XON/XOFF flags. 40:An XOFF has been sent. 20:An XOFF has been received, or an ETX has been sent; output is delayed. Other bits may be changed asynchronously by the device control ROM, and should be ignored.

Status Call

byte	*Meaning for call to RCTL*
0	Unused.
1	Function code (=4).
2-3	Unused.

byte	*Meaning for return from RCTL*
0	Traffic bit & return code.
1	Status byte. Meanings of bits are: 80:1->IRQ asserted, 0->not. 40:1->UART data read, 0->not. 20:unused 10:1->carrier detect, 0->no carrier. 08:0->parity err detected, 1->no err. 04:0->framing err detected, 1->no err. 02:0->overrun err detected, 1->no err. 01:unused
2-3	Unused.

14.7 TIMER SERVICES

14.7.1 The Real-time Clock

The HHC's basic clock facilities are a hardware counter that counts down continuously whether the HHC is on or not, and a software **time of century clock** which stores a binary representation of the elapsed time since 1 January 1980 in a 5 byte field. The time of century clock may be used to display the current date and time.

Application programs may define timers which will generate an event at a specified time. The HHC maintains a queue of ECBs representing the time of century clock and any timers which the application has defined.

The basic timing units are seconds and **clock ticks**. One clock tick equals 1/256 second.

The hardware counter is set to count down from its maximum interval or the least interval remaining in any timer, whichever is less. When it reaches zero it generates an interrupt, allowing the CPU to update all the timer ECB's by processing each entry in the timer queue.

14.7.2 Using Timer Facilities

The HHC provides a number of words for using the timer facility and the time of century clock. These words and their functions are identified below. Their use is described in the "Glossaries" section at the end of this book.

To get the approximate time of century, use GET.GMT (for Greenwich Mean Time) or GET.LOCAL (for local time) To get the exact time of century (Greenwich Mean Time only), use GMTF.

To create a timer, allocate an ECB 6 bytes long. Set the timer interval and queue the ECB with SET.DELAY or SET.DELAY.LONG; or set the time when the timer is to go off with PACKUP (*see* below) and queue the ECB with SET.TIMER.

To cancel a timer and dequeue its ECB, use CANCEL.TIMER.

To wait for a timer to go off, use WAIT or WAITM on the timer's ECB; or use SECS to put the HHC to sleep for a specified number of seconds.

Note: all timer ECB's except those belonging to the CLOCK/CONTROLLER are cancelled and posted when CLEAR is pressed, or when the task that started them finishes executing.

14.7.3 The Auto-Off Facility

The HHC's **auto-off facility** conserves battery power by turning the HHC off automatically when no keystroke has been entered for a period of between 10 and 20 minutes.

The auto-off facility uses a control bit and a 10-minute timer. Every time a keystroke is entered, the bit is turned on. Every 30 seconds, the HHC checks whether the bit is on; if on, it resets the timer and turns the bit off. If the bit is still off the next time the timer expires, the HHC turns itself off.

This is fine for most kinds of applications, but occasionally there is a legitimate need to run the HHC for long periods of time without any keystrokes being entered. One example is a

telecommunications program, which may transmit or receive a long file without keyboard intervention.

For these situations, your program can turn the "keystroke received" bit on. Turn the bit on like this:

```
FLAG3 AOTOKE SET.BITS
```

In a few rare situations you may need to disable the auto-off facility completely. You can do so like this:

```
FLAG3 AOENB CLR.BITS
```

To re-enable the auto-off facility:

```
FLAG3 AOENB SET.BITS
```

14.8 I/O AND TIMER SERVICES WORDS

14.8.1 I/O Words

ATTACH	(DEVCODE LUN --- B)
ATTACHX	(DEVCODE LUN --- B)
EMIT.ESC	(DATACHAR CTRLCHAR ---)
ESCxx	(--- N)

(Includes any of the following: ESCCC, ESCDC, ESCDR, ESCFL, ESCHM, ESCIR, ESCSF, ESCSI, ESCUF, ESCUI, ESCUN, ESCWB.)

RCLOSE	(ECBADR LUN --- B)
RCTL	(ECBADR LUN --- B)
RIP	(ECBADR LUN --- B)
ROP	(ECBADR LUN --- B)
ROPEN	(ECBADR LUN --- B)
WAIT	(ECBADR ---)
WAITM	(ECBADR[N-1] ... ECBADR1 ECBADR0 N --- N')

14.8.2 Timer Services Words

AOECB	(--- A)
AOENB	(--- FL)
AOTOKE	(--- FL)
GET.GMT	(--- DSEC TICKS)
GET.LOCAL	(--- DSEC)
GMT	(--- A)
GMTF	(--- A)
NAP	(TICKS SEC --- [C] B)
POZECB	(--- A)
SECS	(N ---)
SET.DELAY	(TICKS ECBADR ---)
SET.DELAY.LONG	(TICKS SEC ECBADR ---)
SET.TIMER	(ECBADR ---)

CHAPTER 15: FLOATING POINT OPERATIONS

15.1 INTERNAL STORAGE

The format used to represent a floating point number is:

byte.bit	*contents*
0.0	Sign bit; 0 -> positive, 1 -> negative.
0.1-1.3	Exponent. Stored in binary format with radix- 10 biased representation.
1.4-7.7	Mantissa. Stored in binary coded decimal (BCD), digit-normalized form. The mantissa's implied decimal point is between the first and second decimal digits.

Byte 0 is oriented toward low memory. On the parameter stack, therefore, bytes 0 and 1 are on top.

The exponent represents values from 10^{-1024} to 10^{1023}

The mantissa allows 13 decimal digits of significance. Since all arithmetic operations truncate the result, however, the last digit is more properly viewed as a guard digit on a mantissa with 12 decimal digits of significance.

The mantissa's packed-decimal format saves substantial amounts of CPU time during internal/external conversions, and also makes rounding errors more comprehensible to users used to decimal arithmetic. Since the 6502's packed-BCD arithmetic is about as fast as its binary arithmetic, there is no speed penalty.

15.2 ERROR HANDLING AND SPECIAL CASES

When the HHC executes floating point words, a variety of nasty things can happen. The software flags all error conditions by setting bits 1.4-1.7 (the first decimal digit of the mantissa) to zero, and bits 0.1-0.7 to a code indicating the type of error that has occurred. A floating point number set in this way is called a **special case**.

Thus, an erroneous operation is indicated by the ***value*** of the result. This causes an interesting quirk in terminology; the names of error conditions are also the names of the corresponding special cases. For example, an operation that

raises in a zero divide condition returns the "zero divide" special case as its result, and this value may be stored in a FP variable, from which it may be displayed or used in further operations.

Possible values of the error code are:

value	*meaning*
0	**Zero**. The numeric value 0.0. Arithmetic operations which produce 0 always set the sign bit to 0 (positive). The user may push a negative 0 on the stack if he wishes; the software will treat it the same as a positive 0. The ASCII representation of ZERO is '0'.
1	**Underflow**; $10^{-1024} > \text{abs (result)} > 0$. An underflow converted to ASCII is represented as 'U'.
2	**Overflow**; $\text{abs (result)} > 10^{1023}$. An overflow converted to ASCII is represented as 'V'.
3	**Complex number**; occurs when an attempt is made to take the square root of a negative number, or when a complex number is used in a calculation. A complex number converted to ASCII is represented as 'i'.
4	**Zero divide**; occurs when a number, overflow, or underflow is divided by a 0 or by a zero divide. A zero divide converted to ASCII is represented as 'Z'.
5	**Enigma**; any undecidable result not covered above, ***e.g.***, the result of converting your name to a floating point number. An enigma converted to ASCII is represented as '?'.
6-7F	**Undefined**. Application programs may use these values to represent their own "special cases." Any of these codes converted to ASCII is represented as '?'.

Operations on special cases and valid floating point numbers generally obey the rule that ***the type of the result is the same as the type of the higher-valued special case operand.*** The rationale is to propagate the "worst" error through any chain of floating point operations. Exceptions to the rule are:

1. Operations resulting in overflow or underflow, which produce a special case although both operands are numbers.
2. Division by zero, which produces a zero divide (code 4) although the higher-valued operand is zero (code 0).
3. Taking an even root of a negative number, which produces a complex number although both operands are numbers.
4. Adding/subtracting a valid number and zero: the result is a valid number although zero is a special case.

15.2.1 Defining New Special Case Types

You can use special case codes 6 to 0FFH to represent your own special cases.

The word FSTANDTYPE, which extracts the special case code from a floating point number, reports any of these codes as an enigma (code 5). All other floating point operations process them according to the standard rules given above.

15.2.2 Some Missing Operations

Due to space restrictions in the HHC's ROM, several rather basic functions are absent from the floating point vocabulary. Following are suggested substitutes for these absent words.

Test for equality: 'F=' word, which might be expected to perform this function, is absent. The only floating point comparison word on the HHC is 'F<'. 'F=' may be defined like this:

```
: F=    ( FP1 FP2 --- B )
  F-  >R DROP 2DROP R>    0= ;
```

This word will produce the correct result for any combination of values except two non-zero special cases of the same type, which it will report to be unequal.

Setting a special case: if you have a need to define your own special cases (with codes 6 through 07FH), you will need a way to turn a floating point number into a special case. The following word will do so:

```
HEX
: FP>SC    ( FP N --- FP' )
  7F AND SWAP    80 AND    OR ;
\ FP = a floating point number;
\ N = desired special case code;
\ FP' = FP turned into the special case
DECIMAL
```

Or, if you have no need to preserve the sign and the last 6 bytes of the mantissa, drop the floating point number and stack a special case like this:

```
: FSC    ( N ---- FP )
  7F AND >R 0 0 0 R> ;
\ N is the special case code desired;
\ FP is the special case value.
```

15.3 FLOATING POINT OPERATIONS WITHOUT SPECIAL CASE CHECKING

The HHC's software includes three words that do floating point addition, multiplication and division without checking for special cases. These words are about 4 times faster than the corresponding words that do check for special cases. When you know an operation will not result in a special case, you can safely improve your program's performance by using the non-checking words.

The non-checking words are not in the HHC's tag table. If you want to use them, your program must define words that locate them by referring to near-by words that are in the tag table. (You cannot safely code a non-checking word's address into your program, since the address may change from one version of the HHC to another. A change in the address of an intrinsic word does not force application programs to be recompiled, so long as the tag table does not change.)

Following is the code you need to call the non-checking floating point words:

15.3.1 Calling the Non-Checking Version of F+

```
HEX
 CODEC (F+)
   'X FNEGATE >< FF AND 2*
   IDICT SHINT 2* + + DUP
                 LDA,
     1+          LDY,
   LABEL (FIETS)
   N  10 +     STA,
               DEY,
   N  11 +     STY,
   FE       #  LDY,
```



```
   N  10 + )Y LDA,
   N  12 +    STA,
             INY,
   N  10 + )Y LDA,
   N  13 +    STA,

   N  12 + )  JMP,
```

Note that no (F-) instruction exists. To subtract, use '80H XOR (F+)'.

15.3.2 Calling the Non-Checking Versions of F* and F/

```
 HEX
   CODEC (F*) (S FP1 FP2 ---- FP1*FP2)
     'X F/ >< FF AND 2*
     IDICT SHINT 2* + + DUP
                LDA,
       1+       LDY,
     (FIETS)    BNE,       \ Always branches

   CODEC (F/)
     'X FROUND >< FF AND 2*
     IDICT SHINT 2* + + DUP
     LDA,
     1+ LDY,
     (FIETS) BNE,          \ Always branches

   CODEC LEGAL?      (S FP ---- FP BOOL)
     DEX,                 \ Put Boolean on stack
     DEX,
     XSAVE STX,
     0 # LDA,
     TOP 1+ STA,
        1 # LDA,
        TOP STA,
     SEC 1+ LDA,
     F # AND,
     0= IF,
        SEC LDA,
     0= NOT IF,          \Error
        0 # LDA,
        TOP STA,        \ Set false
        NEXT JMP,
     THEN,
        0= IF,
        8 # LDY,
        BEGIN,
          2 ,X STA,    \ Stores 8 byte zero
```

```
        INX,
        DEY,
     0= UNTIL,
     XSAVE LDX,
  THEN,
  NEXT JMP,
```

?LEGAL should be executed after an operation such as (F*), e.g.;

```
  (F*) ?LEGAL
  NOTIF
     BEEP ." ERROR"
     KEY CLEAR
  ENDIF
```

15.4 FLOATING POINT WORDS

Following is a list of words that relate to floating point operations. See the "Glossary" section for complete descriptions of these words.

10EXP	(N --- EXP)
ASC>FP	(ADR LEN --- FP ENDLOC NDIGITS)
F!	(FP ADR ---)
F#	(FP N --- FP N)
F*	(FP1 FP2 --- FP3)
F+	(FP1 FP2 --- FP3)
F@	(ADR --- FP)
F-	(FP1 FP2 --- FP3)
F.	(FP NDIGITS --- N) (see warning in Glossary)
F.EXT	(N M ---)
F/	(FP1 FP2 --- FP3)
F<	(FP1 FP2 --- B)
FABS	(FP --- FP')
FDROP	(FP ---)
FDUP	(FP --- FP FP)
FLIT	(--- FP)
FNEGATE	(FP --- FP')
FOVER	(FP1 FP2 --- FP1 FP2 FP1)
FP>ASC	(FP NDIGITS --- ADR LEN NEXP)
FROUND	(FP NDIGITS FENCE --- FP' NDIGITS')
FSTANDTYPE	(N --- TYPE)
FSWAP	(FP1 FP2 --- FP2 FP1)
FVAR	(---)

CHAPTER 16: THE SnapFORTH ASSEMBLER

16.1 INTRODUCTION

The SnapFORTH assembler is provided for coding routines that would otherwise run too slowly if written as colon definitions. It is designed so that assembly code may be written in a manner similar to FORTH code. Routines may be written first in FORTH, and then recoded in assembler when experience shows that greater speed is needed.

To put the assembly process in context, it is useful to look at a typical assembler language definition. Following is the definition of ?DUP, which DUPs the value on the stack if it is non-zero:

```
  CODE ?DUP ( N -- N  (N) )
        TOP                 LDA, \OR the two bytes
        TOP 1+              ORA, \ on the stack.
  0= NOT IF,
        TOP 1+              LDA, \ Dup if non-zero
                           PHA,
        TOP                 LDA,
        PUSH                JMP,
  THEN,
        NEXT                JMP,
  ENDCODE
```

We will go into detail about the meaning of this code a little later. For now, notice the following things:

1. The word definition begins with CODE and the name of the word being defined, and ends with ENDCODE. This is analogous to a high level definition beginning with ":" and the name of the word, and ending with ";".
2. Processing is done with names similar to 6502 opcodes—LDA, ORA, JMP, etc.—but all of these words end with commas to distinguish mneumonics from hexadecimal numbers like ADC or DEC.
3. Flow of execution is controlled with words similar to those used in FORTH proper: IF, THEN, BEGIN, UNTIL, etc.

16.2 INTERFACE WITH FORTH ROUTINES

High level routines can be divided into 3 classes:

1. Colon definitions and :C definitions.
2. <BUILDS DOES> definitions
3. :P routines

The inner interpreter executes colon definitions (or :C definitions) by jumping to their addresses (in the case of a colon definition it also selects the memory bank where the routine is found). Therefore, colon definitions must start as a machine code routine.

Colon and :C definitions start with a BRK (break) instruction, which calls the inner interpreter to interpret the definition. The inner interpreter therupon saves the address and the bank number where it was previously interpreting on the return stack, and interprets the routine.

<BUILDS DOES> routines also start as machine code definitions, but this time the routines start with a JSR, to the DOES> part instead of a BRK instruction. This puts the address of the PFA minus one on the return stack (i.e. the return address pushed by JSR,).

The DOES> part starts with another JSR, it calls the low level routine DODOES, which increments the address on the return stack (so that it points to PFA instead of PFA-1, and puts it on the parameter stack, so that it can be easely accessed by the DOES> part. Then DODOES invokes the inner interpreter to interpret the DOES> part.

Both colon (:C) definitions and <BUILDS DOES> routines end with EXIT (";"). EXIT pops the inner interpreter's previous bank number and address from the return stack, so that the inner interpreter can continue interpreting the calling high level routine.

:P definitions are not called by a jump from the inner interpreter but by a JSR, from a low level routine. ;P returns to the calling low level routine. When you write a low level routine which must be called by a high level routine, it starts like any other machine code since it is called by a jump. When your low level routine returns to the high level routine that called it, jump into the inner interpreter with 'NEXT JMP,', instead of an RTS, return.

The main entry point of the inner interpreter is called NEXT (because it starts interpreting the next tag). But there are some alternative entry points which allow you to push or pop a result on/from the parameter stack before going to the inner interpreter. We will discuss these alternative entry points a little later.

We will also discuss the low level equivalent of <BUILDS DOES>—the pair CREATE ;CODE.

You can also define standard low level subroutines which may be called by other assembly routines with the JSR, instruction, and which return by executing the RTS instruction. However, such subroutines cannot be called directly by colon definitions.

A low level routine can call a :P definition with a JSR, instruction. :P routines cannot be called from other high level definitions because of differences in the calling sequence.

Assembly routines can be divided into classes analogous to the three kinds of high level routines:

1. Routines called by high level routines, which exit by jumping to the inner interpreter (for instance to NEXT).
2. CREATE ;CODE definitions, same calling sequence.
3. Subroutines which can only be called by other assembly routines.

16.3 THE RETURN STACK

The 6502 machine stack is used to implement the return stack. Since assembly routines are usually entered via a JMP instruction, and exited with a jump to NEXT, the return stack is not used for linkage with calling high level routines.

However, the return stack may be used to save intermediate results, as long as the results are removed prior to returning. A low level routine may call a :P routine via a JSR instruction. In that case, the :P definition can return via ;P. (:P definitions may also be ended with ";" instead of ";P", but then the return address is left on the stack.) For example:

```
:P HIGH CR ." Hello world!" ;P

CODE LOW     HIGH        JSR,
             NEXT JMP,     ENDCODE
```

16.4 THE PARAMETER STACK

The 6502 has only one machine stack, which is reserved by SnapFORTH for the return stack. But SnapFORTH also needs a parameter stack. Therefore, the X register is used as a stack pointer by SnapFORTH. It always points to the top of the stack. Thus the high order byte of the top element is:

```
1 ,X
```

and the low byte is

```
0 ,X
```

The low and high byte of the second element on the parameter stack are:

```
2 ,X and 3 ,X
```

Two macros are provided to address the top and the second 16-bit element of the stack:

```
TOP          LDA,
```

is the same as

```
0 ,X         LDA,
```

and

```
SEC          LDA,
```

is the same as

```
2 ,X         LDA,
```

To illustrate the operation of the parameter stack, consider a nonsensical routine wich pushes 177H on the stack:

```
HEX
CODE 177H       DEX,    \ Allocate stack space
                DEX,      for number
      1 #       LDA,
      TOP 1+    STA,    \ Push high byte
      77 #      LDA,
      TOP       STA,    \ Push low byte
      NEXT      JMP,    ENDCODE
```

The following code produces the same effect:

```
HEX
CODE 177
      1 #       LDA,
                DEX,
      0 ,X      STA,
      77 #      LDA,
                DEX,
      0 ,X      STA,
      NEXT      JMP,    ENDCODE
```

16.5 SOME MORE EXAMPLES

Now that the principles have been explained, you'll probably agree that the basic routines of FORTH are easy to implement:

```
CODE DROP       INX,
                INX,
                NEXT      JMP,    ENDCODE
```


Another example is DUP:

```
CODE DUP            DEX,     \ Allocate stack space
                    DEX,
         SEC        LDA,
         TOP        STA,
         SEC 1+     LDA,
         TOP 1+     STA,
         NEXT       JMP,     ENDCODE
```

But we will eventually come up with a simpler implementation of DUP.

16.6 PARAMETER PASSING

Like high level routines, most assembly routines have both input and output parameters. For convenience, the input parameters can be copied to a scratch area in the zero page called "N" upon entry.

Output parameters must be pushed on the parameter stack before jumping to NEXT. Since such actions are required so often, some dedicated routines for parameter passing have been provided.

16.6.1 SETUP

SETUP is used to transfer arguments from the parameter stack to N.

The desired number of (16-bit) arguments must be loaded in the accumulator. The top element of the stack will be transfered to N and N+1, the second element to N+2 and N+3 etc. There is room for 16 parameters (32 bytes) in N. SETUP checks for stack underflow, the Y register is set to 0 and the carry flag is cleared.

The carry flag will also get cleared when a low level routine is entered via NEXT (i.e. via the inner interpreter).

The low level implementation of "!" demonstrates the features of SETUP:

```
CODE !   2 #        LDA, \ Move 2 arguments
         SETUP      JSR, \ to N..N+3
         N 2+       LDA, \ Store low byte
         N )Y       STA, \ Y = 0
                    INY, \ Advance pointer
         N 3 +      LDA,
         N )Y       STA,
         NEXT       JMP,    ENDCODE
```

16.6.2 NEXT

We have encountered NEXT several times before. As explained, assembly routines can exit to the calling high level routine by jumping to NEXT.

An extremely simple example is presented by the following routine:

```
CODE DUMMY   NEXT   JMP,   ENDCODE
```

When a routine is called through NEXT, the Y register is 0, and the carry flag is cleared.

Caution: When a CODE routine is entered, the X register points to the top of the parameter stack. A routine is permitted to use X as a counter or an index, but always restore the parameter stack pointer to X before calling NEXT.

16.6.3 PUSH

A routine can add a new 16-bit value to the parameter stack by jumping to PUSH instead of NEXT. The low-order byte of the result must be on the return stack, and the high-order byte in the accumulator. DUP illustrates this calling sequence:

```
CODE DUP   TOP   LDA,
                 PHA,
 TOP 1+          LDA,
 PUSH            JMP,   ENDCODE
```

16.6.4 PUT

PUT has the same calling sequence as PUSH, but it replaces the top element of the stack instead of pushing a new value on the stack.

Thus, to exchange the two bytes of the top 16-bit element, PUT can be used in the following way:

```
CODE FLIP TOP 1+    LDA,
                    PHA,
          TOP       LDA,
          PUT       JMP,   ENDCODE
```

16.6.5 CPUSH and CPUT

PUSH and PUT have 8-bit equivalents called CPUSH and CPUT. The 8-bit value must be loaded in the accumulator before jumping to CPUSH or CPUT, it is expanded to a 16-bit value on the stack by adding a high-order byte of zero.

Again, an example is given for each routine:

```
CODE 1    1 #      LDA,
          CPUSH    JMP,   ENDCODE
CODE C@   0 X)     LDA,
          CPUT     JMP,   ENDCODE
```

16.6.6 POP and 2POP

Either one or two stack elements are dropped (removed) by a jump to these routines. By now, an example will probably be enough to explain the details:

```
CODE DROP    POP    JMP,   ENDCODE
CODE 2DROP   2POP   JMP,   ENDCODE
```

16.6.7 PUTFLS

Replaces the top element of the stack with FALSE (0). Example:

```
CODE FALSIFY  PUTFLS  JMP,  ENDCODE
```

16.6.8 PUTTRU

Replaces the top element of the stack with TRUE (1).

```
CODE TRUEIFY  PUTTRU  JMP,  ENDCODE
```

16.7 TEMPORARY STORAGE

The HHC has several storage areas in page-0 RAM that are available for use by low level code.

XSAVE is a one-byte area set aside for storing the X register. A CODE word or low level procedure may use XSAVE for any purpose, but may not safely assume that it will be preserved across any call to a lower-level routine.

N is a 32-byte area which a CODE word or low level procedure may use for any purpose. You may not safely assume that the contents of N will be preserved across any call to a lower-level routine.

16.8 THE SnapFORTH INSTRUCTION POINTER

The SNAP instruction pointer is kept in RAM at a location addressed by the label 'IP'. It contains the address of the tag that will be interpreted next by the inner loop (NEXT).

16.9 THE ASSEMBLY PROCESS

The word CODE creates a tag table entry for an assembler word definitions, and also initiates the assembly process. We will call a word defined by CODE a code definition or a code word.

The assembly process consists of interpreting code with the context vocabulary set to ASSEMBLER.

The ASSEMBLER vocabulary contains definitions of the words that make up assembler programs (e.g. it contains the 6502 opcodes). The principal function of these words is to deposit machine code in the dictionary.

16.10 THE ASSEMBLER DOES NOT WORK IN COMPILE MODE

It is important to understand that the interpreter is not in compile mode while it is assembling a CODE word.

During assembly, each assembler word is executed. If it is a label, it stacks an address. If it is an opcode, it generates a 6502 instruction. Other kinds of assembler words select an addressing mode.

A naive implementation of some elementary instructions is presented in the following examples:

```
HEX : NOP, EA C, ;
    : SEI, 78 C, ;
    : JMP, 4C C, , ;  \"," For destination.
```

Of course the actual assembler is more complicated (our first two examples could be combined using <BUILDS DOES>, while the JMP, instruction must also deal with forward labels etc.)

Consider the following assembler code, which generates one machine instruction:

```
N 2+ )Y          STA,
```


When the CODE definitions containing this code are interpreted, the code is executed, producing the following results:

1. N stacks the address of N.
2. 2+ Adds 2 to the address of N.
3.)Y indicates that this instruction should use indrect indexed Y addressing.
4. STA, generates a STore Accumulator (STA) operation, using indirect indexed Y addressing, with an address taken from the top of the stack—value N+2.

16.11 MACROS

Macros can be implemented by deferring the execution of ASSEMBLER routines by placing them in between ":" and ";". When invoked, the following macro compiles the increment of a 16-bit number located at N through N+1.

```
: INN,  ASSEMBLER
                N       INC,
        0= IF,  N 1+    INC,
        THEN,            ;
```

Another useful macro is

```
: NEXT, ASSEMBLER NEXT JMP, ;
```

16.12 GETTING IN AND OUT OF THE ASSEMBLER VOCABULARY

CODE makes the ASSEMBLER vocabulary the SnapFORTH compiler's context vocabulary; it also defines a CODE word (i.e. it makes an entry in the current tag table). ENDCODE resets the context vocabulary to be the same as the current vocabulary.

LABEL also makes the ASSEMBLER vocabulary SnapFORTH's context, even when executed outside a CODE definition. This may cause confusion, since the routines "0=" and "0<" are redefined within the ASSEMBLER vocabulary with a completely different meaning than the ones they have in the FORTH vocabulary! Since LABEL makes no entry in the tag tables, you can use it to define a low level subroutine which will only be called by other low level routines.

Of course you can also set context to ASSEMBLER by executing ASSEMBLER.

16.13 OPCODES AND ADDRESSING MODES

The routines described in this section generate the actual machine code. Their names are like the opcodes used by other assemblers, except for the comma (which is Forth's convention for "generating code").

16.14 SIMPLE MACHINE INSTRUCTIONS

The following routines assemble some simple 6502 instructions. They need no operands.

SEC, Set carry.
CLC, Clear carry.
SED, Set decimal mode.
CLD, Clear decimal mode.
SEI, Disable interupts.
CLI, Enable interupts.
CLV, Clear overflow flag.
NOP, Do nothing.
DEX, Decrement X register (stack pointer).
INX, Increment X.
DEY, Decrement Y.
INY, Increment Y.
TAX, Accumulator—>stack pointer.
TXA, Stack pointer—>accumulator.
TAY, Accumulator—>index register Y.
TYA, Index reg Y—>accumulator.
TXS, Stack pointer—>return stack pointer.
TSX, Return sp—>stack pointer.
PHA, Push accumulator on return stack.
PLA, Pop accumulator from return stack.
PHP, Push status reg on return stack.
PLP, Pop status reg from return stack.
RTS, Return from subroutine.
RTI, Return from interupt.

16.15 IMMEDIATE ADDRESSING

Immediate addressing is used for constant operands, and is indicated by "#"

```
10 #          LDA, \ Set accumulator to 10.
```

Do not confuse this mode with zero page addressing, which is something entirely different!

```
10            LDA, \Load CONTENTS of 10 into
                    accumulator
```


As a last example, check whether the value of the accumulator is "a"

```
&a #        CMP,
```

16.16 INDEXED ADDRESSING

The two index registers "X" and "Y", can be used to add an offset to the other operand at runtime. Thus

```
2 #         LDY,
500 ,Y      LDA,
```

will load the contents of address 502 into the accumulator.

16.17 STACK ADDRESSING

The X register is a special case, since it is used by Forth as stack pointer for the parameter stack. The first operand tells which element of the stack is being addressed. Some examples are gathered in the following table

0 ,X Top element low byte.
1 ,X Top element high byte.
2 ,X Second element low byte.
3 ,X Second element high byte.
etc.

You should now have enough knowledge of the 6502 assembler to understand the following routine:

```
CODE 3PICK          DEX,
                    DEX,
            6 ,X    LDA,
            0 ,X    STA,
            7 ,X    LDA,
            1 ,X    STA,
            NEXT    JMP, ENDCODE
```

or

```
CODE 3PICK

            4 ,X    LDA,
                    PHA,
            5 ,X    LDA,
            PUSH    JMP, ENDCODE
```

16.18 ZERO PAGE AND ABSOLUTE ADDRESSING

Zero page and absolute addressing are used when the operand of an instruction is a memory location. If this location is on the zero page (i.e. less than 100 hex), the assembler selects zero page addressing, otherwise absolute addressing is selected. An instruction using zero page addressing uses one less byte than absolute addressing.

Zero page: load the contents of N into the accumulator.

```
N          LDA,
```

Absolute: test the contents of C000

```
C000       BIT,
```

16.19 INDIRECT X)

X is added to the first operand (which must be a zero page location), the sum is treated as a pointer to the actual operand.

Since X is the stack pointer, we can use this mode to implement C@.

```
CODE C@     0 X)      LDA,\ Load indirect.
            TOP       STA, \ Store in low byte.
            0 #       LDA, \ High byte becomes
            TOP 1+    STA, \zero.
            NEXT      JMP, ENDCODE
```

or

```
CODE C@     0 X)      LDA,
            CPUT      JMP, ENDCODE
```

16.20 INDIRECT)Y

The contents of the first operand is fetched, and Y is used as an index to it. If locations N and N+1 contain 6500, then

```
2 #     LDY,
N )Y    LDA,
```

will load the contents of 6502 into the accumulator.

16.21 ACCUMULATOR ADDRESSING

If the accumulator is the only operand of an instruction, ".A" should be used. For example, to rotate the accumulator

```
.A      ROL,
```


Notice that this is different from other 6502 assemblers, which do not require explicit specification of this particular addressing mode.

16.22 SOME MORE INSTRUCTIONS

The following instructions can be used with one or more of the addressing modes we have discussed in the previous sections.

LDA,	Load accumulator with memory or constant.
STA,	Store accumulator in memory.
BIT,	Test bits in memory with accumulator.
ADC,	Add memory to accumulator with carry.
SBC,	Subtract mem from accumulator with borrow.
CMP,	Compare memory and accumulator.
EOR,	Exclusive OR memory with accumulator.
AND,	"AND" memory with accumulator.
ORA,	"OR" memory with accumulator.
INC,	Add one to memory.
DEC,	Decrement memory by one.
ASL,	Algebraic shift left.
LSR,	Logical shift right.
ROL,	Rotate left.
ROR,	Rotate right.
LDX,	Load stack pointer or index register Y
LDY,	with memory or constant.
STX,	Store stack pointer or index register Y
STY,	in memory.
CPX,	Compare stack pointer or index register Y
CPY,	with memory or constant.

16.23 CONTROL FLOW—CONDITIONAL SPECIFIERS

FORTH-like control flow statements, which test the condition codes in the 6502 CPU, are available, reducing the need for unstructured GOTO's. In the following example, the part of the program after "IF," (note the comma), will only be executed if the zero flag is set

```
0= IF,           DEX,
                 DEX,
           TOP   LDA,
THEN,
```

NOT is used to invert the condition code. Thus, in the next example, the statements between IF, and THEN, will only be executed if the zero flag is clear

```
        0= NOT IF,        INX,
                          INX,
              0 ,X        STA,
        THEN,
```

Of course, ELSE, can be used if an action is specified for both cases

```
        0= IF,            DEX,
          \ This part is executed only if the zero flag is set.
                          DEX,
              0 ,X        LDA,
        ELSE,
                          INX,
          \ This part only if it's clear.
                          INX,
              0 ,X        STA,
        THEN,
```

Other condition codes known to the assembler are

CS—Carry set	CS NOT—Carry clear.
VS—Overflow set	VS NOT—Overflow clear.
0<—Negative flag set	0< NOT—Negative flag clear

16.24 CONTROL FLOW—CLAUSES

The words

BEGIN, WHILE, REPEAT, UNTIL, and AGAIN,

(the commas are part of the names), are used in the same fashion as their Forth counterparts. The following example searches through a text for a comma, it stops when more than 100H bytes have been scanned without finding one.

```
        BEGIN,   N )Y        LDA,
                 &, #        CMP,
        0= NOT WHILE,
                             INY,
        0= UNTIL,
```

The loop clauses known to the assembler are:

```
        BEGIN, UNTIL,
        BEGIN, AGAIN,
        BEGIN, WHILE, REPEAT,
        BEGIN, WHILE, UNTIL,
```

16.25 SPAGHETTI

Gotos and labels provide a way to turn a decent program into spaghetti. But they can also be used to optimize your program,

or to solve problems which resist every attempt to be solved with the control structures discussed in the previous chapter.

Labels are defined by

```
        LABEL name
```

They behave like literals. i.e. if you refer to a label in an assembly program, like

```
                         FF #        LDY,
                  \Count number of non zero characters.
        LABEL >CNT                   INY,
                          N )Y       LDA,
                 >CNT                BNE,
                                     TYA,
```

its value is pushed onto the stack (from which it is removed by BNE,).

The following branch instructions take an expression and a condition code as operands. The value of the expression cannot differ from the current location by more than 127 bytes ahead or 128 bytes behind.

BCC,	Branch on carry clear.
BCS,	Branch on carry set.
BEQ,	Branch on result zero.
BNE,	Branch on result non zero.
BMI,	Branch on result minus.
BPL,	Branch on result plus.
BVC,	Branch on overflow clear.
BVS,	Branch on overflow set.

16.26 JUMP INSTRUCTIONS

The instructions

```
        JMP,
        JSR,
```

can only be used with the absolute addressing mode. A jump to the contents of a memory location can be made by

```
        ) JMP,
```

(pronounce "jump indirect"). Example, jump to the routine pointed to by IP

```
        IP      ) JMP,
```

16.27 ;CODE

A remarkable feature of Forth, shared by only a few other programming languages, is its use of generic routines.

We assume that you are already acquainted with the operation of <BUILDS DOES>.

These two words allow you to create a new class of objects by supplying some compiling code, which generates a new object of this class and stores all necessary information concerning it in its Parameter Field. There is also run time code, invoked when the new object is executed, which can retrieve the information in the PFA and undertake the appropriate action.

The low level version of <BUILDS DOES> is formed by the two words CREATE and ;CODE. CREATE is similar to <BUILDS and ;CODE to DOES>. But the run-time code for the newly defined class is written in assembly language instead of plain Forth code.

The passing of parameters to the run time code, however, is different. PFA MINUS 1 is pushed on the RETURN stack! If you keep this in mind, everything will behave just as you expected.

Example: Low level definition of CONSTANT

```
: CONSTANT CREATE ,
  ;CODE              PLA,
   N                 STA,
                     PLA,
   N 1+              STA,
                     INY,
   N )Y              LDA,
                     PHA,
                     INY,
   N )Y              LDA,
   PUSH              JMP,   ENDCODE
```

16.28 EXAMPLES

Below are several examples of CODE words and low level procedures to help you become familiar with the SNAP assembler and the techniques of writing low level code for the HHC.

After each example are annotations which expand on the comments in the code. The annotations are keyed to the line numbers that accompany the code.

Example 1: SETUP

SETUP is the procedure, described above, which pops a specified number of entries from the parameter stack to the work area, N. The top entry goes to N, the next entry to N+2, etc.


```
0 LABEL SETUP       \Y=0 on exit; X is updated.
1   0 # LDY,    .A ASL,    N 1- STA,
2   BEGIN,
3     TOP LDA,    N ,Y STA,
4     INX,    INY,    N 1- CPY,
5     0= UNTIL,
6     0 # LDY,    CLC,    RTS,
```

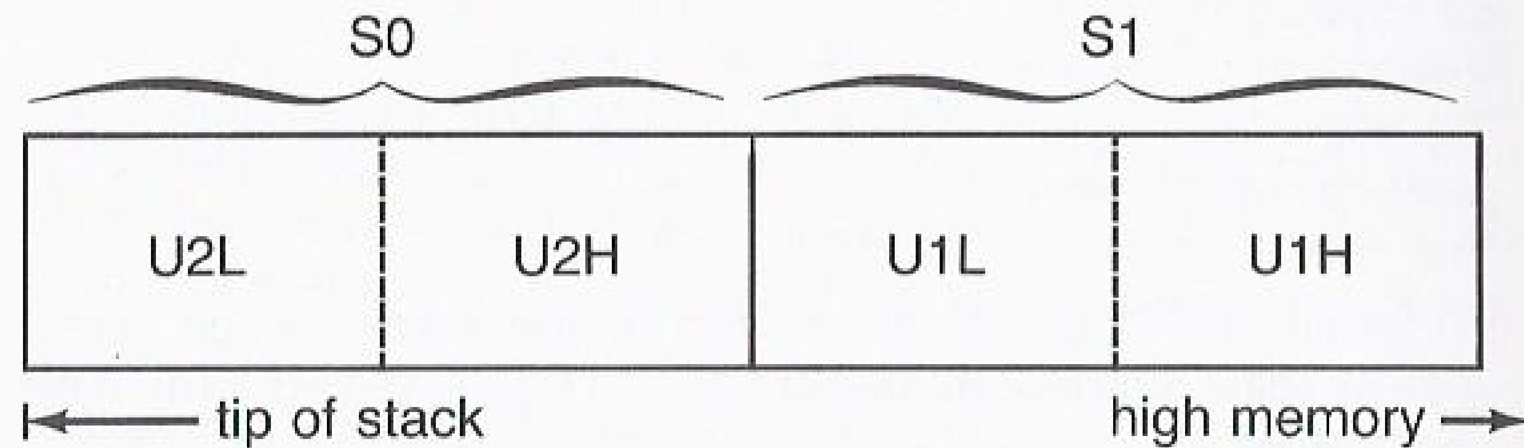
0. Recall that X points to the top of the stack. Since some number of entries are being popped off the stack, X must be updated.
1. 0 is loaded into Y. The accumulator contains the number of words to pop. It is shifted left 1 bit, producing the number of bytes to pop; this is stored at N-1, where a byte is reserved for this purpose.
3. The tip byte of the stack is loaded into AC and then stored at (N,Y).
4. Both X and Y are incremented. Y is compared to the contents of N-1, the number of bytes to pop.
5. Until Y = (N-1) (the meaning of 0= after a compare), we loop back to line 2 to pop the next byte.
6. When we have popped the proper number of bytes, we load Y=0, clear the carry flag, and return to the caller.

Example 2: U*

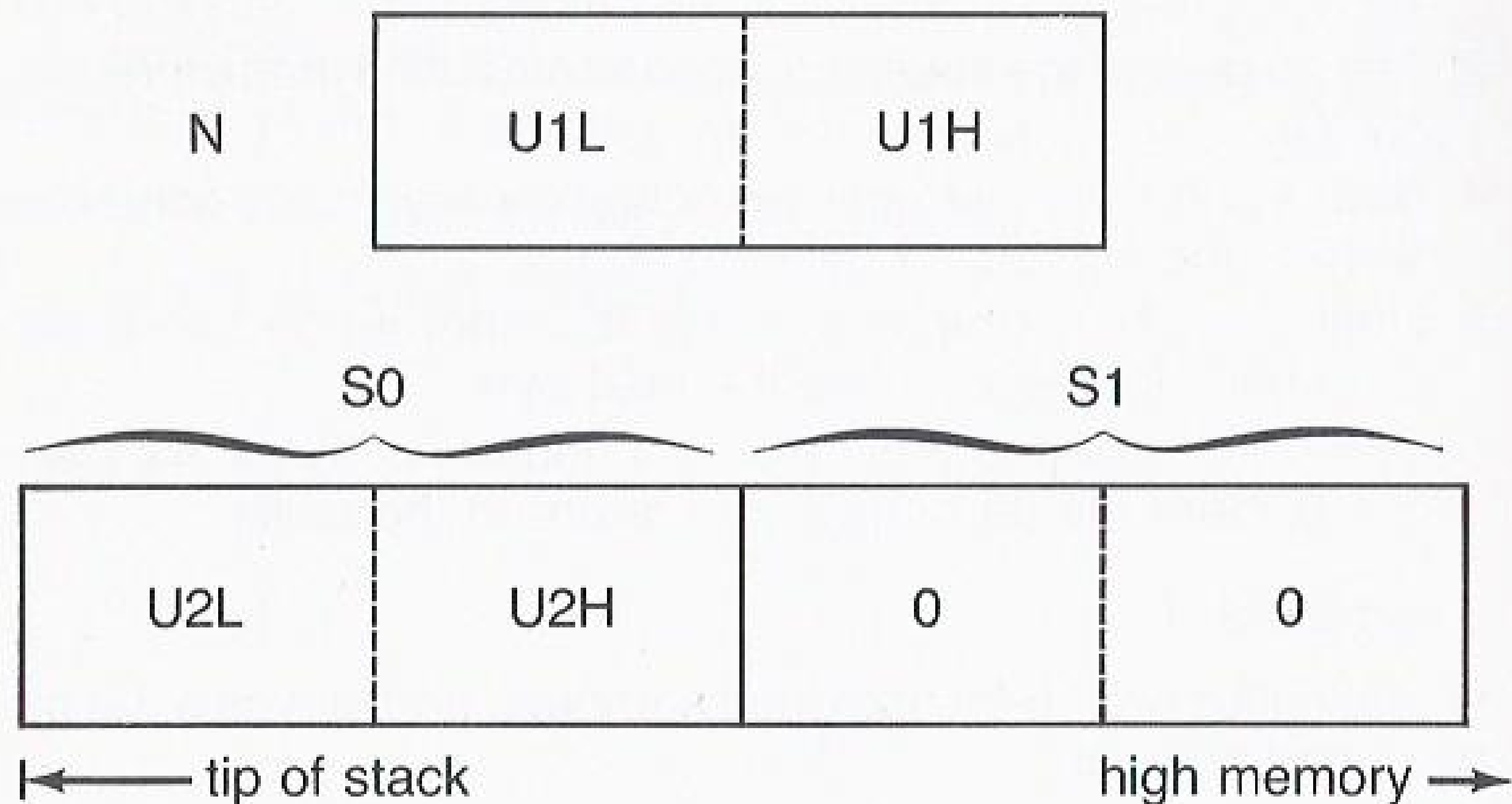
U* multiplies two 16-bit unsigned numbers, and leaves a 32- bit unsigned product.

```
0  HEX
1  CODE U*    ( U1 U2 --- DU )
2    SEC LDA,    N STA,    SEC STY,
3    SEC 1+ LDA,  N 1+ STA,  SEC 1+ STY,
4    10 # LDY,
5    BEGIN,
6      TOP 2+ ASL,
7      TOP 3 + ROL,  TOP ROL,  TOP 1+ ROL,
8      CS IF,
9        CLC,
10         N LDA,  TOP 2+ ADC,  TOP 2+ STA,
11         N 1+ LDA,  TOP 3 + ADC,  TOP 3+ STA
12         CS IF,
13           TOP INC,
14             0= IF,
15               TOP 1+ INC,
16           THEN,
17         THEN,
18     THEN,
19     DEY,
20   0= UNTIL,
21     NEXT JMP,
22   ENDCODE
```

Consider the stack on entry to U*. It looks like this (L and H stand for "low byte" and "high byte"):



U* first moves U1 to a scratch area, N, and zeroes it on the stack:



Next U* begins a loop. The loop shifts the top 4 bytes of the stack (S0/S1) left one bit. A 0 goes into the lowest bit of S1, and the highest bit of S0 goes into the carry flag, C. Then, if C is set (*i.e.*, if the bit shifted out of S0 was 1) it adds U1 to S1.

The loop is executed 16 times, once for each bit in U2. When it is done, U2 has been shifted out of existence, and S0/S1 contains the product, DU.

2-3. U1 is copied from S1 to N. S1 is set to 0. (The value of Y is always set to 0 by NEXT.)

4. Y is set to 16. This is the number of times the loop will iterate.

5-7. The loop begins. The 32-bit unit S0/S1 is shifted left one bit. Notice how S1 is addressed as 'TOP 2 + ' and 'TOP 3 + ', rather than 'SEC' and 'SEC 1 + '. This makes no difference in the assembled code, but it emphasizes that we are treating S1 as an extension of S0, and not as a separate 16-bit parameter.

8-11. If C (the bit just shifted out of S0) is 1, we add U1 to S1.

12-17. If there was a carry out of S1, we increment S0.

19-20. End of the loop. Y is decremented; we loop until Y = 0, then we jump to NEXT.

Example 3: SHIFT

SHIFT shifts a word, X, left a specified number of bits, N. If N is negative, X is shifted right.

```
0    CODE SHIFT   ( X NS --- X' )
1      TOP 1+ LDA,
2      0< NOT IF,
3        BEGIN,
4          TOP LDA,     0= NOT WHILE,
5          SEC ASL,     SEC 1+ ROL,
6          TOP DEC,
7        REPEAT,
8        POP  JMP,
9      THEN,
10      BEGIN,
11        TOP LDA,     0= NOT WHILE,
12        SEC 1+ LSR,  SEC ROR,
13        TOP INC,
14      REPEAT,
15     POP  JMP,
16     ENDCODE
```

The word loads NS's high byte (NSH) into AC. It uses the "negative" or "sign" flag to choose a left shift (N = 0) or a right shift (N = 1). Then it executes one of two loops to shift X the proper number of bits in the proper direction.

1. Loads NSH into AC.

2. If the load set N = 0, the left-shifting loop is executed.

3-4. Begin the left-shifting loop. If NSL has been decremented to 0, escape from the loop; else proceed.

5-7. Shift NS left one bit; decrement NSL and repeat. End of the left-shifting loop.

8. Jump to POP. POP will drop NS and jump to NEXT.

10-15. The right-shifting loop. Control falls to here if the IF, ... THEN, in 2-9 is not executed. The procedure is the same, except that we shift right, not left. N<0, so we increment it toward zero.

CHAPTER 17: SAMPLE PROGRAMS

This chapter contains several annotated HHC application programs written in SnapFORTH.

These programs serve two functions. First, they can serve as reference points as you develop your own SnapFORTH coding style. Second, they contain many passages of ready-made code that you can copy in order to solve some of the common problems you will encounter, such as how to display a standard-format HHC menu.

17.1 SAMPLE 1: ORDER ENTRY PROGRAM

The order entry program lets a customer of a hypothetical retail store order goods through an HHC. It guides the customer through the process of identifying himself, then displays a list of goods available and lets the customer enter the quantity he wants to order for each item.

The program is designed to be entirely self-teaching; that is, a customer who knows nothing about it can use it without referring to any instructions outside those provided by the program itself.

17.1.1 Use (Technical)

Inventory File

The HHC must have an **inventory file** in its virtual file space before the order entry program may be used. This file defines the items the customer will be able to order. The name of the file must be 'O.E.INV'.

There are two ways you can create the inventory file: by running a separate program from the capsule's main menu, or by using the file system editor.

The separate program in the capsule's main menu is titled something like "create order entry file." This program creates an inventory file containing three hard-coded records, for "binocular bump pads," "tiffin guns," and "hypnoglyphs."

If you want to create a larger or more realistic file (***e.g.***, for demonstrating the HHC at a trade show), use the file system editor and follow the file format explained in the next section.

Format Of the Inventory File

The inventory file contains one record per item. The order of the records is unimportant, except that it determines the order in which the items will be presented to the customer.

The format of a record is:

```
name-of-item\qty-price\qty-price\...
    \qty-price
```

"name-of-item" is the name of the item this record represents.

Each "qty-price" gives a range of quantities and a unit price for orders in that range. The ranges should be contiguous; the range of the left-most "qty-price" should begin at quantity = 1, and the range of the right-most "qty- price" should be open-ended.

For example:

```
Tiffin guns\1-3 55.00\4-9 47.50\
    10+ 40.00
```

The format of each "qty-price" should conform to the example above. In particular, observe the following rules:

1. Fields should be separated by backslashes. The last field should ***not*** be followed by a backslash.
2. Each quantity range except the last should consist of two numbers separated by a hyphen, with no spaces.
3. The last quantity range should consist of a beginning number followed by a '+' sign and a blank.
4. The quantity and dollar amount should be separated by one blank.
5. Each price should consist of a dollar figure, a decimal point, and two "cents" digits.

The Order File

The HHC must have an **order file** in its virtual file space in which to record orders. Unlike the inventory file, the order file need not be prepared beforehand; the program will create it automatically if it does not exist.

The name of the order file is 'O.E.INP'.

The order file consists of a series of record groups. Each record group contains information about one order.

Each record group begins with one record containing the name and address of the person making the order. This is followed by one record per item ordered.

The format of the name-and-address record is:

```
name\adr-line-1\adr-line-2\...
    \adr-line-n
```

For example:

```
Joe Schlumpf\PO BOX 0\Dogpatch, AK
```

The address may have up to 5 lines.

The format of an item-ordered record is:

```
inv# qty price
```

where "inv#" is the origin-0 number of the inventory file record identifying the item being ordered; "qty" is the quantity ordered; and "price" is the unit price in cents that applies to the quantity ordered.

For example, suppose a customer orders 15 of the the item in record #3 of the inventory file, and the unit price for orders of 15 is $20.00. The item-ordered record will be:

```
3 15 2000
```

When the order file is nearly full (less than 100 bytes remaining in the file space), an attempt to start the inventory program will produce a warning message like this on the LCD:

```
FILE SPACE LOW
PLEASE NOTIFY A SALESMAN
```

The two lines of the warning will appear alternately until the someone presses CLEAR. The "salesman" should copy the order file to a Programmable Memory Peripheral if he wants to keep it; then he should delete it. (The program specs state that the salesman will use an "upload" function to send the order file to a host system, where orders would be processed. To date the upload function has not been implemented.)

17.1.2 Use (Customer Interface)

When you start the order entry program, it gives you a menu with 3 options: "take an order," "print last order," and "print summary of orders." Select "take an order."

Next the program prompts you for your name and address. If the address has fewer than 5 lines, enter a null line after the last line of the address.

Next the program asks you if you want instructions. If you answer Y for "yes," it displays brief instructions on how to run the program. The instructions are displayed one line at a time. You may terminate the instructions by pressing the S key.

The program then displays the name of the first item in the inventory file. You can scroll the inventory file forward with the ↓ key, and backward with the ↑ key. The file is circular; i.e. the record after the last is the first, and the record before the first is the last. The program displays a warning message before wrapping in either direction.

You can order any item by entering the desired quantity while that item is being displayed, and pressing ENTER. Alternatively, you can just press ENTER. Then the program displays the first quantity-price range and the associated price. You can scroll the quantity-price ranges with the ← and → keys. Again, you can order an item by entering the desired quantity and pressing ENTER; or you can just press ENTER, in which case the program prompts you for a quantity.

After you enter the quantity, the program computes the extended price (product of quantity and applicable unit price) and displays it.

When you are done looking at the extended price, you may return to the inventory menu by pressing any key.

When you are done entering the order, press the E (for "end") key. The program displays the total, sales tax (at 6%), and sum of total and tax.

Print Order Function

This function prints the order most recently entered.

To use this function, you must have an 80-column printer attached to the HHC through a Serial Interface Adaptor. The printer should be loaded with 9-1/2 x 11" paper.

Print Summary Function

This function prints a one-line summary of each order that has been entered since the order file was last dumped.

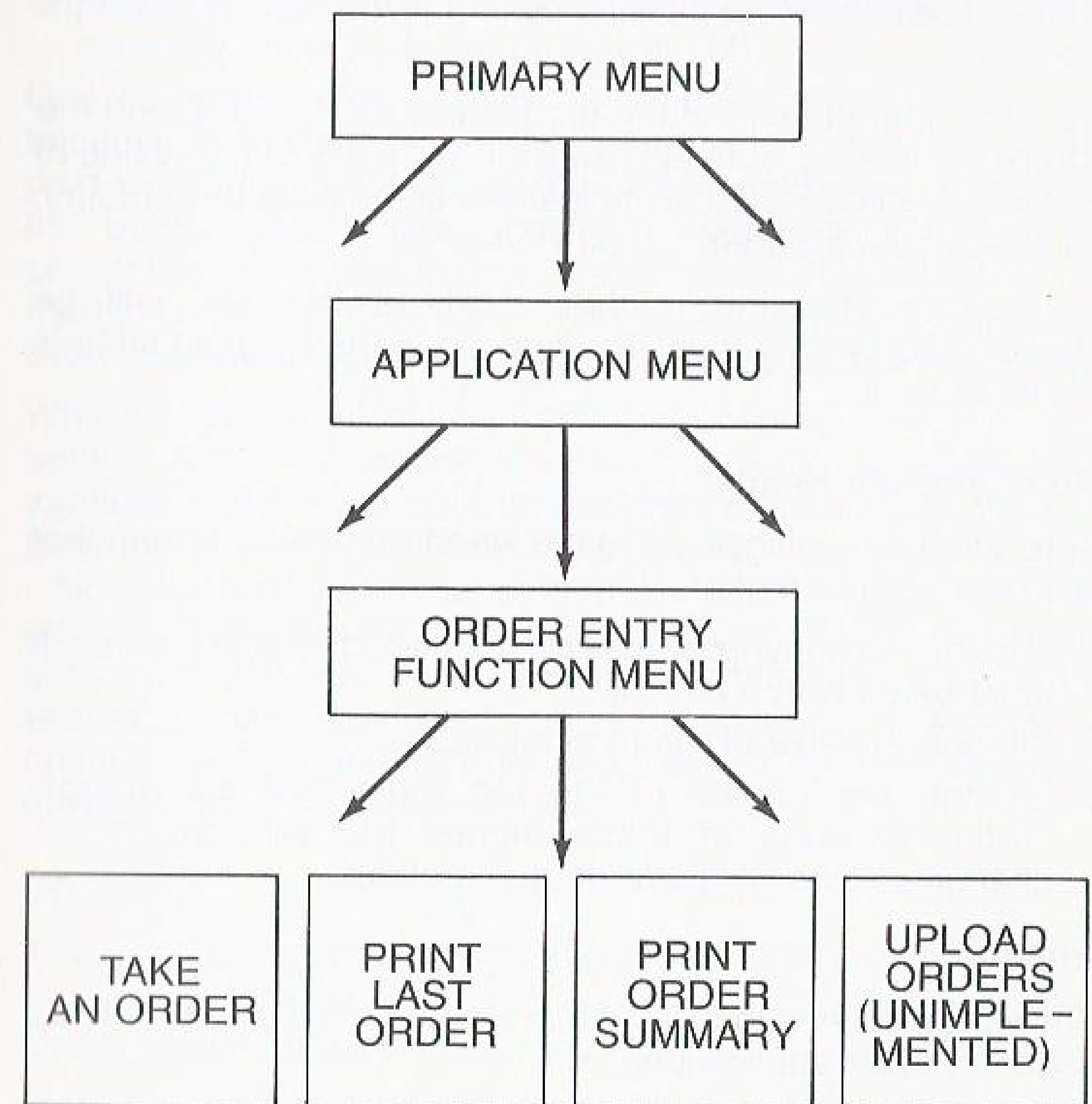
To use this function, you must have an 80-column printer attached to the HHC through a Serial Interface Adaptor. The printer should be loaded with 9-1/2 x 11" paper.

17.1.3 Program Structure

The order entry program is actually several programs in a single capsule. There is one program for each of the following functions: taking an order, printing the most recent order, and printing a summary of all orders.

To avoid confusion, we will use the word **program** to refer to the software in a capsule. We will use the word **function** to refer to the distinct parts that one program may consist of; we would say, for example, that the order entry program has 4 functions.

The HHC has no facility for selecting individual functions, so the capsule itself must select them in some manner. It customarily does this by presenting another menu, which operates just like the primary menu and the application menu in the HHC.



Program With Function Menu

The Header

The first thing compiled in any application program is the program header. This includes the copyright notice (the "C" which must be present if the HHC is to agree to execute the program) and the tag table.

For a detailed explanation of the header's format, see the discussion of the header in the chapter on "General Technical Information."

We define our offset from capsule space by subtracting HERE from ROMADDR. Then we compile a string constant containing the copyright notice.

We set the dictionary pointer to ROMID relative to HERE, and compile a string constant giving the title of the capsule. The title will appear in the primary menu when this program's capsule is plugged into the HHC. When the customer selects this capsule from the HHC's application menu, the HHC will display the title before passing control to the mother tag of the capsule.

We store the address of the tag table at ROMVECT, and the number of the tag table at ROMEXT. We set 80H (the value of PROGBIT) in CSPEED, to indicate what the capsule contains: an executable program in fast ROM.

We specify that the mother word, MAIN.MAN, will be represented by long tags. The first tag in the long tag table is reserved for it.

Modifying the Header

Here are the modifications to the standard header format that you may wish to make:

1. Change the copyright message to reflect the your desires. It must begin with a capital C.
2. Change the size of the long tag table.
3. Change the names of the tag table and the strings. Ordinarily none of these names are referred to, so changing them will have no ripple effect.

Utilities

The words defined in this section are ones that are likely to be useful in other applications.

Most of these words relate to formatting numbers for I/O, and to managing keyboard input. Notice the definition of the EOL word, which is used later in a call to EXPECT.

Data and Constants

Here begin words specifically written for the order entry application. The major words are:

- INBUF#, a buffer used for all input operations from files.
- KBBUF, a buffer used for all input operations from the keyboard.
- OUTBUF, a buffer used for all output operations.
- The device numbers and ECB's of the modem (MDMxxx) and printer (PTRxxx).

  Notice that the ECB's are declared as string variables. This is the most convenient way to declare variables of a specific length, even if the variables will not be used as strings. Remember, STRING allocates an actual amount of space one byte greater than the length, to allow for a length byte.
- Variables for storing different aspects of the state of the program, such as the number of the inventory item currently being displayed (O.CURITM).

One colon definition (O.INBUF) is included in this section because its function is to define a "variable" located one byte past the beginning of one of the buffers (INBUF#). INBUF is the buffer actually used for input operations. INBUF# is present to allow one byte of space before the beginning of INBUF to hold a binary zero. The binary zero acts as a record delimiter during backward scans of a string held in the buffer.

Why not simply allocate a separately named one-byte field before INBUF? Because the HHC does not allocate TSA variables in the order they are declared! It happens to allocate them in reverse order. Rather than write a tricky piece of code which allocates a delimiter byte ***after*** INBUF to make the byte compile into the address space ***before*** INBUF, we have chosen to allocate a single field and then sub-divide it by writing a colon definition. The resulting code is somewhat obscure, but it frees us from dependence on a detail of the cross-compiler's implementation.

I/O Words

The next section of the program contains words relating to I/O.

Order Entry Function

Here, at long last, comes the guts of the order entry program.

It will be a useful exercise for you to read the entire program well enough to understand the function of each word. We will only point out some tricks and global considerations that will help you understand the code.

FIND and -FIND are ***not*** related to the same-named words in FORTH, which search the dictionary for a given word. (Recall that there is nothing like a dictionary in a stand-alone SnapFORTH application at run time.) They scan a data string for a delimiter character. FIND scans forward; -FIND scans backward. Either one is stopped by a binary zero, which signifies end-of-string.

FIND and -FIND are used to move forward and backward within inventory file records. For example, when the HHC is displaying the first quantity-price field in an inventory item, it has a "current position" pointer (O.CURITM) which is pointing at that item:

```
Tiffin guns\1-3 $55.00\4-9 $47.50\
   10+ $40.00
```

When the customer presses the ➡ key, the program calls FIND to move the pointer forward to the next backslash; then it moves the pointer one byte further to the beginning of the following field.

FETCH.PRICE is used after an item is ordered to find the price that applies to whatever quantity of that item the customer has specified.

MOVEDN, MOVEUP, MOVEL, and MOVER ("move down, up, left, and right") allow the customer to "move around" in the inventory file. They update the program's state variables to reflect the requested motion, and update the LCD display. They are called by INVKEY, which reads a key from the keyboard and acts on it while the customer is looking at inventory information and ordering items.

INVKEY calls ORDER.ITEM when the customer orders an item. If the customer orders by simply keying in a number, INVKEY receives the first character of the number; it passes the character to ORDER.ITEM on the stack. If the customer orders by pressing ENTER and then keying in a number, INVKEY gets the ENTER and has no character to pass to ORDER.ITEM; it indicates this by passing ORDER.ITEM a binary zero.

INIT.ORDER initializes the program's state variables at the start of each order. Recall that HHC variables are NEVER initialized automatically; they are allocated on the temporary stack, and they initially contain whatever garbage was left there by other programs.

Print Order Function

The next part of the program implements the print-an-order function.

Function Menu

The next part of the program is the function menu. It takes advantage of some standard HHC utilities which all applications can use to create standard format menus.

The first definition in the function menu is O.PGMS, which executes the functions represented by the menu options. It takes one item on the stack, a number indicating which option to execute. A 0 indicates the first option, a 1 indicates the second, etc.

The second definition is a table of string constants, O.MSGLST. The strings identify the menu options, and correspond to the options executed by O.PGMS.

The third definition, O.MSG, types one of the strings in O.MSGLST. The string it types is determined by the number passed to O.MSG on the stack.

The fourth definition, O.MSGS, allows for wraparound in the message display. Given a number, it determines whether the number represents a message. If so, it displays the message and returns TRUE. If not, it returns FALSE.

The fifth definition, ORDER.MENU, performs the entire function of the program except for initialization. It stacks a 1 (the origin-1 number of the first option to display), the address of O.PGMS, and the address of O.MSGS. Then it calls an HHC word, MENU-DRIVER, that performs all the processing associated with a standard HHC menu.

The sixth definition, ORDER.ENTRY, performs initialization for the order entry program, and executes ORDER.MENU. Notice that it sets the tag number of ORDER.MENU at SOFTAG. This makes ORDER.MENU the word that will be executed by a soft clear.

File Initialization Function

Next is a short section of code which implements the file initialization function. This creates the inventory file that offers hypnoglyphs, tiffin guns, etc.

Main Program Menu

Last is a section of code that implements the top-level menu for this program. The logic of this code is the same as the logic of the preceding menu.

CAP.INIT initializes the tag table vector's short extrinsic table entry. (Recall that the HHC emulator in the development system will execute this word, the second word in the long tag table, before the mother word is ever executed.)

MAIN.MAN is the mother word. The first thing it does is call CAP.INIT, which makes it safe to execute short tags. Then it sets up a call to the menu driver for the capsule's top-level menu.

Modifying the Function Menu

You can adapt the function menu to your own applications quite easily. Copy the code from O.PGMS through O.MENU, with the following modifications:

1. Replace the IF.ITS lines in O.PGMS with equivalent lines that execute your program's functions.
2. Replace the strings in O.MSGLST with equivalent strings that describe your program's functions. Note, there should be the same number of string in O.MSGLST as there are IF.ITS lines in O.PGMS, and they should be in the same order.
3. Change the '4' in O.MSGS to the number of IF.ITS lines you have in O.PGMS.
4. Change the names of the words to follow any naming conventions you have adopted for your program. Be sure to change all references to the changed names.

17.2 SAMPLE 2: SKETCH PROGRAM

The sketch program allows the user to "sketch" a picture on the HHC's LCD. There are functions to save pictures in files, restore them, etc.

17.2.1 Use

When you run SKETCH, you get a function menu with the following entries:

```
MAKE NEW PICTURE
MODIFY EXISTING PICTURE
CONTINUE W/ SAME PICTURE
SAVE THE PICTURE
DELETE A SAVED PICTURE
```

The operation of SKETCH is organized around two important concepts. The first concept is **edit mode**. This is a mode of operation in which SKETCH lets you create or modify a picture. Several of the SKETCH menu functions put SKETCH into edit mode. There is a command in edit mode for returning to the function menu.

The second important concept is the **current file**. This is the picture file which SKETCH is currently working with. SKETCH

can have a current file even when not in edit mode; it is the file that SKETCH will operate on if you select the function "CONTINUE WITH SAME PICTURE" or "SAVE CURRENT PICTURE."

The Editing Functions

MAKE NEW PICTURE lets you enter the name of a file in which SKETCH will save the picture you create. SKETCH creates this file and makes it the current file, and places you in edit mode so that you can create a picture.

MODIFY EXISTING PICTURE lets you enter the name of an existing picture file. SKETCH makes this file the current file, reads the file, and displays the file on the LCD. Then it places you in edit mode so that you can modify the picture.

CONTINUE W/ SAME PICTURE enters edit mode without changing the current file.

SAVE THE PICTURE saves the current picture file, including any modifications you have made to it. After editing a picture and returning to the program's menu, you save the edited picture by selecting this menu option.

DELETE A SAVED PICTURE lets you enter the name of an existing picture file. This file becomes the current file, and then is deleted.

Safety Features

If you create or edit a file, and then work on another file without saving the first one, whatever you have done to the first file will be lost. This is because nothing is actually written to the file until you run the SAVE function.

But SKETCH always warns you when you ask it to do anything which may make you lose work in this way. For example, suppose you edit one file, then try to edit another file without saving the first file. SKETCH will ask you if you want to save the first file before editing the second one. If you reply Y (for "yes"), SKETCH saves the current file before proceeding. If you reply N (for "no"), SKETCH proceeds anyway; but when the new file becomes the current file, any changes you made to the original file are lost.

SKETCH also stops you from running any functions that do not make sense. For example, you cannot MAKE NEW PICTURE with the same name as an existing picture; nor can you CONTINUE WITH SAME PICTURE after DELETE SAVED PICTURE (because the current file is the one just deleted).

Edit Mode

In edit mode, SKETCH displays a special cursor that consists of only one dot. You can move the cursor around with the following keys:

Key	Function
←	Move left 1 column.
→	Move right 1 column.
↑	Move up 1 row.
↓	Move down 1 row.

In its initial "draw" state, the cursor leaves a "trail" of black dots behind it as it moves. This is how you draw a picture.

SKETCH has two other states: "neutral" and "erase". In neutral state, the cursor leaves no trail. In "erase" state, the cursor leaves a trail of clear dots. This enables you to correct errors in your picture without starting over.

You can change the state of SKETCH with the following keys:

Key	Function
INSERT	Puts SKETCH in draw mode (its initial mode).
ENTER	Puts SKETCH in neutral mode.
DELETE	Puts SKETCH in erase mode.

There are two additional commands. One makes SKETCH begin rotating the entire picture from left to right. The other ends EDIT mode, and returns you to the program's menu.

Key	Function
ROTATE	Begins rotating the picture left to right.
e or E	End edit mode, returning to menu.

17.2.2 Program Design

The program is organized in roughly the same way as the order entry program in example 1. There is a header; then generally useful utility routines; then data and program-specific utility routines; then the routines which implement the program's functions; then a menu driver.

Data

A picture file contains one 156-character record. Each character in the record represents one column in the LCD display. Each bit in each character represents one dot in a column of the display.

Note: the last 3 columns of the LCD are not used because of a bug in the HHC's cursor handling code. Although SKETCH makes the cursor invisible, the HHC insists that the cursor is present somewhere in the LCD, and will not allow any dots to be turned on in that position. SKETCH puts the cursor in position 26 of the LCD, of which only 3 dot columns are visible, because this is the least objectionable place for the bug to appear.

The current picture file is kept in PICBUF, a "string" variable in the TSA. BUF>LCD copies it from there directly to the LCD buffer, with !DISPLAY. Editing is actually done on the data in the LCD buffer, using @DISPLAY to fetch the contents of a byte (*i.e.*, a column of dots) and !DISPLAY to store one. The contents of the LCD buffer is moved back to PICBUF just before SKETCH leaves edit mode (see EXEC.KEY).

The variable FNAME holds the name of the current file; the variable L'NAME holds the length of the name.

The variable PICSTAT holds a code indicating the current contents of the picture file buffer:

0 -> empty (no current picture).
1 -> loaded (there is a picture).
2 -> changed (picture has been edited since loaded).

The Program

The mother word is SKETCH; it calls a menu driver structured just like the one in Example 1. This calls a separate word for each function.

DRAW is the word that implements edit mode. Each of the words that puts SKETCH into edit mode does so by executing DRAW.

DRAW consists of a loop that processes one keystroke per iteration. Each keystroke is passed to EXEC.KEY for processing.

DRAW and EXEC.KEY maintain all relevent state variables on the stack. Working from the root toward the top, they are:

X X-co-ordinate of cursor (0 to 155).

Y Y-co-ordinate of cursor. This is expressed as a bit mask which may be applied to a byte in the picture file buffer representing one column in the picture. A value of 1 represents the top bit, 2 the next bit down ... 80H represents the bottom bit.

M Edit sub-mode. A value of 2 represents "delete;" 1 represents "draw;" 0 represents "neutral" (neither erase nor draw.)

In addition, WAIT.KEY receives a keystroke on the stack, and leaves a boolean indicating whether or not DRAW is to return to the function menu driver.

Notice the use of NAP in WAIT.KEY. This is how .SKETCH blinks the cursor while it is waiting for you to press a key. Each iteration of the BEGIN ... UNTIL loop toggles the cursor and does one 64-tick (0.25 second) NAP. When NAP is ended by a keystroke, rather than by timer expiration, the loop ends, the cursor is turned off, and WAIT.KEY returns control to DRAW.

Also notice the extensive use of colon definitions for common functions. For example, .FILE.NAME is an 8-word definition that displays the current file's name; it is used in several of the function words. For another example, .NOFILE does nothing but print a string constant. We could have made .NOFILE a string constant and coded ".NOFILE COUNT TYPE" for each reference to it; but that would take two extra tags in each reference, with no gain in simplicity or generality. It is usually good practice to make extensive use of colon definitions for repetitive fragments of code; it makes your programs shorter, easier to write, and easier to read.

17.3 SAMPLE 3: INFLATION CALCULATOR

The inflation calculator computes the price of an item with a specified initial value after a specified period of inflation at a constant, specified rate.

This program is interesting mainly because it illustrates some fundamental techniques for working with floating point data.

17.3.1 Use

The program compounds inflation monthly.

The program presents a menu which lets you select among the following operations:

- Specify annual rate of inflation; for example, to specify 10.5%, enter '10.5'.
- Specify time in months.
- Specify principal amount in dollars & cents; for example, to specify $36.50, enter '36.50'.
- Display rate, time, and principal amount most recently entered.
- Compute inflated amount from rate, time, and principal amount most recently entered.

To perform a computation, select "annual rate," "time in months," and "principal amount;" enter an appropriate value for each. Then select "compute."

After performing one computation, you can perform another computation by changing one or more factors and selecting "compute" again.

17.3.2 Program Design

The program has the same general structure as the programs in previous examples.

In the TSA, the three variables I.TX, I.RX and I.PX indicate whether the time, rate, and principal amount have been entered yet. The variables I.T, I.R and I.P contain the time in months, the annual rate, and the principal amount, when they have been entered. I.MR contains the monthly rate, calculated from the annual rate.

The function menu contains entries for:

```
ENTER INITIAL PRICE
ENTER ANNUAL INFLATION RATE
ENTER TIME IN MONTHS
DISPLAY INPUT
COMPUTE RESULT
```

Each of the three "enter" functions calls a word that prompts you for a value, converts it to internal form, stores it in the appropriate variable (I.T, etc.), and sets the appropriate "entered" switch (I.TX, etc.).

DISPLAY INPUT and COMPUTE RESULT perform the obvious functions.

17.4 SOURCE LISTING FOR THE ORDER ENTRY PROGRAM

```
***************************************

 * HEADER & COMMON CODE *
 * FOR SNAP DEMONSTRATION PROGRAMS *

***************************************

***** The header *****

HEX
ROMADDR HERE - TO 0

S" CoPyright companyname, inc. 1981"

ROMID T>H HERE - ALLOT
S" ORDER ENTRY DEMO"

040 2 088 TABLES TABLE.OF.TAGS
TT.ORIGIN ROMVECT T>H !
        2 ROMEXT  T>H C!
       80 CSPEED  T>H C!
LONG MAIN.MAN  \ENTRY POINT

DECIMAL

: ?EOL   ( C --- B )            \End of line character
        &CR = ;                 \routine for EXPECT

'X ?EOL == 'EOL

                                \Prompt user for a 'Y' or 'N'

: Y/N     ( --- B)
        BEGIN KEY L>U
           DOCASE
              &Y CASE 1  1 ELSE
              &N CASE 0  1
              ELSE DROP 0
           ENDCASE
        UNTIL ;
: 0!  ( Adr --- )               \Store 0 in word at Adr
        0 SWAP ! ;
: .:    ." : " ;                \Write colon-space to LCD
: PRO      ( --- )              \ Prompt user to press
        ." --->"                \ any key to proceed with
        BEGIN KEY DROP          \ instructions
              ?KEY 0=
        UNTIL ;
```



```
: LASTKEY ( --- C )             \Drains the Keyboard
        KEY                     \ buffer & returns the
        BEGIN ?KEY              \ last key in the buffer.
        WHILE DROP KEY
        REPEAT ;

    ***** Arithmetic utilities *****

: DIGIT    ( C --- [N] B )      \Convert an ASCII
        [ HEX ]7F AND &0 -      \digit to binary
                                \according to current
        DUP BASE C@ U< DUP      \numeric base (given
                                \by the value of BASE).
                                \If C is valid, N is its
        NOTIF UNDER             \binary value & B is
        THEN  ;                 \true, otherwise N is
DECIMAL                         \absent & B is false.

700TAGS : -ROT COMPILE ROT COMPILE ROT ;
    IMMEDIATE                   \Macro, stack
TABLE.OF.TAGS                   \behaviour: n1 n2 n3 -
                                \n3 n1 n2
```

ASC>N converts an unsigned number from ASCII to short binary. The ASCII number is at adr LEN. N is the result. ADR' is the adr of the next position after the last valid digit; thus all digits are valid, ADR' = ADR + L else ADR' < ADR + L. Note: this routine demonstrates the technique of using COUNT to walk along a string.

```
: ASC>N ( Adr Len -- N Adr' )
      0 -ROT 0                  \Binary value := 0
      DO COUNT DIGIT            \Convert next character
         IF ROT 10 * + SWAP     \Add to result if
         ELSE LEAVE             \valid digit
         THEN
      LOOP ;
: (D.CHK) ( W D --- W D' )      \Internal words
                                \for N>ASC conversion.
    0 MAX OVER 1- MIN ;         \Forces 0<=D <=W-1.)
: (WD.CHK) ( W D --- W' D' )    \Forces 1 <=W <=12 &
                                \0 <= D <= W-1
    SWAP 1 MAX 12 MIN SWAP (D.CHK) ;
HEX
```

Converts a long integer to ASCII. DN is the integer; D is the # of digits to put right of the decimal point. ADR LEN points to the result, which is as long as necessary to hold the number.

```
: DN>ASC(D)    ( DN D --- ADR LEN )
   12 SWAP (D.CHK) UNDER           \Check D.
   3 PICK 2SWAP DABS               \D SIGN DN.
   <#
   4 PICK 0 ?DO # LOOP             \Format decpart
   &. HOLD                         \Process decpt.
   2DUP OR
   IF #S                           \Wholepart.
   ELSE 4 PICK NOTIF # THEN        \Equivalent to
   THEN                            \0= IF # THEN
   ROT SIGN #> ROT DROP ; \DO # & SIGN.
```

Converts a long integer to ASCII. DN is the integer; W is the desired width; D is the # of digits to put right of the decimal point. ADR LEN points to the result. If too short, result is padded left with blanks; if too long, field looks like '******.**'.

```
: DN>ASC(W,D) ( DN W D --- ADR LEN )
   (WD.CHK)                        \Check W & D.
   SWAP OVER 2>R                   \DN D; D,W -> R.
   DN>ASC(D)                       \ADR LEN
   R - NEGATE ?DUP
   IF                              \ADR W-LEN
      DUP 0<
      IF                           \W<>LEN.
         DROP DUP R &* FILL        \Overflow!
         R> 2DUP + R> - 1-         \Fill with
         &. SWAP C!                \asterisks
      ELSE >R  R - DUP R>          \LEN<W;
         BLANKS 2R> DROP           \Lpad with blanks
      THEN
   ELSE 2R> DROP
   THEN ;                          \LEN=W.
DECIMAL
: .FP(N) ( FP N --- )              \Displays FP with N
                                   \digits right of the
   >R                              \decimal point.
   14 R 1+ MINUS FROUND
   F.
   R> F.EXT ;
```

****

* ORDER ENTRY APPLICATION *

****

***** CONSTANTS *****

```
120 == O.L'OUTBUF    \L'OUTPUT bfr
120 == O.L'INBUF     \L'INPUT bfr
16 == O.L'KBBUF      \L'KEYBD bfr

HEX

STRING" O.E.INV" O.INV \Inventory file name
STRING" O.E.INP" O.INP \Data file name

42 == MDM.D#        \Modem device #
46 == PTR.D#        \Printer.

2 == MDM.U#         \Modem unit #.
3 == PTR.U#         \Printer.

AREA O.TSA

O.L'OUTBUF 2+   STRING O.OUTBUF \Out bfr
O.L'INBUF  2+   STRING O.INBUF# \Ipt bfr
O.L'KBBUF       STRING O.KBBUF
4              STRING MDM.ECB    \Modem ECB
4              STRING PTR.ECB    \Print ECB
               VAR VID.ECB       \Video ECB

VAR O.CURITM    \CURRENT INVENTORY ITEM#
VAR O.CURFLD    \ADR NEXT FLD IN INBUF
VAR O.QTY       \QUANTITY OF THIS ITEM
VAR O.PRICE     \PRICE OF THIS ITEM
DVAR O.TOTAL    \TOTAL PRICE
VAR O.1STITEM   \REC# THIS ORDER'S 1ST ITEM
VAR O.NAME\ REC# THIS ORDER'S CUSTOMER NAME

\PRINTER VARIABLES
VAR LINE#     VAR PAGELEN
VAR COL#      VAR LINELEN

ENDAREA

: O.INBUF O.INBUF# 1+ ;

DECIMAL
```

***** MISCELLANEOUS UTILITIES *****

```
HEX

: .C.B.A ." CAN'T BE ATTACHED!" ;

: OPEN.L ( --- B )
   PTR.D# PTR.U# ATTACH IF \TRY ATTACH.
     PTR.ECB PTR.U# ROPEN       \TRY OPEN.
   ELSE FALSE ENDIF ;
( "OPEN" EPSON PRINTER.)

: EMIT.L ( C --- )
   1 COL# +!
```

```
5C CASE 2F ENDIF
PTR.ECB WAIT   PTR.ECB 1+ C!
PTR.ECB PTR.U# ROP DROP ;
( EMIT CHARACTER TO EPSON PRINTER.)

 : TYPE.L ( A L --- )
   OVER + SWAP  ?DO I C@ EMIT.L LOOP ;
( "TYPE" A STRING TO EPSON PRINTER.)

 : CR.L    ( --- )
   &CR EMIT.L
   COL# 0!   1 LINE# +! ;

 : SPACES.L ( N --- )
   0 ?DO   &BL EMIT.L   LOOP ;

 DECIMAL
  ***** MISCELLANEOUS UTILITIES *****

 : .AT   ." @ "   ;

 : OPEN.INV  ( --- )
   O.INV COUNT OPEN NOTIF
     CR ." NO INVENTORY FILE!" ENDIF ;

 : OPEN.ORDER ( --- )O.INP COUNT 2DUP
   OPEN NOTIF MAKE DROP %TEXT GET-TYPE
   C!   ELSE 2DROP   ENDIF ;

 : PROMPT>OUTBUF ( --- B )
   O.OUTBUF COUNT SWAP OVER +\LN CL
   O.L'OUTBUF ROT -       \CL LEN-REMAIN
   &LBUF SWAP        \CL &LBUF LEN-REMAIN
   0 0 'EOL %OVR EXPECT \CL +A +L E' K M
   2DROP                \CL +A +L
   >R SWAP R CMOVE R>        \MOVE IT; +L

   O.OUTBUF +C! 1 ;          \INCR LENBYTE
( READS A STRING FROM KEYBD & APPENDS IT
TO O.OUTBUF.)

 : +OUTBUF ( C --- )
   O.OUTBUF COUNT + C! 1 O.OUTBUF +C! ;
( APPENDS A CHARACTER TO O.OUTBUF.)

 : STR>OUTBUF ( ADR LEN --- )
   O.OUTBUF COUNT SWAP OVER +\ TO-LN TO
   O.L'OUTBUF ROT -\TO TO-LN-REMAINING
   ROT MIN                \ ADR TO LN
   DUP O.OUTBUF +C! CMOVE ;\MOVE & INCR
( APPEND A STRING TO O.OUTBUF.)

 : FIND ( ADR C --- [ADR'] B )
   SWAP BEGIN               \C ADR
     2DUP C@                \C ADR C C'
     >R R =   R> 0=  OR 0=\C'=C OR NUL
```



```
     WHILE 1+ REPEAT
   SWAP OVER C@ = IF           \LEAVE ADR 1
     1   ELSE DROP 0 ENDIF ; \OR 0.
( FIND 1ST OCCURENCE OF CHAR C AT/AFTER ADR. IF
NULL IS FOUND 1ST, ADR' IS ABSENT & B IS FALSE.)

 HEX
 HERE H>T == O.HLST
   S" PRESS 'S' KEY TO START."
   S" FOR NEXT ITEM PRESS X."\UPARROW
       -2 ALLOT-V 83 C, &. C,
   S" FOR LAST ITEM PRESS X."\DOWNARROW

       -2 ALLOT-V 80 C, &. C,
   S" FOR NEXT PRICE PRESS XXX." \->
 -4 ALLOT &ESC C, ESCDC C, 82 C, &. C,
   S" FOR LAST PRICE PRESS XXX." \<-
 -4 ALLOT &ESC C, ESCDC C, 81 C, &. C,
   S" TO ORDER AN ITEM,"
   S" PRESS ENTER,"
   S" THEN TYPE QUANTITY,"
   S" AND PRESS ENTER AGAIN."
   S" TO END ORDER,"
   S" PRESS 'E' KEY."
   0 C,
 DECIMAL

 : O.HELP  ( --- )
   O.HLST BEGIN
     DUP C@ NOTIF DROP O.HLST ENDIF
     COUNT 2DUP CR TYPE
     +
     ?KEY IF KEY L>U &S = ELSE 0 ENDIF
     UNTIL   DROP ;
( ROTATE THROUGH O.HLST UNTIL USER PRESSES 'S'.)

  ***** TAKE AN ORDER. *****

 : INIT.ORDER ( --- )
   0 0 O.TOTAL 2!
   O.CURITM 0!   O.CURFLD 0!
   O.INBUF# 0!
   LINE# 0!   COL# 0!
   66 PAGELEN !   85 LINELEN ! ;
( INITIALIZATION FOR AN ORDER.)

 : -FIND ( ADR C --- [ADR'] B )
   SWAP BEGIN
     2DUP C@
     >R R =   R> 0=   OR 0=
     WHILE 1- REPEAT
   SWAP OVER C@ = IF
     1   ELSE DROP 0 ENDIF ;
```

(LIKE FIND BUT SCANS RIGHT TO LEFT.)

```
: .TYPE   ." TYPE "   ;
```

```
: N>OUTBUF ( N --- )
  S>D <# # #S #> STR>OUTBUF ;
```

(CONVERTS A NUMBER TO ASCII & APPENDS IT TO O.OUTBUF.)

```
: D.$&C ( DN --- )
  <# # # &. HOLD   2DUP OR IF      \.
    #S  ENDIF  &$ HOLD #>    TYPE ; \$
```

(INTERPRETS N AS A NUMBER OF CENTS & PRINTS A PRICE IN THE FORM '$DD.CC'.)

```
: PROMPT.NAME ( --- )
  OPEN.ORDER
  REC-CNT O.NAME !
  0 O.OUTBUF C!    &\ +OUTBUF
  CR ." YOUR NAME: "
  PROMPT>OUTBUF DROP    &\ +OUTBUF
  CR ." HOUSE ADDRESS: "
  PROMPT>OUTBUF DROP    &\ +OUTBUF
  CR ." CITY, STATE ZIPCODE: "
  PROMPT>OUTBUF DROP
  O.OUTBUF COUNT REC-CNT INSERT DROP ;
```

(PROMPTS FOR NAME & ADDRESS AT START OF AN ORDER.)

```
HEX
```

```
: LAST.FLD ( ADR --- ADR' )
  BEGIN
    DUP 5C FIND
    WHILE SWAP DROP 1+ REPEAT ;
```

(MOVE PTR FROM 1ST PRICE FIELD TO LAST PRICE FIELD IN AN INVENTORY RECORD.)

```
: \FIND     ( ADR --- ADR' )
  5C FIND DROP 1+ ;
```

(FIND BEGINNING OF NEXT FIELD, DELIMITED BY A BACKSLASH.)

```
: FETCH.PRICE ( --- N )
  O.INBUF 1+ \FIND            \ SKIP NAME FLD
  BEGIN                       \ SEARCH QTY/$S
    DUP 7F ASC>N SWAP DROP \ AMIN AMAX-1
    1+ 7F ASC>N DROP          \ A-MIN MAX
    DUP 0 > SWAP              \ A-MIN ?MAX MAX
    O.QTY @ <                 \ A-MIN ?MAX ?<
  AND WHILE                   \UNTIL MAX>QTY
    \FIND REPEAT              \ ADV FIELD.
  DUP 5C FIND IF              \ POINT TO '\'
    SWAP DROP  ELSE             \OR NULL
    0 FIND DROP ENDIF           \DELIMITER.
```



```
  2- DUP 2 ASC>N DROP     \A-CC CC
  SWAP 1- DUP             \ CC C-$$ C-$$
  20 -FIND DROP 1+        \ CC C-$$ A-$$
  DUP C@ 24 = IF 1+ ENDIF \ SKIP '$'.
  SWAP OVER - ASC>N DROP \ CC $$
  64 * + ;                \ TOTAL IN CC
```

(LOCATES 1ST PRICE FIELD W/MAX-QTY>= O.QTY & RETURNS ITS PRICE IN CENTS.)

```
: RECORD.ITEM ( --- )
  0 O.OUTBUF C!          \ NULL >OUTPUT
  O.CURITM @ N>OUTBUF \ITEM# >OUTPUT
  20 +OUTBUF             \ BLANK >OUTPUT
  O.QTY @ N>OUTBUF       \ QTY >OUTPUT
  20 +OUTBUF             \ BLANK >OUTPUT
  O.PRICE @ N>OUTBUF       \ UNIT PRICE
  OPEN.ORDER
  O.OUTBUF COUNT REC-CNT INSERT DROP
  OPEN.INV ;
```

(WRITES AN ORDERED ITEM TO THE ORDER FILE.)

```
DECIMAL
```

```
: WRAPMSG ( B --- )
  CR ." YOU HAVE REACHED THE "
    DUP 0< IF ." START" ELSE ." END"
      ENDIF
  CR ." OF THE INVENTORY LIST."
  CR ." THE NEXT ITEM YOU SEE"
  CR ." WILL BE THE "
    DUP 0< IF
      ." LAST." ELSE ." FIRST." ENDIF
  0 > IF
    CR ." PRESS 'E' TO END ORDER."
    ENDIF ;
```

(INVENTORY LIST WRAP MESSAGE. B=1 -> WRAP FROM END TO START; B=-1 -> WRAP FROM BEGINNING TO END.)

```
: INVREAD ( --- )
  OPEN.INV                 \OPEN INV. FILE.
  O.INBUF O.L'INBUF        \READ CUR. RCD.
  O.CURITM @ READ DROP
  O.INBUF + 0 SWAP C!  \PUT NUL AT END.
  O.INBUF O.CURFLD ! ; \SET FIELD ADR.
```

(READS CURRENT RECORD FROM INVENTORY FILE INTO INBUF.)

```
: INVDISP ( --- )
  O.CURFLD @ DUP &\      \A A '\'
  FIND NOTIF             \A A' (A('\')).
  DUP 0 FIND DROP ENDIF
  OVER -                 \A L
  CR TYPE ;              \TYPE IT.
```

(DISPLAYS CURRENT FIELD OF CURRENT RECORD.)

```
: MOVEDN ( --- )
  OPEN.INV  O.CURITM @ 1+  \ITEM #.
  DUP REC-CNT < NOTIF    \>MAX.?
    1 WRAPMSG DROP 0 ENDIF \YES; WRAP.
  O.CURITM !               \STORE #.
  INVREAD ;                \READ RCD.
```

(MOVE DOWN TO NEXT INVENTORY ITEM.)

```
: MOVEUP ( --- )
  OPEN.INV   O.CURITM @ 1- \ITEM #.
  DUP 0< IF                \<0?
    -1 WRAPMSG DROP        \YES; WRAP.
    DROP REC-CNT 1-
    ENDIF
  O.CURITM !               \STORE #.
  INVREAD ;                \READ RCD.
```

(MOVE UP TO PREVIOUS INVENTORY ITEM.)

```
: MOVEL ( --- )
  O.CURFLD @ 2-     \A(LAST CHR) PRV FLD.
  DUP C@ NOTIF      \IS THERE ONE?
    DROP            \NO.
    O.INBUF 0 FIND DROP 1- \C(RCD)-1.
    ENDIF
  &\ -FIND IF 1+  \GO BACK TO '\'+1;
  ELSE O.INBUF ENDIF \TO START IF NONE.
  O.CURFLD ! ;      \SET A(FIELD).
```

(MOVE LEFT 1 FIELD IN CURRENT INVENTORY RECORD.)

```
: MOVER ( --- )
  O.CURFLD @ &\ FIND IF
  1+                        \A(NEXT FIELD).
  ELSE O.INBUF ENDIF        \NONE; USE STRT
  O.CURFLD ! ;              \SET A(FIELD).
```

(MOVE RIGHT 1 FIELD IN CURRENT INVENTORY RECORD.)

```
: ORDER.ITEM ( C --- )
  &LBUF C!                    \SAVE 1ST CH
  CR ." ENTER QUANTITY: "
  &LBUF 10 OVER C@ IF 1 2 ELSE 0 0 ENDIF
  'EOL %OVR EXPECT 2DROP  \READ ++
  2DUP + >R ASC>N        \ CVT->NUM
  R> = IF                      \VALID?
    DUP O.QTY !                \YES; SAVE
    FETCH.PRICE DUP O.PRICE ! \QTY,
    S>D ROT SD*                \PRICE,
    2DUP O.TOTAL 2@
    D+ O.TOTAL 2!              \CUM. TOT;
    CR ." EXTENDED PRICE IS " D.$&C
    RECORD.ITEM MOVEDN         \RCD & ADV
  ELSE
    CR ." THAT'S NOT A VALID NUMBER."
    ENDIF ;
```

(PROMPTS FOR QTY; SAVES QTY, DISCOUNTED PRICE CUMULATIVE TOTAL; DISPLAYS EXTENDED PRICE; RECORDS ITEM IN FILE.)

```
: INVKEY ( --- B )
  KEY L>U
  IF.ITS &<   MOVEL 1   ENDIF \MOVE LEFT
  IF.ITS &>   MOVER 1   ENDIF \MOVE RIGHT
  IF.ITS &U   MOVEUP 1  ENDIF \MOVE UP
  IF.ITS &D   MOVEDN 1  ENDIF \MOVE DOWN
  IF.ITS &CR
    0 ORDER.ITEM 1 ENDIF        \ENTER
  DUP GETDIGIT IF               \A DIGIT.
    DROP DUP ORDER.ITEM 1 ENDIF
  IF.ITS %E   0           ENDIF \END.
  SWAPDROP ;
```

(PROCESS USER RESPONSE.)

***** PRINT ORDER SUMMARY *****

```
: RETRIEVE.ITEM ( A L --- A L' )
  2DUP + 0 SWAP C!        \PUT NULL AFTER RC
  ASC>N                   \ITEM#: N CEIL
  SWAP O.CURITM !         \SAVE ITEM#; CEIL
  1+ O.L'INBUF ASC>N      \QTY: N CEIL
  SWAP O.QTY !            \SAVE QTY; CEIL
  1+ O.L'INBUF ASC>N      \PRICE (CC):N CEIL
  DROP O.PRICE !          \SAVE PRICE.
  OPEN.INV                \OPEN INV. FILE.
  O.INBUF DUP O.L'INBUF \INBFR: A A L'
  O.CURITM @ READ 2DROP \ITEM RCD: A
  DUP \FIND 1- OVER - ;
```

(RETRIEVE ITEM: GIVEN ORDER RECORD IN INBUF, SETS O.CURITM, O.QTY & O.PRICE & RETURNS ADR/LEN OF NAME OF ITEM [WHICH IS IN INBUF].)

```
: SUMMARY.NAME  ( A L --- )
  1- SWAP 1+ SWAP   TYPE.L CR.L ;
```

(PRINT A NAME RECORD IN THE QUEUE SUMMARY.)

```
: SUMMARY.ITEM ( A L --- )
  RETRIEVE.ITEM   \FORMAT RECORD.
  O.QTY @ S>D 5 0 DN>ASC(W,D) 1-
  TYPE.L   2 SPACES.L
  >R R TYPE.L
  28 R> - 1 MAX SPACES.L
  O.PRICE @ S>D  6 2 DN>ASC(W,D) TYPE.L
  3 SPACES.L
```

```
    O.PRICE @ S>D    O.QTY @    SD*
    8 2 DN>ASC(W.D) TYPE.L CR.L ;
```
(PRINT AN ITEM RECORD IN THE QUEUE SUMMARY.)

```
HEX

: PRINT.SUMMARY ( --- )
    INIT.ORDER
    80 PTR.ECB C!
    OPEN.ORDER CR.L
    REC-CNT 0 ?DO
       OPEN.ORDER
       O.INBUF DUP O.L'INBUF I READ DROP
       OVER C@ &\ =
       IF SUMMARY.NAME
       ELSE SUMMARY.ITEM
       ENDIF
    LOOP
    CR ;
```
(PRINT A SUMMARY OF THE ORDER QUEUE.)

```
DECIMAL

: PROMPT.ORDER ( --- )
  OPEN.ORDER
  REC-CNT O.1STITEM ! \START WRITE HERE
  INVREAD             \READ 1ST INV RCD
  BEGIN   INVDISP INVKEY   WHILE REPEAT ;
```
(PROMPT USER FOR HIS ORDER; CONTINUE UNTIL INVKEY RETURNS 0, INDICATING LAST ITEM ENTERED.)

```
: REVIEW.ORDER ( --- )
  CR ." SUMMARY OF YOUR ORDER:"
  OPEN.ORDER
  REC-CNT O.1STITEM @ ?DO \FOR EACH RCD
    OPEN.ORDER
    O.INBUF DUP O.L'INBUF  \READ,
    I READ DROP            \TYPE NAME,
    RETRIEVE.ITEM CR TYPE  \GET DATA,
    .: SPACE O.QTY @ . .AT \TYPE QTY,
    O.PRICE @ S>D D.$&C .: \PRICE, ':'
    O.PRICE @ S>D          \EXTENDED
    O.QTY @ SD* D.$&C      \PRICE.
  LOOP ;
```
(REVIEWS THE ORDER AFTER LAST ITEM HAS BEEN ENTERED)

```
: SHOW.TOTALS ( --- )
  CR ." TOTAL IS "          O.TOTAL 2@ D.$&C
  O.TOTAL 2@ 6 SD* 100 UM/ SWAP DROP S>D
  CR ." SALES TAX @ 6% IS "   2DUP D.$&C
  O.TOTAL 2@ D+
  CR ." TOTAL + TAX IS " D.$&C ;
```
(SHOWS TOTAL, TAX, & TOTAL + TAX.)


```
: ACCEPT.ORDER? ( --- )
  CR ." TAKE ORDER AS SHOWN (Y/N)?"
  Y/N IF
     CR ." THANK YOU FOR YOUR ORDER!"
  ELSE
     CR ." CANCELLING ORDER."
     OPEN.ORDER  REC-CNT
     BEGIN
       DUP O.NAME @ < NOT
     WHILE
       DUP DELETE DROP
       1-
     REPEAT
     DROP
  ENDIF    ;
```
(ASK USER IF THE ORDER JUST ENTERED SHOULD BE ACCEPTED.)

***** PRINT AN ORDER *****

```
: DN>$&C ( DN L --- A L' )
  >R                          \DN; R= W
  <# # # &. HOLD              \FMT .CC
    2DUP OR IF #S ENDIF       \FMT $$$$
    &$ HOLD #>                \FMT '$'; A L
  R> OVER - >R                \N'BLNKS TO ADD
  R 0 > IF
    R + SWAP   R - SWAP        \A-N L+N
    OVER R> &BL FILL           \BLANK FILL.
  ELSE R> DROP ENDIF ;
```
(GIVEN A DOUBLE NUMBER & DESIRED LEN, FORMATS THE NUMBER AS $NNNN.NN & RETURNS ADR/LEN. NOTE, L' WILL BE LONGER THAN L IFF DN WON'T FIT IN L POSITIONS.)

```
: SKIP.TO.COLUMN ( N --- )
  DUP COL# @ > IF          \ANY WORK TO DO?
    DUP LINELEN @ >          \YES; TOO MUCH?
    IF DROP CR.L              \YES; DO CR.
    ELSE                      \NO; DO
      COL# @                  \N-COLNUM
      DO &BL EMIT.L LOOP      \SPACES.
    ENDIF
  ELSE DROP                  \NO WORK TO DO.
  ENDIF ;
```
(SKIP TO COLUMN N ON EPSON PRINTER.)

```
: FORM.FEED     ( --- )
  BEGIN
    CR.L                          \DO CR'S
    LINE# @ PAGELEN @             \UNTIL END
```

```
   < WHILE REPEAT
   LINE# 0! ;                               \ OF PAGE.
```
(DO "FORM FEED" ON EPSON PRINTER BY CR'S UNTIL LINE# = PAGELEN.)

```
 : SKIP.TO.LINE
   DUP LINE# @ > IF        \ ANY WORK TO DO?
      DUP PAGELEN @ >        \ YES; TOO MUCH?
      IF DROP FORM.FEED        \ YES; FF.
      ELSE                     \ NO; EMIT
         LINE# @                \ N-LINE#
         DO CR.L LOOP           \ CR'S.
      ENDIF
   ELSE DROP                 \ NO WORK TO DO.
   ENDIF ;
```
(SKIP TO LINE N ON EPSON PRINTER.)

```
 HEX

 : PR.CUST ( --- )
      OPEN.ORDER
      O.INBUF DUP O.L'INBUF
      O.NAME @ READ DROP  \ GET NAM,ADR
      OVER + 5C SWAP C!    \ TERMINATE
      1+                   \ SKIP 1ST BACKSLASH
      3 0 DO               \ *** RISKY
         DUP \FIND
         SWAP 2DUP - 1- \ CIEL OLD CIEL-OLD
         0C SKIP.TO.COLUMN
         TYPE.L CR.L    \ LINE OF CUST
      LOOP DROP ;
```
(PRINT A CUSTOMER'S ORDER.)

```
 : RTYP.L     ( A L MAXLEN COL --- )
      SKIP.TO.COLUMN
      OVER - 0 ?DO
         &BL EMIT.L      \ PAD FIELD
      LOOP
      TYPE.L ; \ NOTE: L > MAXLEN IS PERMITTED

 DECIMAL

\ EXTENDED PRICE COLUMN
 : EX.PR
    8  72 RTYP.L ;

 HERE H>T == THE.HHC
 S" THE PANASONIC HAND HELD COMPUTER"

 HERE H>T == O.E.D.P.
 S" ORDER ENTRY DEMONSTRATION PROGRAM"

 : PR.O.1    ( --- )
   6 SKIP.TO.LINE
   12 SKIP.TO.COLUMN
      THE.HHC COUNT TYPE.L CR.L
```



```
   12 SKIP.TO.COLUMN
      O.E.D.P. COUNT TYPE.L CR.L
   13 SKIP.TO.LINE
   PR.CUST ;
```
(PRINTS HEADING & NAME ON AN ORDER FORM.)

```
 : PR.O.2  ( --- )
   24 SKIP.TO.LINE
   OPEN.ORDER
   REC-CNT O.1STITEM @ ?DO
      OPEN.ORDER
      O.INBUF DUP O.L'INBUF I READ DROP
      RETRIEVE.ITEM
         O.QTY @ S>D 5 0 DN>ASC(W,D)
         1- 5 1 RTYP.L
         O.CURITM @  230 +
         S>D 4 0 DN>ASC(W,D) 1-
         4 21 RTYP.L  OVER C@ EMIT.L
      130 SKIP.TO.COLUMN TYPE.L
      O.PRICE @ S>D
         7 DN>$&C 8 60 RTYP.L
         O.PRICE @ S>D
         O.QTY @ SD* 2DUP
         O.TOTAL 2@ D+ O.TOTAL 2!
         7 DN>$&C EX.PR CR.L
   LOOP ;
```
(PRINTS THE ITEMS-ORDERED PART OF AN ORDER FORM.)

```
 : PR.O.3    ( --- )
   58 SKIP.TO.LINE
   O.TOTAL 2@
      7 DN>$&C EX.PR
   61 SKIP.TO.LINE
   O.TOTAL 2@ 6 SD*
      100 UM/ SWAP DROP S>D
      2DUP 7 DN>$&C EX.PR
   64 SKIP.TO.LINE
   O.TOTAL 2@ D+
      7 DN>$&C EX.PR CR.L ;
```
(PRINT TOTALS PART OF AN ORDER FORM.)

```
 HEX

 : PRINT.ORDER ( --- )
      80 PTR.ECB C!
      INIT.ORDER
      PR.O.1 PR.O.2 PR.O.3
      FORM.FEED ;
```
(PRINT AN ORDER.)

```
 DECIMAL
```

```
***** MENU *****

: TAKE.ORDER    ( --- )
   INIT.ORDER PROMPT.NAME O.HELP
   PROMPT.ORDER REVIEW.ORDER SHOW.TOTALS
   ACCEPT.ORDER? ;
( TAKE AN ORDER [CALLED BY MAIN MENU].)

: O.PGMS
   IF.ITS 0    TAKE.ORDER ENDIF
   IF.ITS 1    PRINT.ORDER ENDIF
   IF.ITS 2    PRINT.SUMMARY ENDIF
   DROP FALSE ;

HERE H>T == O.MSGLST
   S" TAKE AN ORDER"
   S" PRINT LAST ORDER"
   S" PRINT SUMMARY OF ORDERS"

: O.MSG
   O.MSGLST @SA TYPE ;

: O.MSGS
   3 <IF    'X O.MSG TRUE
   ELSE    FALSE ENDIF ;

: ORDER.MENU
   INIT.ORDER          \INITIALIZATION!
   100 ?ROOM NOTIF
     BEGIN
       ." FILE SPACE LOW" CR
       ." PLEASE NOTIFY A SALESMAN" CR
     AGAIN
   ENDIF
   BEGIN
     1 'X O.PGMS
       'X O.MSGS
     MENU-DRIVER
   AGAIN ;
( WHOLE PROGRAM EXCEPT INITIALIZATION. 'FLEE'
GOES HERE, & CONTROL RETURNS HERE WHEN
'CLEAR' IS PRESSED.)

HEX

: ORDER.ENTRY
   O.TSA ?ENOUGH-ROOM
   80 PTR.ECB C!
   80 POZECB C!
   O.1STITEM 0!    O.NAME 0!
   OPEN.L
   'X ORDER.MENU FLEE
( MOTHER WORD FOR ORDER ENTRY.)

DECIMAL
```



```
****************************************
* BUILD INVENTORY FILE FOR ORDER ENTRY *
****************************************
HERE H>T == LINES
S" BINOCULAR BUMP PADS\1-5 .50
   \6-10 .40\11+ .35"
S" TIFFIN GUNS\1-3 55.00\4-8 40.00
   \9+ 30.00"
S" HYPNOGLYPHS\1-9 10.00\10+ 8.00"
0 ,

: PUT.IT.OUT    ( --- )
   O.INV COUNT MAKE ?ENOUGH-ROOM
   %TEXT GET-TYPE C!
   LINES BEGIN
     DUP COUNT REC-CNT INSERT ?ENOUGH-ROOM
     COUNT +
     DUP C@ WHILE REPEAT
   DROP ;
****************************************
* TRAILER & MAIN MENU FOR DEMO PROGRAMS *
* ORDER ENTRY *
****************************************
: ESKEY    ( N --- B )
   IF.ITS  0  PUT.IT.OUT       ENDIF
   IF.ITS  1  ORDER.ENTRY      ENDIF
   DROP FALSE ; \CONTINUE

\MENU TITLE TABLE
HERE H>T == ESMENU
   S" ORDER ENTRY CREATE FILE"
   S" ORDER ENTRY APPLICATION"

: .ES    ( N --- )    ESMENU @SA TYPE ;

: ESDSP  ( N --- )
   2 <IF    'X .ES TRUE
   ELSE     FALSE   ENDIF
   ;
               \SET UP TAG TABLE VECTORS
: MAIN.MAN    ( --- )
   CAPINIT
   BEGIN
    1 'X ESKEY
      'X ESDSP
    MENU-DRIVER
   AGAIN ;

DECIMAL
```

17.5 SOURCE LISTING FOR THE SKETCH/INFLATION PROGRAM

Header & common code for SnapFORTH demonstration programs.

***** The header *****

```
HEX

SET %FORW CLR %INT

ROMADDR HERE - TO 0

S" Copyright ...."

ROMID T>H HERE - ALLOT
S" Sketch/Inflation demo"

040 2 088 TABLES TABLE.OF.TAGS
TT.ORIGIN ROMVECT T>H !
        2 ROMEXT  T>H C!
       80 CSPEED  T>H C!

LONG MAIN.MAN  \ENTRY POINT
```

****** Miscellaneous Routines *****

```
: ?EOL  (S C ---- B )
        &CR = ;
```

(End of line character routine for EXPECT)

```
 'X ?EOL == 'EOL
```

(Prompt user for a 'Y' or 'N')

```
 : Y/N    ( ---- B)
        BEGIN KEY L>U
           DOCASE
             &Y CASE 1 1 ELSE
             &N CASE 0 1
             ELSE DROP 0
           ENDCASE
        UNTIL ;
 : 0! ( Adr ---- )
```

(Store 0 in word at Adr)

```
        0 SWAP ! ;

 : .:    ." : "   ;
```

(Write colon-space to LCD)

```
 : PRO      ( ---- )
        ." -->"
        BEGIN KEY DROP
           ?KEY 0=
        UNTIL ;
```

(Prompt user to press any key to proceed with instructions)

```
 : LASTKEY ( ---- C )
        KEY
        BEGIN ?KEY
        WHILE DROP KEY
        REPEAT ;
```

(Drains the Keyboard buffer & returns the last key in the buffer.)

***** Arithmetic utilities *****

```
 : DIGIT    ( C ---- [N] B )
        [ HEX ]7F AND   &0 -
        DUP BASE C@ U< DUP
        NOTIF UNDER
        THEN ;
 DECIMAL
```

(Convert an ASCII digit to binary according to the current numeric base given by the value of BASE). If C is valid, N is its binary value & B is true, otherwise N is absent & B is false.)

```
 700TAGS : -ROT COMPILE ROT COMPILE ROT ;
     IMMEDIATE              \Macro, stack behaviour:
 TABLE.OF.TAGS              \n1 n2 n3 - n3 n1 n2
```

(ASC>N converts an unsigned number from ASCII to short binary. The ASCII number is at adr LEN. N is the result. ADR' is the adr of the next position ofter the last valid digit; thus is all digits are valid, ADR'=ADR+L else ADR'<ADR+L Note: This routine demonstrates the technique of using COUNT to walk along a string.)

```
 : ASC>N ( Adr Len -- N Adr' )
        0 -ROT 0
```

(Binary value := 0)

```
        DO COUNT DIGIT
```

(Convert next character)

```
           IF ROT 10 * + SWAP
```

(Add to result if valid digit)

```
           ELSE LEAVE
           THEN
        LOOP ;

 : (D.CHK)  ( W D ---- W D' )
```

(Internal words for N>ASC conversion.)

```
        0 MAX OVER 1- MIN ;
```

(Forces 0 <= D <= W-1.)

```
 : (WD.CHK)  ( W D ---- W' D' )
```

(Forces 1 <= W <= 12 & 0 <= D <= W-1)

```
     SWAP 1 MAX 12 MIN   SWAP  (D.CHK) ;

 HEX
```

(Converts a long integer to ASCII. DN is the integer; D is the #

of digits to put right of the decimal point. ADR LEN points to the result, which is as long as necessary to hold the number.)

```
: DN>ASC(D)     ( DN D ---- ADR LEN )
  12 SWAP (D.CHK) UNDER          \Check D.
  3 PICK 2SWAP DABS              \D SIGN DN.
  <#
  4 PICK 0 ?DO # LOOP            \Format decpart
  &. HOLD                        \Process decpt.
  2DUP OR
  IF #S                          \Wholepart.
  ELSE 4 PICK NOTIF # THEN       \Equivalent to
  THEN                           \0= IF # THEN
  ROT SIGN #> ROT DROP ;         \DO # & SIGN.
```

(Convert a long integer to ASCII. DN is the integer; W is the desired width; D is the # of digits to put right of the decimal point. ADR LEN points to result. If too short, result is padded left with blanks; If too long, field looks like '*****.**'.)

```
: DN>ASC(W,D) ( DN W D ---- ADR LEN )
  (WD.CHK)                       \Check W & D.
  SWAP   OVER 2>R                \DN D; D,W->R.
  DN>ASC(D)                      \ADR LEN
  R - NEGATE ?DUP
  IF                             \ADR W-LEN
     DUP 0<
     IF                          \W<>LEN.
        DROP DUP R &* FILL       \Overflow! Fill
        R> 2DUP + R> - 1-        \with asterisks
        &. SWAP C!
     ELSE >R  R - DUP R>         \LEN<W;
        BLANKS 2R> DROP          \Lpad with blanks
     THEN
  ELSE 2R> DROP
  THEN ;                         \LEN=W.
DECIMAL

: .FP(N)     (S FP N ---- )      \Displays FP with N
        >R                       \digits right of the
        14 R 1+ MINUS FROUND     \decimal point
        F.
        R> F.EXT ;
```

`*************************************`

* 'SKETCH' LCD GRAPHICS PROGRAM *

`*************************************`

***** Constants *****

```
156 == L'LCD                     \L'LCD in dot columns.
 26 == L'FNAME                   \L'FILE name buffer.
```


***** Variables *****

```
AREA SK.AREA
    L'LCD 1+    STRING PICBUF  \Picture buffer.
    L'FNAME 1+ STRING FNAME  \Filename buffer.
    VAR  L'NAME          \L'INPUT filename.
    CVAR PICSTAT         \PICBUF status:
                         \ 0 - Empty
                         \ 1 - Loaded
                         \ 3 - Changed since loaded
    CVAR PICCOL          \ Temp.stor. an LCD column.
ENDAREA

: SK.INIT    ( ---- )          \Initialize program
        PICBUF L'LCD ERASE
        FNAME L'FNAME BLANKS
        0 L'NAME  !
        0 PICSTAT C! ;

: FILE.NAME  ( ---- [A L] B )
        CR ." Enter Picture file name."
        FNAME L'FNAME               \Adr Maxlen
        L'NAME @ DUP                \A Ml Initl Initl
        'EOL  %OVR                  \A Ml Il Il Eol M
        EXPECT 2DROP                \A L
        DUP L'NAME !                \Update name len.
        DUP
        IF 1                        \A L 1
        ELSE 2DROP 0
        THEN ;
```

(Display last-entered file name & prompt used for new file-name)

```
: .FILE.NAME ( ---- )
        &" EMIT
        FNAME L'NAME @ TYPE
        &" EMIT SPACE ;
```

(Type filename enclosed in quotes)

```
: .NOFILE    ." No file Present" ;

: .NOFIND    ." Can't find that file" ;

: @DISPLAY ( CN ---- F )
        DSPLY + PICCOL 1 DMOVE
        PICCOL C@ ;
```

(Stack col. image F from LCD column buffer.)

```
: !DISPLAY ( F CN ---- )
        SWAP PICCOL C!
        DSPLY + PICCOL SWAP 1 DMOVE ;
```

(Put column image F into LVD column buffer.)

```
: BUF>LCD  ( ---- )
        PICBUF DSPLY L'LCD DMOVE ;
```

(Refresh LCD from picture buffer.)

```
: LCD>BUF   ( --- )
        DSPLY PICBUF L'LCD DMOVE ;
```
(Refresh picture buffer from LCD)

```
: ERASE.LCD ( --- )
        PICBUF L'LCD ERASE BUF>LCD ;
```
(Erase the buffer & clear the LCD)

Cursor is leaving [X,Y]. Set the corresponding LCD bit to:

0 : Neutral—>No change
1 : Insert—>1
2 : Erase—>0

```
: LEAVE.DOT ( X Y MODE -- X Y MODE )
        >R R           \If mode is zero, do nothing
        IF OVER @DISPLAY OVER\ X Y Colimage Y
           R 1-     \ Test whether mode is 1 or 2
           IF [ HEX ] FFFF XOR  \2 = Erase
              AND
           ELSE OR                 \1 = Insert
           THEN 3 PICK !DISPLAY \Update LCD
        THEN R> ; \Push Mode back to parameter stack
DECIMAL
```

(Blink cursor dot until a keystroke is received. Then leave display in its original state.)

```
: WAIT.KEY   (X Y MODE --- X Y MODE KEY)
        3 PICK @DISPLAY            \X  Y M Col
        DUP >R                     \Rpush Col
        BEGIN 3 PICK XOR           \Flip crsr bit
           DUP 5 PICK !DISPLAY \Update LCD.
           64 0 NAP                \Wait a while.
        UNTIL
        UNDER R>                   \X' Y M K C
        5 PICK !DISPLAY ;          \Restore LCD.
```

```
: MC.ROTX    ( X Y M --- X' Y M )
        L'LCD 1- @DISPLAY          \Save last column.
        DSPLY DUP 1+   L'LCD 1-
        DMOVE                      \ Move the rest.
        0 !DISPLAY                 \Set 1st column.
        ROT 1+ L'LCD MOD ROT ROT ; \Rot crsr.
```
(Rotate display right one column)

```
: MC.ROT     ( X Y M --- X' Y M )
        BEGIN MC.ROTX
              64 0 NAP
        UNTIL DROP ;
```
(Rotates display until key hit)

Given a keystroke, update the program's state [except for display update, which is done by wait.key]. B= TRUE -> end sketch mode. First, define keyboard codes for 'Up', 'Down' etc:


```
HEX 07 == %R      \Rotate key
    80 == %U      \Up key
    81 == %<      \Left
    82 == %>      \Right
    83 == %D      \Down
    84 == %I      \Insert
    85 == %k      \Delete
DECIMAL
: EXEC.KEY ( X Y M KEY --- X' Y' M' B )
  DOCASE ( KEY )
  %< CASE  LEAVE.DOT
           ROT 1- 0 MAX ROT ROT 0
              ELSE
  %> CASE  LEAVE.DOT
           ROT 1+ L'LCD 1- MIN ROT ROT 0
              ELSE
  %U CASE  LEAVE.DOT
           SWAP 2 / 1 MAX SWAP 0
              ELSE
  %D CASE  LEAVE.DOT
           SWAP 2* 128 MIN SWAP 0
              ELSE
  &CR CASE DROP 0 0
              ELSE   \Neut.
  %I CASE  DROP 1 0
              ELSE
  %K CASE  DROP 2 0
              ELSE
  %R CASE  MC.ROT 0
              ELSE
  L>U  &E CASE  LCD>BUF 2DROP DROP 1 \Exit
  ELSE     DROP 0        \Unknown command
 ENDCASE ;
```

There's a litte bug in the HHC software; when you execute 'STOP.CURSOR', the cursor disappears, but the field where the cursor used to be keeps blinking. Therefore we must move the cursor out of the display when we want it to disappear:

```
: DRAW      ( --- )
   LCD.CR BUF>LCD STOP.CURSOR
   26 BUFPOSN C!            \Fix the buf
   L'LCD 2/    16 1         \Initial XY Mode
   BEGIN WAIT.KEY           \Wait for key
         EXEC.KEY           \Process the key
   UNTIL                    \'E' key ends ftn.
   LCD.CR START.CURSOR ;
```

***** Picture status checkers *****

```
HEX
```

```
: ?PICLOAD ( --- B )
        PICSTAT C@ 1 AND ;
```
(Determines if a picture is loaded)

```
: !PICMOD ( --- )
        PICSTAT C@ 3 OR PICSTAT C! ;
```
(Flags picture buffer as loaded & modified.)

```
: !PICUNMOD ( --- )
        PICSTAT C@ FD AND PICSTAT C! ;
```
(Flags picture as unmodified.)

```
: !PICUNLOAD ( --- )
        PICSTAT C@ FC AND PICSTAT C! ;
```
(Flags picture as unloaded & unmodified.)

```
DECIMAL

: CONT.PIC ( --- )
        ?PICLOAD
        NOTIF CR .NOFILE
              EXIT
        THEN !PICMOD  DRAW ;
```
(Continue drawing a picture already in the buffer.)

```
: SAVE.PIC ( --- )
        ?PICLOAD
        NOTIF CR .NOFILE
              EXIT
        THEN !PICUNMOD
        0 DELETE DROP
        PICBUF L'LCD 0 INSERT    DROP
        CR .FILE.NAME ." Saved" ;
```
(Save a picture from the LCD buffer to the current file. This is cassed by '?SAVEFIRST' as well as by the menu driver.)

```
: ?SAVEFIRST ( --- )
        PICSTAT C@ 2 AND
        IF CR ." Save "
          .FILE.NAME ." first?" Y/N
            IF SAVE.PIC
            THEN
        THEN !PICUNLOAD ;
```
(Prompt user to save current picture if it is flagged as modified. Unload the file even if it doesn't save, since its caller is going to destroy the contents of the file name buffer.)

```
: GET.PIC    ( --- )          \ Get a picture file & read
  ?SAVEFIRST !PICUNLOAD       \ the picture into
                                the buffer:
  FILE.NAME                   \ Read filename.
  IF OPEN                     \ Try to open.
     IF PICBUF                \ Read REC#0,
        L'LCD 1+ 0            \ L'LCD + 1.
        READ
        IF L'LCD =
           IF !PICMOD DRAW\If LEN = LCD it's ok.
Following are error conditions.
           ELSE CR ." That isn't a
             picture file"
           THEN                  \ REC#0 LEN wrong
        ELSE CR ." That file is empty"
        THEN                     \ Read failed.
     ELSE CR .NOFIND
     THEN                        \ Open failed.
  THEN ;

: MAKE.PIC ( --- )          \ Make a new picture file
  ?SAVEFIRST !PICUNLOAD\(should not already exist).
  FILE.NAME                   \ Get fname.
  IF 2DUP OPEN                \ Shouldn't find it.
     NOTIF MAKE               \ Should make it.
      IF PICBUF L'LCD 0 INSERT \Enuf space?
         IF !PICMOD ERASE.LCD DRAW \Yes.
         ELSE CR ." No room to
           make file"
              CFILE DELETE-FILE
         THEN
      ELSE CR ." Can't make file"
      THEN
     ELSE CR ." File already exists"
          2DROP
     THEN
  THEN ;

: DEL.PIC   ( --- )     \Delete a picture file

  ?SAVEFIRST
  FILE.NAME
  IF OPEN                     \ Ask name.
     IF !PICUNLOAD            \ Try open
        CFILE DELETE-FILE \ It worked
        CR .FILE.NAME ." Deleted"
     ELSE CR .NOFIND
     THEN
  THEN ;
```

***** Menu *****

```
HERE H>T == SK.MLIST
        S" Make new picture"
        S" Modify existing picture"
        S" Continue w/ same picture"
        S" Save the picture"
        S" Delete a saved picture"

: SK.KEY
     IF.ITS 0   MAKE.PIC        THEN
     IF.ITS 1   GET.PIC         THEN
     IF.ITS 2   CONT.PIC        THEN
```

```
        IF.ITS 3   SAVE.PIC          THEN
        IF.ITS 4   DEL.PIC           THEN
        DROP FALSE ;

: SK.DS    ( N --- )
        SK.MLIST @SA TYPE ;

: SK.MTXT ( N --- )
        5 < DUP IF 'X SK.DS SWAP THEN ;

: SK.MENU ( --- ) \Everything except initialization.
        SK.INIT
        BEGIN 1 'X SK.KEY
              ' 'X SK.MTXT
              MENU-DRIVER
        REPEAT ;

: SKETCH
        SK.AREA ?ENOUGH-ROOM
        'X SK.MENU FLEE ;
```

****

* Inflation calculator demo program *

****

***** Constants *****

```
DECIMAL
HERE H>T == FP1200 1200. F,
HERE H>T == FP0       0. F,
HERE H>T == FP1       1. F,
```

***** TSA *****

```
AREA I.TSA
  CVAR    I.TX       \Time entered?
  CVAR    I.RX       \Rate entered?
  CVAR    I.PX       \Price entered?
   VAR    I.T        \Time (months)
  FVAR    I.R        \Annual rate.
  FVAR    I.MR       \Comp. monthly %
  FVAR    I.P        \Price
ENDAREA
```

***** Miscellaneous help words *****

```
: READINBUF ( --- ADR LEN )
        &LBUF %LLEN 0 0 'EOL %OVR
           EXPECT 2DROP ;
```

(Read a line from keyboard)

```
: .ENTER
        ." Enter " ;
```



```
: .--
        ." --" ;

: I.INVKEY  ( --- )
        CR ." That's an invalid key." ;

: I.ERR       ( --- )
        CR ." That isn't valid." ;
```

***** Menu options & associated words *****

```
: I.REVU ( --- )
        CR ." Price = " I.PX C@
        IF    &$ EMIT I.P F@ 2 .FP(N)
        ELSE .--
        THEN CR ." Rate = " I.RX C@
        IF    I.R F@ 2 .FP(N)    &% EMIT
        ELSE .--
        THEN CR ." Time = " I.TX C@
        IF    I.T @ . ." Months"
        ELSE .--
        THEN ;
```

(Reviews factors set so far)

```
: I.TIME ( --- ) \Accepts & validates time in
                 \months
    .: SPACE
    READINBUF ?DUP               \Read Keyboard
    NOTIF DROP
          EXIT
    THEN                         \LL- > \QUIT
    2DUP + >R   ASC>N R>         \N END' END
    <IF I.ERR DROP
         EXIT
    THEN                         \Valid?
    DUP 0<                       \N< 0?
    IF I.ERR DROP
        EXIT
    THEN
    DUP 9999 >
    IF I.ERR DROP
        EXIT
    THEN I.T !    1 I.TX C! ;
```

(Accepts and validates time in months)

```
: H.N.B.E. ." Hasn't been entered." ;

: I.DATA? ( --- B )
    I.PX C@  I.RX C@  I.TX C@ AND AND
    DUP NOTIF
        CR ." Can't compute result"
        CR ." Because"
        I.PX C@ NOTIF CR ." Price "
           H.N.B.E. THEN
        I.RX C@ NOTIF CR ." rate "
```

```
      H.N.B.E. THEN
   I.TX C@ NOTIF CR ." time "
      H.N.B.E. THEN
THEN ;
```

(Ensures that all 3 factors have been entered before computing inflation)

```
: I.PRICE ( --- )
  ." : $" READINBUF              \Read price
  ?DUP NOTIF DROP EXIT THEN      \Null?
  2DUP + >R                      \CEIL -> R.
  ASC>FP DROP                    \CVT -> FP.
  R>
  <IF I.ERR FDROP                \Err, data.
  ELSE FDUP FP0 F@ F<
     IF I.ERR FDROP              \Err,< 0.
     ELSE I.P F! 1 I.PX C!       \Good.
     THEN
  THEN ;
```

(Reads and stores prices)

```
: I.RATE ( --- )
      .: READINBUF
      ?DUP NOTIF DROP EXIT THEN
      2DUP + >R
      ASC>FP DROP
      R>
      <IF I.ERR FDROP
      ELSE FDUP I.R F! FP1200 F@ F/
         I.MR F!   1 I.RX C!
      THEN ;
```

(Accepts & saves annual interest rate; computes & saves monthly compound interest rate.)

```
: I.COMP ( --- )
      I.DATA?
      IF CR ." Final Price = $"
         I.MR F@ FP1 F@ F+   I.P F@
         I.T @ 0
         DO FOVER F*
         LOOP
         2 .FP(N) SPACE FDROP
         LASTKEY DROP
      THEN ;
```

(Computes & prints inflated price)

***** MENU *****

```
: I.MKEY ( N --- 0 )
      IF.ITS 0   I.PRICE   THEN
      IF.ITS 1   I.RATE    THEN
      IF.ITS 2   I.TIME    THEN
      IF.ITS 3   I.REVU    THEN
      IF.ITS 4   I.COMP    THEN
      DROP FALSE ;
```



```
HERE H>T == I.MLIST
      S" Enter initial Price"
      S" Enter annual infl. rate"
      S" Enter time in months"
      S" Display input"
      S" Compute result"

: I.DS  ( N --- )
      I.MLIST @SA TYPE ;

: I.MTXT ( N --- )
      5 < DUP IF 'X I.DS SWAP THEN ;

: I.INIT  ( --- )
      FLAME.ON
      0 I.PX C! 0 I.TX C!  0 I.RX C! ;
```

(Initializes the program)

```
: I.MENU ( --- )
      I.INIT
      BEGIN 1 'X I.MKEY
              'X I.MTXT
           MENU-DRIVER
      AGAIN ;

: INFLATION ( --- )          \ Mother word
      I.TSA ?ENOUGH-ROOM
      'X I.MENU SOFTAG !
      I.MENU ;
```

****

* Trailer & main menu for demonstration programs *

* SKETCH - INFLATION *

****

```
HEX

: ESKEY   ( N --- B )
      IF.ITS  0  INFLATION      THEN
      IF.ITS  1  SKETCH         THEN
      DROP FALSE ;              \Continue

DECIMAL
```

\MENU TITLE TABLE

```
HERE H>T == ESMENU
      S" Inflation calculator"   \0
      S" The electric sketchPad"

: .ES    ( N --- )   ESMENU @SA TYPE ;

: ESDSP ( N --- )
      2 <IF   'X .ES TRUE
      ELSE    FALSE   THEN ;
```

```
: MAIN.MAN    ( --- )
        CAPINIT
        BEGIN 1 'X ESKEY
                'X ESDSP
              MENU-DRIVER
        AGAIN ;
```

(First tag—entry point)

GLOSSARIES

The following glossaries are designed for reference. They will not provide an introduction to SnapFORTH, even if you are fluent in FORTH. Read the preceding chapters and the SnapFORTH Tutorial Manual for introductory material before you try to use the glossaries.

Two glossaries are provided;

- A glossary of SnapFORTH capsule words that require the SnapFORTH capsule to execute.
- A glossary of HHC words you can use in a stand-alone SnapFORTH program.

The conventions used in these glossaries are described in the following section.

Glossary Format

Here is an example of a glossary entry with a description of each part:

OVER (X Y --- X Y X) NOTE

Copies the second word on the stack to the top of the stack.

OVER

The word being described.

(X Y --- X Y X)

Shows stack input to the left of the hyphens, and stack output to the right. The rightmost symbol corresponds to the topmost on the stack. This example shows that the word expects two items on the stack, and leaves three. Conventions for naming the stack items are intuitive rather than formal. Descriptive names, such as "LEN" for a length, are used where appropriate. Where no conventions are used:

A for an address.

B for a boolean (TRUE/FALSE).

C for an ASCII character.

..D for a suffix indicating a double-length (32- bit) operand, e.g. ND for a double-length integer.

FL for a byte of bit flags.

FP for a floating point number.

L for a length or byte count.

N for an integer.

U for an unsigned integer.

X,Y,Z for 16-bit quantities of unspecified type.

[ ] encloses an operand that may or may not be present. Example: '(X --- X [X])'. X may or may not be duplicated.

The sequence '(... --- [X] B)' customarily means that X is present if B is TRUE, and absent if B is FALSE.

If the same name appears on both sides of the hyphens, it represents the same value (i.e. the value is not changed). If a name appears on the left side of the hyphens, and appears on the right side with a prime after it, it represents a corresponding (but changed) value. Example:

(A L --- A L')

If two or more values of the same type appear in the same specification, they are distinguished by numbers. Example:

(N1 N2 --- N3)

NOTES

Notes explain how this word fits into the HHC system.

none

Absence of any note implies that the word is a colon definition in the HHC.

C

The word is a code definition in the HHC.

Examples: DUP, ROT.

=

The word is a symbolic constant that stacks a scalar value. It may be used inside or outside a colon definition.

Examples: &BL, the binary value of the ASCII character "blank;" %TEXT, the file type code for a text file.

Note that symbolic constants which return an address of data are tagged 'K' (for constants in ROM) or 'D' (for variables in RAM).

Vn

The word returns the address of a variable in the target address space. "n" is the variable's length in bytes. The

word may be used inside or outside a colon definition.

Example: &LBUF, an 80-byte area meant to be used as an input buffer for EXPECT.

Kn

The word returns the address of a constant in the target address space (in ROM). "n" is the constant's length in bytes. The word may be used inside or outside a colon definition.

Example: ROMADDR, the address of the beginning of the part of the HHC address space reserved for ROM capsules.

I

The word is an immediate word. It is executed at compile time whether used in a target colon definition or elsewhere. It may be used inside or outside a colon definition, but is customarily used only inside.

Examples: ';', '[','\', IF, THEN etc.

L

The word is a label. It stacks an address in the target address space at compile time. The address is most often the entry point to a subroutine that is called from low-level words, using the standard 6502 calling linkage rather than the SNAP (high-level) linkage.

The word may be used inside or outside a colon definition, but is customarily used only outside, in connection with a code-word definition.

DEF

The word is a defining word. It creates a new word in the target vocabulary. Examples: ':', 'STRING'.

NOTES

There is something else to note about this word. See the following description.

On the second and following lines of the glossary entry is a description of the word.

Many words in the glossary begin with non-alphabetic characters; these words are listed in the following order:

! # % & ' (*
+ , - . / (digits)
< = > ? @ [\ ^]
(from left to right)

GLOSSARY OF SnapFORTH CAPSULE WORDS

SnapFORTH Capsule Words

Dictionary variables (using TO)

NOTE:

All the variables below, set up and used by the SNAP capsule, are of a special type called "TO" variables. Fetch (@) and Store (!) are not necessary. The name of the variable returns the contents rather than the address, and a statement like 45 TO MYTOVAR stores 45 in "TO" variable MYTOVAR.

These variables are declared with VARIABLE rather than VAR, and the declaration need not be in an AREA. Of course these variables may not be used outside the dictionary. They are useful for temporary compiler extension routines that will not be part of your final program.

UR@ is a special constant that returns a pointer to the base of these "TO" variables. also: UR@ @ gives the address of the current tag table.

DP	Dictionary pointer (only ALLOT can change this variable).
HDP	Symbol table pointer.
HDPO	Pointer to end of symbol table.
CONTEXT	As in FORTH.
CURRENT	As in FORTH.
TIB	Address of internal input buffer. Default value is given by &LFUF. You can change TIB, but it must point into internal RAM.
TIBLEN	Length of internal input buffer. Default value is 99 decimal.
IN	Offset into TIB pointing to next input character.
LASTIN	Pointer to beginning of previous word in TIB.
STATE	Flag as in FORTH (compile or interpret mode). STATE is 0 when interpreting and C0 when compiling.
CSP	Compiler state pointer (saves stack pointer during compile).
V-LINK	Points to linked list of vocabularies.
L1	Vector to routine that moves lines from file to TIB.
L2 L3 L4	Variables that can be used by L1.
O	Offset for target compilation, difference between target and host location of program. See T>H and H>T.
MAXSZ	Maximum size of unused memory in the dictionary file.
WIDTH	Maximum number of characters in symbol names must be =< 31.
(NUM)	Vector to routine for number conversion, defaults to (NUMBER), which converts to integers.
(LIT)	Vector to routine for making literals, defaults to (LITERAL).
(QUIT)	Vector to a warm start routine, defaults to tag of ((QUIT)).
FENCE	Vector to LFA of symbol protected against "FORGET".
AREAPNT	Vector to last created "AREA".
CON-LINK	Points to linked list of predefined constants.
FWD	Points to location containing patch address of last forward reference.
%HIGH	Flag which determines if last forward is low/high part of last forward.

Compiler control bits:

SET <name> INT

Set a compiler control option.

CLR <name> INT

Reset a compiler option.

%REDEF =

When set, SnapFORTH gives a message when you redefine a name.

%INT =

When set, internal tags can be compiled.

%FORW =

When set, forward references are allowed.

%AUTOCAP =

When set, SAVESNAP generates a CAPINIT word (when needed).

Note, %AUTOCAP must be set before defining the TABLES in which your primary tag table apears.

%0OPT =

When set, compiler performs BRK optimizing.

BEEPER =

When set, beeps at end of each line when loading.

'TO' concept words:

%VAR (--- N)

Flag containing the state of "TO".
0: FROM
1: TO
-1: +TO

EXECTO (ADDR --- {N})

Runtime routine to fetch/store into 16-bit "TO" variables.

TO

Word to store into all "TO" variables.

+TO

Word to add into all "TO" variables.

FROM

Resets %VAR to fetch mode.

((

Saves state of %VAR on the return stack, and sets %VAR to zero. Useful for indexing into a table with a "TO" VARIABLE.

))

Restores state of %VAR.

VARIABLE <name> INT,DEF

Define <name> as a new dictionary variable.

'O' concept words:

H>T (Hostaddr—Targetaddr)

Converts address from host to target by adding contents of "O".

T>H (Targetaddr—Hostaddr)

Converts address from target to host by subtracting "O".

Nucleus extension words:

(Cannot be used in target programs)

U> (U1 U2 --- B)

Unsigned "greater-than" comparison operator.

0> (N --- B)

Greater-than-zero operator.

UMIN (U1 U2 --- U3)

Unsigned minimum of two 16-bit numbers.

UMAX (U1 U2 --- U3)

Unsigned maximum of two 16-bit numbers.

D. (D ---)

Double "."—print a double integer.

U. (U ---)

Unsigned "."—print an unsigned integer.

B. (B ---)

Byte "."—prints lower byte as two hex digits.

H. (U ---)

Hex "."—prints 16-bit number as four hex digits.

G. (F ---)

Free format "."—prints floating point number.

SEMIT (CHAR ---)

Safe "emit"—prints non-ASCII bytes as a dot.

STYPE (ADDR COUNT ---)

Safe "type"—same as type, but non-ASCII bytes are printed as dots.

SP! (X1 X2 ... XN ---)

Empties the parameter stack.

?S (--- N)

Returns number of 16-bit parameter stack entries.

.S (---)

Non-destructive print of parameter stack using "U.".

? (ADDR ---)

Print 16-bit contents of address using "."—same as "@ ."

C? (ADDR ---)

Print 8-bit contents of address using "."—same as "C@ ."

DUMP (ADDR ---)

Dumps 4 bytes of memory starting at address on top. Use right arrow for next byte, left arrow for previous byte, down for next four bytes up arrow for previous four bytes. Type ENTER to exit from DUMP.

Compile time extensions

GET-TYPE I

A compile time macro, see also HHC glossary

." "<text>" I

See HHC glossary, Compiles to (.").

BL (--- 20H)

Pushes ASCII 20H—same as &BL.

Dictionary words:

HERE (--- A)

Address of next free byte in dictionary.

ALLOT (N ---)

Increments the dictionary pointer DP by N bytes.

F, (F ---)

Puts a floating point number in the dictionary.

, (N ---)

Puts a 16-bit number in the dictionary

C, (N ---)

Puts a 8-bit number in the dictionary.

TAG, (TAG ---)

Puts a tag (8 or 16-bit) in the dictionary.

S" "<string>"

Begins the definition of a count format string constant (i.e. a constant consisting of a one-byte length field followed by an ASCII value). Used like this:

```
S" "TEXT OF THE MESSAGE"
```

S" first starts skipping blanks and then starts skipping quotes. The string is terminated by the next " character in the input stream. This implies that you cannot use S" to denote a zero string. Use 0 C, instead.

Used outside a colon definition, S" simply deposits a string value in the target address space. To get the address of the string (at compile time), stack HERE before defining the string. Here is an example which defines a table consisting of 3 strings, terminated by a null string, with the label C-TABLE representing the target address of the first string:

```
HERE
  S" FIRST STRING"
  S" SECOND STRING"
  S" THIRD STRING"
  0 C,  \ S" does not accept null string
 == C-TABLE
```

To define a single string constant, the word STRING" is preferred.

The leading quote is optional, but if your string includes leading blanks, you must add an extra delimitting quote:

```
S" "   Preceded by blanks..."
```

STRING" "<text>" <name>   INT,DEF

Defines a string constant. Input up to the next double quote is prefixed with a length byte and moved to virtual memory. The next word is defined to be the string's name. For example:

```
STRING" PLEASE WAIT..." WAITMSG
```

When the string's name is referenced at run time, the string's address (the address of the count byte) is placed on the stack.

String handler words:

EXPND N1 N2 C --- N3

Given N1 as source address, N2 as destination and C as separator, EXPND first skips in the source until a non-'C' is found and then will copy and expand every byte starting at address N1 to location N2 until it hits 'C' or a null in the source.
It also skips 'CR's in the source unless 'C' is a CR.

It leaves the source expanded as a string with count at location N2 and the number of bytes taken from the source on the stack. Used by LOAD to compile from PORTAWRITER files.

INCH   (--- C)

Gets the next character out of TIB and puts it on the stack.

WORD   (C ---)

Gets the next word separated by 'C' an puts it at HERE as a string with count.

-WORD   (---)

Calls WORD with a blank as delimiter.

EXPCT   (N1 N2 ---)

Given N1 as the address and N2 as the length of a buffer, EXPCT fills the buffer with whatever the user types in, like EXPECT in the nucleus. The difference is that the right arrow lets the user duplicate information from whatever was in the buffer before.EXPCT does not print a CR at the beginning.

QUERY   (---)

Uses EXPCT to fill TIB and resets IN.

Symbol table words:

Description of symbol:

	bits 76543210		
LFA		LINKL	Lower byte of link to previous symbol in vocabulary
		LINKH	Higher byte
NFA	1IF	LEN	I --- 1 If immediate word F --- 1 If symbol is/was a forward reference LEN - Total length of symbol
	0	CH1	CH1 .. CNn First n characters of symbol name
	0	: :	
	0		
	1	CHn	
PFA		TYPE	Type of symbol: 0 is TAG, 1 is = =, 2 is call, 3 .. FF user definable
		VALL	VALL --- Lower part of symbol value.
		VALH	VALH --- Higher part of symbol value.

For short-tag-symbols VALL is tag#, VALH is 00
For long-tag-symbols VALL is extension#, VALH is tag#
For = =-symbols VALL,VALH is value
For call-symbols VALL,VALH is address of routine

LFA   (A1 --- A2)

Given the PFA of a symbol, LFA will convert that address to the LFA of that symbol.

NFA   (A1 --- A2)

Given the PFA of a symbol, NFA will convert that address to the NFA of that symbol.

PFA   (A1 --- A2)

Given the LFA of a symbol, PFA will convert that address to the PFA of that symbol.

CFA   (A1 --- N2)

Given the PFA of a symbol CFA will leave the value of that symbol on the stack as follows:

Tag symbol : The address of the routine
CALL symbol : The address of the routine
= = symbol : The value of the = =

FIND (A1 A2 --- [A3] B)

Given A1 as the address of a string with count and A2 as the start of a linked list of symbols, FIND will search for a match in the list of symbols. If found, it leaves the PFA of the symbol as A3 and a TRUE, when not found FIND leaves FALSE.

-FIND (<symbol> --- [A] B)

Gets the next word out of the input buffer and searches for it first in the CONTEXT and then in the CURRENT vocabulary. Leaves the same result as FIND.

?DEF (<symbol> --- BOOL)

Takes the next word out of the input buffer and searches for it in the CONTEXT and CURRENT vocabularies.Leaves TRUE when found, and FALSE when not found.

' (<symbol> --- A) I

Gets the next word out of the input buffer and searches for it first in the CONTEXT and then in the CURRENT vocabulary.

If found, leaves the PFA of the symbol; if not found gives the error message 'CAN'T FIND'.

When ' is used inside a definition it will execute immediately and generate a literal from the result.

When you want to call this word from your program use: [COMPILE] '

'X (<symbol> --- N) I

Uses ' to find the next word in the input buffer, and leaves the tag value of the symbol on the stack as follows:

Short tag: Lower byte tag-value, upper byte=zero

Long tag: Lower byte=extension tag, upper byte= tag-value

When the symbol does not represent a tag or the symbol is not found, it will give the error message "CAN'T FIND".

When 'X is used inside a definition it will execute immediately and generate a literal from the result. When you want to call this word from your program use: [COMPILE] 'X

'V (<symbol> --- A) I

Uses ' to find the next word in the input buffer, and uses CFA to leave value of the symbol on the stack.

When 'V is used inside a definition it will execute immediately and generate a literal from the result. When you want to call this word from your program use: [COMPILE] 'V

LATEST (--- A)

Leaves the LFA of the last symbol in the CURRENT vocabulary.

LAST (--- A)

Leaves the LFA of the last symbol defined.

SMUDGE (---)

During its compilation, a ':' definition is not available by name yet (it is only accessable through "LAST") because ':' changed the symbol. SMUDGE will undo this change when the definition is finished. (Called by ";", ENDCODE and ;P) If there is any error during compilation of a definition QUIT will reset the dictionary and symbol table pointers to the state existing before the definition was entered.

ID. (A ---) INT

Given the LFA "ID." will print the symbol name.

VLIST INT

Will start a print of all the symbols in the CONTEXT vocabulary. Any keystroke shows the next symbol, LOCK followed by a keystroke scrolls the symbols. Aborted by ENTER.

VOCABULARY <name> INT

Defining word creating a new vocabulary with given name. When <name> is used it will make the new vocabulary the CONTEXT vocabulary.

FORTH

The root vocabulary on which all other vocabularies are built. "FORTH" will make this vocabulary the CONTEXT vocabulary.

ASSEMBLER

When used will make the ASSEMBLER vocabulary the CONTEXT vocabulary. ASSEMBLER is default chained TO FORTH.

FORWARD

System vocabulary where all unresolved symbols are kept. When used, it will become the CONTEXT vocabulary so VLIST will give a list of all current forward symbols.

"FORWARD" is chained to nothing, so never say: FORWARD DEFINITIONS, because then you are locked out of FORTH.

DEFINITIONS INT

Makes the CONTEXT vocabulary the CURRENT vocabulary.

CHAIN <vocname> INT

Chain the vocabulary with the name <vocname> to the CURRENT vocabulary so the first symbol in the CURRENT vocabulary will be linked to the last symbol of the vocabulary <vocname>.

FORGET <name> INT

FORGET will delete all symbols entered since <name> up to and including <name>.

When tag-symbols are forgotten, FORGET will reset the dictionary pointer, and reset tag table pointers.

When forward references are forgotten, it will reset the status of that forward symbol. When vocabularies are forgotten forget will unchain them from the vocabularies they were chained to. And so on.

In short- FORGET will reset the status of the symbol table and the dictionary (if the word you forget is a tag) to the point prior to <name>'s entry.

FORGET will not reset the numeric base, SHORT/LONG.TAGS, the values of CURRENT and CONTEXT, variables created by the user, etc. to the state where the <name> was defined. Nor does FORGET reset the length of an AREA when parts of the AREA are forgotten. When you FORGET your motherword, the automatic CAPINIT word created by TABLES is reset, and you will have to make a new CAPINIT word yourself.

(FORGET) (A ---) INT

Call FORGET with an address.

SHRNK <name1> <name2> INT

Deletes everything in the symbol table between and including <name1> and <name2>. This includes forward references and SET.CONSTANTS.

(CREATE) <name> (N1 N2 ---)

A word used to create new symbols in the symbol table. It takes N2 as the type and N1 as the value of the symbol as

discussed above. It will link this new symbol to the CURRENT vocabulary.

Number handling words:

SET.CONSTANT (N1 <name> ---) INT

A defining word used to notify the SnapFORTH compiler that a certain word stacks the constant value N, and should be compiled from references to that constant.

SET.CONSTANT lets you make your code more compact by taking advantage of tags which stack constant values instead of compiling longer references to CLIT or LIT.

For example, suppose you defined the following word:

```
: 2^10 1024 ;
```

Later you could notify the compiler that the word '2^10' stacks the constant value 1024 by executing the following code:

```
1024 SET.CONSTANT 2^10
```

Then, whenever the cross-compiler encountered the constant value 1024, it would compile the tag for '2^10' (one or two bytes, depending on whether the tag was short or long) instead of the tag for LIT (one byte) plus the constant value 1024 (two bytes).

An alternative way to achieve the same result is to define a colon definition with the same name as the constant it replaces:

```
: 1024 1024 ;
```

This alternative has the advantage of simplicity, but it has two disadvantages. First, it prevents you from giving the constant a mnemonic name. Second, it prevents you from changing the value of the constant, once it is defined, except by the extremely dirty means of defining a word like 1024 that stacks some value other than the one it appears to represent.

See also CONSTANT and '= ='.

LITERAL (N --- [N]) I

When compiling, "LITERAL" will take N from the stack and compile code into the dictionary that will put N on the stack when executed.

"LITERAL" first checks if N is available as a constant previously defined, if not it will compile a 'LIT' 'CLIT' 'CLIT2' or 'CLIT3' according to the value of N.

When LITERAL is called in immediate mode, it will not do anything.

The intended use of LITERAL is to perform compile time calculations, and compile a 16-bit or 8-bit result into the code:

example:

```
: TRY [ 3 4 * ] LITERAL ;
```

same as

```
: TRY 12 ;
```

not same as

```
: TRY 3 4 * ;
```

DLITERAL (D1 --- [D1]) I

Similar to LITERAL, but for double precision numbers. At compile time, takes a 32-bit value from the stack and compiles it into the code. At execution time, this 32-bit value is treated in whatever way its context dictates. See LITERAL for examples of use.

FLITERAL (F1 --- [F1]) I

Same as LITERAL but for floating point values.

(LITERAL) (N1/D1/F1 TYPE --- [N1/D1/F1])

Default routine of the "(LIT)" vector. Takes TYPE as indicator to what type of number is on the stack.

0 --- INTEGER
1 --- DOUBLE
2 --- FLOATING POINT

It will call the appropriate literal routine to compile the number.

It is part of the scheme to add new types of numbers to the compiler. When INTERPRET hits an unknown symbol it calls NUMBER to convert it to a number. NUMBER in its turn will call the vectored routine residing in "(NUM)" to do the conversion. This will leave the number and a type on the stack as indicator to the vectored "(LIT)" routine what type of literal to compile.

HEX (---)

Sets the system variable BASE to 10H so most number converting routines like CONVERT . .R etc. convert everything from/to hex

DECIMAL (---)

Sets BASE to decimal.

OCTAL (---)

Like 'HEX', but sets BASE in octal.

DCONVERT (A1 --- D1 A2)

Converts string at A1 (no count format) to double number. A2 is the address where conversion stopped. Example:

```
1234    DCONVERT 1234
```

A1 A2

A comma at the end of the string is optional.

FCONVERT (A1 --- F1 A2)

Works like DCONVERT, only it always converts the string at A1 into a floating point number in base ten. General format of floating point :

```
[+/-] ddd.ddddd E yyyy
```

CONVERT (A1 --- N1 A2)

Same as FCONVERT and DCONVERT, but always leaves single precision number.

(NUMBER) (A1 --- N1/D1/F1 TYPE A2)

Converts a string to a number. The type of the number depends on the format of the string. TYPE is the type of the number:

0 - Single precision number
1 - Double precision number
2 - Floating point number

Types are determined as follows:

Floating point numbers are only recognized in DECIMAL. When there's an 'E' or a '.' in the string, its type is 2. When there's a ',' in the string, its type is 1.

In all other cases, the type of the string is 0.

(NUMBER) is the default value of the (NUM) vector. When you store another routine in this vector, you can change the way in which the compiler scans numbers.

NUMBER (A1 --- N1/D1/F1 TYPE)

Takes a string with count and calls the convert routine pointed to by "(NUM)" to convert the number.

It checks the end address left by the conversion routine against the length of the string, and will generate an error message if they are inconsistant.

&^ (--- C) I

&C leaves the ASCII value of the character C on the stack: &A will leave 65

^^ (--- C) I

^C leaves the control character value of the character on the stack: e.g. ^A will leave a 1.

CONSTANT (X ---) INT

Defines a constant. Used like this:

```
80 CONSTANT RECLEN
```

RECLEN is defined as a constant with the value 80.

== (N <name> ---) DEF

Creates a new symbol and assigns the value of N to it, so that using <name> is the same as using N. Example:

```
HEX 7FFF == MAXINT
```

This does not generate a "constant".

Tag Table Words

TABLES (N1 N2 N3 <name> ---) INT

Defines a new set of tag tables with N1 as the number of short tags, N2 as the extension number where the long tags start and N3 as the number of long tags.So, N2 can be any one of 2 to 6. Example:

```
HEX 10 2 110 TABLES MYTABS
```

Defines tag tables containing short tags C0 (SHINT) to D0 and long tags 200 up to 310. The short table is allocated in memory before the long tables.

TT.ORIGIN (--- A)

'A' is the pointer to the start of the current tag table.

STAG# (--- N)

N is the next available short tag in this tag table .

LTAG# (--- N)

N is the next available long tag in this tag table.

EXT# (--- N)

N is the extension number of the first long tag in the tag tables.

#SHORTS (--- N)

N is the total number of short tags in the short tag table.

#LONGS (--- N)

N is the total number of long tags in the tag tables.

SHORT.TAGS (---) INT

Compiler control: Instructs the compiler to compile defining words as short tags until further notice. LONG.TAGS is the opposite of SHORT.TAGS.

LONG.TAGS (---) INT

Compiler control: next tags will be long.

CAPINIT (---) I

When the %AUTOCAP option is set, TABLES will generate code to automatically set up the tag table vectors when the program is running as a capsule or as a RUN SNAP PROGRAM. Your mother word can call this code by executing CAPINIT.

SHORT <name> INT

Used to predefine a short tag in current tag table so that the tag number of the word is already known, even when its code comes later :

```
SHORT MONKEY
.
.
: MONKEY .... ;
```

LONG <name> IN

Reserves a long tag number for a specific word, and assigns that tag number to the word regardless of whether it is assigning long or short tags to other words when the word is defined. LONG is used like this:

```
LONG word
```

It must be used ***before*** the word in question is defined.

Use LONG before the first colon definition in your program to identify the word that is to be the program's entry point. This works because SnapFORTH uses the first tag in the program's long tag table vector as the entry point.

TAG <name> (N ---) INT

Defines <name> as tag with N as the tag number. When N is between C0H and FFH, <name> the tag will be a short

one, otherwise (i.e. when N is between 100H and 8FFH) the tag will be long.

When <name> is used inside a definition, tag N will be compiled.

Can also be used to predefine a tag like SHORT and LONG, only this time before the tag table is defined. (Or when the tag table for the tag is in another capsule.)

700TAGS INT

Name of the default tag table that the system creates at cold start, containing tags 700 to 73F. The compiler starts allocating tags in this table when you execute its name:

```
700TAGS
```

To start allocating tags in your own table again, execute the name of your own table.

Object Time Words

CREATE word ... word words:

CODE <name>

Starts an assembler definition with name <name> and assigns a tag to it from the current tag table. It automatically sets CONTEXT to ASSEMBLER. The word must be ended with 'ENDCODE', before it can be used.

CODEC <name>

Same as CODE, only CODEC does not assign <name> a tag number, but instead compiles a 'CALL' to an absolute address when <name> is interpreted. This does not use up tags in the tag table, but each reference takes 3 bytes. The defined word is called the same way as a word defined by ':C'.

ENDCODE INT

Used to end a CODE or CODEC word. The new word is appended to the CURRENT vocabulary.

: <name> I

Starts a new SnapFORTH definition and assigns a tag number to it from the current tag table. It sets the CURRENT vocabulary to be the CONTEXT vocabulary and sets the 'STATE' to C0H so INTERPRET will compile every word until a ';' is encountered. (Compiling is done by the text

interpreter as long as STATE is non-zero. Words with the precedence bit set are executed rather than compiled.)

:C <name> I

Used like ':', but it defines a word without creating an entry for it in any tag table. When the compiler encounters such a word, it compiles the tag of '(CALL)', followed by the address of the word. This makes your program one byte shorter if you refer to the word only once in your program.

If you refer to a word before you define it, the compiler assumes that the word will be defined later as a :C word. Therefore, if you want to make a forward reference to a colon definition word, "predefine" the word by declaring it with LONG or SHORT before the first reference to it. (see the discussion on forward references in "SnapFORTH general technical information".)

; I

Used to end a ':' or ':C' definition so the name of the definition will be added to the CURRENT vocabulary. Like ENDCODE.

:P <name> I

Defines a passage of high level code that is callable by a JSR from a low level routine. It is used like this:

```
:P high-level-name
```

SnapFORTH words go here . . .

```
;P
.
.
LABEL any-label
.
.        \ X= top of parameter stack.
high-level-name JSR,
.
.
```

:P creates a label named by the following word. At run time, the X register must contain the address of the top of the parameter stack before JSRing to the label defined by :P.

:P assembles a 'BRK,' instruction. At run time, the "BRK," causes a transfer of control to an interpreter routine that pushes the return address onto the return stack and goes to NEXT. NEXT begins interpreting the following SNAP code.

;P assembles code to pop the return address off the return stack and transfer control there.

;P I

Used to end a ':P' definition, see ':P'. Compiles to the HHC tag (;P).

CREATE <name>

Used together with ';CODE' to make a <BUILDS DOES> like construct with the <BUILDS part between CREATE and ;CODE and the DOES> part written in ASSEMBLER between ;CODE and ENDCODE.

When no ;CODE part is given it defaults to a routine that leaves the address of where the word was created:

```
CREATE ABC 10 , 20 , 30 ,
```

Now ABC will leave the address of where 10 was put in the dictionary. However, you may not use CREATE in this form in a stand-alone application. What actually happens when you do not supply a ;CODE part is that the compiler itself inserts a ;CODE part. But since that code resides in the SnapFORTH capsule ROM, it will not be around when your stand-alone application must run!

CREATEC <name>

Like CREATE,except that it makes a call-word.

;CODE I

Used together with CREATE and CREATEC to generate new snap words.

<BUILDS <name>

Same as CREATE. See Chapter 9 for a full discussion.

<BUILDSC <name>

Same as CREATEC.

DOES>

Same as ;CODE, except that DOES> part is high-level. See Chapter 9.

AREA Words and TABLES:

AREA <name>

Used to define a temporary storage area (TSA) at compile time, and allocate it at run time.

The user may create different kinds of variables within the TSA with words such as VAR, DVAR, CVAR, VECTOR, STRING, etc.

The AREA is terminated by the word ENDAREA. Its maximum length is 250 bytes.

See the section on TSA's in "General Technical Information" for more details.

CTABLE <name> (N ---) INT,DEF,NOTE

Used to define a character table and to assign a tag for <name>in ROM. The table must be initialized by storing data into it, for example:

```
4 CTABLE TRIBBLE
  3 C,  2 C,  1 C,  0 C,
```

N is used by the compiler to check the range of constant subscripts in references to CTABLE. At run time the code

```
1 TRIBBLE
```

will push a 2 on the stack. Note: a CTABLE should not be defined within a TSA.

CVAR <name> INT,DEF

Used in a TSA to reserve one byte of storage for a variable and to give <name> a tag. For example:

```
CVAR ERRFLG
```

This reserves one byte for a variable named ERRFLG. At run time, ERRFLAG will put an address on the stack.

CVECTOR <name> (N ---) INT,DEF

Used in a TSA to reserve N bytes of storage for an array of byte-length values. For example:

```
8 CVECTOR ERRTBL
```

This reserves 8 bytes for an array of 8 byte-length values named ERRTBL.

At run time, the vector expects one item on the stack; it interprets this item as an origin-0 index. For example,

```
7 ERRTBL
```

would stack the address of the 7th (and last) element of the vector, ERRTBL.

DVAR <name> (---) INT,DEF

Used in a TSA to reseve 4 bytes of storage for a long numeric variable. For example:

```
DVAR ERRCOUNT
```

This reserves two words for a variable named ERRCOUNT. At run time, ERRCOUNT's address is placed on the stack.

ENDAREA (---) INT

Used to terminate a TSA. All of the VAR's etc., defined between AREA and ENDAREA are allocated in the TSA.

FVAR <name> (---) INT,DEF

Used in a TSA to reserve 8 bytes of storage for a floating point variable. For example:

```
FVAR ERRVAL
```

This reserves a 8 bytes for a variable named ERRVAL. ERRVAL's address will be put on the stack at run time.

STRING <name> (N ---) INT,DEF

Used in a TSA to reserve storage for a string.

The word reserves N bytes. By convention, this is space for a string up to N-1 bytes long, preceded by a count byte containing the string's actual length. For example:

```
24 STRING ERRMSG
```

This reserves 24 bytes for a string named ERRMSG. The address of ERRMSG's first byte is placed on the stack at run time.

TABLE <name> (N ---) INT,DEF

Used to create a word-element table in ROM (not in a TSA). The table must be initialized by storing data into it, for example:

```
4 TABLE TRIBBLE
3 , -2 , 1 , 6498 ,
```

N is used by the compiler to check the range of constant subscripts in references to TABLE. At run time, the value on top of the stack is used to index into the TABLE named TRIBBLE. The entry is left on the stack, e.g. 0 TRIBBLE yields 3.

VAR <name> (---) DEF

Used in a TSA to reserve one word of storage for a variable. For example:

```
VAR ERRCODE
```

This reserves a word for a variable named ERRCODE; its address is pushed on the stack at run time.

VECTOR (N ---) DEF

Used in a TSA to reserve 2*N bytes of storage for an array of word-length values. For example:

```
8 VECTOR ERRTBL
```


This reserves 16 bytes for an array of 8 word-length values named ERRTBL.

At run time, the vector expects one item on the stack; it interprets this item as an origin-0 index. For example,

```
7 ERRTBL
```

would stack the address of the 7th (and last) element of the vector, ERRTBL.

File words:

LOAD {n:}<fname> INT

Used to 'LOAD' a text file from a given RAM bank. When n: is not specified, it will load from the current RAM bank.

RAM banks are numbered 0..n where 0 is the internal RAM, 1 is the external RAM in the lowest slot number etc.

The file may be either generated by the internal file system editor or by PORTA-WRITER, the HHC's word-processor. The file can only contain legal SnapFORTH code.

LOAD" "<string>" {n:}<fname> --- INT

Same purpose as LOAD,but it starts loading from the first line whose beginning words match with the given string.

DEVLOAD (N1 ---) INT

Same as LOAD, except that it does not load from a file, but instead loads from the device with device code N1. Source lines should be 80 characters or less, followed by a carriage return. Linefeeds are ignored. Tabs are converted to blanks. All other control characters are used as end of file markers, and terminate execution of DEVLOAD.

DEVLOAD" "<string>" (N1 ---) INT

The combination of DEVLOAD and LOAD".

(LOAD) (N1 N2 N3 N4 N5 ---)

The basic load word used by the four words above.

N1 is a flag that when true tells (LOAD) that &SPAT and &SLEN contain the string and length of a string that needs to match with the first part of a line before loading starts.

N2 The tag# of a 'L1' routine that takes a line from somewhere and puts it in TIB. It can use L2, L3 and L4 as parameters to tell where to get the information from. 'L1' must be non-zero because it serves as a flag to tell the system that it is loading a program.

N2, N3, and N4 are the initial values of L2, L3 and L4; three variables reserved for use by L1.

EDIT n:<fname> INT

Calls the internal file system editor to edit the file <fname> in the RAM bank 'n'. When n: is not specified it will take the current RAM bank.When <fname> does not exist yet it will be created.

SAVESNAP {n:}<fname> INT

Takes the dictionary contents from the start of the current tag table to the end of the dictionary and turns this into a RUN SNAP Program with name <fname> in RAM bank n. If the file already exists and is a RUN SNAP Program, it will be overwritten, otherwise an error message will be generated.

This program is not allowed to contain any absolute addresses and the first long tag should contain its entry point.If the auto-capinit flag is on, SAVESNAP will insert a routine with tag number equal to the entry point's tag plus one. It will initialize all tag table vectors in the system when called. In this case,the entry point should call this word with the name 'CAPINIT'.

Conditional compilation:

IFTRUE BOOL --- INT

Conditional compilation feature, only used in immediate mode. When BOOL is IFTRUE will continue interpretting until an OTHERWISE or an IFEND is encountered. When BOOL is FALSE, IFTRUE will search for an OTHERWISE or IFEND and will continue after that. Example:

```
?DEF ROUTINE IFTRUE FORGET ROUTINE
   IFEND
```

Note that IFTRUE, OTHERWISE and IFEND cannot be nested.

OTHERWISE INT

When executed OTHERWISE will search for an IFEND and continue INTERPRETTING there.

IFEND INT

End of an IFTRUE-IFEND or IFTRUE-OTHERWISE-IFEND construction.

Choice clauses:

DOCASE I

Used in conjunction with ENDCASE to build an arbitrary complex clause from the following elements:

<case clause> ::= CASE <choice clauses> ENDCASE

<choice clauses> ::= <empty> | <choice clause> <choice clauses>

<choice> <clause>::= <selector> <statements> { ELSE }

<selector> ::= CASE | IF | <IF | NOTIF | =IF IF.ITS | IF.L

ENDCASE I

Generates all the THENs needed to balance the control structures defined between DOCASE and ENDCASE.

Examples of well formed DOCASE-ENDCASE clauses:

```
DOCASE
   CASE                 ELSE
   CASE                 ELSE
   CASE
   ELSE
ENDCASE

DOCASE
   IF.ITS 3             ELSE
   <IF                  ELSE
   =IF
   ELSE
ENDCASE
```

CASE (N1 N2 --- [N1]) I

Tests if N1 = N2 If TRUE drops both and control passes to the next word after CASE. Otherwise drops N2 and execution skips to the matching ELSE, THEN (ENDIF) or ENDCASE. For example:

```
: .DIGIT    ( N --- )
DOCASE
   0 CASE    ." ZERO "   ELSE
   1 CASE    ." ONE "    ELSE
   2 CASE    ." TWO "
   ELSE DROP  ." NOT A VALID CHOICE "
ENDCASE ;
```

Compiles the tag (CASE).

=IF (N1 N2 ---) I

Tests if N1 = N2 and drops N1 and N2. If TRUE, it continues execution after = IF otherwise skips to next ELSE, THEN or ENDCASE. Compiles (=IF).

<IF (N1 N2 ---) I

Tests if N1<N2 and drops both. If TRUE, it continues execution after = IF otherwise skips to next ELSE, THEN or ENDCASE. Compiles (<1F).

NOTIF (N1 ---) I

Equivalent to '0= IF'. For example: Tests if N1 = 0 and drops N1. If N1 = 0, continues execution after NOTIF; otherwise it skips to next ELSE, THEN or ENDCASE. Example:

```
A @ NOTIF XYZ ENDIF
```

XYZ is executed if the contents of A is zero. Compiles (NOTIF).

IF.ITS <C> (C1 --- C1) I

A CASE-like statement used to make a choice based on a character value. IF.ITS is useful where clarity or compactness is more important than efficiency, and where the possible values are scattered. It is most often used to execute code based on a menu selection.The word is used like this:

```
X C@  \ Value to choose with.

IF.ITS 1 code-for-choice-1 ENDIF
IF.ITS 2 code-for-choice-2 ENDIF
IF.ITS 3 code-for-choice-3 ENDIF
:
:
IF.ITS n choice-n ENDIF
DROP
```

The word IF.ITS must be followed by a constant or symbolic constant in the range 0-255, inclusive. If C1 is equal to the constant, IF.ITS executes the code between the constant and the ENDIF. Compiles (IF.ITS).

JUMP-TAB (N ---) INT,DEF,NOTE

Defines a "jump table" that can be used to execute one of several words depending on the value of a word on the stack.

JUMP-TAB is used like this:

```
4 JUMP-TAB FOUR.WORDS
   WORD1
   WORD2
   WORD3
   WORD4
```

This example defines a jump table with 4 entries, named FOUR.WORDS. The entries in the jump table are four words (defined elsewhere, with tags) named WORD1, WORD2, WORD3, and WORD4.

A jump table defined by JUMP-TAB is used like this:

```
: JUMPING.LIZARDS ( --- )
  .
  .
    JUMP.TO.ME @ FOUR.WORDS
  .
  .
  ;
```

In this example, JUMP-TO-ME is presumably a variable containing a jump table index between 0 and 3. If the index is 0, the illustrated line of code executes WORD1; if the index is 1, the code executes WORD2; and so forth.

Caution: A jump table word defined by JUMP-TAB does not do range checking. Executing a jump table word with an invalid index produces unpredictable results. You can conveniently check the range before executing a jump table word by using ?RANGE.

IF (B ---) I

Used to begin an IF construction. Used in the form:

```
IF tttt... ELSE eeee... THEN or
IF tttt... THEN eeee...      or
DOCASE IF tttt.. ENDCASE     etc.
```

If B is TRUE, "tttt..." is executed and "eeee..." (if any) is skipped. If B is false, "tttt..." is skipped and "eeee..." (if any) is executed. Compiles (IF).

IF.L (B ---) I

Tests if B is non-zero (i.e. TRUE). If TRUE, it continues execution after IF.L otherwise skips to next ELSE, THEN or ENDCASE.

The difference with IF is that IF can generate only jumps up to 255 bytes forward and IF.L has no limit on the range. Compiles the tag ?JUMP.

ELSE (---) I

Used to separate the IF clause and ELSE clause in an IF, CASE or similar construction. Example:

```
IF tttt... ELSE eeee... ENDIF
```

If the entry IF takes off the stack is TRUE, "tttt..." is executed and "eeee..." is skipped. If it is FALSE, "tttt..." is skipped and "eeee..." is executed. Compiles (ELSE).

ELSE.L (---) I

Like ELSE, but has no limit on the range. Compiles JUMP.

ENDIF (---) I

Synonym for THEN.

THEN (---) I

Used to terminate an IF, CASE or similar construction. Examples:

```
IF tttt... ELSE eeee... THEN or
IF tttt... ENDIF           or
CASE tttt. ELSE eeee... THEN etc.
```

THEN is optional at the end of a DOCASE .. ENDCASE construction, since ENDCASE will generate the required THENs. Example:

```
a CASE aaaa... ELSE
b CASE bbbb... ELSE
c CASE cccc... ELSE
xxxx...
THEN THEN THEN
```

Can also be written as:

```
DOCASE
  a CASE aaaa... ELSE
  b CASE bbbb... ELSE
  c CASE cccc... ELSE
  xxxx...
ENDCASE
```

The latter form is preferred because it is much more readable.

Repetition clauses

BEGIN (---) I

Begins an iterative loop in a colon definition. Used in one of the forms:

```
BEGIN

  WHILE!<WHILE! NOTWHILE ! =WHILE !
  WHILE.ITS ! CASEWHILE ! WHILE.L

REPEAT ! AGAIN ! UNTIL
```

BEGIN ... UNTIL loops until UNTIL finds a nonzero value on thestack. Then it passes control to the next word after UNTIL.

BEGIN ... REPEAT loops forever, unless terminated by EXIT.

BEGIN ... WHILE ... REPEAT loops until WHILE finds a 0 on the stack. Then it passes control to the next word after REPEAT.

(At compile time, BEGIN leaves its return address and an error checking code on the stack for UNTIL, AGAIN, or REPEAT to use.)

UNTIL (B ---) I

Jumps back to BEGIN if B is FALSE, otherwise continues execution. Compiles (UNTIL).

REPEAT (---) I

Jumps back to BEGIN. Compiles (AGAIN).

AGAIN (---) I

Synonym for REPEAT.

CASEWHILE (N1 N2—[N1]) I

Same parameters as CASE, used to exit from a BEGIN clause. Compiles (CASE).

=WHILE (N1 N2 ---) I

Same parameters as =IF, used to exit from a BEGIN clause. Compiles (=IF).

<WHILE (N1 N2 ---) I

Same parameters as <IF, used to exit from a BEGIN clause. Compiles (<IF).

WHILE (B ---) I

Ends iteration of a BEGIN loop in a colon definition. Occurs in a colon definition, for instance in the form

```
BEGIN ... WHILE ... REPEAT
```

BEGIN ... WHILE ... REPEAT loops until WHILE finds a 0 on the stack. Then it passes control to the next word after REPEAT. Compiles (IF).

NOTWHILE (B ---) I

Same parameters as NOTIF, used in BEGIN clause to exit when B is TRUE. Compiles (NOTIF).

WHILE.ITS (N1 <N2> --- N1) I

Same parameters as IF.ITS, used in BEGIN clause to exit when N1 = N2. Compiles (IF.ITS).

WHILE.L (B ---) I

Like IF.L, exits from BEGIN clause if B is FALSE. Compiles ?JUMP.

DO (LIMIT INITIAL ---) I

Used in the construction:

```
DO ... LOOP or
DO ... INCR +LOOP
```

iterates the code between DO and LOOP or +LOOP until the loop limit is reached.

The initial value of the loop index is INITIAL. Each time LOOP is reached, the index is incremented by 1. Each time +LOOP is reached, the index is incremented by INCR.

If LOOP is used, or if +LOOP is used with a positive INCR, iteration continues until control returns to DO with index >= LIMIT. If +LOOP is used with a negative INCR, iteration continues until control returns to DO with index <= LIMIT.

Within a loop, the word 'I' will copy the index onto the stack. Within two nested loops, the word 'J' will copy the index of the outer loop onto the stack.

Within a loop, the word LEAVE will reset the LIMIT to the current value of the index, forcing the loop to terminate at the end of the current iteration.

(The run time routine of DO is (DO). At execution time, (DO) initializes the index, and pushes the index and limit onto the Loop Stack.) See also '?DO'.

LIMIT, INITIAL and INCR are treated as 16 bit unsigned numbers. When the difference between LIMIT and INITIAL is greater than 8000H, DO..LOOP performs only once.

?DO (LIMIT INITIAL ---) I

Like DO, but it will not execute the loop if LIMIT =< INITIAL initially. Another difference with DO is that LIMIT and INITIAL are treated as 16 bit SIGNED numbers, which means that ?DO never loops when LIMIT U> INITIAL but LIMIT < INITIAL.

LOOP (---) I

End of a loop clause. The index is incremented by 1. If the new index value < limit, control returns to the first word after the corresponding DO or ?DO. If the new index value >= limit, the loop parameters are discarded and control passes to the next word after LOOP.

+LOOP (N1 ---) I

Used to terminate a DO loop when the index is to be incremented by a value other than 1.

N is added to the index. If the new index value < limit, control returns to the first word after the corresponding DO or ?DO.

If the new index value >= limit, the loop parameters are discarded and control passes to the next word after +LOOP.

Compiler/misc words:

?PAIRS (N1 N2 ---)

Generates the error message 'ILLEGAL CONSTRUCTION' if N1<>N2.

!CSP (---)

Saves the current parameter stack-pointer in the variable CSP.

?CSP (---)

Does a ?PAIRS on the current stack-pointer and CSP.

?COMP (---)

Checks the state, and if in immediate mode will generate the error message

```
' CAN'T EXECUTE'.
```

?EXEC (---)

Checks the state, and if in compile mode will generate the error message

```
' CAN'T COMPILE'.
```

COMPILE <name>

Only used inside a compiler definition, COMPILE will compile the code to call <name> into the dictionary.

[COMPILE] <name> I

Used to compile an immediate definition into a definition. Normally immediate routines are executed inside a definition.

[(---) I

Switches SnapFORTH from compile mode to execute mode. This word has two uses. One is to execute some words while compiling. This is most commonly done to resolve the value of an expression at compile time. For example:

```
HEX
      80 == ALL.DONE
      40 == PART.DONE

: FOOBAR  ( xxxxxxxx... )
     ...
  FLAGBYTE C@
  [ ALL.DONE PART.DONE OR ]
     LITERAL AND IF
     ...
;
```

The code 'ALL.DONE PART.DONE OR' is executed (in the middle of the compilation of FOOBAR). It leaves a value on the stack which LITERAL stores into the dictionary as a literal, so that the line beginning with FLAGBYTE executes as though it read like this:

```
  FLAGBYTE C@ C0 AND IF
```

The second use of '[' is to return Snap to execute mode after compiling a piece of high level code outside a colon definition. This is most commonly done when a code (i.e. assembler) definition must call a high level (i.e. SnapFORTH) word. The technique is explained in the chapter "General Technical Information."

] (---) I

Switches SNAP from execute mode to compile mode. Used with '[' to switch SNAP from compile mode to execute mode, then back to compile mode; or from execute mode to compile mode, then back to execute mode. See description of '['.

MYSELF (---) I

Allows a routine to call itself when MYSELF is used instead of the routine name.

IMMEDIATE (---) INT

Used after a definition is defined to declare this an immediate routine.

CEXECUTE (ADDR ---) INT

A routine to call call-words with the address of the routine on the stack.

INTERPRET (---)

Interprets a line (when entered from keyboard) or file with SnapFORTH words in it.

\ <comment> I

Makes the rest of this source screen line a comment. Example:

```
2DUP + 1-     \ Point to the last byte.
```

'\' must be followed by a blank. No terminating delimiter is needed.

(<comment>) I

Allows comment until the next ')'.

ABORT" "<string>" I

Generates the error message <string> and returns to the routine in the (QUIT) vector.

?ABORT" (BOOL "<string>" ---) I

Generates the error message <string> when the boolean on the stack is FALSE.

CLEANUP INT

A routine called by your quit vector to reset the status of the compiler.

((QUIT)) (---)

Default value of the (QUIT) vector.

QUIT (---)

Resets the current RAM bank, checks the dictionary file and jumps to the user quit vector in (QUIT).

COLD INT

Deletes the current dictionary file and goes back the the FORTH-menu.

VARIABLE <name> INT

Creates a new 'TO' variable (not in an area).

WHO (---)

Displays the author's moniker.

Assembler Glossary

Assembler words:

ASS	Switches to the ASSEMBLER vocabulary and initializes some assembler variables.
LABEL	Generates an assembler label.

Addressing modes:

#	Immediate addressing mode.
#L	Immediate mode, use low byte of value on stack.
#H	Immediate mode, use high byte of value on stack.
.A	Accumulator mode, e.g. .A ROL,
,X	Parameter stack mode
,Y	Index value on stack with Y register.
X)	Parameter stack indirect.
)Y	Zero page indirect, indexed with Y register.
)	Used to generate) JMP,

Opcodes:

CLC, SEC, CLI, SEI, NOP, CLV, CLD, SED,
DEX, INX, DEY, INY, TAX, TXA, TAY, TYA,
TXS, TSX, PHP, PLP, PHA, PLA, RTS, RTI,
ORA, AND, EOR, ADC, STA, LDA, CMP, SBC,
ASL, ROL, LSR, ROR, DEC, INC,
CPX, CPY, LDY, STY, BIT, LDX, STX,
BPL, BVC, BVS, BCC, BCS, BNE, BEQ,
JMP, JSR,

Control structures:

NOT 0= 0< CS VS	--- Used with IF, WHILE, and UNTIL, as : 0= IF, 0= NOT IF, VS NOT WHILE, CS IF, 0< NOT IF, CS NOT UNTIL, etc.
IF, THEN,	IF, ELSE, THEN,
ENDIF, ELSE,	Same AS THEN,
BEGIN, UNTIL, AGAIN, WHILE, REPEAT,	BEGIN, UNTIL, BEGIN, AGAIN, BEGIN, WHILE, REPEAT,

Interface words to nucleus:

TOP	0 ,X
SEC	2 ,X
PUTTRU	(PUTTRU))
PUTFLS	(PUTFLS))
CPUT	(CPUT))
CPUSH	(CPUSH))
POPPUT	(POPPUT))
2POP	(2POP))
PUT	(PUT))
POP	(POP))
PUSH	(PUSH))

GLOSSARY OF HHC WORDS

HHC Words Beginning With Punctuation Marks or Numbers

! (X A ---) C

Stores X at A. Pronounced "store" or "bang."

(ND1 --- ND2)

Used to convert binary numbers to ASCII. See description of <# for details.

#> (ND --- A L)

Used to convert binary numbers to ASCII. See description of <# for details.

#DECB (--- A)

Stacks the address of DECB, an ECB used by SECS and the LCD display routines. #DECB is a short tag; thus you can use #DECB instead of DECB to save a byte in your code.

#S (ND1 --- ND2)

Used to convert binary numbers to ASCII. See description of <# for details.

%BASIC (--- FL) NOTE, =

File-type mask used with GET-TYPE to designate the "Microsoft BASIC program file" attribute. Note: %BASIC is the customary symbol for this mask, but the symbol is not pre-defined. If you need it, define it as 10H.

%DELETE (--- N) =

"Delete mode" mask for EXPECT's mode parameter. See EXPECT for details.

%EXECUTE (--- N) =

File-type mask used with GET-TYPE to designate the "executable" or "RUN SNAP PROGRAM" attribute.

%FLEN (--- N) =

A value used during nucleus testing. No use to applications programming.

%INSERT (--- N) =

"Insert mode" mask for EXPECT's mode parameter. See EXPECT for details.

%INVISIBLE (--- N) =

File-type mask used with GET-TYPE to designate the "invisible" attribute.

%LD (--- N) =

"Lock-delete mode" mask for EXPECT's mode parameter. See EXPECT for details.

%LI (--- N) =

"Lock-insert mode" mask for EXPECT's mode parameter. See EXPECT for details.

%LLEN (--- N) =

The length of &LBUF - 80 bytes.

%OVR (--- N) =

"Overstrike mode" mask for EXPECT's mode parameter. See EXPECT for details.

%TEMPORARY (--- N) =

A file type identifier assigned for future expansion. See GET-TYPE.

%TEXT (--- N) =

File-type mask used with GET-TYPE to designate the "text" attribute.

&APS (--- A) V6

A 6-byte area in page zero provided for use by application programs. This area is not used by any part of the HHC's operating system. Its contents are destroyed by a CLEAR.

&BL (--- C) =

The ASCII value input by the "space" key. The name stands for "blank."

See the description of 20H.

&BSP (--- C) =

The binary value input representing the ASCII character "backspace" (not represented on the HHC keyboard). When EMITted, this value causes a destructive backspace.

&BUF-ADR (--- A) V2

Pointer to the buffer used by EXPECT.

&BUF-LEN (--- A) V1

Location containing the length of the buffer used by EXPECT. (length <256)

&C1 (--- C) =

&C2 (--- C) =

&C3 (--- C) =

&C4 (--- C) =

The binary values input by the HHC keys 'C1', 'C2', 'C3', and 'C4'. Their HHC graphic representations are ä, ö, ü and ñ respectively.

&CFILE (--- A) V2

Pointer to the currently open virtual file. If no file has been opened since the most recent soft or hard clear, the pointer is zero.

&CR (--- C) =

The binary value input by the ENTER key. It represents the ASCII control character RETURN. Sent to an output device, it performs a carriage return; sending this character to the LCD is equivalent to executing CR.

&CURPOS (---A) V1

Byte used to hold the position of the cursor within EXPECT's buffer.

&DATA-LEN (---A) V1

Used by EXPECT to hold the length of the data held in the buffer.

&EOF-TAG (--- A) V2

Used by File Editor to hold the tag which determines the End-of-File condition.

&EOL-TAG (--- A) V2

Vector for the File Editor's End-of-Line check tag. Typically holds 'X EOL.

&ESC (--- C) =

The binary value representing the ASCII control character ESCAPE (not represented on the HHC keyboard). It begins an escape control sequence for devices recognizing them.

&EXTRINSIC (--- A) V1

Indicates whether the current virtual file space is an extrinsic or the intrinsic file space (i.e., in a Programmable Memory Peripheral or in intrinsic RAM). A zero value means intrinsic. A non-zero value indicates extrinsic; the value is the number of the RAM bank currently selected (see CURRAM).

&FP (--- A) V6

A 6 byte area of RAM used by some floating point words. It is available for application programs to use, but its contents may be destroyed by the execution of any floating point word.

&HLP (--- C) =

The binary value input by the HELP key (14H).

&INIT-CURPOS (--- A) V1

Used to hold EXPECT's initial cursor position.

&IO (--- C) =

The binary value input by the I/O key (0BH).

&LBUF (--- A) V80

An 80-character area which you may use as an input buffer for EXPECT or as application workspace.

&LF (--- C) =

The value of the ASCII control character LINE FEED (not represented on the HHC keyboard).

&LINE (--- A) V2

Holds the record number of the line currently being edited by the File Editor.

&LK (--- C) =

The binary value input by the LOCK key (86H). Its value is not returned by KEY until a second key is pressed. If that second key is SHIFT or 2ND SFT, the nucleus handles the lock condition automatically.

&LOCK (--- A) V1

A byte used by EXPECT to implement its handling of the LOCK key.

&MAX-LEN (--- A) V1

Contains the maximum line length permitted by the File Editor (set upon entry).

&MEM (--- A) V8

Used by the intrinsic CALCULATOR for "memory." Do not use this area; if you do, you will destroy the contents of the CALCULATOR's memory. This memory is not affected by the CLEAR key.

&MODE (--- A) V1

A byte used by EXPECT to implement its handling of the INSERT and DELETE keys.

&SLEN (--- A) V1

A data byte available to application programs for any purpose.

(Search pattern length. Used by the SEARCH key function in the file system editor.)

&SPAT (--- A) V18

A data area available to application programs for any purpose.

(Search pattern. Used by the SEARCH key function in the file system editor.)

&STP (--- C) =

The character value input by the STOP/SPEED key (0EH).

'ABORT (--- A) V2

Tag of a word that is to be called when a serious logical error occurs in the HHC software. Examples of conditions that cause an abort are stack underflow or overflow and return stack underflow or overflow.

The tag in 'ABORT is executed by calling ABORT. You can change it to get control when a serious error occurs in order to perform your own error processing.

The default value of 'ABORT displays the message "RESET" and then performs a hard clear. 'ABORT is reset to this value by a clear.

'EMIT (--- A) V2

An area containing the tag of a word which EMIT will call to do all of its processing. This area normally contains the tag of (EMIT), and is reset to that tag by a soft clear.

You can "redefine" EMIT for your application by setting 'EMIT to a tag of your choice.

The following code illustrates the use of 'EMIT. After it is executed, EMIT will execute FOOBAR instead (EMIT). It will continue to do so until 'EMIT is changed again, or until the next soft clear.

```
: FOOBAR . . . ;
: INITIALIZATION-WORD
  . . .
  'X FOOBAR 'EMIT !
  ; . .
```

'FILE-COND (--- A) V2

An area used to store the tag of a file-selecting word for .FILE to use in determining which files are to be listed in a menu.

The linkage to the file-selecting word must be:

(A --- B)

where A is the address of a file and B is TRUE if the file is to be listed in the menu, or FALSE if it is not.

The default file-selecting word returns TRUE for non-invisible files and FALSE for invisible files. This results in the HHC's usual behavior, which is to list all files except invisible files. You can change this behavior by storing the tag of a word of your choice in 'FILE-COND.

Example:

```
:FCOND
   2+ C@ %INVISIBLE
     AND NOT
;
:FKEY
   OPEN-FILE TRUE
;
:FMENU
   'X FCOND  'FILE-COND !
   1 'X FKEY 'X .FILE
   MENU-DRIVER
;
```

'FILEORG (--- A) V2

An area containing the tag of the word which returns the origin of filespace. It is called by FILEORG. The default word is normally (FILEORG).

'KEY (--- A) V2

An area containing the tag of a word which KEY will call to do all of its processing. This area normally contains the tag of (KEY), and is reset to that tag by a soft clear.

You can redefine KEY's function by writing your own character processing routine and storing its definition tag at 'KEY.

'V (--- A) I

Compiles code to stack the address of the instruction that begins the following definition. For example, to stack the address of the colon definition MENU-DRIVER:

```
'V MENU-DRIVER
```

'X (--- TAG) I

Returns tag of the following word. Example of use:

```
'X YOUR-TOP-WORD FLEE
```

This returns the tag of the word YOUR-TOP-WORD, and then FLEEs to it. FLEEing transfers control to YOUR-TOP-WORD, and also stores its address (or tag) at SOFTAG, causing YOUR-TOP-WORD to be re-executed if the CLEAR key is pressed.

'X may be used either inside or outside a target colon definition.

((---) I

Begins a comment. The comment will be ended by the first ')' encountered (whether or not it is surrounded by blanks). Note: ')' is not a vocabulary word; it is just a character that '(' scans for.
See also '/'.

(+LOOP) (N ---) C

Compiled from +LOOP. Increments loop index by N and tests for loop completion.

(.") (---) C

Compiled from '."' (dot-quote). Displays the COUNT format string that follows (.") in the program; passes control around the string.

(;P) (---) C

Compiled from ';P'. Pops IP from the return stack and goes to NEXT.

(<IF) (N1 N2 ---) C

Compiled from <IF. If N1<N2, executes the following code. If N1>=N2, passes control around the following code to the point representing the matching ELSE or ENDIF.

(=IF) (N1 N2 ---) C

Compiled from =IF. If N1=N2, executes the following code. If N1<>N2, passes control around the following code to the point representing the matching ELSE or ENDIF.

(?DO) (N1 N2 ---) C

Compiled from ?DO. If N1=<N2, control is passed to the code following the next LOOP or +LOOP.

(AGAIN) (---) C

Compiled from REPEAT and AGAIN. Followed by a one-byte unsigned displacement which is subtracted from the address of the displacement to get the address of the corresponding BEGIN.

(CALL) (---) C

Compiled from any reference to a colon definition that was defined with ':C' instead of ':'. Interprets the next two bytes as an address, and "calls" the colon definition beginning at that address.

(CASE) (N1 N2 --- [N1]) C

Compiled from CASE. If N1=N2, executes the following code and N1 is dropped. If N1<>N2, branches around the following code to the point representing the next CASE, the matching ELSE, or the matching ENDIF. N1 remains on the stack in this case.

(D.) (DN --- A L)

Formats a double integer in ASCII, using the current base (see BASE). A L is the field containing the ASCII formatted number.

(DO) (N1 N2 ---) C

Compiled from DO. Moves loop control parameters to the return stack.

(ELSE) C

Compiled from ELSE. Followed by a one-byte unsigned number giving the displacement to the next ENDIF.

(EMIT) (C ---)

Performs the character output function of the LCD. The address of (EMIT) is normally stored at 'EMIT.
You can redefine EMIT's function by writing your own character processing routine and storing its tag at 'EMIT.

(FILEORG) (--- A)

Stacks address of the current virtual file space. See FILESPACE and 'FILEORG.

(FREEZE) (--- A) V1

Number of the first character position of the LCD that participates in rotation. The normal value is 0, causing the entire LCD to rotate. You can "freeze" the left part of the LCD by setting (FREEZE) to a non-zero value. (FREEZE) is reset to 0 by a clear.

(IF) (B ---) C

Compiled from IF. Followed by a one-byte unsigned number giving the displacement from the displacement byte to the next ELSE (if any) or ENDIF.
If B is TRUE, does nothing. If B is FALSE, passes control to the point representing the matching ELSE or ENDIF.

(IF.ITS) (N --- N) C

Compiled from IF.ITS. If N is equal to the constant compiled

after IF.ITS, passes control to the code following the constant. If N is not equal to the constant, passes control to the point representing the matching ELSE or ENDIF.

(KEY) (--- C)

Performs the entire character input function of KEY. The tag of (KEY) is normally stored at 'KEY.

You can redefine KEY's function by writing your own character processing routine and storing its tag at 'KEY.

(LOOP) (---) C

Compiled from LOOP. Increments loop index by 1 and tests for loop completion.

(NOTIF) (B ---) C

Compiled from NOTIF. Followed by a one-byte unsigned number giving the displacement from the displacement byte to the next ELSE (if any) or ENDIF.

If B is FALSE, does nothing. If B is TRUE, passes control to the point representing the matching ELSE or ENDIF.

(ROTMODE) (--- A) V1

A variable that controls how the LCD will be rotated when the cursor is past the last character column and another character is written with EMIT. Meaningful values are:

0: fill mode
1: rotate mode (the normal setting)
2: rotate and fill mode

Note that (ROTMODE) is not reset by a soft clear.

For details of use, see the discussion of "Rotation Mode" in the chapter on the LCD.

(UNTIL) (B ---) C

Compiled from UNTIL. Followed by a one-byte unsigned displacement. If B is FALSE, passes control back to the corresponding BEGIN, which is at the address of the displacement minus the value of the displacement. If B is TRUE, passes control to the address following the displacement.

* (N1 N2 --- N3)

N3 = N1 * N2.

*/MOD (N1 N2 N3 --- N4 N5)

N4 = remainder of (N1*N2)/N3.
N5 = quotient of (N1*N2)/N3.
Intermediate result is a double number.

+ (N1 N2 --- N3) C

N3 = N1 + N2.

+! (N A ---) C

Adds N to the word found at A and stores the sum back at A. Pronounced "plus-store." Equivalent to

```
SWAP OVER @ + SWAP !
```

+C! (N A ---) C

Adds N to the byte found at A and stores the sum back at A. Pronounced "plus-C-store." Equivalent to

```
SWAP OVER C@ + SWAP C!
```

+USED (N ---)

Internal word used by the file system. Adjusts the number of bytes currently used by all of the files in the file space.

- (N1 N2 --- N3) C

N3 = N1 - N2.

-1 0 1 2 3 4 5 6 7 8 9 (--- N) C

These small numbers are used so often that they are defined by name in the dictionary.

. (N ---)

Display a number (16-bit two's complement notation) in ASCII, according to the current BASE. A trailing space follows the number. Pronounced "dot."

." (---) NOTE

Begins a string constant. The constant is the text following ." and one blank, up to the next "".

If used in a colon definition, ." compiles to the (.") word, and a string.

Here is an example of how ." is used in a colon definition:

```
." two words"      (no leading blanks)
."   two words"    (with leading blanks)
```

If used in immediate mode, ." prints the string immediately.

The maximum number of characters that may appear in the string is 255.

.File (N --- [A L TAG] B)

Word used to set up a menu of files (see example in MENU-DRIVER). .FILE searches for the Nth file satisfying

the condition-testing word in 'FILE-COND (e.g. text files). If it is found, B is TRUE and A and L are the address and length of an ASCII string containing the file's name. TAG is the tag number of TYPEDROP.

If an Nth file cannot be located in the file space, only B=FALSE is returned.

.NO-ROOM (---)

Displays the message "NO ROOM", generally when there is insufficient file space.

.R (N1 N2 ---)

Display N1 (16-bit two's complement notation) in ASCII, according to the current BASE. The number is right-aligned in a field N2 positions wide.

If the "natural" width of the number exceeds N2, no truncation occurs, and no leading spaces are printed.

/ (N1 N2 --- N3)

N3 = N1 / N2. Remainder is dropped.

If N2=0 and N1<0 then N3=1. If N2=0 and N1>=0 then N3=-1.

/MOD (N1 N2 --- N3 N4)

N3 = remainder of N1/N2.
N4 = quotient of N1/N2.

If N1<0 or N2<0, /MOD may produce results that are not meaningful.

0< (N --- F) C

Test if N < 0.

0= (N --- B) C

Test if N = 0.

1+ (N --- N') C

N' = N + 1.

1- (N --- N') C

N' = N - 1.

10EXP (N --- EXP)

Extracts the unbiased exponent from a copy of a floating point number's topmost word copied to the top of the stack. EXP is the unbiased exponent from the number.

E.g., '3.14 DUP 10EXP' leaves a zero on the top of the stack, since 3.14 = $3.14*10^0$.

If you want to look at a floating point number further down the stack: '718.0 3.14 5 PICK 10EXP' leaves an integer 2 on the stack (728.0=$7.28*10^2$), EXP=0 for zero, underflow, etc.

2! (X Y A ---)

Stores X at A+2 and Y at A. (The reverse order of storage reflects the 6502's convention of orienting the low-order part of a number toward the low end of memory.)

2* (N --- N') C

N' = N + N.

2+ (N --- N') C

N' = N + 2.

2- (N --- N') C

N' = N - 2.

2/ (N --- N') C

N' = N / 2. Note that the result is truncated downward, not toward zero; thus, for example, "-3 2/" leaves -2, and "-1 2/" leaves -1.

20H (--- N) NOTE

Stacks the value 20H.

Note: 20H produces the same result as &BL: it stacks the numeric value 20H (which is also the ASCII character "blank"). 20H generates a short tag (one byte of code), as does &BL.

2@ (A --- X Y)

Fetches X at A+2 and Y at A.

2>R (X Y ---)

Transfers X and Y from the parameter stack to the return stack. X will be on top of the return stack.

2DROP (XD ---) C

Drops top double entry on stack (or 2 single entries).

2DUP (XD --- XD XD)

Copies top double entry on stack (or 2 single entries).

2OVER (XD YD --- XD YD XD)

Copies 2nd double entry on stack.

2PICK (XDy ... XD3 XD2 XD1 y --- XDy ... XD3 XD2 XD1 XDy)

Copies y'th double entry on stack to the top.

2ROLL (XDy ... XD3 XD2 XD1 y --- ... XD3 XD2 XD1 XDy)

Moves y'th double entry on stack to the top.

Inverse of 2>R—takes two words from the top of the

2R> (--- X Y)

Inverse of 2>R—takes two words from the top of the return stack and puts them on top of the parameter stack.

An advanced technique that requires some care is shown in the following example:

```
: WORD3
    IF
      2R> 2DROP EXIT
    ENDIF
    .
    .
    ;
: WORD2
    X @ WORD3
    .
    .
    .
;
: WORD1
    .
    .
    .
   WORD2
    .
    .
    .
;
```

If the contents of X are nonzero, WORD3 will return directly to WORD1 instead of WORD2, thereby avoiding further processing of WORD2 code. The programmer must be careful about what is on both the parameter and return stacks when returning to WORD1.

2ROT (XD YD ZD --- YD ZD XD)

Moves 3rd double entry on stack to top.

2SHFTBLIP (--- MASK) =

A mask for the "second shift" blip. See ANDBLIP for details.

2SWAP (XD YD --- YD XD)

Exchanges top 2 double entries on stack.

30H (--- N)

Stacks the value 30H.

80H (--- N)

Stacks the value 80H.

< (N1 N2 --- B) C

Test if 'N1 < N2'.

<# (---)

Used to convert a number from binary to ASCII. Associated words are # , #S , SIGN , and #> .

The conversion process uses the text output buffer(TOB) at the top of the Temporary Stack. Caution: Since this area is also used by '.',CFIND, (D.) and FP>ASC, remove the data as soon as conversion is completed.

<# performs setup for conversion.

generates the next ASCII character and stores it in the TOB. Conversion is performed according to the current value of BASE.

Conversion is performed right to left (i.e. least to most significant digit). The word takes a double integer from the stack, divides it by BASE, deposits the ASCII representation of the remainder in the buffer, and leaves the quotient on the stack.

#S repeats # until the remainder is zero (but always executes it at least once).

SIGN deposits a minus sign in the TOB if the sign (indicator (an integer on the stack below the double integer being converted) is negative, and drops the sign indicator from the stack.

HOLD takes a character from the stack and deposits it in the TOB, adjusting pointers so that the next character will not overlay it. For example, '2EH HOLD' may be used to insert a decimal point in the number.

#> drops the quotient and leaves the address and length of the ASCII number on the stack.

The use of these words is best explained by documenting sequences of words as though they were single words.

<# #S #>   (DU --- A L)

Converts a non-negative double integer to ASCII representation in current BASE.

<# # #S #>   (DU --- A L)

Converts a non-negative double integer to ASCII representation in current BASE. The number is padded with leading zeros, if necessary, to a length of 2. (Using the # word N times will cause the number to be padded to a length of N+1.)

DUP ROT ROT DABS <# #S SIGN #>   (DN --- A L)

Converts a signed double integer to ASCII representation. If the number is negative it is given a leading minus sign.

DUP S>D DABS <# #S SIGN #>.   (N --- A L)

Converts a signed integer to ASCII representation.

DUP ROT ROT DABS   (DN --- A L)
<# # # 2E HOLD #S SIGN 24 HOLD #>

Converts a signed double integer to ASCII. A minimum of 3 digits are produced. "2E HOLD" inserts a decimal point (period) before the next-to-last digit. SIGN inserts a minus sign before the first digit if the number is negative. "24 HOLD" inserts a dollar sign before the first digit and sign (if any).

Typical formatting action for this sequence would be: 2595 to '$25.95'; 95 to '$0.95'; 1 to '$0.01'; -5 to '$-0.05'; -13950 to '$-139.50'.

Note: BASE should be 16 at compile time so that "2E" and "24" will be interpreted correctly. BASE should be 10 at run time so that the numbers will be printed in decimal.

<>   (N1 N2 --- B)

Test if 'N1 <> N2' (N1 not equal to N2).

<JUMP-TAB>   (N ---)

Compiled from JUMP-TAB. Executes the N'th word in the jump table.

=   (N1 N2 --- B)   C

Test if 'N1 = N2'.

>   (N1 N2 --- B)   C

Test if 'N1 > N2'.

><   (X --- X')   C

Synonym for FLIP. Reverses the order of the two bytes in a 16- bit word.

>R   (X ---)   C

Pushes X onto the top of the return stack.

?DUP   (X --- X [X])   C

If X is nonzero, DUPs it, otherwise no effect.

?ENOUGH-ROOM   (B ---)

Use ?ENOUGH-ROOM after ?ROOM to give a stubborn error message if there is not enough room in the intrinsic file space to allocate a TSA.

If enough room exists, B is TRUE and ?ENOUGH-ROOM simply drops it.

If too little room exists, B is FALSE and ?ENOUGH-ROOM presents the user with a menu of non-invisible files. The user may choose to delete any one of these files. If he selects a file to delete, the HHC will clear itself after deleting the file.

If the user does not select a file to delete, the only way to escape from the ?ENOUGH-ROOM menu is to press CLEAR.

?FILE   (A L --- [A'] B)

Given the name of a virtual file, returns its address.

A L are the address and length of the file's name. If the file exists in the current file space, A' is the address of its header, and B is TRUE. If the file does not exist in the current file space, A' is absent and B is FALSE.

?JUMP   (B ---)   C

Compiled from 1F.L and WHILE.L. Followed by a 2-byte signed displacement.

If B is TRUE, passes control to the address of the displacement plus the value of the displacement. If B is FALSE, passes control to the address of the displacement +2.

?KEY   (--- B)   C

Return TRUE if a keystroke is available for KEY, else return FALSE.

?KEY-HIT   (--- B)

If a keystroke is waiting to be read, reads the keystroke, throws it away, and returns TRUE. If no keystroke is waiting to be read, returns FALSE.

?RANGE (N MIN MAX --- N B)

Returns TRUE if MIN=<N=<MAX, FALSE if not.

?ROOM (N --- B)

Determines whether there is enough room in RAM to allocate N bytes of storage or file space. If there is enough room, B is TRUE; else B is FALSE.

?SF (A --- B)

Given the address of a file (i.e. of the file-length word), determines whether the file is an executable SNAP file.

?VISIBLE (A --- B)

Given A, the address of the first byte of a virtual file (see CFILE), ?VISIBLE returns a TRUE if the file is visible (i.e. the % INVISIBLE flag is not set).

@ (A --- X) C

Stacks the word stored at A.

@SA (N A --- A' L)

"Fetch string array." A is the address of an array of count-format strings, such as the one that would be compiled by this code:

```
HERE
    S" STRING ARRAY ELEMENT #0"
    S" STRING ARRAY ELEMENT #1"
    S" STRING ARRAY ELEMENT #2"
    S" STRING ARRAY ELEMENT #3"
== EXAMPLE.STRING.ARRAY
```

N is an index to the array.

A' is the address of string #N in the string array at A.

L is its length.

Example:

```
2 EXAMPLE.STRING.ARRAY @SA TYPE
```

would type, 'STRING ARRAY ELEMENT #2' on the LCD.

^CURSOR (--- A) V2

Stacks the address of the address of the current cursor image definition. You can define a new cursor image by executing the following words:

```
YOUR-CURSOR-DEFINITION ^CURSOR !
```

where YOUR-CURSOR-DEFINITION stacks the address of a 5-byte cursor image definition block (which cannot be in a capsule).

^CURSOR is reset to the HHC's standard cursor definition by a clear.

^HDT (--- A) V2

Pointer to the first HDT.

A

A-ECB (--- A) V2

The ECB used by ALL-EMIT, See FLAME.ON.

A>B (C --- N)

Performs ASCII character to binary conversion according to the following rules:

'0' through '9' are converted to 0 through 9.

'A' through 'Z' and 'a' through 'z' are converted to 10 through 36.

All other input values are converted to binary zero.

A>B is convenient for interpreting a character indicating the user's selection from a menu.

ABORT (---) C

The word executed when a serious error occurs. See 'ABORT for more information.

ABS (N --- N')

N' = ABS(N).

ACVECT (--- A) V2

Stacks the address of the pointer to the current LCD accent table. You may give the HHC a new LCD accent table by changing the contents of ACVECT.

ALL-EMIT (C ---) C

Sends a character to the LCD and to all attached peripheral devices. FLAME-ON stores the tag of this word in 'EMIT.

ALRMBLIP (--- N) =

A mask for the "alarm" blip. See ANDBLIP for details.

AND (X Y --- Z) C

Bitwise logical AND.

ANDBLIP (N ---) C

ANDs the low-order byte of N into the LCD blip byte. If a bit is

1 in N, the corresponding blip is unchanged; if a bit is 0 in N, the corresponding blip is turned off.

Following are the symbolic constants that may be used to designate the bits in the blip byte:

BATBLIP	leftmost blip; may be defined by user
SHFTBLIP	keyboard in shifted state
LOCKBLIP	keyboard in locked state
2SHFTBLIP	keyboard in second-shifted state
RECBLIP	"delete" blip (not used by system)
MEMBLIP	"insert" blip (not used by system)
ALRMBLIP	"alarm" blip (used only by POCSEC)
RUNBLIP	"on line" blip (not used by system)

To turn on a blip, turn on the corresponding bit with ORBLIP. Similarly, to turn off a blip, turn off the corresponding bit with the complement of the corresponding bit mask and ANDBLIP.

The HHC nucleus manipulates SHFTBLIP, 2SHFTBLIP, LOCKBLIP, and ALRMBLIP asynchronously with applications, so it is not a good idea for applications to try to use those blips. RECBLIP and MEMBLIP are safer; only EXPECT uses them from among the HHC's intrinsic software, and so an application that uses them is less likely to get into trouble.

AOECB (--- A) V6

A 6-byte area containing a timer ECB used by the auto-off function.

AOENB (--- FL) =

A flag in FLAG3 that controls the auto-off feature. 1 -> the feature is enabled; 0 -> the feature is disabled.

AOTOKE (--- FL) =

A flag in FLAG3 that is used by the auto-off feature. 1 -> an event has occurred that warrants resetting the auto off timer; 0 -> no such event has occurred. The bit is checked, and the timer may be reset, every 30 seconds. The timer is set to a 10 minute interval; if it reaches the end of the interval, the HHC is turned off.

Note that AOTOKE is meaningful only when AOENB is 1.

AP (--- A) V2

A variable which points to the ceiling of the current TSA.

ASC>FP (A L --- FP ENDLOC NDIGITS)

Converts a floating point number from ASCII format to

internal format. (No exponent part, just signed decimal input).

A and L are the address and length of the ASCII format number. L must be in the range 0<L=<255.

FP is the floating point value. ENDLOC is the address of the first location beyond the [valid part of the] number. NDIGITS is the number of digits and leading zeros in the number.

If ENDLOC<A+L, ASC>FP encountered an error during the conversion; ENDLOC points to the first character considered erroneous, and FP and NDIGITS reflect the value of the number up to that point.

ATTACH (DEVCODE LUN --- B) C

Attaches a device of the type specified by device code DEVCODE to logical unit number LUN. B indicates success or failure of the operation.

If ATTACH finds a device of the specified type that is already attached to a LUN, it will attach the device to the new LUN without detaching it from the old one. For an output device, this causes output from the two LUNs to be mixed together. For an input device, it may cause the new LUN to take the device away from the old one, or may have no effect. See ATTACHX for a word that avoids these undesirable kinds of behavior.

ATTACHX (DEVCODE LUN --- B) C

Identical to ATTACH except for the behavior if it finds a device of the requested type that is already attached to a LUN. ATTACHX ignores the device and searches for another of the same type, returning B=FALSE if it cannot find one.

AVAIL (--- N)

Returns the number of free bytes in the current file space. If the intrinsic file space is current, N is reduced by a safety factor to allow room for numeric conversions between the temporary stack and the files.

For the total number of bytes in the current file space, see FILELEN.

B

B>A (N --- C)

Performs binary to ASCII character conversion according to the following rules:

0 through 9 are converted to '0' through '9'.

10 through 36 are converted to 'A' through 'Z'. 37 is converted to '[' (the next ASCII character after 'Z'), 38 is converted to '\', and so on.

B>A is the inverse of A>B.

BASE (--- A) V1

The current number base used for input and output conversion. Initial value is 10; this value is restored by CLEAR.

BATBLIP (--- MASK) =

A mask for an unassigned blip, which is available for use in applications. See ANDBLIP for details.

BEEP (---)

Produces a "beep" on the HHC's noise maker. Equivalent to sending 07H (ASCII "bell" code) to the LCD.

If LCD output is redirected to another device (by changing the value of 'EMIT, for example), BEEP is redirected too.

BEEPER (--- N) =

A mask which may be ANDed into the system control byte (LATCH) to isolate the squeaker bit. If the bit is 1, a DC-high voltage is placed on the squeaker. If the bit is 0, a DC-low voltage is placed on the squeaker. A program can make sounds by switching the bit on and off, but it is recommended to use SQUEAK to do this.

BIGMOVE (BANK FROMADR TOADR LEN ---)

BIGMOVE moves the contents of a section of bank-switched memory to internal or external RAM.

Bank	high byte is ID of ROM bank (without 80H bit), low byte is ID of destination RAM bank.
Fromadr	the address of the source
Toadr	the address of the destination
Len	the length to move

Note: BIGMOVE will not move data directly from one external RAM bank to a different external RAM; do an intermediate move to internal RAM to accomplish this.

BLANKS (A N ---)

Deposits N blanks in RAM, beginning at address A. Equivalent to '32 FILL'.

BOUNDS (N1 N2 --- N2' N1)

Used to set up the initial and limit for a DO loop. Converts an address and length to ceiling (i.e. N2' = N2 + N1 + 1) and address (i.e. initial value N1).

BUFPOS (--- N)

Fetches the contents (1 byte) of &BUFPOS.

BUFPOSN (--- A) V1

Gives the cursor's current character position in the display, i.e. the position where the next character EMITed will appear. The left-most position is #0. Application programs may move the cursor by changing this variable with the word POSN.

C

C! (C A ---) C

Stores C at address A. Pronounced "C-store."

C@ (A --- C) C

Fetches the character stored at address A.

CALC (---)

Executes the intrinsic calculator.

CALT (--- N) =

"Alternate character set" bit mask for EMIT. See EMIT for details.

CAN.LIST (A C B ---)

Cancels one or more ECB's.

A is the address of a list of ECB addresses; each address points to an ECB to be cancelled or preserved (see B, below).

C is the length of the ECB list, in words.

B = FALSE means "cancel every ECB in the list." B = TRUE means "preserve every ECB in the list, but cancel every other ECB."

CANCEL (ECBADR ---) C

Post ECB as completed, regardless of whether the associated event has occurred.

CAPON (--- N) =

A mask which may be ANDed into the system control byte to isolate the capsule-power bit. If the bit is 1, the hardware is giving the capsules power. If the bit is 0, the hardware is not giving the capsules power.

CBIT (--- FL) =

A flag bit in FLAG2 used to indicate whether the cursor is on. As the cursor flashes, this bit is toggled on and off. Tested by >ENCODE.

CCNT (--- N)

Fetches the contents (1 byte) of &DATA-LEN.

CEMIT (C ---)

Emits a character to the video and LCD. Usable only in the CALCULATOR, which binds the video.

CFILE (--- A)

Returns the address of the current virtual file. After a clear and before any file has been opened, the address is zero. CFILE is used to identify the file in various I/O operations, most importantly in DELETE-FILE. Equivalent to '&CFILE @'.

CFIND (A L C --- [N] B)

Searches string of length L beginning at A for the first occurance of the character C. B is TRUE if C is found and N indicates that C lies at A+N. If B is FALSE, N is not put on the stack. Note: CFIND uses the byte just before the top of the Temporary Stack as a scratch area.

CFLSH (--- MASK) =

"Flash" bit mask for EMIT. See EMIT for details.

CGRTN (---) C

Internal word used by EMIT to process the carriage return character.

CHARBUF (N --- A)

Adds the offset N to the address DBUF of the display buffer.

CHESAV (--- A) V2

Pointer to the current I/O ECB. Used internally by I/O routines.

CHSAV (--- A) V1

Holds the character being output to a peripheral. Used internally by I/O routines.

CHVECT1 (--- A) V2

Stacks the address of the address of the current LCD primary character set table. You can change the HHC's primary character set by changing the contents of CHVECT1.

CHVECT2 (--- A) V2

Stacks the address of the address of the current LCD alternate character set table. You can change the HHC's alternate character set by changing the contents of CHVECT2.

CLEAR (---) C

Simulates the effect of the user pressing the CLEAR key once. This effect is a hard clear if this is the second CLEAR (from software or keyboard) after any other keyboard key was pressed; otherwise it is a soft clear.
See also SOFT.CLR and HARD.CLR.

CLIT (--- C)

Compiled from a literal with a value in the range 0-0FFH. Followed by one byte giving the value of the literal. At run time, CLIT stacks the contents of the next byte, left-padded with a zero byte.

CLIT2 (--- N) C

Compiled from a literal with a value in the range 200H-2FFH. Followed by one byte giving the value of the literal minus 200H.

CLIT3 (--- N) C

Compiled from a literal with a value in the range 300H-3FFH. Followed by one byte giving the value of the literal minus 300H.

CLR.BITS (A X ---)

Changes the value of the word at A by zeroing each bit that corresponds to a bit that is on in X. See also TOGGLE and SET.BITS.

CMOVE (FROM TO LEN ---) C

Moves LEN bytes from address FROM to address TO. The

bytes are moved one at a time, lowest first. Thus, CMOVE will cause the field to overlay itself if TO is greater than FROM but less than FROM+LEN .

MOVE is a similar instruction that does not overlay data.

CNEG (--- N) =

"Negative" bit mask for EMIT. See EMIT for details.

COLD.START (X X X ... X ---) C

Called during a cold start of the HHC. Among other functions, it empties the parameter stack.

COPY-FILE (---)

The "COPY FILE" function of the FILE SYSTEM application.

COUNT (A --- A' LEN)

Converts the address of a string to the address of the string's data and the length of the string's data.

CPUON (--- N) =

A mask which may be ANDed into the system control byte to isolate the CPU-on bit. If the bit is 1, the CPU is on. If the bit is 0, the CPU is off.

CQMK (--- N) =

A mask for the "flash/question-mark" bit in the high-order byte of an EMIT parameter. If the bit is on, EMIT displays a character alternating with a '?'. If the bit is off, EMIT displays a character in the usual way.

CR (---)

Causes a carriage return on the LCD (and devices bound to the LCD). If the device is the LCD, clears the cursor from the freeze point (if any) to the right edge, and returns the cursor to the leftmost position (depending on rotation mode).

If the the most recent keystroke was input after the most recent LCD character was displayed, CR causes an immediate carriage return.

If the most recent LCD character was displayed after the most recent keystroke was input, CR works by turning on a flag which is later tested by the LCD output routines. The next time a character is sent to the LCD, the routines test this flag to see if a carriage return is pending; if so, they do the carriage return (and reset the flag) before writing the character.

In addition, each time a character is written to the LCD, a timer is set; and if user writes a CR and then another character before the timer expires, the routines wait until the timer expires before doing the carriage return. This gives the user a guaranteed minimum length of time to look at a line on the LCD after it is completely displayed and before it is erased. (RUNBIT in FLAG2 can be used to avoid waiting, see also FAST.CR).

CRBIT (--- FL) =

Bit in FLAG2. 1 -> an LCD carriage return is pending.

CRESTORE (---)

Pops two words from the return stack to the parameter stack, then stores the low character of the first word at the address indicated by the second word. See SAVE.

CSBIT (--- N) =

A mask for the byte in a ROM capsule at the address given ROM. If the bit is off, this ROM capsule has fast ROM. 16K ROMs are slow ROMs; all other ROMs are fast ROMs (slow ROMs take longer to fetch data).

CSPEED (--- A) K1

A field in the header of a ROM capsule. This field contains bit flags which describe the capsule. The following symbols represent bit masks for the bits currently used:

PROGBIT 1: capsule name should appear in Main Menu. Generally because it contains a program.

CSBIT 1: capsule uses slow ROM. 0: capsule uses fast (normal) ROM.

CTEXT (--- A) K1,NOTE

Tag table number of the primary external tag table of an application in a control ROM.

Note that the value of the "constant" will vary depending on what control ROM is currently bank-switched in the address space. SET.CTRL or SET.CROM can be used to specify which ROM.

CTRFLAG2 (--- A) K1

Control ROM byte containing complex I/O flags.

CTRFLAGS (--- A) K1

Control ROM byte containing the peripheral's hardware device type code.

CTRID (--- A) K1

The address in a control ROM where the I/O key's name of

the device the name (e.g. 'MODEM') is stored in count format.

CTROM (--- A) =

Stacks the address of the part of the address space where device control ROMs go.

CTVECT (--- A) K2

A vector used to point to the main tag table (if any) of an application in a Control ROM. See CTEXT, ROMVECT.

CURBIT (--- FL) =

A bit in FLAG2. 1 -> turn the cursor on.

CURCTL (--- A) V1

Bank ID of the ROM bank currently switched into the peripheral control ROM address space. (ID=4*Slotnumber)

CURPOS (--- N)

Returns the contents (one byte) of &CURPOS.

CURRAM (--- A) V1

Bank ID of the RAM bank currently switched into the extrinsic RAM space. (ID = 8*(Slot number +1)+N where N is a number from 0 through 7 specifying the bank within the RAM module. Internal RAM ID is zero.)

CURROM (--- A) V1

Bank ID of the ROM bank currently switched into the ROM capsule space. (See CURRAM to compute external ROM IDs, just add 80H to the result. The ID's for the 3 slots in back of the HHC are 81H, 82H, 80H from left to right.)

D

D+ (ND1 ND2 --- ND3) C

ND3 = ND1 + ND2.

D- (ND1 ND2 --- ND3)

ND3 = ND1 - ND2.

D.R (ND N ---)

Display ND in ASCII, according to the current BASE. The number is right-aligned in a field N positions wide.

If the "natural" width of the number exceeds N, no truncation occurs, and no leading spaces are printed.

D= (ND1 ND2 --- B)

Compares two double numbers. B is TRUE if ND1 = ND2.

DABS (ND --- ND')

ND' = ABS(ND).

DBUF (--- A) V27

Character buffer for the LCD. Contains ASCII characters representing the 27 character positions of the LCD. (Only the first half of the 27th position is displayed.)

DBUF is used internally to manage the LCD display. It is not normally used by applications.

Putting a character in DBUF does not update the LCD until the next time the affected part of the LCD is redisplayed. See 'UPDISP'.

(See also DBUF1 and CHARBUF.)

DBUF1 (--- A) V27

Control buffer for the LCD. Each byte contains bit flags that control whether the character in the corresponding byte of DBUF is displayed with blink, flashing, etc. The control byte normally is deposited in DBUF1 by EMIT, which gets it from the high-order byte of its parameter.

See also DBUF and UPDISP.

DBUFL (--- N) =

The length of the LCD in characters (27).

DECB (--- A) V6

A timer ECB used by SECS and the LCD control routines. See #DECB.

DEL.CH.R (N ---)

Shifts the right part of the LCD display left one character. The character in position N+1 is shifted to position N; all following characters are shifted after it. The the last position is set to the most recently emitted character. See also INS.CH.R.

DELETE (N --- B)

Deletes the N'th record in the current virtual file. If the N'th record does not exist, B is false.

DELETE-FILE (A ---)

Deletes a file. A is the address of the first byte of the file to be deleted.

Use ?FILE to find the address of the first byte of the file, given its name.

DISP.CH (N --- N')

Used internally by (EMIT) to display a character. Given buffer position of cursor before display, returns buffer position of cursor after display.

DISPON (--- N) =

A mask which may be ANDed into the system control byte to isolate the LCD display bit. If the bit is 1, the LCD display is on. If the bit is 0, the LCD display is off.

DMOVE (FROM TO LEN ---) C

"Display move." Used to read and write the LCD at the dot level.

You read the LCD at the dot level by moving data from the LCD buffer, a memory-mapped area, to RAM. You write into the LCD at the dot level by moving data from RAM or ROM to the LCD buffer.

The LCD buffer is 160 bytes long. The buffer's address is given by the symbolic constant DSPLY. Each byte represents one LCD dot column; the byte at DSPLY represents the leftmost dot column; the byte at DSPLY + 158 represents the rightmost dot column; and DSPLY + 159 represents the blips.

Within each byte of the buffer, the high-order bit (80's bit) represents the bottom dot, and the low-order bit (1's bit) represents the top dot.

FROM is the address data is to be moved from; TO is the address data is to be moved to. Either FROM or TO can be an address in the LCD buffer. LEN is the number of bytes of data to be moved, that is, the number of dot columns to be read or written.

DMOVE is actually more general than just allowing direct access to the display. It also enables the I/O pages, and can be used to read and write them as well.

Caution: before starting to use DMOVE for LCD graphics, clear the LCD by executing the following code:

```
LCD.CR          \ Clears LCD at 'DMOVE' level.
STOP.CURSOR     \ Makes cursor invisible.
26 POSN         \ Moves cursor to char posn 26.
```


When you are ready to begin writing characters to the LCD again (e.g. with EMIT), execute the following code:

```
LCD.CR          \ Clears LCD at 'DMOVE' level.
START.CURSOR    \ Makes cursor visible again.
CR              \ Moves cursor back to left edge.
```

Note: '26 POSN' moves the cursor into character position 26. This places it half in dot columns 156-158, and half off the LCD. With the cursor in this position, dot columns 156-158 are not available for your use. This is necessary because (1) an HHC bug makes it impossible to write to dot columns under the cursor, and (2) it is impossible to move the cursor entirely off the LCD.

DNEGATE (ND --- ND') C

ND' = -1 * ND.

DO-EDIT (N ---)

Used to edit a text file. N is the line number where the editing should begin. You can only exit DO-EDIT with a CLEAR.

DODOES (--- A) L

Compiled from DOES> in a <BUILDS / DOES> word. Stacks the address of the data following the next tag; then executes the next tag, which represents the DOES> code.

DOLATCH (--- A)

Vectored assembler routine used to update the LATCH. Load the value to be stored into the Accumulator, DOLATCH will set both LATCH and its software shadow LATCHS.

DROP (X ---) C

Drops the top entry from stack.

DSIZE (--- N) =

Size of LCD in characters (the value is 26).

DSPFLAG (--- FL) =

Bit in the HDT byte 0; 1 -> this HDT is for the LCD.

DSPLY (--- A) =

Address of the 160-byte memory mapped LCD I/O area. You can read from and write to the LCD at the dot level by moving data to and from this area with DMOVE.

DUP (X --- X X) C

Duplicates X.

DVCON (--- FL) =

Bit in HDT byte 1; 1 -> this device is on, or will be turned on the next time DVCSET is called.

DVCSET (---) C,NOTE

Scans the HDT's and turns all devices on or off, depending on the value of the DVCON bit. Note: DVCSET will not execute in a capsule, but you can execute it by moving the following code to internal RAM and executing it there:

```
LABEL 'DVCSET
]
   LATCHS C@
   DVCSET
   DUP LATCHS C!
   LATCH C!
   EXIT
[
HERE 'DVCSET - == DVLEN

:  TO.DVCSET
   'DVCSET &LBUF DVLEN MOVE
   &LBUF IP !
;
```

DVSAV (--- A) V1

Page 0 location used by I/O routines to store the on/off status of a device during an I/O operation.

E

E-L (TAG N --- [A] B)

Locates the N'th file in the current file space that satisfies the criteria of the file-selecting word associated with TAG. (See the discussion of 'FILE-COND regarding file-selecting words.)

If the file exists, A is its address and B is TRUE. If the file does not exist, A is absent and B is FALSE.

E.T (---)

Routine used by the CLOCK/CONTROLLER to set the time. STM is preferable.

EDIT-FILE (MAXLEN EOLTAG EOFTAG LINE# ---)

Invokes the file system editor on the current file, uses EXPECT and &LBUF.

MAXLEN is the maximum length a record may have.

EOLTAG is the tag of a word that takes an input character from the stack and returns TRUE iff that character indicates "end of line." The word EOL is most commonly used.

EOFTAG is the tag of a word which takes an input character from the stack and returns TRUE iff that character indicates "end of file."

LINE# is the number of the first line (*i.e.*, the first record) in the file to be edited. Note that the user may move the cursor forward or backward from this line if he wishes.

Note that EOFTAG is called only when EOLTAG returns TRUE. Thus every character that is to be recognized as an EOF character must also be recognized as an EOL character.

EMIT (C ---)

Send character C to display.

The high-order byte of C consists of flags which determine how the character is to be displayed. Turning each bit on applies the corresponding attribute to the character:

CALT	displays from the alternate character set.
CFLSH	displays a flashing character.
CNEG	displays the character in inverse-image.
CQMK	displays the character alternately with a question mark.
8,4,2,1	constitutes a number, 0-15. This number is used as an offset into the floating accent table if one is defined; the accent is superimposed on the character.

EMIT.ESC (DATACHAR CTRLCHAR ---)

EMITs an escape control sequence with the escape control character CTRLCHAR and the data character DATACHAR. Equivalent to 'HEX 1B EMIT EMIT EMIT'.

ENCODE (---) C

Used internally to manage LCD display. Not normally used by applications.

(Converts ASCII character to LCD dots.)

EOL (C --- B)

The HHC's standard end-of-line word for EXPECT and EDIT- FILE. An end-of-line word defines what characters will be considered to end a line of input when read by EXPECT. See EXPECT for details of how to use the word.

EOL accepts 4 keys as end-of-line characters: ENTER, ♠, ♥, and SEARCH.

ERASE (A LEN ---)

Clear an area at A, of length LEN bytes. Each byte in the area is set to binary 0.

ESCCC (--- N) =

Escape control sequence opcode "Set Control Character Mode." If the data byte is TRUE, subsequent non-executable characters sent to the device will be displayed; if FALSE, subsequent control characters sent to the device will not be displayed. Executable control characters such as Bell, ESC and CR are not affected.

ESCDC (--- N) =

Escape control sequence opcode "Display Character." DATACHAR (see EMIT.ESC)will be displayed, even if it is a control character that would normally be executed. For example, if the next character is 0DH (carriage return) and it would normally cause a physical end-of-line on the device, it causes no carriage end-of-line, but is written as an inverse M.

ESCDR (--- N) =

Escape control sequence opcode "Delete Right." The character under the cursor is deleted; following characters on the line are moved left. The data byte is used as a fill character at the end of the line.

ESCFL (--- N) =

Escape control sequence opcode "Flush I/O Buffer." Characters in the device's I/O buffer are read or written, emptying the buffer. (This operation is generally applied only to output devices that write a line of data at a time. Writing a line would normally be triggered by a CR character.) Data byte ignored.

ESCHM (--- N) =

Escape control sequence opcode "Home Cursor." This normally returns the cursor to the upper left corner of a device with a two-dimensional display. Data byte ignored.

ESCIR (--- N) =

Escape control sequence opcode "Insert Right." The data byte is displayed at the cursor. The character under the cursor, and all following characters on the line, are pushed to the right. The rightmost character is pushed off the end of the display (or buffer).

ESCSF (--- N) =

Escape control sequence opcode "Set Flash Mode." Subsequent output will be displayed in flashing characters. Reverses the effect of ESCUF. Data byte ignored.

ESCSI (--- N) =

Escape control sequence opcode "Set Inverse Mode." Subsequent output is displayed in inverse. Reverses the effect of ESCUI. Data byte ignored.

ESCUF (--- N) =

Escape control sequence opcode "Unset Flash Mode." All subsequent output is displayed in non-flashing characters, until the mode is changed by ESCSF. Reverses the effect of ESCSF. Data byte ignored.

ESCUI (--- N) =

Escape control sequence opcode "Unset Inverse Mode." Subsequent output will be displayed normally (non- inverse). Reverses the effect of ESCSI. Data byte ignored.

ESCUN (--- N) =

Escape control sequence opcode "LCD unescape." On the LCD, the data byte is written normally. On all other devices, it is ignored.

ESCWB (--- N) =

Escape control sequence opcode "Set Word Break." The data byte defines the word break character that can trigger automatic word wrap. (This means that when an output line becomes longer than a device's line length, the device automatically starts a new line and moves the last word of the old line to the start of the new line so that the word will not straddle a line break.)

The word break character is initially blank (20H) in all cases. The LCD does not support word breaking.

Setting the word break character to 0FFH turns automatic word breaking off.

EVECT (--- A) =

Stacks address of a JMP instruction used by the inner loop to decode tag tables.

EVFLAG (--- A) V1

Event processor service flag. 1 -> call event processor the next time NEXT is executed. 0 -> do not call event processor.

This byte may not have any value except 0 or 1. It is changed with increment and decrement instructions, so that an interrupt cannot occur while it is being updated.

EVFLAG1 (--- A) V1

Event processor service flag. 1 -> an event is being processed. This flag prevents the HHC from re-entering the event processor when an event occurs while another event is being processed.

EXECUTE (TAG ---) C

Executes the word denoted by the TAG.

Here is a simple (and pointless) example of how EXECUTE can be used:

```
: YOUR-TOP-WORD ... ;
'X YOUR-TOP-WORD EXECUTE
```

This code fragment executes a word named YOUR-TOP-WORD. 'X YOUR-TOP-WORD stacks the tag of YOUR-TOP-WORD.

See also FLEE.

EXIT (---) C

Used inside a colon definition, terminates execution of that colon definition, returning control immediately to its caller. The contents of the parameter stack are not disturbed.

Note: do ***not*** use ;S in this context, as you would in Fig-FORTH.

Note: do ***not*** use EXIT inside a DO loop. It will cause the loop stack to grow by one entry each time it occurs.

EXPAND (A L --- B)

Expands the used file space by L bytes starting at address A. B is TRUE if this can be done successfully. B is FALSE if there is no room to expand or if A is not in file space.

EXPECT (A L I C E M --- A L' K M')

Reads a line of text from the keyboard. EXPECT permits the

user to edit the line of text extensively. EXPECT can also display a line of initial text, and allow the user to add to it and/or edit it.

A: Address of the buffer where EXPECT is to deposit the line it reads. This buffer can be &LBUF.

L: Length of the buffer indicated by A. The maximum allowed length is 256. (The length of &LBUF is 80.)

EXPECT will deposit up to L characters in the buffer. Unlike many FORTH implementations of EXPECT, it will ***not*** deposit a NULL after the last character.

I: Length of initial text in buffer. If I=0, EXPECT reads a completely new line. If I>0, EXPECT considers the buffer to contain I characters, and displays that many characters on the LCD before beginning to read input.

C: Initial position of cursor.

If C=0, EXPECT scrolls the cursor through the line from the beginning to the end, stopping when it reaches the end or when the user presses any key.

If 0<C<I, the cursor is placed in position (C-1), origin 0. For example, if C=3, the cursor is placed in position 2 (over the third character in the buffer).

If C>=I, the cursor is placed just after the last character in the buffer.

If C>26, EXPECT cannot position the cursor without scrolling the beginning of the line off the left edge of the LCD. In this case it scrolls the line as much as necessary to bring position (C-1) under the cursor. It positions the cursor in the right half of the LCD (if C-1 is near the end of the text) or in the approximate center of the LCD (if not).

E: Tag of a word written to detect the end-of-line (EOL) character.

EXPECT calls this EOL word each time a character is read from the keyboard. EXPECT finishes editing the line when the EOL word finds an EOL character.

The linkage to the EOL word is:

(C --- B)

where C is a character that has been read from the keyboard, and B is TRUE if the character is an EOL character.

The standard EOL word used by the file system editor is named 'EOL'.

M: The initial editing mode EXPECT should operate in. The following symbolic constants represent the possible modes:

%OVR	Overstrike. This is EXPECT's "normal" mode.

Each input character replaces the character under the cursor, and causes the cursor to move right.

If the cursor is at end-of-line, each input character is appended to the line.

The left- and right-cursor keys move the cursor; but an attempt to move the cursor past the beginning or end of the text will not succeed, and will make the HHC beep.

%INSERT — Insert. Each input character is inserted just before the character under the cursor. All following characters are bumped one position to the right. EXPECT returns to overstrike mode after receiving one character.

If the cursor is at end-of-line, each input character is appended to the line.

%DELETE — Delete. The "cursor left" key deletes the character under the cursor, and moves the cursor to the left, over the previous character; the "cursor right" key deletes the character under the cursor and leaves the cursor where it was which is now the next character. Any other key is considered an error, and causes EXPECT to beep.

EXPECT returns to overstrike mode after receiving one character.

%LI — Lock-insert mode. Identical to insert mode except that EXPECT does not return to overstrike mode after receiving one character.

%LD — Lock-delete mode. Identical to delete mode except that EXPECT does not return to overstrike mode after receiving one character.

L': Length of the text in the buffer when EXPECT finished processing.

K: The character which caused the EOL word to return TRUE.

M': Editing mode that applied to the last character received from the keyboard.

F!   (FP A ---)

Stores a floating point number.

F#   (FP N --- FP N)

Used by FP>ASC to assist in formatting the ASCII representation of a floating point number.

Used between <# and #> to put a digit of the mantissa in the TOB. To format the entire mantissa, FP>ASC executes a loop for N from number of significant digits desired minus 1, to 0. For each digit, if the digit is not a trailing 0, F# converts the digit to ASCII, deposits it at the address indicated by HLD, and decrements HLD by 1.

F*   (FP1 FP2 --- FP3)

Leaves the product of two floating point numbers. Performs special case handling.

F+   (FP1 FP2 --- FP3)

Leaves the sum of two floating point numbers. Performs special case handling.

F-   (FP1 FP2 --- FP3)

Leaves the difference of two floating point numbers. Performs special case handling.

F.   (FP NDIGITS --- N)

Displays a floating point number in standard notation (not exponential notation), using EMIT.

FP is the number to display. NDIGITS is the number of significant digits to display. N is the number of digits displayed after the decimal point. (e.g. N=1 for FP=706.3)

Note: If FP equals zero, F. trashes the top byte of the TSA. So execute FSTANDTYPE to detect special cases and branch accordingly. For example:

```
1 FSTANDTYPE
IF
   FDROP
   ." 0"
ELSE
   8 F.
   ENDIF
```

F.EXT (N M ---)

Used after 'F.' to pad a floating point number to a specified number of digits to the right of the decimal point.

N is the number of digits to the right of the decimal point in the floating point number just displayed. (It is the value left on the stack by 'F.'.) M is the desired number of digits. F.EXT extends the number to the desired length by displaying M-N zeroes. For example:

'493.1 7 F.' displays

493.1

'493.1 7 F. 2 F.EXT' displays

493.10

F/ (FP1 FP2 --- FP3)

Leaves quotient of two floating point numbers. Performs special case handling.

F< (FP1 FP2 --- B)

Tests if FP1 < FP2.

F@ (A --- FP)

Fetches a floating point number.

FABS (FP --- FP')

Sets absolute value of floating point number.

FALSE (--- B) =

Pushes a FALSE onto the stack. The binary value of FALSE is 0. FALSE is a synonym for 0.

FALSIFY (X --- FALSE)

Drops the top word on the stack and stacks a FALSE (0).

FAST.CR (---)

Performs a carriage return on the LCD immediately. Distinguish FAST.CR from CR, which sets flags which will cause a carriage return to occur at some future time.

FDROP (FP ---)

Drops four words (one floating point number).

FDUP (FP --- FP FP)

Duplicates the floating point number (four words) on the top of stack.

FECB (--- A) V6

A timer ECB used by the LCD control routines to control cursor flashing.

FILE-TAG (A --- A' L TAG B')

Used by the FILE SYSTEM. The code is equivalent to COUNT 'X TYPEDROP TRUE.

FILELEN (--- LEN)

Returns length of the current virtual file space (i.e. of the intrinsic or extrinsic file space—***not*** the length of the current file).

If the intrinsic file space is current, its length is reduced by a safety factor to allow room for numeric conversions between the end of the last file and the top of the temporary stack.

For the amount of free space remaining in the current file space, see AVAIL.

FILEORG (--- A)

Address of the origin of the current virtual file space (i.e. of the intrinsic or extrinsic file space). See 'FILEORG.

FILESPACE (--- A) =

An internal word in the virtual file system.

Stacks address where the intrinsic virtual file space begins.

FILL (A LEN C ---)

Fill an area in memory at address A, length LEN, with the byte value C.

FIO-ERR (X Y --- B)

Equivalent to 2DROP FALSE.

FLAG1 (--- A) V1

A byte of flags relating to keyboard status.

FLAG2 (--- A) V1

A byte of flags relating to keyboard and LCD status. It controls the function of the function keys, the functions of the SHIFT, 2nd SFT and LOCK keys, and the LCD rotation/carriage return delay, among other things. See FUNBIT and PASSRAW.

FLAG3 (--- A) V1

A byte of flags relating to various functions of timer inter-

rupts. It controls the auto-off feature, among other things. See AOTOKE and AOENB.

FLAME.ON (---)

Attaches all available output devices and binds them to the LCD. Use this word if you want your application to echo the LCD's display on anything that happens to be available, e.g. the video.

Once the available devices are bound to the LCD, they remain so until the next soft clear.

FLEE (TAG ---)

Sets SOFTAG equal to TAG, then executes the word identified by the tag. Note, control never returns to the word that executed the FLEE; the word FLEE'd to returns control directly to the operating system.

Here is a simple example of how FLEE can be used:

```
'X YOUR-TOP-WORD FLEE
```

The primary menu starts an application by doing a FLEE, so that pressing the CLEAR key while the application is running will restart the application. If an application has several sub-modes, it may use FLEE to start each of the sub-modes, so that pressing CLEAR will re-start a sub- mode instead of re-starting the whole application.

FLEE.CAP (---)

Executes the mother tag of the external tag table in the capsule currently switched into the capsule address space.

FLEE.CROM (DEVCODE ---)

FLEEs to the application program, if any, stored in the control ROM for the peripheral with the specified device code.

FLIM (--- A)

Returns the address of the first free byte in the current virtual file space.

FLIP (X --- X') C

Exchanges the high- and low-order bytes of X.

FLIT (--- FP) C

"Floating point literal." Places the next four words after this tag on the stack, and passes control around the words.

At run time, FLIT works like the run-time words LIT (which stacks a word literal) and CLIT (which stacks a character literal). FLIT is compiled by FLITERAL, e.g. when you type

```
: PI 3.1415 ;
```

or when you type

```
[ 3.1415 2. F/ ] LITERAL
```

FNEGATE (FP --- FP')

Stacks the negative of FP (i.e. -1*FP).

FOVER (FP1 FP2 --- FP1 FP2 FP1)

Copies the second floating point number on the stack to the top of the stack.

FP (--- A) V2

The address of the first free byte in intrinsic file space. Maintained by the file system.

FP>ASC (FP NDIGITS --- A L NEXP)

Converts a floating point number from internal form to exponential notation.

FP is the number to be converted. NDIGITS is the number of significant digits desired in the result.

ADR LEN are the address and length of the mantissa, which is in ASCII. NEXP is the exponent, which is a (binary) integer.

The structure of the ASCII string is of the form

```
Sn.nnnn
```

S is the sign: '-' for negative; a blank for positive. The decimal point is always immediately to the right of the first digit in non- zero numbers. No trailing zeros are appended. A floating point zero has the form S0, where S is generally a space.

WARNING: ADR is always on top of the HHC's temporary stack, and a portion of RAM is also used by the word '.', 'F.', and by FP>ASC. If you are not going to use the ASCII string immediately, it is a good idea to move it to your own RAM area or make sure that '.', 'F.', and 'CFIND' are not executed until the string is used.

FROUND (FP NDIGITS FENCE --- FP' NDIGITS')

Used before F. to round a floating point number to a specified number of digits.

FP is the number to be rounded.

NDIGITS is the maximum desired number of significant digits. If NDIGITS>13, only FENCE rounds.

FENCE is the desired position of the last significant (i.e.

possibly non-zero) digit in the rounded number, relative to the decimal point. The value of FENCE is interpreted as follows:

.
.

-2 last significant digit is in 0.1's column.
-1 last significant digit is in 1's column.
0 last significant digit is in 10's column.
1 last significant digit is in 100's column
2 last significant digit is in 1000's column.

.
.

If FENCE<-400H, only NDIGITS rounds,

FP' is the rounded number.

NDIGITS' is the actual number of significant digits.

Note: FP is not actually rounded. The last significant digit is fixed so that when the number is printed with NDIGITS' significant digits (and truncation) it will ***appear*** rounded.

Example: we want to print 1234.567 with 5 significant digits, leaving one digit right of the decimal point. We can do this as follows:

```
1234.567 5 -2 FROUND F.
```

FROUND changes the number to 1234.667, and leaves a 5 on the stack. F. prints 1234.6 and leaves a 1 on the stack.

Another example: we want to print 1234.567 with 2 significant digits, with the last significant digit 3 left of the decimal point. We can do this as follows:

```
1234.567 2 1 FROUND F.
```

FROUND leaves the number as 1234.567 and leaves a 4 on the stack. F. prints 1200 and leaves a 0 on the stack.

Note: if $|FP| < 10^{FENCE}$, then FP' will be an underflow.

FSPACE (--- A)

Returns the address of the first file in the current file space. If the file space is empty, returns the address where the first file will go.

FSTANDTYPE (N --- TYPE)

Returns the special-case type of a floating point number.

N is the stack position at which the number starts. For example, if the number starts immediately under N, then N should be 1 (N itself occupies position 0!). If the number starts under N and one other floating point number, then N should be 5, and so forth.

The possible values of TYPE are:

0 -> valid non-zero number.
1 -> zero.
2 -> underflow.
3 -> overflow.
4 -> complex number.
5 -> zero divide.
6 -> enigma or user-defined code (number contains a codes 6 to 7F).

Notice that the codes representing special cases are all 1 greater than the numbers of the cases they represent. This is because "valid number" and "zero" have distinct codes.

FSWAP (FP1 FP2 --- FP2 FP1)

Swaps two floating point numbers.

FUDGE-FLIM (--- A)

Modification of FLIM used to protect the Temporary Stack.

FUNBIT (--- FL) =

A flag in FLAG2 which controls operation of the function keys. 0 -> the keys are processed normally. 1 -> iF passraw is also set, the ASCII value of the function key is passed through the keyboard buffer.

If FUNBIT is set and PASSRAW is clear, the result is undefined.

G

GET-TYPE (--- A) I

Returns the address of the file-type byte for the current virtual file. You can read the file-type byte with C@ and set it with C!.

The value of the file-type byte is the logical OR of the file-type attributes that apply to this file. The symbolic constants defining the file-type attributes are:

%INVISIBLE: file will not be displayed in file selection menus.

%EXECUTE: file contains executable SNAP code.

%TEXT: file contains ASCII text. (This is the only type of file that may be edited. Note that %TEXT and %EXECUTABLE are logically incompatible.)

%BASIC file contains a Microsoft Basic program. The program is stored in ASCII except for Basic keywords, which are represented by one-byte binary codes. ***Note***: the symbol %BASIC is not pre-defined. If you need it, assign it the value 10H.

Note: this word compiles into TWO tags, representing the words CFILE and 2+.

GET.GMT (--- DSEC TICKS) C

Get approximate time of year (Greenwich Mean Time). DSEC is an unsigned double integer giving seconds since midnight on 1 January 1980. TICKS is an unsigned integer giving clock ticks.

GET.LOCAL (--- DSEC)

Get approximate time of year (local time).

At present, GET.LOCAL returns the same value as GET.GMT. It may return a different time in a future version of the HHC.

GETMEM

Vectored assembler routine used to request page zero workspace. Put the number of bytes you want into the Accumlator; if GETMEM returns nonzero, that is the address of the available space (taken from the parameter stack-SL is changed). If a zero is returned, the space is not available.

GETMEMT

Similar to GETMEM, except that it reserves RAM from the Temporary Stack instead of page zero. Again, put the number of bytes into the Accumulator when calling GETMEMT, but now the Carry Flag is set when GETMEMT is successful and cleared when it isn't. The contents of TP are set to point to the space made available. Used by GRAB and I/O driver routines.

GMT (--- A) V4

Exact time of century of the next timer interrupt, in seconds.

GMT points one byte past GMTF; that is, GMT points to the 2nd byte of the 5-byte area at GMTF.

GMTF (--- A) V5

Exact time of century of the next timer interrupt, in clock ticks.

GMTF may be used to get a time of century that is more accurate than that obtainable with GET.GMT, the usual choice for getting time of century. GET.GMT is accurate only as of the most recent interrupt.

To get an accurate time of century, do a SET.DELAY for a short time period (e.g. 0.1 second). Immediately after setting the delay, check to see that the delay has not yet expired (i.e. the associated ECB is not yet posted). If not, get the time with GMTF; it is in the future by no more than the duration of the delay that you set.

GRAB (N --- [A] B) C

Determines whether there is enough space in intrinsic RAM to allocate N bytes of storage space. If there is enough space, returns the address of the space and TRUE. If there is not enough space, returns FALSE.

The allocated space is, in effect, an extension to the current TSA. If the program releases the TSA with LETGO, it will release the space allocated by GRAB, as well. GRAB resets TP.

H

HARD.CLR (---) C

Performs a hard clear, as though the user had pressed the CLEAR key twice in succession.

HDTSAV (--- A) V1

An internal word used by the I/O routines. Saves offset to current HDT.

HDTSZ (--- N) =

The size of an HDT in bytes (6).

HINTFLAG (--- FL) =

An unused bit flag mask in HDT byte 1.

HLD (--- A) V2

Used by HOLD to maintain a pointer.

HOLD (C ---)

Used to insert an ASCII character into an ASCII number being created by "<# #S #>". May be used to insert a decimal point, for example. See "<#" for details. Uses HLD as pointer.

I

I   (--- N)   C

Used in a DO loop to copy the loop index onto the stack.

I/O   (--- A)   =

Address of the "I/O page," a memory mapped region used by device drivers and intrinsic I/O routines. Note that the so-called I/O page does not begin on a 6502 page boundary.

IDICT   (--- A)   =

Stacks the address of the short intrinsic tag table.

You can stack the entry point of a word represented by a short intrinsic tag (00H to 0BFH) by loading the word at the following address, where "tn" represents the tag number:

```
IDICT tn 2* +
```

You can stack the entry point of a word represented by a long intrinsic tag (100H to 1FFH) by loading the word at the following address, where "tn" represents the tag number:

```
HEX
IDICT SHINT 2* +
tn 100 - 2* +
```

INS.CH.R   (N ---)

Shifts the right part of the LCD display right one character. the character in position N is shifted to position N+1; all following characters are shifted after it. The original contents of the last position is lost; the contents of position N is set to the last character emitted. SEE ALSO DEL.CH.R.

INSERT   (A L N --- B)

Inserts a new record before the N'th record of the current virtual file. The new record consists of the data at address A, length L. B is TRUE if the insertion was successful.

To add a record to the end of the file, use N = the number of records in the file (one greater than the record number of the last record). The word REC-CNT returns this value.

INTAPS   (N --- TAG)

A TABLE holding the tags of the HHC's four intrinsic applications. For example,

```
1 INTAPS EXECUTE
```

would enter the CLOCK/CONTROLLER

INTFLAG   (--- FL)   =

A flag in PCA byte 3. 0 -> an interrupt is being requested from this peripheral; 1 -> no interrupt is being requested.

IOSAV   (--- A)   V1

Used internally by I/O routines to hold a code identifying the peripheral controller ROM entry point to call.

IP   (--- A)   V2

The SnapFORTH Interpretation Pointer. It points to the next tag to be executed. See IPBANK.

IPBANK   (--- A)   V1

Bank ID of the memory bank the tag now being interpreted is in.

IPBIT   (--- FL)   =

A bit in FLAG2. 1 -> the HHC is in input mode; i.e. a keystroke has been encountered since the last EMIT was executed.

IRQVECT   (--- A)   V3

Address of a JMP instruction that jumps to the entry point of a user IRQ routine. You can intercept IRQ's by changing IRQVECT to jump to your own IRQ handler. Use extreme caution when you do this.

IRQX   (--- A)   V1

Area where the nucleus saves the X register before calling the IRQ handler in a peripheral control ROM. The X register always points to the PCA in this context (when added to I/O). See also IRQY.

IRQXV   (--- A)   V3

Address of a JMP instruction that jumps to the entry point of the IRQ routine in the peripheral control ROM currently switched into the address space.

IRQY   (--- A)   V1

Area where the nucleus saves the Y register before calling the IRQ handler in a peripheral control ROM. The Y register always points to the HDT in this context (when added to the contents of ·HDT). See also IRQX.

IVECT1   (--- A)   V3

Used internally by NEXT to decode tags and jump to the code.

IVECT2 (--- A) V3

Used internally by NEXT to decode tags and jump to the code.

J

J (--- N) C,NOTE

Copies the value of the index of the second outer nested DO loop onto the stack.

Note: in other implementations of FORTH, there is a restriction that both loops and the reference to 'J' must be in the same colon definition. In SNAP, this restriction does not apply.

JUMP (---)

A run time support word generated by ELSE.L. Followed by a 2- byte signed displacement.

Transfers control to the address of the displacement plus the value of the displacement.

This word may also be used to patch compiled high-level code, although (CALL) is usually more convenient because it returns control to the point after the patch.

K

K (--- N)

Copies the value of the index of the third outer nested DO loop onto the stack. See J.

KB (--- A) =

Address of the keyboard's hardware I/O area. When the user presses a key, a binary value is stored somewhere in this area. The "raw keyboard code" is derived from the value that is stored and the address of the byte that the value is stored in.

KBFLAG (--- FL) =

A flag in HDT byte 0. 1 -> this HDT represents the keyboard.

KBREST (---)

Paired with KBSAVE to restore keyboard variables (e.g. KBVECT, ACVECT) from the return stack.

KBSAVE (---)

Saves keyboard vectors for restoration by KBREST.

KBVECT (--- A) V2

Address of the current keyboard translation table. You can change the HHC's current keyboard translation table by changing the contents of KBVECT. (Except if your table is in capsule space.)

KECB (--- A) V2

An ECB used by the keyboard input routines.

KEY (--- C) NOTE

Read key. (Actually read from keyboard buffer.)

If the user has pressed LOCK and SHIFT, or if he has just pressed SHIFT, the keystroke returned is case-shifted. See 'KEY.

KQ (--- A) V8

The keyboard buffer. This is a ring buffer with KQI pointing to input (from keyboard) and KQO pointing to output (for KEY).

KQHI (--- N) =

Length of the keyboard buffer in characters. This is the maximum number of characters that the HHC can buffer between keyboard input and program processing. The current value is 8.

KQI (--- A) V1

The keyboard buffer input pointer. Contains the displacement of the byte in the keyboard buffer where the next character read from the keyboard will be stored. Note that keys are stored backwards, e.g. the value of KQI is initially 7, then becomes 6, 5, 4, ... 0, 7, ... See also KQ, KQO.

KQO (--- A) V1

The keyboard buffer output pointer. Contains the displacement of the byte in the keyboard buffer from which the next character passed to KEY will be fetched.

The buffer is empty when KQO and KQI (the input pointer) are equal.

See also KQ, KQI.

KS (N --- B)

Used to execute the Nth RUN SNAP PROGRAMS file.

L

L>U (C --- C')

Converts a lower case character to upper case. L>U has no effect on characters which are not lower case.

LASTFLAG (--- FL) =

A flag in HDT byte 0. 1 -> this is the last HDT in the HDT list.

LATCH (--- A) V1,NOTE

Stacks address of the LATCH, a special one-byte segment of the address space whose bit settings control various aspects of the HHC's hardware status. Symbolic constants defining the functions of the bits in LATCH are listed in the following table.

Note 1: Any program which needs to read the latch should do so by fetching LATCHS, a copy of the latch that is kept current by software. The latch is implemented with write-only hardware, and cannot be fetched directly.

Note 2: Application programs should not try to manipulate the latch. System programs that need to manipulate the latch should use the low level code routine DOLATCH to manipulate the latch. Update LATCHS when LATCH is changed.

Word	Value (hex)	Meaning
	1	Indicate which capsule
	2	Socket (if any) is currently bank- switched into the address space. Values 00B, 01B, and 10B represent sockets 0, 1, and 2. Value 11B indicates an extrinsic capsule.
CAPON	4	1-> a capsule is bank- switched into the address space. 0 -> The I/O page is bank-switched into the address space.
DISPON	8	1-> LCD is turned on, 0-> off.
	10	1-> addressing slow ROM, 0-> normal.
BEEPER	20	High/low DC to speaker. SQUEAK toggles this bit to generate audio.
CPUON	40	1-> CPU power on, 0-> off.
	80	Selects function of memory mapped area for keyboard/LCD functions. 1-> LCD control, 0->keyboard control.

(Note that the latch is write-only. To interrogate the latch's status you must read LATCHS below.)

LATCHS (--- A) V1

The "latch shadow," a copy of the latch maintained by the HHC nucleus. A program which needs to interrogate the latch must do so by reading this byte.

LATSAV (--- A) V1

A byte in N used internally by I/O routines to save latch byte settings.

LBUF (--- N)

Gets the address stored in &BUF-ADR and places it on the stack.

LCD.CR (---) C

Clears the LCD at the dot-graphics level. Use LCD.CR to clear the LCD before beginning to do dot-graphic display with DMOVE.

LEAVE (---) C

Used to force early termination of iteration in a DO loop. LEAVE sets the loop limit equal to the current value of the index. This causes termination of the loop at the end of the current iteration. ***Note***: it does ***not*** cause immediate termination of the loop.

LETGO (---)

Releases the TSA most recently allocated and not yet released. This permits you to use variables defined in any previously-allocated remaining TSA. You may also re-use the space in the released TSA (by allocating a new TSA, for example).

LIT (--- N)

Compiled from a literal that cannot be compiled to a CLIT, CLIT2, or CLIT3. Followed by two bytes giving the value of the literal. At run time, LIT stacks the contents of the next two bytes.

LLEN (--- N)

Pushes the one-byte contents of &BUF-LEN onto the stack.

LOBAT (--- A) V1

A byte whose 80H bit is turned on to indicate the "battery low" condition.

LOC.NXT.CAP (--- B)

Searches through ROM ID's for the next available ROM capsule after the one now switched into the capsule address space.

To search all ROM ID's for ROM capsules, execute -1 SET.ROM. This sets the "current ROM ID" to -1 (a meaningless value) but does not affect the ROM actually switched into the address space.

Now execute 'LOC.NXT.CAP'. LOC.NXT.CAP returns TRUE, switches the first available capsule (in ROM ID order) into the capsule address space, and stores that capsule's ROM ID in CURROM.

Continue executing LOC.NXT.CAP until it returns FALSE. FALSE indicates that there are no more capsules available. The capsule address space is not switched and CURROM is not changed.

Note that you cannot execute LOC.NXT.CAP from a program in a ROM capsule, since LOC.NXT.CAP will switch the ROM containing the program out of the address space!

See also CURROM, LOC.PRG.

LOC.PRG (N --- [[N'] B'] B)

Searches through ROM ID's for a program in a ROM capsule or a peripheral control ROM.

N is a number identifying the program to be switched in. If 'P' is the number of programs available, N may have any value from 0 to P-1.

If program #N is available (i.e. if 0<=N<=P-1), B is TRUE; B' is present, and N' may be present. If not, B is FALSE; B' and N' are both omitted.

If the program is in a peripheral control ROM, B' is TRUE and N' is present. If not, B' is 0 and N' is absent.

N' is the software device code of the peripheral control ROM containing the program. You can FLEE to the program by placing this code on the stack and executing FLEE.CROM.

See LOC.NEXT.CAP, FLEE.CROM.

LOCATE (BIGADDR BIGLEN SMADR SMLEN --- [DISP] B)

Locates a substring in a string.

BIGADDR and BIGLEN are the address and length of the string to be searched. SMADR and SMLEN are the address and length of the string to be searched for.

If the string is found, DISP is the origin-zero displacement of the substring within the string being searched, and B is TRUE.

If the string is not found, DISP is absent and B is FALSE.

LOCK.IT (--- FL) =

A flag in FLAG1. 1 -> LOCK key is in effect, 0 -> not.

LOCKBLIP (--- MASK) =

A mask for the "lock" ("keyboard locked") blip. See ANDBLIP for details.

LOCKED? (--- FL) =

A flag in FLAG1. 1 -> locked in SHIFT or 2nd SFT mode, 0 -> not. Meaningful only when SHIFT? is true.

LOOKUP (N --- [A] B)

Returns the address of the N'th file in the current file space.

If file N exists, A is the address of the file (*i.e.*, of the "length of file" field) and B is TRUE. If file N does not exist, A is absent and B is FALSE.

LP (--- A) V1

Displacement of the first free byte on the loop stack from the beginning of page 1. Add 100H to this value to get the address of the first free byte on the loop stack.

The address of the base of the loop stack is given by LSTK; thus, if the value of LP is equal to LSTK, the loop stack is empty.

LSTK (--- A) =

The address of the base of the loop stack.

M

M* (N1 N2 --- ND)

ND = N1 * N2.

M/ (ND N1 --- N2 N3)

N2 = signed remainder of ND/N1. (The remainder takes its sign from the dividend.)

N3 = signed quotient of ND/N1.

M/MOD (UD1 U2 --- U3 UD4)

Unsigned equivalent of M/.

U3 = remainder of UD1/U2.

UD4 = quotient of UD1/U2.

MAKE (A L --- B)

Creates a new virtual file.

A and L are the address and length of the file name. B is TRUE if the file can be created; FALSE if there is no space for it. (Note that a duplicate file name will not prevent MAKE from creating a new file and returning TRUE.)

MAKE-ROOM (L --- A)

Used to create space for copying a file. L is the length of the file to be copied. A is the address to which it can be written. Note: use ?ROOM or AVAIL to determine whether space is available before executing MAKE-ROOM.

MAX (N1 N2 --- N3)

N3 = the greater of N1,N2.

MAXBANK (--- N) =

The maximum number of RAM and ROM banks that the HHC may support at one time. The current value is 64.

MAXDEV (--- N) =

The maximum logical unit number that the HHC can support; one greater than the highest valid logical unit number. The current value is 16.

MEMBLIP (--- MASK) =

A mask for the "insert" blip. See ANDBLIP for details.

MENU-DRIVER (OFFSET TAG1 TAG2 ---)

Main entrypoint to the HHC's standard menu driver word. After setting up appropriate tables and lower-level words, you can call MENU-DRIVER to display a standard HHC menu that presents (and executes) the functions your application program is to perform.

OFFSET is the number of the menu item MENU-DRIVER is to display first (the usual value is one). OFFSET is added to an item number before printing it on the display, and subtracted from it after MENU-DRIVER has read it from the keyboard. So as far as your application is concerned, item numbers start from zero.

The HHC's file system uses this feature; when you enter the file system, the menu reads something like:

```
1=NEW FILE
2=COPY FILE
3=FILE1
4=FILE2
```

When you select COPY FILE, the file system once again calls MENU-DRIVER, but this time OFFSET is three, and the menu reads:


```
3=FILE1
4=FILE2
```

If it had not done so, the file numbers would suddenly start from one instead of three!

TAG1 is the tag of a routine with the following linkage:

(N --- B)

TAG1 is the "executer". When the user has selected a function, MENU-DRIVER calls TAG1 to perform the function. N is the number of the function (remember item/function numbers start from zero).

After the function is performed, TAG1 leaves a flag on the stack, telling MENU-DRIVER what to do next. If it is FALSE, MENU-DRIVER continues the menu, if it is TRUE MENU-DRIVER returns to the routine that called it.

Thus, if you want MENU-DRIVER to perform an endless loop, always leave FALSE on the stack.

TAG2 IS is the "selecter". It is called when the user has entered an item number, or when MENU-DRIVER wants to print a menu item. Its linkage is:

(N --- [[Parameters] TAG3] b)

B is true when N is an item that appears in the menu, it is false when N is illegal, i.e. out of range. If B is true, TAG3 is the routine that will print the name of the Nth item. (Only the name, MENU-DRIVER itself will print "M=", where M is N+OFFSET

The linkage of TAG3 is

(N [Parameters] ---)

TAG3 is executed by MENU-DRIVER in order to print the N+1th item from the menu.

When all menu items are file names, you can use ".FILE" for TAG2. .FILE will use TYPEDROP for TAG3. The linkage of .FILE is:

(N --- [Address Length 'X TYPEDROP] B)

When there are no valid files (see also "'FILE-COND"), .FILE prints "NO FILES", and B will be FALSE. Otherwise the address and the length of the Nth file are left on the stack for TYPEDROP. So the linkage to TYPEDROP is:

(Dummy Address Length ---)

Dummy is the N supplied by the menu driver.

Let's put everything together in a working example. Routine -REMOVE uses a menu from which the user can select which files to throw away. EXEC performs the actual removal, .FILE is used by the menu driver to select a file, REMOVE is the caller to the menu driver:

```
: EXEC ( Filenumber --- Boolean )
    \TAG1
     OPEN-FILE
          \Find file
     CFILE DELETE-FILE
     FALSE ;
          \Continue menu

: SELECT ( Address --- Boolean )
    \ Accepts every file type
     DROP TRUE ;

: REMOVE
     'X SELECT 'FILE-COND !
     1
          \Offset
     'X EXEC
          \TAG1
     'X .FILE
          \TAG2
     MENU-DRIVER ;
```

Notice that in our example the menu can only be left by pressing CLEAR.

If you only want to only remove text files, you must define SELECT as

```
: SELECT 2+ C@ %TEXT AND ;
```

MIN (N1 N2 --- N3)

N3 = the lesser of N1,N2.

MOD (N1 N2 --- N3)

N3 = N1 modulo N2 (remainder of N1/N2). Sign of N3 is same as N1.

MOVE (A1 A2 L ---)

Move data from A1 to A2, length L. The move is managed so that the data cannot overlay itself, no matter what values of A1, A2, and L are used. This allows overlapping source and destination areas to be MOVEd successfully.

L is an unsigned integer; thus a "negative" length will be interpreted as a length > 32565. This is a sure way to wipe out a program.

CMOVE is a similar instruction that will overlay data if $A1 < A2 < A1+N$.

MOVE-IT (A1 A2 L ---)

Used by the COPY FILES subfunction to copy a file at A1 of length L to the space at A2. The source's RAM bank should be stored in &EOL- TAG and the destination RAM bank in &EOF-TAG. &LBUF is used as an intermediate buffer for the move.

N

N (--- A) V32

A 32 byte RAM area in page zero, available for scratch space by assembler routines. Contents do not necessarily survive passage through NEXT.

NAP (TICKS SEC --- [C] B)

Puts the HHC to sleep for a specified time or until a keystroke is entered.

SEC is seconds to sleep. TICKS is the number of clock ticks to sleep; it must be in the range 0-255.

If the nap is ended by time expiration, C is absent and B is FALSE.

If the nap is ended by a keystroke, C is the character entered and B is TRUE. A subsequent KEY or equivalent word will ***not*** get the same character.

NATTACH (--- A) V1

A counter used internally by FLAME.ON

NEGATE (N --- N') C

N' = -1 * N.

NEVER (X --- FALSE)

Drops one word from the stack and returns FALSE. Same as FALSIFY, but NEVER is a long tag and FALSIFY a short one.

NEW-FILE (---)

Routine used by the FILE SYSTEM to allow entry of a new text file. Only CLEAR can terminate it.

NEXT (--- A) V3

Address of a JMP instruction that points to the SNAP inner loop. Changed by the HHC's nucleus to handle certain types of interrupt conditions.

NEXTV (--- A) V1

Address of a variable used to repoint NEXT with a bank switching code when SNAP must execute a tag defined in another memory bank.

NFILES (--- A)

Stacks the address of a word containing the number of files existing in the current file space.

NMIVECT (--- A) V3

Non-maskable interrupt vector. May be changed to point to a low level NMI processing routine. The default NMI processing routine is a null routine; is returns control immediately to the caller.

NOP (---) C

"No operation." Generates an HHC tag which has no effect when executed.

If a program is being executed from RAM , such a tag is useful as a placeholder for space where another tag can be patched in at run time. Patching code in this way is not a legitimate coding technique, but is occasionally useful during debugging.

NOT (X --- X')

Reverses results of a previous logical test. Functionally identical to 0= . Changes 0 to 1, non-zero to 0.

NREC (N --- [A] B)

A very useful file system word. Given N, the record number, NREC returns the address of the first byte of the record and True, if the record exists. Otherwise it returns False. In non-text files, you can use 0 NREC to return the address of the first byte of data. -1 NREC will return the address of a count prefaced string which is the file name.

O

ONFLAG (--- A) V1

A byte used to turn the CPU on and off. The 80's bit controls the on/off status.

OPEN (A L --- B)

Opens an existing virtual file.

A and L are the address and length of the file name. B is TRUE if the file is found; FALSE if not.

OPEN-FILE (N ---)

Opens the N'th file in the current file space, relative by the routine in 'FILE-COND.

OPFLAG (--- FL) =

A flag in HDT byte 1. 1 -> this HDT represents a device capable of output.

OR (X Y --- Z) C

Performs a bitwise logical OR.

ORBLIP (N ---) C

ORs the low-order byte of N into the LCD blip mask. If a bit is 1 in N, the corresponding blip is turned on; if a bit is 0 in N, the corresponding blip is unchanged.

For a list of symbolic constants that may be used to designate the bits, see ANDBLIP.

OVER (X Y --- X Y X) C

Copies second entry on stack.

P

PASSRAW (--- FL) =

A bit in FLAG2 which controls the operation of the SHIFT, 2nd SFT, and LOCK keys. 0 -> the keys have their normal functions. 1 -> the keys place binary codes in the keyboard buffer, like ordinary character keys.

PATLEN (--- L) =

Length of the "pattern" used to re-initialize several I/O-related vectors, such as the address of the keyboard translation table, during a clear.

PCASAV (--- A) V1

A byte in N used internally by I/O routines to save the displacement of the current PCA.

PCH (--- A) V1

Value of the previous raw keyboard character code. Used internally by keyboard debouncing routines.

PENDFLAG (--- FL) =

A flag in HDT byte 1. 1 -> I/O operation is pending on this device.

PICK (Xy ... X3 X2 X1 y --- Xy ... X3 X2 X1 Xy) C

Copies y'th entry on stack to the top.
1 PICK = DUP
2 PICK = OVER

PIPFLAG (--- FL) =

A flag in PCA byte 0. 0 -> this PCA represents a device capable of doing input.

PKCT (--- A) V1

The pushkey buffer pointer. Contains the number of characters currently in the pushkey buffer, PKQ. If the value is 0, the buffer is empty, and the next character in the input stream will come from the keyboard buffer. If the value is 4, the pushkey buffer is full. See also PKQ.

PKEY (--- A) V1

A location in which input characters are deposited when a RIP is done on the keyboard.

PKQ (--- A) V4

The PUSHKEY buffer. This is a 4-byte stack containing characters "pushed" onto the keyboard input stream by the program.

The stack grows upward; thus the first character pushed into the buffer goes at PKQ, and the 4th (and last) character pushed into the buffer goes at PKQ+3.

See also PKCT.

POPFLAG (--- FL) =

A flag in PCA byte 0. 0 -> this PCA represents a device capable of doing output.

POSN (N ---) C

Positions the cursor in LCD character position N. The next EMIT will put a character at the new cursor position.

POSTECB (--- A)

Vectored subroutine used by I/O routines to post an ECB and reset PENDFLAG. Upon entry, the Y-register should hold an offset pointing to byte 1 of the HDT.

POZECB (--- A) V6

The timer ECB used by NAP. In application routines it is safe to use this ECB for any purpose if NAP is not used.

PROGBIT (--- F) =

A bit mask for CSPEED. See CSPEED for details.

PSECB (--- A) V6

Timer ECB used by the CLOCK/CONTROLLER. If any CLOCK/ CONTROLLER alarms are defined, this ECB is in the timer queue, and contains the time of the next alarm to come due. If no alarms are defined, this ECB is not in the timer queue.

When the timer represented by this ECB is exhausted, the HHC issues its trumpet fanfare and resets the timer for now plus 10 minutes. When the CLOCK/CONTROLLER cancels the alarm that has come due, it also adjusts the ECB to reflect the next remaining alarm in the queue (if any).

PUSHKEY (C ---) C

Pushes a character into the pushkey buffer. In this way, software can induce the effect of a key having been pressed. See PKQ, PKCT.

R

R (--- X) C

Copies X from top of return stack.

R> (--- X) C

Pops X from top of return stack.

RAM.N (N --- B)

Given the relative bank ID of a RAM bank, N, tries to switch the RAM bank into the extrinsic RAM address space. Available RAM banks are numbered consecutively from 1.

If the RAM bank exists, RAM.N switches it in and returns B=TRUE. If not, RAM.N returns B=FALSE.

RAMADDR (--- A) =

Address of the beginning of the extrinsic RAM address space (8000H).

RAMBIT (--- FL) =

A bit in FLAG3 indicating whether to do a hard clear when exiting from the I/O key processor. 1 -> do a hard clear; 0 -> do no clear.

(A hard clear is necessary if a virtual file is open and the user has used the I/O key to change the current file space.)

RAMEXFLAG (--- FL) =

A flag in PCA byte 0. 0 -> the PCA represents a programmable memory peripheral.

RCLOSE (ECBADR LUN --- B)

Initiates a "close" operation on the device identified by logical unit number LUN, using an ECB at ECBADR. B indicates success or failure of the operation.

A successful RCLOSE must be followed (eventually) by a WAIT and a test for successful conclusion of the closing operation.

RCLOSE is not normally needed by applications, since the CLEAR key forces an RCLOSE on all devices.

RCSAV (--- A) V1

A byte in N used internally by I/O routines to save an ECB return code.

RCTL (ECBADR LUN --- B) C

Initiates a "read status" or "write control" operation on the device identified by logical unit number LUN, using an ECB at ECBADR. B indicates success or failure of the operation.

The length of the ECB depends on the operation being performed. Its minimum required length is 2 bytes; the 2nd byte contains a command code indicating the type of control operation to be performed.

A successful RCTL must be followed (eventually) by a WAIT and a test for successful conclusion of the operation.

If the operation is a "read status" type, byte 1 of the ECB contains the status byte on return from a successful WAIT.

READ (A L N --- [L'] B)

Read a record from the current virtual file.

A L are the address and length of a buffer area in RAM. N is a record number.

If the record exists, READ moves the record to A, truncating it to length L if necessary. L' is the length of the record (after truncation if any); B is TRUE.

If the record does not exist, the data at A is not changed. L' is absent and B is false.

REC-CNT (--- N)

Returns number of records in the current virtual file. This may be used to add a record to the end of the file, like this:

```
adr len REC-CNT INSERT
```


REC-MOVE (A1 A2 L ---)

Moves an ASCII string of length L, at A1, to A2 + 1. The value L is stored at A2 in order to produce an S" - type string.

RECBLIP (--- MASK) =

A mask for the "delete" blip. See ANDBLIP for details.

REFRESH (---) C

Used internally to manage the LCD display. Not normally used by applications.

Checks contents of LCD display buffer; ensures that all characters that are marked "flash on" in the control buffer are, in fact, flashing.

Use this word if you have updated CHARBUF directly, and want the LCD to show its current contents.

RELOC (A --- TAG)

Relocates a RUN SNAP PROGRAM stored at A and initializes TVECT0.

```
CFILE RELOC EXECUTE
```

will RUN the program.

REST.KEY (---)

Restores variables indicating status of keyboard from values saved on the return stack. See SAVE.KEY.

RESTORE (---) NOTE

Restores a word from the return stack to memory. Note: does not leave the address on the stack. See SAVE.

RESTORE.RAM (---)

Switches the current RAM bank into the address space. Used by the I/O key to restore the current RAM bank after switching in each available bank while displaying the I/O menu.

RESTORE.TASK (---)

Restores variables indicating HHC status that are saved by SAVE.TASK.

REVISE (A N --- B)

Adds or deletes space within virtual files. (Also used internally by the file system to insert and delete records in text files.)

If N>0, creates an open space N bytes long at A by taking all the data between A and the end of the last virtual file, and moving it N bytes up. B is FALSE if this cannot be accomplished.

If N<0, "deletes" a space ABS(N) bytes long at A by taking all the data between A+ABS(N) and the end of the last virtual file, and moving it ABS(N) bytes down. Also updates the length of the current file and filespace.

RIP (ECBADR LUN --- B) C

Initiate a read operation on the device identified by logical unit number LUN, using an ECB at ECBADR. B indicates success or failure of the operation.

A successful RIP must be followed (eventually) by a WAIT and a test for successful conclusion of the operation.

The ECB must be 2 bytes long. On return from a successful WAIT, the second byte contains the character that has been read.

ROLL (Xy ... X3 X2 X1 y --- ... X3 X2 X1 Xy) C

Moves y'th entry on stack to the top.
'3 ROLL' is equivalent to 'ROT'.

ROLLBIT (--- FL) =

A flag in FLAG1 used by the keyboard decoding routine to support keyboard rollover.

ROMADDR (--- A) =

Stacks the location at which ROM capsules begin in the HHC address space.

ROMEXFLAG (--- FL) =

A flag in the PCA byte 0. 0 -> this PCA represents a ROM extender.

ROMEXT (--- A) K1

The location in a ROM capsule header which contains the number (2 through 7) of the capsule's primary long tag table. See ROMVECT for more information.

ROMID (--- A) K1

The location in a ROM capsule header where a string containing the ROM capsule's name should begin.

ROMVECT (--- A) K2

The location in the header of a ROM capsule which contains the address of the capsule's primary long tag table. When

the HHC executes the program in the capsule, it stores this address in the proper entry of the tag table vector. The proper entry is the one indicated by ROMEXT.

ROP (ECBADR LUN --- B) C

Initiate a write operation on the device identified by logical unit number LUN, using an ECB at ECBADR. B indicates success or failure of the operation.

The ECB must be 2 bytes long. The second byte holds the character to be written.

A successful ROP must be followed by a WAIT and a test for successful completion of the write operation.

ROPEN (ECBADR LUN --- B) C

Initiate an "open" operation on the device identified by logical unit number LUN, using an ECB at ECBADR. B indicates success or failure of the operation.

A successful ROPEN must be followed by a WAIT and a test for successful completion of the operation.

Not normally needed by applications, since the first RIP, ROP, or RCTL forces an ROPEN on the device.

ROT (X Y Z --- Y Z X) C

Moves third entry on stack to top, the first and second are moved down.

RP! (---) C

Empties the return stack. Dangerous to use.

RP@ (--- A) C

Stacks the current value of the return stack pointer, i.e. the address of the current top entry on the return stack. May be used with caution to fetch data from the return stack.

RSTK (--- A) =

Address of the base of the return stack.

RUNBIT (--- FL) =

A bit in FLAG2 which controls the STP/SPD delay for rotation and carriage returns. 0 -> the delay is performed normally. 1 -> the delay is disabled; CR's and rotation occur immediately.

RUNBLIP (--- MASK) =

A mask for the "on-line" blip. See ANDBLIP for details.

S

S+ (A L N --- A' L')

Returns the address and length of a substring extending from a specified character in a string to the end of the string.

A and L are the address and length of the string. N is the origin-0 number of the character where the substring starts. It can be negative.

A' and L' are the address and length of the substring extending from character #N of the string to the end of the string.

A' = A+N
L' = L-N

S.T (---)

Shows a date and time in the format used by the CLOCK/CONTROLLER's "time" function. Does not update the time.

Before executing S.T, you must store the time you wish to display in the 39th and 40th words of &LBUF. The time should be time of century in seconds. For example, to display the current time:

```
GET.LOCAL  &LBUF 39 + 2!    S.T
```

S= (A1 L1 A2 L2 --- B)

Compares the string at address A1 with length L1 to the string delimitted by A2/L2. B is TRUE iff the two strings are equal.

S>D (N --- ND)

N is converted to a double integer.

SAVE (A ---)

Saves address A and the contents of the word at address A on the return stack. The contents of the word (byte) may be restored to memory by RESTORE (CRESTORE).

SAVE.KEY (---)

Saves variables indicating status of keyboard on the return stack. See REST.KEY.

SAVE.TASK (---)

Saves the a number of variables that determine attributes of the HHC's status.

Saved variables include (ROTMODE) and (FREEZE); 'KEY and 'EMIT; characters displayed on the LCD (but not their display attributes, e.g. reverse image); cursor position; and the capsule ROM's bank ID.

(Used the preserve the HHC's status by the I/O key routine. Variables are restored by RESTORE.KEY.)

SCOMP (--- A)

Signed comparison - a vectored assembler subroutine. See UCOMP.

SCROLL (---)

Used internally to manage LCD display. Not normally used by applications.

Rotates the LCD display left by 1 character. Matching dot columns at the right end of the display are cleared. The display output pointer, BUFPOSN, is shifted left with the display.

SCROLR (---)

Used internally to manage LCD display. Not normally used by applications.

(Same as SCROLL, but scrolls right.)

SD* (ND1 N --- ND2)

ND2 = ND1 * N.

SDT (--- A) V16

Stacks the address of the software device table (SDT). The SDT contains one one-byte entry per logical unit number (LUN), stored in LUN order.

Each entry is the displacement from the beginning of the hardware device table (HDT) area to the HDT for the device connected to this LUN. A value of 0 indicates the keyboard; 6 represents the LCD; 0CH, 12H, 18H, ... represent peripherals; 0FFH represents an unattached LUN.

SECS (N ---)

Puts the HHC to sleep for N seconds.

SET-BLIPS (---)

An internal word used by EXPECT to refresh the INSERT and DELETE blips.

SET.BITS (A X ---)

Changes the value of the word at A by turning on each bit that corresponds to a bit that is on in X. See also TOGGLE and CLR.BITS.

SET.CROM (DEVCODE --- B) C

Switches the control ROM for the peripheral with hardware device code DEVCODE into the peripheral control ROM address space. B indicates whether such a control ROM was available or not. DEVCODE may be either a hardware or software device code. If two or more such peripherals are attached, the one in the lowest numbered slot is chosen.

SET.CTRL (DISP --- B)

Switches the control ROM for the peripheral in a slot related to DISP into the peripheral control ROM address space. B indicates whether or not the 'C' of the copyright message was found. DISP is the displacement from the origin of the I/O page, and corresponds to the slot number. If the peripheral is plugged in directly, DISP = 0. Otherwise it is a 4x(slot number).

SET.DELAY (TICKS ECBADR ---)

Set expiration time in a timer ECB, and queue the ECB. TICKS is an unsigned integer giving delay in clock ticks (1/256 sec).The maximum delay allowed is 65,535 ticks, or about 4 minutes, 16 seconds.

SET.DELAY.LONG (TICKS DSEC ECBADR ---)

Set expiration time in a timer ECB, and queue the ECB. DSEC is an unsigned DOUBLE integer giving delay in seconds. TICKS is an unsigned integer giving additional delay in clock ticks; it must be in the range 0-255.

SET.FLSH (---)

Causes any following EMIT with zero in the character parameter's high byte to display a flashing character. SEE ALSO UNSET.INV and SET.INV.

SET.INV

Causes any following EMIT with zero in the character parameter's high byte to display a character in inverse image form. SEE ALSO UNSET.INV and SET.FLSH.

SET.RAM (N --- B) C

Used by the I/O key's support routines. Not ordinarily used by application programs.

Bank-switches memory bank N into the extrinsic RAM space. TYPE is an integer indicating what type of memory bank N was found to contain. See CURRAM for N's format.

If B is FALSE, bank N was empty, nonfunctional, or contained ROM. The bank-switching did not take place.

If B is TRUE, bank N contains RAM, and was bank-switched into the RAM capsule space (beginning at 8000H).

The usual method of using SET.RAM is to test every memory bank in turn for the desired type of memory. To ensure forward compatibility with any hardware enhancements which may be made to the HHC in the future, you should test every bank (01H to 0FFH) until you find the capsule you want. At present only the first few bank numbers are used, but others may be used in the future.

SET.ROM (N --- B) C,NOTE

Used by capsule controller in nucleus. Not ordinarily used by application programs (BIGMOVE is a safer alternative.)

Similar in function to SET.RAM, but bank-switches memory bank N into the ROM capsule space. Caution: you will crash if you deselect your own capsule with this word. See CUR-ROM for format of N.

If B is FALSE bank N was empty, nonfunctional, or contained RAM. The bank-switching did not take place.

If B is TRUE bank N contains ROM, and was bank-switched into the ROM capsule space (beginning at 4000H).

Note that when you want to bank switch in a ROM capsule, you generally are looking for a particular capsule. Thus, you must test the contents of the ROM after switching it in, to see if it is the right one. If not, keep searching!

SET.RP (X ---) C

Word used by the system to set the Return Stack pointer to 100H + X (low order byte of X only). Dangerous to use.

SET.SP (X ---) C,NOTE

Sets the parameter stack pointer to the address indicated by the low-order byte of X (relative to the start of page zero). This may be used ***with caution*** to empty the stack to a predetermined level, regardless of how many entries have since been pushed onto it.

To get a value of X, use SP@. For example, in the following piece of code:

```
SP@ SDEPTH !
  .
  .
  .
SDEPTH @ SET.SP
```

SET.SP@ sets the stack to the depth that it had immediately before SP@ was executed.

```
SPINIT SET.SP
```

empties the stack.

Caution: if SET.SP moves the stack pointer ***away*** from the root of the stack, effectively adding entries to the stack, the added entries will be meaningless and might cause problems by trashing system variables.

SET.TIMER (ECBADR ---)

Queue a timer ECB. The ECB contains the time at which the ECB should be posted, measured in clock ticks from the beginning of 1 January 1980.

SETBANKS (X ---)

The two bytes of X are interpreted as bank ID's. The high-order byte is used to switch a ROM bank into the ROM capsule address space. The low-order byte is used to switch a RAM bank into the extrinsic RAM address space.

SETUP (--- A)

Vectored assembler subroutine used to move parameters from the stack into N. Enter SETUP with the number of ***words*** to move in the Accumulator.

SHFTBLIP (--- MASK) =

A mask for the "shift" blip. See ANDBLIP for details.

SHIFT (X N --- X') C

Shifts X left by N positions. A negative N causes a shift right.

SHIFT? (--- FL) =

A bit in FLAG1. 1 -> we're in SHIFT or 2nd SFT mode; 0 -> we're not. If SHIFT? is TRUE, WHICH? tells which mode we're in.

See also WHICH? and LOCKED?

SHINT (--- N) =

A symbol giving the number of the first short tag that is free for use in the short extrinsic tag table.

SHOW.TIME (---) NOTE

Shows the current time in the format used by the CLOCK/CONTROLLER's "time" function; updates the time each second.

Uses &LBUF, the buffer provided for EXPECT, as a work area.

Note: to leave this word, you must do a soft clear.

SHRINK (A L --- B)

Deletes the segment of File space beginning at A, of length L. B is TRUE if the deletion was successful. B is FALSE when the segment does not lie within the file space.

SIGN (N D --- D)

Used in conjunction with <# # #> to convert a number's sign to ASCII form.

N is a sign indicator. If N is negative, SIGN puts a '-' before the ASCII number in the text output buffer. If N is non-negative, SIGN puts nothing in the buffer.

See the description of '<#' for examples of SIGN in use.

SIZERAM (---) C

Used by the CLEAR key to determine the size of a RAM. Resets TP to point to the end of RAM +1. This routine should not be used in general.

SL (--- A) V1

"Stack limit." The displacement from the beginning of page 0 to the first word below the limit of the parameter stack. (Recall that the parameter stack grows down from the end of page 0.)

In hex;

```
SL C@ 2+
```

gives the address of the lowest word where a parameter stack entry may be stored.

Note that the value of SL may change if a routine allocates work space from the bottom of the parameter stack area with GETMEM.

SLEEP (--- A)

Vectored assembler subroutine used by the nucleus to turn off the CPU.

SLPFLG (--- A) V1

A byte used to control "sleep," i.e. the turning off of the CPU. If a non-zero value is put in SLPFLG, only the OFF-key will turn the CPU off. This increases power consumption and should be avoided.

SMART-POSN (N ---)

Moves the LCD cursor to character position N. SMART-POSN functions by EMITing a string of <-'s or ->'s. It affects bound devices as well as the LCD. N should be in the range 0 to 26 (see POSN).

SNAP-FILE (---)

Implements the "RUN SNAP PROGRAMS" option from the primary menu.

SOFT.CLR (---) C

Performs a soft clear, as though the user had pressed the CLEAR key once.

SOFT.EMIT (C ---)

Same function as EMIT, but treats all control characters (including the cursor control codes beginning at 80H, and escape control sequences beginning with 1BH) as displayable characters. For example, 81H (normally a non-destructive backspace) is displayed as '<-'; 1BH (normally "escape") as an inverse image '['; 0DH (normally "carriage return") as an inverse image 'M'.

SOFTAG (--- A) V2

Holds the tag that was executed when the current application was entered. Set by FLEE.

This tag will be re-executed if the application is restarted by a soft clear.

SOFTAG may be changed directly, or by another FLEE.

See also SOFTROM, SOFTCTL, and SOFTVECT.

SOFTCTL (--- A) V1

Bank ID of the peripheral control ROM that was selected when the current application was entered. Set by FLEE.

This control ROM will be re-selected if the application is restarted by a soft clear.

SOFTCTL may be changed directly by the application program.

See also SOFTROM, SOFTVECT, and SOFTAG.

SOFTFLAG (--- A) V1

A location whose contents determine whether the next clear will be a soft clear or a hard clear.

If SOFTFLAG contains 5AH, the next clear will be soft. If the contents of SOFTFLAG is anything other than 5AH, the next clear will be hard.

A soft clear sets SOFTFLAG to 0; any other keystroke sets it to 5AH.

An application can prevent hard clears from taking place by setting SOFTFLAG to 5AH as soon as the restart word indicated by SOFTAG is executed. (***Caution***: this violates the HHC's customary user interface design, and forces the

user to take drastic steps, such as unplugging a program capsule, if the program malfunctions or he forgets how to use it.)

SOFTROM (--- A) V1

Holds the bank ID of the capsule ROM that was selected when the current application was entered. Set by FLEE.

This capsule ROM will be re-selected if the application is restarted by a soft clear.

SOFTROM may be changed directly by the application program.

See also SOFTCTL, SOFTVECT, and SOFTAG.

SP@ (--- A) C

Returns the address of the top of the parameter stack, i.e. of the lowest entry on the stack.

SPACE (---)

Displays a space on the current output device (normally the LCD).

SPACES (N ---)

Displays N spaces on the current output device (normally the LCD).

SPEED (--- A) V1

A variable containing a number that controls speed of EMIT, CR, LCD scrolling, etc.

The value of SPEED is set by the STOP/SPEED key like this:

value	**STOP/SPEED setting**
10	1 (slowest setting)
9	2
:	:
:	:
2	9 (second fastest setting)
1	0 (fastest setting)
0	faster than fastest STOP/SPEED setting

SPINIT (--- A) =

Stacks the address of the root of the parameter stack. To empty the stack, execute the following code:

```
SPINIT SET.SP
```

SQUEAK (PITCH TIME ---)

Squeaks at pitch PITCH for time TIME through the speaker.

PITCH is an integer from 0 to 36. 1 gives the lowest note and 36 gives the highest note; 0 gives the sound of silence. 32 is approximately middle C; a difference of 1 is approximately one semitone.

TIME is an integer; the time unit is about 5.88 msec. There are 10,200 counts/minute, of 170 counts/second.

START.CURSOR (---)

Enables flashing cursor (makes it visible).

STKBIT (--- FL) =

A bit in FLAG1 indicating that the HHC status is on the return stack.

STM (---)

Performs the entire "set time" function of the clock controller. It returns control to its caller when the user presses ENTER (if he wants to set the time) or gives up control to a soft clear (if he wants to escape from the word without setting the time).

Application programs may call this word to let the user set the HHC's clock without going to the CLOCK/CONTROLLER.

STOP.CURSOR (---)

Disables flashing cursor (makes it invisible). See DMOVE if the cursor is being turned off for LCD graphics.

SWAP (X Y --- Y X) C

Exchanges top 2 entries on the stack.

SWAPDROP (X Y --- Y)

Drops the second entry on the stack.

T

T! (---)

Initializes (empties) the temporary stack. ***Note***: application programs should ***not*** use this word.

TDKFLAG (--- FL) =

A flag in HDT byte 0: 1 -> this HDT belongs to the LCD or keyboard.

THI (--- A) =

High order byte of the hardware timer. See TLO, TMID.

TIMEFLAG (--- FL) =

A bit in FLAG2 used by the HHC nucleus to help support timer updating.

TLO (--- A) =

Low order byte of the hardware timer. See THI, TMID.

TMID (--- A) =

Middle order byte of the hardware timer. See THI, TLO.

TOGGLE (A X ---)

Changes the value of the word at A by reversing each bit that corresponds to a bit that is on in X. See also SET.BITS and CLR.BITS.

TP (--- A) V2

Returns the address of a variable that holds the address of the top of the temporary stack. Bytes higher than this are used for AREAS and GETMEMT memory. Filespace should ***never*** cross into this region.

TPSAFE? (N --- A | 0) C

Determines if there is enough space in intrinsic RAM to allocate N bytes of storage space. If there is, A is the address of the space; if there is not, 0 is returned.

Note that TPSAFE? does not allocate the space; it merely returns the address at which space WILL be allocated, if you allocate it. To allocate space, see GRAB.

TQUEUE (--- A) V12

Timer queue. Consists of 6 2-byte entries, each containing the address of a timer ECB or 0.

Each unexpired timer ECB in the system is represented in this queue. Whenever a timer interrupt occurs, the queue is searched for expired ECB's; if any are found, they are posted and removed from the queue. Each unexpired ECB is updated to reflect the amount of time remaining until it expires.

TRUE (--- TRUE) =

Stacks a boolean TRUE. The numeric value of TRUE is 1.

TURN.OFF (---) C

Turns the HHC off as if the OFF key had been pressed. Applications should not use this word; it will cause the HHC to execute the word pointed to by 'ABORT. See 'ABORT, ABORT.

TVECT0 (--- A) V24

Stacks the address of the first entry in the tag table vector (the entry representing the short extrinsic tag table).

The table consists of 9 three-byte entries: #0 for the short extrinsic tag table, #1 for the long intrinsic tag table, and #2 through #8 for long extrinsic tag tables. For further details, see the discussion of tag tables in the chapter on "General Technical Information."

Each entry in the tag table is 3 bytes long:

byte	bit	contents
0-1		Address of the tag table.
2	80	1->tag table is in external ROM, 0-> nucleus, RAM or control ROM
	40-01	Bank ID of memory bank containing the tag table.

TYPE (A L ---)

Displays characters from memory, starting at address A, for length L.

Control characters may be included in the character string to perform control functions on the TV adaptor if the adaptor is bound to the display. These control codes have no effect on the LCD.

See the section on control characters for details.

TYPEDROP (X A L ---)

Types the string at address A, length L, and drops X.

U

U* (U1 U2 --- UD) C

UD = U1 * U2.

U< (U1 U2 --- B) C

Tests if U1 < U2.

UCOMP (--- A)

Vectored assembler subroutine used to perform a unsigned comparison on two stack entries (N1 on top, N2 second).

The negative flag indicates that N1>N2.

UD< (UD1 UD2 --- B)

Tests if UD1 < UD2.

UDMIN (UD1 UD2 --- UD3)

UD3 = the smaller of UD1 and UD2.

UM/ (UD U1 --- U2 U3)

U2 = unsigned remainder of UD/U1.
U3 = unsigned quotient of UD/U1.
No check for U1=O.

UNDER (X Y --- Y)

Synonym for SWAPDROP.

UNCOMMITTED (--- A) V16

A data area in low RAM reserved for future expansion of the HHC nucleus. It currently is not used by any part of the HHC system. Application programs should not use it.

UNSET.FLSH (---)

Reverses the effect of a preceding SET.FLSH, if any.

UNSET.INV (---)

Reverses the effect of a preceding SET.INV, if any.

UPDISP (N ---) C

Updates the LCD after the contents of the LCD character buffer or the LCD control buffer has been modified. N is the position of the first character to be updated.

USED (--- A)

Stacks the address of a 2-byte area containing the number of bytes used in the current file space.

For the number of bytes free in the current file space, see AVAIL.

W

WAIT (ECBADR ---) C

Wait for ECB to be posted.

WAITM (ECBADR[N-1] ... ECBADR1 ECBADR0 N --- N')

Multiple wait. Waits for any of the "N" ECBs to be posted. N'

is the number of the ECB whose posting ended the wait. For example, if the value of N' is 0, that means that ECB #0 (the first one in the WAITM request) was posted.

WHICH? (--- FL) =

A flag in FLAG1. If SHIFT? is TRUE, indicating that the HHC is in SHIFT or 2nd SFT mode, WHICH? indicates which mode it is in: 1 -> SHIFT mode, 0 -> 2nd SFT mode.

WRITE (A L N --- B)

Replaces record #N of the current virtual file with the data at address A, length L.

If record #N already exists, B is true. If record #N does not already exist, the new record is ***not written*** and B is false.

To write a record where none exists, use INSERT.

X

X1SAV (--- A) V1

Used by peripheral I/O routines to save the X register.

XOR (X Y --- Z) C

Bitwise logical exclusive OR.

XSAVE (--- A) V1

A one-byte area set aside for storing the X register (parameter stack pointer). A CODE word or low level procedure may use it for any purpose. But note that any lower level routine may destroy its contents; if your code calls a lower level routine that is not part of your own program, assume that XSAVE will be destroyed by the call, even if it currently appears to be preserved.

Y

YOFF (--- A) =

Address of a location in the I/O page. A store into this address causes the Y-driver to stop asserting IRQ.

Z

Z (--- A) V9

Work space in low RAM, used by IRQ processing routines.

INDEX

2nd SFT key, 11-2, 11-3
6502
  Addressing Modes, 16-10
  Instructions, 16-10, 16-13, 16-15

A

Additional HHC characters, 11-12
Alternate character pattern for cursor, 12-11
Alternate character set for LCD, 12-7
AOENB, 14-31
AOTOKE, 14-31
AREA (TSA word), 7-3
ASCII, 11-4
Asynchronous I/O, 14-6
Asynchronous processing, 14-1
ATTACH, 14-7
  POKE and, 14-11
ATTACHX, 14-8
Auto-off facility, 14-30
AVAIL, 7-7

B

<BUILDS DOES>, 9-1
Bank switching
  Programmable Memory Peripheral, 13-9
BASIC file type, 13-7
Battery
  Conserving, 11-20
Beep, 12-15
Blips, 12-13
  Last 3 columns of LCD, 12-7
BUFPOSN, 12-6

C

C1/C2/C3/C4 keys, 11-3
Capinit, 5-4
Capsule
  Memory map, 2-2
Capsules
  headers, 5-5
  Libraries, 6-1
CASE statement, 8-3, 8-5
CFILE, 13-6
Character
  Numeric representation, 2-4
Character control buffer, 12-5

Character display buffer, 12-5
Character translation table, 11-4
  LCD, 12-7
CHVECT1, 12-10
CLEAR
  Cancels ECB's, 14-30
  Does not reset rotation mode, 12-13
  Hard, 7-8, 7-10
  Inhibiting, 7-10
  Soft, 7-8, 7-10
CLEAR key, 7-8, 7-10, 11-2, 13-8, 14-8
  Restarting an application, 7-8
CLEAR stack, 7-8
Clock, 14-29
Clock tick, 14-29
Compilation
  from a file, 10-3
Compile Time, 5-1
Compiler options, 3-1
Complex number, 15-2
Constant, 7-1
Control byte
  Peripheral, 14-10
Control characters, 11-7, 11-8, 11-9
  EMIT, 12-2
  Peripheral, 14-13
Control key, 11-1
Control structure
  Scope of, 8-1, 8-6
Current file, 13-2
Cursor
  Alternate character pattern, 12-11
  LCD, 12-1
Cursor control keys, 11-3

D

DBUF, 12-5
DBUF1, 12-5
Definitions
  Types, 4-11
DELETE, 13-5
Delete cursor, 11-16
DELETE key, 11-3, 11-17
DELETE-FILE, 13-6
Dictionary
  headers, 4-13
  structure, 4-1
  variables, 3-3
Displayable characters, 11-10

DMOVE, 12-6, 12-7
DOCASE ENDCASE statement, 8-4
Dots in LCD, 12-1
Double word
  Representation of, 2-3
DSPLY, 12-6
Dynamic allocation of RAM, 7-6

E

ECB, 14-1, 14-2, 14-13
  Allocating, 14-4
  CLEAR cancels, 14-30
  Format of, 14-4
  Peripheral I/O, 14-6
  Posting, 14-6
  Return code, 14-12, ***see*** Return code
  Timer, 14-30
  WAITM, 14-11
EDIT-FILE, 13-8
EMIT, 12-2
  Display mode set by, 12-3
  ROP and, 14-9
(EMIT), 12-13
ENDAREA (TSA word), 7-3
Enigma
  Floating point, 15-2
ENTER key, 11-3
EOL word
  EXPECT, 11-15
Escape control sequences, 14-14
  EMIT, 12-3
ETX/ACK protocol
  Serial Interface Adaptor, 14-25
Event control block, ***see*** ECB
EXECUTABLE file type, 13-2, 13-7
EXPCT, 11-17
EXPECT, 11-14
  Dot graphics and, 12-7
Extrinsic
  Virtual file space, 13-9, 13-10
Extrinsic RAM, 2-3

F

f1/f2/f3 keys, 11-3
File
  Current, 13-2
  Header, 13-1
  Multiple files with same name, 13-3
  Non-text, 13-11

Output causes other files to move, 13-1
Files, 1-7
Virtual, 13-1
FLAG3, 14-31
FLAME.ON, 12-13
Flash display, 14-15
FLEE, 7-9
Floating accent
EMIT linkage, 12-3
Table, 12-10
Floating point
Numeric representation, 2-3
Representation of, 15-1
Special case, 15-1
Forward references, 4-19
Function key, 11-3
"Typing" from program, 11-23
Precedence over keyboard & pushkey buffers, 11-23
Function vs. program, 17-5

G

GET-TYPE, 13-7
Glossary
Format of, 1-2
GRAB, 7-6
Graphics
EXPECT and, 12-7
LCD, 12-6
Ground state, 7-8

H

H: (host definition), 12-9
Hardware device code, 14-6, 14-12
Header, 17-6
File, 13-1
HELP key, 11-2
HHC character set, 11-7
Hidden keys, 11-1
Host definition, 12-9

I

I/O
Drivers, 1-6
Intrinsic, 1-5
Peripheral drivers, 1-7
I/O bank, 1-5
I/O key, 11-2
Immediate keys, 11-1
INSERT, 13-4, 13-5

Insert cursor, 11-16
INSERT key, 11-3, 11-17
Integer
Representation of, 2-3
Interrupt, 14-1, 14-6
Interrupts, 1-8
Intrinsic, 1-5
Applications, 1-7
RAM, 2-1, 7-5
ROM, 2-3
Virtual file space, 13-9, 13-10
Intrinsic I/O, 1-5
Inverse display, 14-15
INVISIBLE file type, 13-2, 13-7

K

?KEY, 11-20
KBVECT, 11-5
KEY
RIP and, 14-9
Keyboard, 11-1, ***see also*** names of individual keys
Character translation table, 11-4
Hardware of, 11-1
Hidden and immediate keys, 11-1
Input with EXPECT, 11-14
Keyboard buffer, 11-20
Logical unit number 0, 14-7
Raw keyboard code, 11-4, 11-6
RIP ***versus*** KEY, 14-9
Symbolic constants for keys, 11-24

L

Latch byte, 2-5
LCD
As a peripheral, 14-16
Blips, 12-13
Character control buffer, 12-5
Character display buffer, 12-5
Character translation table, 12-7
Cursor, 12-1, 12-11
Display buffer, 12-6
Displaying accent marks, 12-10
Graphics, 12-6
Logical unit number 1, 14-7
ROP ***versus*** EMIT, 14-9
Slaving peripherals to, 12-13
STOP/SPEED control, 12-13
Used as data buffer, 17-13
LCD.CR, 12-6

LETGO, 7-5
LOAD, 10-3
LOCK key, 11-3
LOCK-KEY, 14-9
Logical unit number, 14-6, 14-7, 14-12
LOOKUP, 13-8
Loop stack
  Size of, 2-1
LUN
  Unattaching and reattaching, 14-11

M

MAKE, 13-3
Memory map, 2-1
Menu driver, 17-9
Micro Printer, 14-17
Modem, 14-19
"Modem initialization file,
  name of", 14-21
Mother tag, 17-6

N

NAP, 11-20
Non-text file
  Format of, 13-11

O

ON/OFF keys, 11-2
OPEN, 13-2
Order entry program, 17-1
Overflow
  Floating point, 15-2

P

Parameter stack, 7-6
  Size of, 2-1
Peripheral
  Interface, 1-6
Peripherals, 1-5
  I/O drivers, 1-7
Pheripheral
  Slaving to LCD, 12-13
Posting an ECB, 14-2
Power, 1-8
Program function keys, 11-3
Program vs. function, 17-5
Programmable Memory Peripheral
  Virtual files, 13-9

Pushkey buffer, 11-22
  Precedence over keyboard buffer, 11-22

Q

Queueing an ECB, 14-2
Queuing an ECB, 14-5

R

?ROOM, 7-6
RAM, ***see also*** Temporary storage area
  Dynamic allocation of, 7-6
  Extrinsic, 2-3
  Intrinsic, 2-1, 7-5
  Memory map, 2-1
Raw keyboard code, 11-4, 11-6
RCLOSE, 14-10
RCTL (Request Control), 14-10
READ, 13-3
Real-time clock, 14-29
REC-CNT, 13-5
Record number, 13-1
Restarting an application, 7-8
Restrictions
  Relocatability, 5-6
Return code, 14-4, 14-12
Return code in ECB, 14-3
Return stack, 2-4
  Size of, 2-1
REVISE, 13-5
RIP (Request Input), 14-8
ROM
  Allocating space in, 7-7
  Effect of write to, 2-2
  Intrinsic, 2-3
  Memory map, 2-1
Root of stack, 2-4
ROP (Request Output), 14-8
ROPEN, 14-10
ROTATE key, 11-3
Rotation mode, 12-13
(ROTMODE), 12-13
RS-232 interface, ***see*** Serial Interface Adaptor
Run Time, 5-1

S

SAVESNAP, 5-9
SDT
  Addresses in, 14-11

SEARCH key, 11-3
SECOND SHIFT key, 11-1
Serial Interface Adaptor, 14-24
Servicing an interrupt, 14-1
SHIFT key, 11-1, 11-2, 11-3, 11-4
SMART-POSN, 12-4
SnapFORTH interpreter, 1-6
SOFTAG, 7-9
SOFTCTL, 7-9
SOFTFLAG, 7-10
SOFTROM, 7-9
SOFTVECT, 7-9
Special case, 15-1
  Program-defined, 15-3
Special HHC keys, 11-13
Squeak, 12-15
Stack
  Tip and root, 2-4
Stacks
  Parameter, 16-3
  Return, 16-3
START.CURSOR, 12-7
STOP.CURSOR, 12-6
STOP/SPEED key, 12-13
STP/SPD key, 11-2
String representation, 2-3
Symbolic constant, 7-1

T

Tag, 1-8
Tag table, 17-6
Tag Tables, 4-5
Tags
  types, 4-2
  vectors, 4-8
TEMPORARY file type, 13-7
Temporary stack, 13-9, ***see*** Temporary storage area
  Used by Micro Printer, 14-17
Temporary storage area, 7-3
Terminal Input Buffer, 10-1
TEXT file type, 13-2, 13-7
THEN and AGAIN, 8-8
Tick, ***see*** Clock tick
Time of century clock, 14-29
Timer, 14-29
  ECB, 14-30
  ECB format, 14-4
Tip of stack, 2-4
Traffic bit, 14-2, ***see*** Wait bit

TSA, ***see*** Temporary storage area
  Overflow of, 7-6
  Size of temporary stack, 7-5
  Temporary stack, 2-4, 7-6
  Variables allocated in reverse order, 17-7
Types of virtual files, 13-2, 13-7

U

Unattaching a LUN, 14-11
Underflow
  Floating point, 15-2

V

Vectored I/O, 12-13
Video interface, 14-19
Virtual file
  Memory map, 2-1
Virtual file space, 13-9
  Structure of, 13-10
Virtual Files, 1-7, 13-1
  File system editor, 13-8
Virtual memory, 7-7
Vocabularies, 4-16
  chaining, 4-18

W

WAIT, 14-2, 14-6, 14-13, 14-30
Wait bit, 14-2, 14-5
WAITM, 14-11
WHILE, 8-7
Word
  Representation of, 2-3
Word break character, 14-16
Words
  Assembler Exits, 16-5
  CODE, 16-1
WRITE, 13-4
  Renaming a file, 13-7

X Y Z

XON/XOFF protocol
  Modem, 14-20, 14-22
Zero divide
  Floating point, 15-2

FRIENDS AMIS, INC.

The program described in this document is furnished under a license and may be used, copied and disclosed only in accordance with the terms of such license.

FRIENDS AMIS, INC. ("FA") EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE RESPECTING THE HHC SOFTWARE PROGRAM AND MANUAL. THE PROGRAM AND MANUAL ARE SOLD "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE AS TO THE MEDIUM ON WHICH THE SOFTWARE IS RECORDED ARE LIMITED TO SIXTY (60) DAYS FROM THE DATE OF LICENSING BY THE INITIAL USER OF THE PRODUCT AND ARE NOT EXTENDED TO ANY OTHER PARTY.

USER AGREES THAT ANY LIABILITY OF FA HEREUNDER, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE LICENSE FEE PAID BY USER TO FA. FA SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS, BUT NOT LIMITED TO, LOSS OR INJURY TO BUSINESS, PROFITS, GOODWILL, OR FOR EXEMPLARY DAMAGES, EVEN IF FA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

FA will not honor any warranty when the product has been subjected to physical abuse or used in defective or non-compatible equipment.

The user shall be solely responsible for determining the appropriate use to be made of the program and establishing the limitations of the program in the user's own operation.

An important note: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or programs are satisfactory.

USA

Panasonic Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

CANADA

Panasonic Canada
Division of Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississaugo,
Ontario L4W2T3

OTHERS

Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center