HP-41 M CODE FOR BEGINNERS

BY:

KEN EMERY

HP-41
MCODE
FOR
BEGINNERS

by Ken Emery

a book for the nonfad machine

Ken Emery

published by:
SYNTHETIX
P.O. Box 1080
Berkeley, CA 94701-1080 USA

ISBN 0-9612174-7-2
Library of Congress 85-61881

Also available from SYNTHETIX:

HP-41 Synthetic Programming Made Easy
by Keith Jarett

HP-41 Extended Functions Made Easy
by Keith Jarett

Inside the HP-41
by Jean-Daniel Dodin

HP-71 Basic Made Easy
by Joseph Horn

ENTER (for the HP-11, 12, 15, and 16)
by Jean-Daniel Dodin

Quick Reference Card for Synthetic Programming

ACKNOWLEDGEMENT

There are six people who, through their tremendous contributions, helped to make this book a reality.

The primary program contributor for this book is a person known only as SKWID. He has written articles for the PPC JOURNAL on beginning MCODE programming, as well as some advanced User code programs. Other programs were written by Clifford Stern, who also served as technical consultant.

David Johanson, Pete Graves, and David Hovik provided a great deal of insight into how the book should be structured, as well as correcting some of the more blatant English errors. I would also like to thank David E. White, Editor of the PPC Journal, for the editorial comments he made during the creation of the book. I would also like to thank Blaine Albios for drawing all of those great pictures of SKWID.

ABOUT THE AUTHOR

Ken Emery is a Chemical Engineer who graduated from Cal Poly Pomona in March of 1985. His first calculator was an HP-41CV purchased in August of 1981 (talk about starting at the top!). He has been addicted to HP calculators ever since. First, he worked on becoming familar with the HP-41 operating system through the use of Synthetic programming. With the advent of MCODE, and the possibility of newer, faster programs, he had to enter this field to satisfy his craving for more speed out of the little box called a 41.

WHAT IS MCODE?

MCODE is the internal machine code used by the HP-41, one level below the set of "user code" instructions that users and programmers are accustomed to dealing with. Some user code instructions like CLX are implemented by the HP-41 in just a few MCODE instructions; other user code instructions like TAN consist of hundreds of MCODE operations.

HISTORICAL BACKGROUND

When Hewlett-Packard announced the HP-41C in July 1979 they described it as: "A Calculator, A System, A Whole New Standard." Six years later we know these bold statements to be true. The HP-41 has been successful beyond HP's most optimistic expectations.

By the end of 1979, only five months after the introduction of the HP-41, the beginnings of a new form of programming appeared. Pioneered by Dr. William C. Wickes, it is now called synthetic programming, or SP. Synthetic programming encompasses the creation and use of new undocumented instructions to which the HP-41 responds. Synthetic programming is only an extension of user code programming. Its study, however, provided an important general overview of the HP-41's operating system and its memory management. The next step was to find ways to list and study the internal machine code, now called MCODE.

User community programming in MCODE was discouraged by HP. "It's too complicated and in many cases doesn't offer an advantage," was the usual reason given by HP's technical support staff. By the spring of 1982, however, the first MCODE programs were written, hand compiled, and burned into EPROM by Jim De Arras.

Four problems had to be overcome before MCODE could become popular. First, the user community had to discover that MCODE programming is not beyond the grasp of talented programmers. The second problem was the documentation of the HP-41's operating system. HP eventually released the annotated operating system listings, but only after Jim De Arras produced his own version, a monumental feat. The third problem was the lack of a means to generate and store MCODE instructions. Several small manufactures now offer the necessary hardware to the user community. The fourth and last problem was documenting in one place the basics of MCODE programming. This book is the result of that effort.

WHY SHOULD YOU USE MCODE?

The first reason to use MCODE is speed. MCODE programs run from 7 to 120 times faster than user code. The second reason is that you get full system control. More efficient data register usage (data packing) and access to all of system memory are but two examples. A third reason to use MCODE is that greater accuracy is possible by using the internal 13-digit math routines. A fourth reason for using MCODE is the ease of dealing with hexadecimal (base 16) numbers. The HP-41 has MCODE instructions to do hexadecimal arithmetic at least as easily as decimal arithmetic. Finally, your MCODE programs are immune to MEMORY LOST because they do not reside in normal user code program memory.

MCODE programming requires additional hardware, costing from $100 to $400. But once you enter the world of MCODE there is nothing you can't do. To get started, however, you need to understand the basics of MCODE. That's where this book fits in. It will give you the background you need to write your own MCODE programs and to start to understand the HP-41's operating system. Understanding the operating system is the key to the most advanced applications of MCODE.

Richard J. Nelson
Editor, CHHU Chronicle

PREFACE

With the introduction of the HP-41C in July of 1979, the world of truly personal computing was set on its ear. In one hand, the computer user was now able to hold what once took an entire room full of hardware. At the time of its introduction, the HP-41C was expected to have a product life of five years. Based on the results of a survey made of the user community in late 1984, the projected life of the current 41 series (CV/CX) is *still* 5 years. The overwhelming success of the 41 is due in large part to enterprising users who managed to tickle ever more power out of their 41. Dr. William Wickes first discovered and utilized "synthetic programming" for the HP-41, with Keith Jarett, Roger Hill, and others expanding the bounds of knowledge significantly. In 1981, members of the Personal Programming Center (PPC) created an astounding collection of programs for the PPC ROM, which combined synthetic programming techniques with improved algorithms to come up with what is still the most advanced non-MCODE ROM around.

Hewlett-Packard has responded to the success of the HP-41 by introducing new products (such as Extended Memory, HP-IL, and the Time module) that expand the capabilities of the 41 manyfold. Pioneering work by Steve Jacobs and Jim De Arras in the disassembly of HP-41 instructions led HP to unofficially release the operating system listings for the 41, along with the original programmers' annotations. Thus was born the art of MCODE programming.

MCODE programs can normally be executed only as part of an internal or plug-in ROM (Read Only Memory) module. As the name implies, ROM modules cannot be reprogrammed. Lynn Wilkins and Paul Lind originally developed the Machine Language Development Lab (MLDL) to enable programmers to conveniently write, test, and use MCODE programs. Later refinements by Lynn Wilkins, Paul Lind, Nelson Crowle, and the ERAMCO company led to today's state-of-the-art MLDL. An MLDL contains ordinary memory (RAM) that looks like ROM to the HP-41. It also contains sockets that allow you to plug in EPROM (erasable, programmable, read-only memory) chips. EPROM's, which can be programmed using third-party hardware that connects to the HP-41, let you create your own custom ROMs inexpensively.

Most of the MLDL-type devices available today have some, if not all, of the following features:

- 4K to 16K of RAM that emulates HP-41 ROM (with battery back-up)
- Sockets for 4K to 24K of EPROM's that emulate HP-41 ROM
- Development software to aid in MCODE programming

Once the hardware problem was solved, software needed to be tackled. MCODE programmers all over the world developed assemblers, dissassemblers, editors, and general-purpose MCODE programming tools. These software development tools, which are standard on computer systems, are now available for the HP-41.

But alas! With all of this programming power available, HP-41 users still had a tough time trying to learn how to program in MCODE. To make it easy on yourself, you needed to speak fluent Jacobs-DeArras, Hewlett-Packardian, and ZENGRANGEish to be able to understand the various mnemonics. Further, the only method of learning for each programmer was to start at the bottom, with all of the appropriate documents in hand, and pull himself up by his bootstraps. One evening, Ken Emery was bemoaning the lack of a tutorial on MCODE to several local PPC members. "Write it yourself!", they told him. So he did, and the rest is history.

This book will do its best to try and guide you through all of the vagaries of HP-41 MCODE programming that you are likely to experience as a beginning MCODE programmer. Intermediate programmers will find a fair amount of useful information as well, perhaps a few little-known tricks that will cut program size or execution time. And advanced MCODE programmers will get a kick out of remembering how they first discovered these secrets.

David E. White
Editor, PPC Journal

TABLE OF CONTENTS

INTRODUCTION

This book will introduce you to machine language programming on Hewlett-Packard Series 40 calculators (the HP-41C, CV, and CX). This book is suitable for total beginners in machine language, but experience in normal HP-41 programming will prove helpful.

Machine language (also known as MCODE) is the language used to program the internal functions of the calculator. With machine language (MCODE), you have total control of the calculator. The execution speed of an MCODE program can be anywhere from 5 to 120 times as much as that of a similar User code program.

To help you better understand HP-41 machine language programming, we will first review the structure of the CPU registers. Next we will discuss the instruction set, and finally we will provide examples of how to use the various instructions. In the process, several practical routines will be demonstrated. Each routine is fully documented to provide a clear understanding of why a particular instruction was chosen at each step.

Throughout this book we shall refer to machine language programming on the HP-41 as MCODE. The term MCODE is derived from both Machine language programming and microCODE. Machine language is the language determined by the instruction set of the CPU. Microcode is the electronic programming that actually determines what the CPU's instruction set will be. When machine language programming first became possible on the HP-41, the term MCODE was coined, and it remains in use to this day.

In order to program in MCODE, you must have an accessory that simulates the ROM (Read Only Memory) of the HP-41. This is because the HP-41's operating system is not designed to run MCODE programs from its normal RAM (Random Access Memory) area. Extensive internal ROM contains the permanent code that determines the function set of the HP-41. Several types of devices are available for this purpose, and they are commonly referred to as MLDL's (short for Machine Language Development Lab). These devices plug into one of the four ports at the top of the 41. They contain RAM, memory that may be altered by the user, suitable for holding MCODE programs. Further explanation will be provided in the hardware section of this book.

THE BASICS

BINARY NUMBER REPRESENTATION

The CPU can only interpret binary numbers. Binary numbers are base 2 numbers. They can only be represented using a one or zero. For example, 6 in base ten would become 110 in binary. Let's examine how this is done. The rightmost digit is the one's place; it may be either one or zero. When we get to 2 we must go to the next digit to the left. This is the two's digit. If it is a 1 then we add 2 to the total. If the one's and two's digits are set to one we have 3 (1 + 2 is 3). If we want to continue counting, then we must move to the next digit to the left, which is the four's digit (four comes after three). If this digit is one, then we add 4 to the total. In our example the four's digit and the two's digit are one. This means that we have 4 + 2 (or 6). Since the one's digit is zero, we don't add one to the total.

As you can see, counting in binary can be rather difficult (unless you only have two fingers). When writing programs for the HP-41's CPU in binary it is very easy to make a mistake. In the CPU of the 41 the instructions are ten binary digits long. Each of these digits is known as a BIT (for <u>BI</u>nary digi<u>T</u>). Now, if you have a program that is 100 instructions long, then you would have to check 1,000 (100 instructions times 10 bits per instruction) bits to make sure that you have made no errors. As you can see, writing programs in binary makes them difficult to debug. Binary numbers all look the same, particularly after a few hours of debugging.

Since computers never get tired, and love to work with binary numbers, we write programs to translate our inputs into binary. We input in hexadecimal (hex for short) or base 16. Since numbers only cover from 0 to 9, we must borrow letters from the alphabet for the last 6 hex digit values. We use the letters A through F, with A corresponding to 10, B to 11, and so on until we get to F, which is 15 in base ten.

Here's an example of how much easier hex is than binary. We will use ten-bit binary numbers since this is what the 41 CPU uses.

Binary	Hex
0110011110	19E
1100101001	329
0000010000	010
1111101001	3E9
1000110111	237

If you make a mistake keying in the binary instructions, then you must examine 50 bits to see where the mistake is. Using hex, only 15 digits must be examined. This is a reduction of 70% in the number of digits you must check.

How do we get the CPU to use these hex digits if it only recognizes binary numbers? We use a program which will translate our hex codes to binary. This program is called a hex assembler. Since computers don't make mistakes, the translation from hex to binary will be performed without any mistakes.

Since most people can't count too well in hex (we haven't seen anyone with 16 fingers), the hexcodes are given alphanumeric representations of the operations that they perform. These alphanumeric representations are called mnemonics. The program that translates these mnemonics into binary is called an assembler. These programs are usually rather elaborate. However, they make programming much easier, since you can actually see what each instruction does, and you may follow the logic of the program. For example, the binary number 0000001110 (00E in hex) is the A=0 ALL instruction in the microprocessor of the 41. It is much easier to figure out what the A=0 ALL instruction does (sets all of CPU register A equal to zero), than to translate 0000001110 to a number which you may then look up on a chart.

The opposite of the assembler is the dissassembler. This is a program which takes the binary codes at specified locations in memory and translates them to mnemonics so that you may easily examine what instructions are in memory.

You may be wondering why the HP-41's main CPU registers are 56 bits wide. The 41 was designed with numerical computation in mind. The number 56 is divisible by 4, therefore it may be partitioned into 14 sections of four bits each. The reason for using four bits is because the numbers zero to nine may be represented using four bits. The leftmost four bits (one nybble) are used to tell whether the number is negative or positive. If this nybble is 0, then the number is positive. If it is equal to nine (1001 in binary), the number is negative.

The next ten nybbles are used to hold the mantissa of the number. Because there are only ten mantissa digits the 41 is accurate in calculations to ten decimal places. For example, the mantissa of PI is 3141592654. These are the ten digits you see when PI is in the display and you are in FIX 9 mode.

The three rightmost nybbles are the exponent sign and the exponent. The leftmost of the three is the sign of the exponent. This is encoded in the same way as the sign on the mantissa. It is nine if the exponent is negative, and zero if it is positive. The next two nybbles form the exponent. The 41 stores all numbers in scientific notation, that is, with the exponent set so that the mantissa has only one number to the left of the decimal point. You may remember that the exponent on the 41 may range from 0 to 99. This is because the largest decimal number in two digits is 99. The CPU cannot handle an exponent greater than 99 because there is no room to store the three digits (100 and greater) needed to represent this. For numbers with negative exponents the number stored in the exponent is 100 minus the exponent. For example, for a negative exponent of 2 the actual number stored is 98 (100-2). The reason numbers aren't always displayed in scientific format is because HP was kind enough to give you a choice of whether you want scientific, engineering, or no exponent (FIX format) displayed. The display routines take care of all of the work to make sure the number is displayed in the format you want.

THE MICROPROCESSOR

A microprocessor is the heart of any computer. The microprocessor chip is made of silicon, just like any of the other integrated circuits that comprise a computer. However, it has been designated as the controller of the whole show. The microprocessor has been manufactured so that it recognizes certain inputs, and then it tells everything else what to do. It is the brain of the computer.

When this chip is manufactured, a set of commands that will delegate the work is etched into the chip. These commands are known as the instruction set. The microprocessor has a set of registers where all of the operations are carried out. These registers are known as the CPU registers. The CPU registers are completely separate from the memory registers, as you'll see later.

In many texts, you may have noticed references to Microprocessor, Micro Processing Unit (MPU), and Central Processing Unit (CPU). These terms all mean the same thing. To maintain some semblance of consistency, we will use the term CPU throughout the book when referring to the HP-41 microprocessor.

In the CPU of the 41, ROM (Read Only Memory which may **NOT** be altered by the user), and User RAM are not the same. In the ROM address space the bytes are each 10 bits long. The CPU has a 64 Kilobyte address space for ROM. Therefore it can have up to 65,536 bytes of functions and programs. The way the 41 CPU was designed was to treat this whole area as ROM. The User RAM is treated as a peripheral by the CPU, and is not part of the 64K ROM address space. The RAM bytes are each eight bits long. The 41 CPU further complicates matters by storing the eight bit bytes of User RAM in 56-bit registers (7 bytes per register).

Each 10-bit word of an MCODE instruction takes 155 microseconds to execute. The only exception is FETCH S&X (introduced on page 50), which takes twice as long. The CPU thus processes an amazing 6452 words of MCODE per second.

THE CPU REGISTERS OF THE HP-41

In order to program in MCODE you MUST know how the internal CPU registers interact with each other. This is not like User RAM, where you do not have to worry about the partitioning of programs and data. Remember, with MCODE you are in command of the calculator at the most fundamental level. Therefore you must know what you are doing in similar detail. Almost anything you want to do can be done. Like a good synthetic programmer, who must know that there are 16 status registers and how they are used by the calculator, you must know how the data flows through the internal CPU registers. A diagram of the flow of data in the CPU registers is given below. The numbers in parentheses are the lengths of each register in bits. Each register is named by a letter(s).

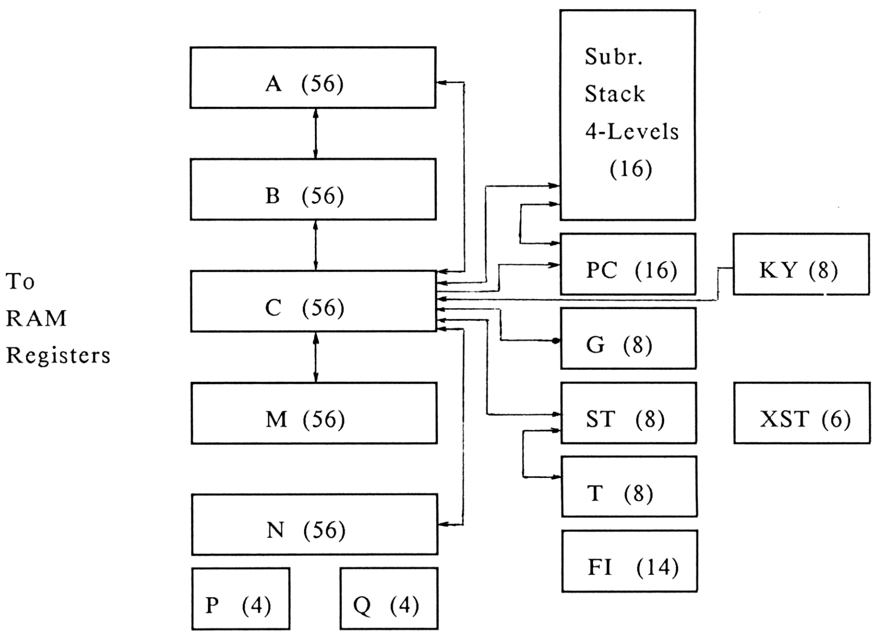


Figure 1

Now for a short vocabulary lesson, followed by a little explanation of the uses of each of these registers.

Word	Definition
Bit	Binary digit. One bit can have a value of either 1 or 0. It is like a switch, either on or off.
BCD	Binary Coded Decimal. This is how the CPU represents the numbers you see. Each decimal digit is represented by four bits (one nybble). Each of the nybbles is separate from the other, and may have a value from zero to nine. When one of the nybbles tries to become ten, a one is added to the nybble to the left, and the original nybble is set to zero.
Hexcodes	The three hex digits used to symbolize the ten-bit MCODE words.
Mnemonics	Alphanumeric representations of what certain hexcodes do. For example, the hexcode 00E has a mnemonic of A=0 ALL. From the mnemonic you can deduce that hex 00E sets <u>all</u> of CPU register <u>A</u> <u>equal</u> to <u>zero</u>. This is much easier than having to memorize what each hexcode stands for.
Nybble	Four bits put together. The highest value that may be obtained is when all 4 bits are set to 1. This is 15 decimal, or F in hexadecimal. One nybble is also one hexadecimal (hex) digit.
NOP	<u>N</u>o <u>OP</u>eration (do nothing instruction).
Byte	Two consecutive nybbles or eight consecutive bits.
Shift	Movement of data within a register, either left or right. Any data pushed off the end of the register is lost forever. For example, if we shift the binary number 10110111 right by 2 bits

the two rightmost bits will be lost and zeros will be placed on the left. We then end up with 00101101.

Wraparound Movement of digits from one side of a register to the other, during rotation of a register. Rotation is like shifting right except instead of losing the rightmost digits they are wrapped around to the left. For instance, if the above example was rotated instead of shifted, we would get 11101101 as our answer. Notice that the last two digits were placed on the left end of the number and were not lost. This is wraparound. You may also be familiar with this term as logical rotation.

Word The CPU instructions of the HP-41 are 10 bits long. So the term <u>Word</u> describes a ROM memory cell that holds a single CPU instruction. The term Byte is avoided in this context in order to distinguish ROM words from the 8-bit bytes in RAM. However, you will occasionally see CPU instructions referred to as bytes, for example when the "byte count" of a routine is quoted.

Underflow Underflow occurs when a negative number would result from an operation. The CPU does not know what negative numbers are, so it gives a result as if it had borrowed a one from the next most significant digit. For example, the operation 1001 minus 1100 would result in an underflow, since 1100 is greater than 1001. The result would be 1101, which is 11001 minus 1100. The Carry, which will be explained later, is set whenever an underflow occurs.

Overflow Overflow is the opposite of the underflow. It is much like the OUT OF RANGE error message we get when a number greater than 9.999999999 E99 would result from a mathematical operation. If the operation were carried out, there would be an overflow, since the wanted number would be too large for the CPU to handle. The CPU just chops off anything that would be larger than it can handle. For example, 1001 plus 1000 would be 10001. But since we are using only four bits for our example, the leftmost bit would be eliminated and the answer would be 0001. The Carry bit is set after one of these operations.

Here is an explanation of how the CPU registers function.

Register	Usage
C	This is the main register. All communication with the RAM registers is done through the C register. This is the only register that can directly interact with all of the other CPU registers (except T). This register can either be shifted one nybble right or the whole register may be rotated from 1 to 13 nybbles to the right. 4-bit digits (0 to F in hex) may be loaded into any nybble of this register. This register corresponds to the accumulator on other CPUs. It may be incremented or decremented by one, and it may also be zeroed.
A	The A register may interact with only the C and B registers. These registers may be added to A and they may also subtracted from A. A can also be added to C. It can be incremented or decremented by one, shifted left or right one nybble, or zeroed.
B	This register may be added to or subtracted from only the A register. However, it may be exchanged with the A and C registers in whole or in part. It may also be shifted right one nybble, or zeroed.
M and N	These registers may interact with only the C register. They can not interact with each other, or with any register other than C. They are usually used for storage.

P and Q	These 2 four-bit registers are the pointers. They may be set to any value from 0 to 13. They are used to point to digits in the A, B, and C registers. Only one of the pointers may be selected as the active pointer at any time. The active pointer may be incremented or decremented by one. The active pointer is sometimes referred to as the 'R' register.
PC	This is the program counter. It contains the address of the MCODE instruction that is currently being executed. It may be modified using certain instructions.
Subroutine Stack	The subroutine stack has space for 4 pending returns. These returns may be popped into the C register. Part of the C register may be pushed onto the subroutine stack. This stack should not be confused with the subroutine stack used for User code programs.
G	This register interacts with the C register at the nybble pointed to by the active pointer, and the next highest nybble. If the nybble pointed to is 13, then wraparound occurs.
ST	This is the flag register. Flags 0 to 7 reside in this register. They may be set, cleared, and tested. The ST register may be zeroed and exchanged with, or set equal to, nybbles 0 and 1 of the C register. Nybble 0 is flags 0-3 and nybble 1 is flags 4-7. Note that these flags are independent from the User flags of the 41, although they are frequently set to match User flags 48 to 55.
XST	This register contains CPU flags 8 to 13. XST cannot be directly accessed by any other register. These flags may be set, cleared, or tested. Note on ST and XST: Flags 0-13 are also referred to as status bits in HP documentation.

KY — This is the keyboard register. When a key is pressed, KY is loaded with a two-digit hexcode from a table built into the CPU (see the table on page 150). Part of registers C and PC may be set equal to KY.

FI — Peripheral flag register. These flags may only be tested by the CPU. They must be set by a peripheral.

Carry — This one bit is set when an overflow or underflow occurs. It is also set if a test is true. After the carry is set, the next MCODE instruction clears the carry, regardless of whether that MCODE instruction tests the carry bit.

What follows is the ROSETTA STONE of MCODE programming. Figure 2 shows the fields of a 56 bit register. These 56 bits are divided into 14 nybbles. These are numbered 0 to 13 (starting from the right). The fields are used extensively to operate on all or part of the A, B, or C registers.

```
Nybble: 13   12   11   10    9    8    7    6    5    4    3    2    1    0

Field:  <---------------------------------------- ALL ---------------------------------------->
Field:  <-MS-><---------------------------- M ------------------------------><-XS->
Field:                                        <------- ADR --------><---- S&X ---->
Field:                                                  <-- KY -->
```

Figure 2

Note that these fields also function as postfixes for a number of instructions. Here are the functions of the fields in Figure 2:

Field	Usage
ALL	All 14 nybbles.
S&X	Exponent and exponent sign (nybbles 0-2).

XS Exponent sign only. (nybble 2)

M The 10 nybbles of the Mantissa (nybbles 3-12).

ADR Nybbles 3-6. This is where the address is taken from when a return is pushed onto the subroutine stack; it is also placed here when a return is popped from the subroutine stack.

KY Nybbles 3 and 4. This is where the contents of the KY register are placed. C cannot be placed into KY.

@R At the nybble pointed to by the active pointer.

P-Q Uses the nybbles pointed to by each pointer. The nybbles used depend on whether P is larger than Q. If P<=Q, digits P through Q are used. If P>Q, digits P through 13 are used.
For example; if P=12 and Q=2 and we execute the instruction C=0 P-Q, then nybbles 12 and 13 of C will be zeroed since P is greater than Q. If the values were reversed, then nybbles 2 through 12 would have been zeroed. For the field designation P-Q it does not matter which pointer is selected as the active pointer.

R< All digits from 0 through the digit pointed to by the active pointer.

The last three items (@R, P-Q, and R<) are not actually fields. They are postfixes to a group of instructions, as are the field definitions. These last three can change position, and can not be rigidly defined as being in one place (like the rest of the postfixes). Table 1, on page 27, contains all of the prefix instructions for use with the postfixes mentioned above. (By the way, a word about prefixes and postfixes. These are not before and after fixes for something you may be considering to do or did do wrong, rather they are descriptions of which half of the mnemonic is being discussed. The first half is the prefix; the second half is the postfix.)

THE HARDWARE

The hardware accessory needed to program in MCODE is called a Machine Language Development Lab, or MLDL for short. This device contains the necessary electronics to interface at least one 4 Kilobyte block of CMOS RAM with one of the ports at the top of the calculator. The total amount of RAM available for writing MCODE depends on the device.

At the present there are several popular versions of this box. One of these, the ERAMCO MLDL, has 8K of RAM (two 4K blocks) and space for 24K of EPROM (Erasable Programmable ROM). This device uses a hex code that the CPU regards as a NOP to trigger its write mode. Reading and writing to this device is very fast. However, in order to write MCODE to this device, you must have software written in MCODE. The ERAMCO MLDL is supplied with one 4K EPROM set to help you get started writing MCODE.

Another MLDL device is called the Protocoder II. This device uses the ABS function in the calculator to trigger its read and write functions. Because of this, it takes longer to read from and write to this unit. However, programs will run at the same speed when they are executed in either device. The main advantage of the Protocoder II is that software written in MCODE is not necessary, it just makes things much easier.

For those of you with an adventurous spirit, Volume 9 Number 3 of the PPC Calculator Journal contains schematics and instructions to build your own 4K RAM MLDL (with provision for 4K of EPROM).

Another type of add-on for the 41 is the EPROM box. This box provides the electronic circuitry enabling you to plug in EPROM (Erasable Programmable Read Only Memory) chips into the interface box. The calculator sees these as Application Pacs. With this capability you can write one-of-a kind ROMs for only the cost of a set of EPROMs (approx. $15 U.S.) and the cost of burning (programming) the EPROMs. This is much cheaper than having a custom ROM manufactured for you by HP (about $10,000+).

The ERAMCO MLDL comes with sockets that allow you to plug in up to 24K (six 4K sets) of these EPROMs. The Protocoder II requires the addition of an extra board that addresses up to 16K of EPROM memory. A company called Hand Held Products makes a variety of EPROM boxes. They even have one that uses an HP Card Reader case. You can put up to 32K of EPROM in this device.

A company called Corvallis MicroTechnology also makes an EPROM box that only uses one EPROM instead of the usual two. This device can hold either 4K or 8K of ROM. CMT also makes a plug-in module that has an EPROM built into it. This module looks exactly like a HP application pac except for the window on one side. With this module there are no extra boxes or extentions of the calculator. This module comes in 4K, 8K, and 16K versions. For more information about these manufacturers see Appendix A.

THE SOFTWARE

In order to efficiently program one of these boxes, some sort of software is needed to allow you to write to the RAM. This can be accomplished using either hexcodes or mnemonics; however, the software for writing to the boxes using hexcodes is much more prevalent. The main piece of software that you will need is an assembler. An assembler takes the mnemonics (alphabetic representations of what the hex instruction does) that you input and calculates the correct hexcodes to place into the RAM of your MLDL. A disassembler will output these hexcodes, along with the corresponding mnemonics, to a printer, video display, or the display of the 41.

The EPROM set that comes with the ERAMCO MLDL has the hexcode kind of assembler. This EPROM set also contains many utility routines not found elsewhere.

A 4K EPROM set written in Australia is known as the Assembler 3 EPROM. This set contains a disassembler, as well as an assembler that can assemble MCODE from mnemonics in the Alpha register. Working with the other functions of this EPROM is also a delight.

The Nelson F. Crowle ROM (NFCROM for short), another such set, is for use with the Protocoder II. It contains read/write functions for this device and many other useful functions.

A new 4K EPROM came out in May of 1984 that allows you to key in mnemonics from the keyboard. This revolutionary ROM is called DAVID ASSEM.

In order to enter instructions directly from the keyboard, each key is redefined with a mnemonic or mnemonic prefix (more on this later). This EPROM makes MCODE program input as easy as keying in a User code program.

With the use of software like this you should have no problem keying in any of the routines in this book.

For those of you who have User code (RPN) programs that you wish to put into your MLDL RAM, Phi Trinh has written a routine that will do this for you. The only input required is the name of the User program you wish to load into the MLDL. The routine compiles all GTOs and XEQs and has the most complete error checking of any routine yet written for this purpose. This routine is intended to be used only for creating User Code ROMs with your MLDL. The ERAMCO MLDL EPROM also has a routine that is somewhat similar to Phi's. ERAMCO's program allows you to mix MCODE and User code.

Instructions on how to use these software packages will not be covered in this book. Review their respective manuals for specifics of operation. The manufacturers' addresses for these software packages may be found in Appendix A.

SOURCE LISTINGS FOR THE HP-41'S OPERATING SYSTEM

Another very important piece of software is the operating system that is built into your HP-41. The so-called "mainframe" of the HP-41 contains 12 kilobytes of delicately interwoven MCODE programs that make the HP-41 what it is. The mainframe contains many routines to read the keyboard, access the display, and perform other frequently needed "housekeeping" functions.

Rather than write a complicated subroutine every time you need a housekeeping function in your programs, you can simply execute one of these mainframe routines as a subroutine from your program. The variety of mainframe functions is practically unlimited. If what you want to do has a counterpart in normal operation of the HP-41, chances are that the task exists as a subroutine in the HP-41's mainframe.

A mainframe routine begins at an **entry point**. In order to correctly use mainframe routines, you need to know the following:

1) The location of the entry point.
2) The initial conditions required, including which registers are used for input, correct flag settings, mode and peripheral selection, etc. Some routines require detailed setup; others do most of their own setup.
3) The routine's register and subroutine stack usage.
4) The output specifications, including what values are output and where, and how the routine ends (return to calling program, or return to the operating system).

To get this information, you need a copy of HP's annotated listings for the operating system. These listings are commonly referred to as the VASM listings (HP's terminology). Appendix A has a list of organizations that sell VASM listings. **Don't ask HP**, because HP does not support MCODE.

All serious MCODE programmers should spend some time studying the VASM listings. The listings will give you a much better idea of how the HP-41 works, and you are bound to run across some entry points that you can use later in your programs. You'll also get an appreciation for the complexity of this operating system, which was written by a team of 2 or 3 very skilled programmers.

THE ROM ADDRESS SPACE

The 64 kilobyte (64K) ROM address space of the 41 is divided into 16 pages - each of which is 4K in length. Each of these pages contains 4096 ROM words that are each 10 bits long. The RAM that is used for User code programs is not included in this 64K, since it is addressed in a different manner.

Some of these 4K pages have been allocated by HP for specific uses. A list of how these pages are allocated is given below in Figure 3.

Page Number	Use	Page Number	Use
0		8	Lower half Port 1
1	Mainframe ROMs	9	Upper half Port 1
2		A	Lower half Port 2
3	Extended Func. (CX only) Not used (CV and C)	B	Upper half Port 2
4	Service module or Disabled IL Printer	C	Lower half Port 3
5	Timer Module	D	Upper half Port 3
6	Printer ROM	E	Lower half Port 4
7	HP-IL Control Functions	F	Upper half Port 4

Figure 3

Note that the first 8 (0-7) 4K pages are reserved for specific purposes. The upper 8 pages are the ROM address space into which we plug all of our HP application PACs. If you plug a 4K ROM into port 1, it will use page 8. This leaves page 9 inaccessible since nothing else can be placed into this port.

THE ROM WORD

In the architecture of the 41, the ROM words are 10 bits long instead of the conventional 8 bits. The nomenclature used in this book will list these 10-bit words in hexadecimal (hex). In order to do this, 3 hex digits must be used. All ROM words will be of the form:

VNN   Where V can range from 0 to 3, and N can be from 0 to F.

There are alphabetic descriptions or mnemonics for each of these different 3 digit hex codes, but that's the subject of another chapter.

HOW A 4K PAGE IS DIVIDED

In addition to assigning specific purposes to pages, HP has assigned specific purposes to individual address areas <u>within</u> each 4K page. The first section of a 4K page assigns the XROM number, the number of functions, and the addresses of the functions within the 4K page.

Let's give the section of the ROM we are about to describe the acronym **FAT**, short for **Function Address Table**. The first word, at address P000, is the XROM number. 'P' is the page number (any value from 5 to F). The number at this address, called the XROM ID, may be from 001 to 01F in hex (1 to 31 in decimal). This is the first number that is displayed when you see a function displayed as an XROM. For example, the Standard Applications Pac function CLSTK is displayed as XROM 05,01 when the ROM is not plugged in. The 05 is the decimal equivalent of the hex number at address P000.

The word at address P001 indicates the number of functions for that 4K ROM. This number may range from 001 to 040 hex (1 to 64 in decimal). The functions include any global labels from User code programs contained in the ROM, as well as any MCODE functions that are programmed into the ROM. This number also includes any headers that are in the ROM. A header is nothing more than an MCODE function with a name that is between eight and eleven characters. A ROM may have more than one header. An example of this is the HP-IL module. It has two headers, **-MASS ST 1H** and **-CTL FNS**.

Now comes the tricky part. This next set of words is grouped into pairs. They indicate to the calculator the address of the first executable instruction in a ROM routine, be it User code or MCODE. The words are of the following format:

Address	Word	Description
P002	UVW	This pair of words specifies a function whose starting
P003	XYZ	address is PWYZ. If U is zero, it is an MCODE function; if U is two, it is a User code program. Digits V and X are normally set to zero. W, Y, and Z correspond to the last three digits of the starting address of the function.
P004	UVW	This pair of words has the same format as the first pair
P005	XYZ	except they point to the address of the second ROM function.

We continue with this format of pairing the words together until all of the functions in our ROM have an address in the FAT. The two words after the last entry are set to 000. This signals to the calculator that the FAT has ended. You may start putting your programs after these final two words in the FAT.

Let's do an example. This ROM will have two functions. The first one, a User code program, will be located at address P119. A function written in MCODE will be at address P387. The XROM number for our ROM will be 14 decimal (0E hex).

Address	Hexcode	Description
P000	00E	This is the XROM number in hex. 00E is 14 in hex. We do not want to put 014 here since this would be an XROM number of 20 in decimal (014 in hex is 20 in decimal).
P001	002	This is the number of functions in our ROM, as specified above. It is also in hex. If we had 31 functions in our ROM this hexcode would be 01F.
P002	201	Since this is a User code program the U digit is set to 2. This tells the calculator to interpret the code starting at this address as RPN instructions. Notice that the V

		digit is zero. The 1 corresponds to the W digit in the starting address of the program.
P003	019	This is the second word of the two word set for the address of the first program. The X digit is set to 0. The 1 corresponds to the Y digit in the starting address, and the 9 is the Z digit.
P004	003	This is the first word of the two word FAT set for an MCODE function, so the U digit is set to zero. The V digit is 0, and 3 is the W digit.
P005	087	Here is the second word of this FAT entry. The X digit is 0. The 8 is the Y digit and the 7 corresponds to the Z digit.

Now come the two 000 words at addresses P006 and P007. You could start programming immediately following these instructions, but you don't have to. It is advisable to leave space between the last FAT entry and your first program so that more entries may be added to the FAT as you add more functions to your ROM. If you were to start programming your ROM at address P008, right after address P007, you would not be able to add any more functions to the FAT, since there would be no space to insert two more words into the FAT for the function. To leave room for a FAT containing the maximum number of functions (64), begin your programming at P084.

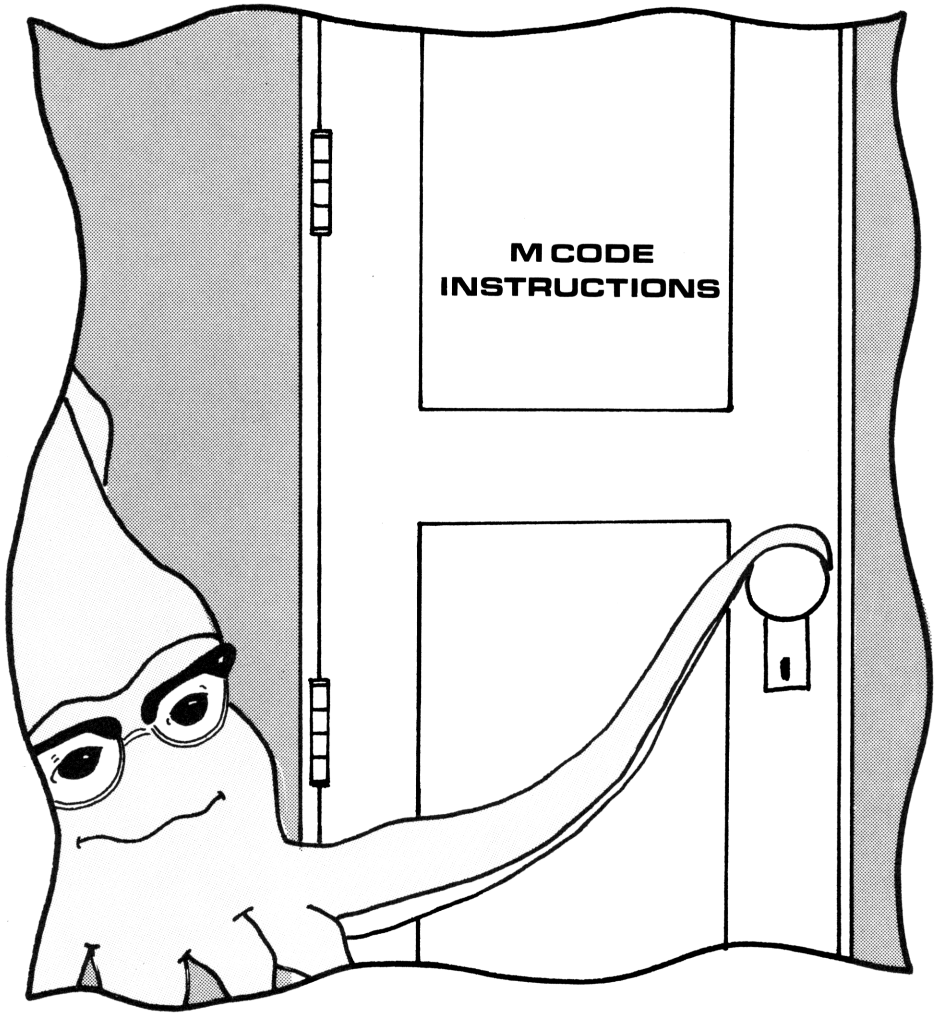
The rest of the 4096 words may be used for programs, until we reach PFF4. PFF4 to PFFA have been defined by HP as polling (interrupt) points. You should always leave these set to zero unless you know <u>exactly</u> what you are doing.

PFFB to PFFE are reserved for the ROM revision. The 4 hexcodes at these addresses correspond to letters which are read in reverse order starting with address PFFE. An example of this is the HP-IL Development ROM. The revision is PD-1B. The '-' is put in the display by the ROM-checking program. An example should help clarify this. Here are the words at addresses PFFB to PFFE in the HP-IL Development ROM.

Address	Hexcode	Alpha code
PFFB	002	B
PFFC	031	1
PFFD	004	D
PFFE	010	P

As you can see, the revision is read from the highest address, the address with the highest number value, to the lowest address.

The last word in the ROM is reserved for the checksum of the ROM. It is used by the Service Module and other modules to verify that a module is good. It is not used by the HP-41 itself. The checksum is calculated by adding the the total of all the words in the ROM up to, but not including, the last one. Anytime there is a carry into the 11th bit (ROM words are only 10 bits long) we add one to the total. To get the final checksum the 2's complement is taken. With the correct checksum in place, this process will give a result of zero if applied to all 4096 words.


M CODE
INSTRUCTIONS

THE TOOLS

THE INSTRUCTION SET

And now, without further ado, the HP-41 Instruction Set!

Instruction	Function
A=0	Sets the part of register A specified by the postfix to zero.
B=0	Does the same as above, but for the B register.
C=0	Does the same but for C.
A<>B	Exchanges the contents of the A and B registers, much like the function X<>Y in User code.
B=A	Copies the specified field of the A register into the B register. The old contents of B at that position are lost.
A<>C	Exchanges the contents of the A and C registers. This is the only direct way to place the contents of A into C.
C=B	Set C equal to B as specified by the postfix. The contents of B remain the same. Only the C register is altered.
C<>B	Exchange the contents of the C and B registers.
A=C	Set A equal to C. The contents of C remain unchanged. A is changed as specified by the postfix.
A=A+B	Adds the A and B registers and puts the result into A. The contents of B are undisturbed.
A=A+C	Same as above except use C instead of B.
A=A+1	Add 1 to A as specified by the postfix.
A=A-B	Subtract B from A. The contents of B are not disturbed. A contains the result.
A=A-1	Subtract 1 from A as specified by the postfix.
A=A-C	Subtract C from A. The result is in A. C is not disturbed.
C=C+C	Add C to itself. This shifts all of the bits in the specified portion of C left by one bit. This is commonly used as a quick multiply-by-2.
C=C+A	Add the C and A registers. The result ends up in C; the A register is left undisturbed.

C=C+1	Add one to the C register as specified by the postfix.
C=A-C	Subtract C from A and put the result into the C register.
C=C-1	Subtract one from the C register.
C=0-C	Gives the 1's or 9's complement of the designated field, according to whether the CPU is in hex or decimal mode. In hex mode, each bit is inverted; in decimal mode each digit is subtracted from 9. For example the 1's complement of 1101 is 0010, and the 9's complement of 43 is 56.
C=-C-1	2's or 10's complement of the specified field, according to the CPU mode (hex or decimal). This is the 1's or 9's complement plus one. For example, the 2's complement of EC is 13+1 = 14 hex; the 10's complement of 67 is 32+1= 33 decimal. Two's complement is ordinarily used to represent negative numbers in computers. In the HP-41, 10's complement is used for both the exponent and mantissa fields of numbers. For example, an exponent of -54 is represented as 946 = 999-054+1. The sign digit can actually be regarded as part of the number under the 10's complement convention.
?B≠0	Sets the carry bit if the specified field is not zero.
?C≠0	Same as above but for the C register.
?A<C	Sets the carry bit if A is less than C. All register comparisons are done on a hex basis, even if the CPU is in decimal mode.
?A<B	Sets the carry bit if A is less than B.
?A≠0	Sets the carry bit if A is not equal to zero.
?A≠C	Sets the carry if A does not equal C.
RSHFA	Shifts the A register right by one nybble. The rightmost nybble of the section being shifted is lost and a zero is put into the leftmost nybble.
RSHFB	Same as above but for B.
RSHFC	Same as above but for C.
LSHFA	Shifts the A register left by one nybble. The leftmost nybble of the section being shifted is lost and a zero is put into the rightmost nybble. The A register is the only register that may be shifted left.

		POSTFIX							
	Instruction	ALL	S&X	M	R<	@R	MS	XS	P-Q
	A=0	00E	006	01A	00A	002	01E	016	012
	B=0	02E	026	03A	02A	022	03E	036	032
	C=0	04E	046	05A	04A	042	05E	056	052
	A<>B	06E	066	07A	06A	062	07E	076	072
	B=A	08E	086	09A	08A	082	09E	096	092
	A<>C	0AE	0A6	0BA	0AA	0A2	0BE	0B6	0B2
	C=B	0CE	0C6	0DA	0CA	0C2	0DE	0D6	0D2
	C<>B	0EE	0E6	0FA	0EA	0E2	0FE	0F6	0F2
	A=C	10E	106	11A	10A	102	11E	116	112
	A=A+B	12E	126	13A	12A	122	13E	136	132
	A=A+C	14E	146	15A	14A	142	15E	156	152
	A=A+1	16E	166	17A	16A	162	17E	176	172
	A=A-B	18E	186	19A	18A	182	19E	196	192
P	A=A-1	1AE	1A6	1BA	1AA	1A2	1BE	1B6	1B2
R	A=A-C	1CE	1C6	1DA	1CA	1C2	1DE	1D6	1D2
E	C=C+C	1EE	1E6	1FA	1EA	1E2	1FE	1F6	1F2
F	C=C+A	20E	206	21A	20A	202	21E	216	212
I	C=C+1	22E	226	23A	22A	222	23E	236	232
X	C=A-C	24E	246	25A	24A	242	25E	256	252
	C=C-1	26E	266	27A	26A	262	27E	276	272
	C=0-C	28E	286	29A	28A	282	29E	296	292
	C=-C-1	2AE	2A6	2BA	2AA	2A2	2BE	2B6	2B2
	?B≠0	2CE	2C6	2DA	2CA	2C2	2DE	2D6	2D2
	?C≠0	2EE	2E6	2FA	2EA	2E2	2FE	2F6	2F2
	?A<C	30E	306	31A	30A	302	31E	316	312
	?A<B	32E	326	33A	32A	322	33E	336	332
	?A≠0	34E	346	35A	34A	342	35E	356	352
	?A≠C	36E	366	37A	36A	362	37E	376	372
	RSHFA	38E	386	39A	38A	382	39E	396	392
	RSHFB	3AE	3A6	3BA	3AA	3A2	3BE	3B6	3B2
	RSHFC	3CE	3C6	3DA	3CA	3C2	3DE	3D6	3D2
	LSHFA	3EE	3E6	3FA	3EA	3E2	3FE	3F6	3F2

TABLE 1

All of the above instructions use the same eight postfixes. Table 1 gives the hexcode of these instructions with these eight postfixes.

There is another class of instructions whose postfixes are numeric.

Instruction	Description
READ n	Reads the contents of a RAM register into C. RAM is divided into 16 register blocks, or chips, that may be individually selected (More on how to do this later.) A READ 3 instruction would put the contents of the fourth register of that chip into the C register (counting starts from zero). Allowed values of n range from 1 to 15. There is no READ 0 instruction.
WRIT n	Same as for a READ except the contents of C are written to the specified RAM register. N ranges from 0 to 15.
RCR n	Rotate register C right by n nybbles. N can range from 1 to 13.
SETF n	Set flag n. The 14 flags are numbered from 0 to 13.
CLRF n	Same as above but will clear the flag.
?FSET n	Sets the carry bit if the specified flag is set. All 14 flags may be tested.
R= n	Sets the active pointer equal to n (0 to 13).
?R= n	Sets the carry bit if the active pointer is equal to n (0 to 13).
LD@R n	Load the value n into the digit pointed to by the active pointer. The active pointer is decremented by one to make loading of consecutive numbers easy. This can only be done in the C register.
?FI n	Sets the carry flag if the specified peripheral flag is set. Peripheral flags can not be set by the User; the peripheral must set them. They range from 0 to 13.
SELP n	Selects peripheral device n. The CPU is inactive during this

time while special instructions are being executed by the selected peripheral.

Now we present a table of the hexcodes for all of these functions.

REG.		READ	WRIT	RCR	SETF	CLRF	?FSET	R=	?R=	LD@R	?FI	SELP
0	T	XXX	028	XXX	388	384	38C	39C	394	010	3AC	024
1	Z	078	068	33C	308	304	30C	31C	314	050	32C	064
2	Y	0B8	0A8	23C	208	204	20C	21C	214	090	22C	0A4
3	X	0F8	0E8	03C	008	004	00C	01C	014	0D0	02C	0E4
4	L	138	128	07C	048	044	04C	05C	054	110	06C	124
5	M	178	168	0BC	088	084	08C	09C	094	150	0AC	164
6	N	1B8	1A8	17C	148	144	14C	15C	154	190	16C	1A4
7	O	1F8	1E8	2BC	288	284	28C	29C	294	1D0	2AC	1E4
8	P	238	228	13C	108	104	10C	11C	114	210	12C	224
9	Q	278	268	27C	248	244	24C	25C	254	250	26C	264
10	-	2B8	2A8	0FC	0C8	0C4	0CC	0DC	0D4	290	0EC	2A4
11	a	2F8	2E8	1BC	188	184	18C	19C	194	2D0	1AC	2E4
12	b	338	328	37C	348	344	34C	35C	354	310	36C	324
13	c	378	368	2FC	2C8	2C4	2CC	2DC	2D4	350	2EC	364
14	d	3B8	3A8	XXX	XXX	XXX	XXX	XXX	XXX	390	XXX	3A4
15	e	3F8	3E8	XXX	XXX	XXX	XXX	XXX	XXX	3D0	XXX	3E4

TABLE 2

Since we now have the hexcodes for the read/write instructions, we should learn how the RAM of the calculator is structured. There are basically three different parts: the status registers, main memory, and extended memory. The status registers receive the most use in MCODE programs since they contain vital information about the structure of the rest of RAM. We will now show two tables in figures 4 and 5. The first will be the memory structure of the calculator as a whole, and the second will highlight the status registers.

Address	RAM	
3FF	Extended Memory #2	RAM address limit
300		
2FF	Extended Memory #1	
200		
1FF	Top of Main Memory -----------data register 0------------ top of User programs ----------------.END.----------------- I/O Buffer area Key Assignments	
0C0		
0BF	Top of X-funct. X-Mem. Bottom of X-Funct. X-Mem	
040		
	Nonexistent Registers (VOID)	
00F	Status Registers	
000		

Figure 4

Now a little explanation on Figure 4. The addresses on the left are the absolute addresses of the register blocks starting from zero. They are given in hex. The solid lines are fixed addresses; the dashed lines are moveable address points. We will explain each section of the diagram, starting from the top of the diagram and working our way down.

Name	Description
Extended Memory #2	This is the location of the second set of extended memory module registers in the addressing scheme of the calculator RAM. The addresses of these registers are from 301 to 3EF. There is one nonexistent register (300) at the bottom of the module. The RAM at addresses 3F0 to 3FF are used by some peripherals and are NONEXISTENT for storing any data.
Extended Memory #1	Just like Extended Memory #2, except that the addresses are changed to protect the innocent. The new addresses of the RAM that exists are from 201 to 2EF.
Main Memory	1FF is the top register in the Main Memory of a 41CV, 41CX, or a 41C with a quad memory module. The bottom of Main Memory is at address 0C0. The main memory is divided into four major sections. They are: data registers, User programs, the I/O buffer, and key assignments. If this order isn't always followed your calculator will probably lock up. The data registers start at address 1FF and go down until the imaginary line between data and program memory is reached. The address of this line is kept in one of the status registers (more on this later). The next area is where the User programs that you write are placed. Then comes the .END.. After this is the free register area, or I/O buffer. These are the unused program registers. This area also includes the buffers set up by some of HP's ROMs, the most famous being the Time module. This is the area where the timer alarm information is stored. Right below these

registers are the User key assignments. They start at register 0C0 and are pushed upward every time a new assignment register is needed. These assignments do not include those for programs in User RAM. Two assignments are put in each register before a new register is used.

Extended Functions/ Extended Memory	This is the Extended memory that comes with the Extended Functions module. It is addressed from 0BF to 040. There are no voids between this and main memory, as there are with the other extended memory modules.
Void	A void occupies the RAM address space from 010 to 03F. These registers are NONEXISTENT.

Here is a diagram of the 16 status registers located at absolute addresses 000 to 00F:

Nybble 13 12 11 10 9 8 7 6 5 4 3 2 1 0

e	Shifted Key Assign. Bit Map PTEMP2 Line #
d	56 User Flags
c	ΣREG start unused Cold start Reg. 0 addr. .END.
b	Return stack Prgm pointer
a	Return stack
⊢	Unshifted Key Assign. Bit Map Scratch
Q	Scratch
P	Scratch Alpha Characters 22 to 24
O	Alpha Characters 15 to 21
N	Alpha Characters 8 to 14
M	Alpha Characters 1 to 7
L	Last X Register
X	X Register
Y	Y Register
Z	Z Register
T	T Register

Figure 5

Here is how the registers listed in Figure 5 are used:

Register	Description
e	The 36 leftmost bits of this register are used for a shifted key assignment bit map. When a shifted key is pressed while in USER mode, the calculator looks in this register to see if the key being pressed has been assigned. If the corresponding bit has been set, then the search for the key assignment starts. If the bit is not set, then the built-in (keyboard) function is executed. Nybbles 3 and 4 contain a set of status bits from the last partial key sequence (see Appendix C). The right three nybbles store the current program line number.
d	This is the register where all 56 User flags of the calculator are kept. Flag zero is on the left and flag 55 is on the far right.
c	This register holds a number of interesting goodies. Starting from the left, the first three nybbles are used as the absolute address of the first register of the Statistics Registers. The next two nybbles are not used by the calculator (they are used by some custom ROMs). Nybbles 6, 7, and 8 are the cold start constant. They are set to 169 hex. If changed from this value, the calculator will give MEMORY LOST (no accommodations for errant MCODE programming). The next three nybbles hold the absolute address for data register zero. The last three nybbles are the absolute address of the register in which the .END. resides. Don't mix this register up with the CPU C register. You will notice that this is a small c and the internal CPU register is a capital C. This is an easy way to tell them apart.
b	The four rightmost nybbles of this register hold the pointer to the address where you happen to be in program memory. The other ten nybbles are the first two and one half return addresses on

the user subroutine return stack. Each return address takes up four nybbles.

a This register is the last three and one half returns on the user subroutine return stack.

⊢ The leftmost 36 bits of this register hold the unshifted key assignment bit map. These are used in the same way as the bits for the shifted keys in register e. The rest of the register is used by the calculator as a scratch area.

Q This register is used by the calculator as a scratch register. Scratch means that there is no set purpose for that register area. It may have several different uses.

P The eight leftmost nybbles are used as a scratch area. The other six nybbles are the last three characters of the Alpha register when there are 24 characters.

M, N, O These three registers are the first 21 characters of the Alpha register. The M register is filled with the first seven characters. At the eighth character the N register starts filling with characters. It will accumulate characters until we get to the fifteenth character. Then the O register starts to accumulate characters. It takes characters until there are 21 of them. Finally, the P register takes the last three characters of the Alpha register.

Last X This is the Last X register and is accessed with the Last X function.

X This is the familiar X register where all of the numbers we see are placed.

Y The second register in the RPN stack.

Z The third register in the RPN stack.

T The top (fourth) register in the RPN stack.

If you don't quite understand this the first time, read it a few times and let the subject matter sink in. This knowledge will be very helpful for creating simple MCODE routines. You might consult a copy of "HP-41 Synthetic Programming Made Easy" for more detailed information on the status registers.

Here is a hexcode list of alpha characters displayable in the names of MCODE functions.

CHARACTER TABLE FOR MCODE FUNCTION NAMES

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
01	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↗	_
02	sp.	!	"	#	%	$	&	'	(	)	*	+	-{	-	}-	/
03	0	1	2	3	4	5	6	7	8	9	▓	,	<	=	>	?
04	⊦-	a	b	c	d	e	‾	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

sp. = blank space

TABLE 3

Let's look at how the name of a function is coded. The name of the function is put in reverse order from what would be read. An example should help. Let's do the name for a Y<>Z function.

Hexcode	Letter
09A	"Z"
03E	">"
03C	"<"
019	"Y"

start of executable code.

You will notice that the letters are in the reverse order from what we would expect. They start with the last letter and work down to the first. Notice that the last letter in the function name (Z) has hex 080 added to its hexcode (09A = 01A + 080 in hex). This signals to the processor that this is the last letter in the function name. Function names may be up to seven characters in length.

Now we have the knowledge to write a Y<>Z routine. But first, let's set up our 4K block of RAM. First set your MLDL address switches to page 8 and clear out the entire 4K block of RAM. The software you have probably has a function to do this. Consult the instruction manual of your software package on how to clear the RAM block.

We are going to use XROM 1, so the hexcode at address 8000 will be 001. We shall leave space in the FAT for the maximum number of functions (64) or 40 hex, so that our ROM name can start at address 8084 (JJ*2+4, where JJ=40 hex). If you don't want to be able to have 64 functions in your RAM, then you just decrease the JJ number to however many functions you want and use that hex number instead of 040 in the formula to find the address of the first instruction. The name of our ROM shall be SKWID 1A. (At least 8 letters must be used so that the header will show up in the CAT 2 listing of a CX. Up to 11 letters may be used in this name). The code for the ROM name is shown in the following listing:

Address	Hexcode	Letter or function	
8000	001	XROM number in hex	
8001	001	Number of functions in the FAT.	
8002	000	Address of the first executable instruction in the ROM	
8003	08C	header.	
8004	000	Indicates end of FAT.	
8005	000		
.		We now jump down to 8084 so that there will be room for	
.		more entries in the FAT. This entire area is clear.	
.			
8084	081	"A"	Recall that hex 080 is added to the hexcode for the letter A.
8085	031	"1"	
8086	020	" "	
8087	004	"D"	
8088	009	"I"	
8089	017	"W"	
808A	00B	"K"	
808B	013	"S"	
808C	3E0	RTN	This is the return function, so that if this function is synthetically entered into a program, the function just executes the return and acts as a NOP.

There is one entry in the FAT, as shown by the hex code at address 8001. This is the ROM header. When you execute CAT 2 you should see SKWID 1A in the display; if you don't, make sure that you keyed everything in correctly. We shall now write our Y<>Z routine. First we must update the FAT. The number at address 8001 must be increased by 1 and the address of the first executable instruction must be added to the FAT. Since the name is 4 letters long and the last instruction was entered at 808C, we will then add 5 to this address to come up with the address of the first executable instruction for the FAT. 808C+5 is 8091 in hex, so the FAT now looks like the following:

Address	Hexcode	Function
8000	001	XROM Number
8001	002	Number of functions in the FAT.
8002	000	Address of ROM header.
8003	08C	
8004	000	Address of Y<>Z function.
8005	091	

The rest of the FAT is zeros since there are no more functions. Now that this is done we can get down to the real business of writing the Y<>Z routine.

"Y<>Z"

Address	Hexcode	Mnemonic	Description
808D	09A	"Z"	Last letter of function name. Has hex 080 added to its hexcode.
808E	03E	">"	The rest of the name is the next 3 hexcodes.
808F	03C	"<"	
8090	019	"Y"	
8091	0B8	READ 2(Y)	Put the Y register into C. We may now manipulate the contents of the Y register or save them for later usage.
8092	10E	A=C ALL	Save Y, which is in C, in A. This will allow us to use the C register for another purpose. The choice of register A is arbitrary; any of the other 56-bit CPU registers would do just as well.
8093	078	READ 1(Z)	Put the Z register into C. The old contents of C, the Y register, are lost from C. This is why we had to save them

			elsewhere.
8094	0A8	WRIT 2(Y)	We shall now write the Z register out to the Y register. We can do this since Z is in the C register.
8095	0AE	A<>C ALL	We now bring back the original contents of the Y register to C. You can only write to RAM registers through the C register.
8096	068	WRIT 1(Z)	Put the contents of the original Y register out to the Z register.
8097	3E0	RTN	Return.

In case you're wondering, the letter behind the number in the read and write instruction is the letter of the status register that corresponds to that number. This is used since these instructions are usually used only on the status registers. The letters would not be appropriate for any other part of RAM.

THE CPU FLAGS

The 14 flags of the CPU should not be confused with the 56 User flags that are in the calculator. Flags zero to seven are contained in the ST register. This register may be zeroed. It may also be set equal to, or exchanged with, nybbles zero and one of the C register. These flags may be set, cleared, and tested. Flags eight and nine have no special meaning. Although they may be set, cleared, and tested, they are contained in a special register (XST) which we cannot access except by instructions that manipulate the individual flags. Flags 10, 11, 12, and 13 are given a special meaning by the CPU. Otherwise they share the same characteristics as flags eight and nine. The designations of these flags are given below.

Flag	If Set
10	The User code program counter (contained in status register b) points to a ROM program.
11	The RPN stack lift is enabled.
12	The User program pointer is in a private program.
13	A User code program is being run.

Now let's write a program to show the use of some of these flags. The program we will write is a "go to .END." program. This program will put you at the top of the last program in User RAM. That is the one with the .END. as its END. This is useful to avoid having to go through Catalog 1 to get to the scratch area at the end of User program memory.

The strategy of this program is to execute the permanent .END. with no pending return in the return stack, so that the program pointer will be set to the top of the last program in User RAM. This is accomplished by forming the address which points to the permanent .END., and placing it along with a zeroed pending return in the status register b. CPU flag 13 is then set to force the HP-41 to execute the .END. as a program instruction.

We now write the program to implement this procedure. It shall be called GE. Here is the annotated listing:

"GE"

Address	Hexcode	Mnemonic	Description
8098	085	"E"	Last letter of name. Hex 080 is added to the hexcode for E.
8099	007	"G"	First letter of name.
809A	378	READ 13(c)	Get the address of the .END. register. It is in nybbles 0-2 of c.
809B	05A	C=0 M	Zero the mantissa of register C. This is nybbles 3-12. This clears the 1st return so that the calculator will return control

			to the keyboard when the .END. is executed.
809C	01C	R= 3	Set the active pointer to 3 so that the required digit may be loaded into nybble 3.
809D	0D0	LD@R 3	Load a 3 into nybble 3 so that the first byte of the .END. will be executed.
809E	0C4	CLRF 10	Clear flag 10 so that the calculator is set to RAM.
809F	2C8	SETF 13	Set flag 13 so the calculator thinks a program is running, even if this routine is executed from the keyboard. This will allow us to execute the .END.
80A0	328	WRIT 12(b)	Write the address of the .END. to the b register. This will put the program pointer, which is in the last four nybbles of status register b, at the first byte of the .END.
80A1	3E0	RTN	Return.

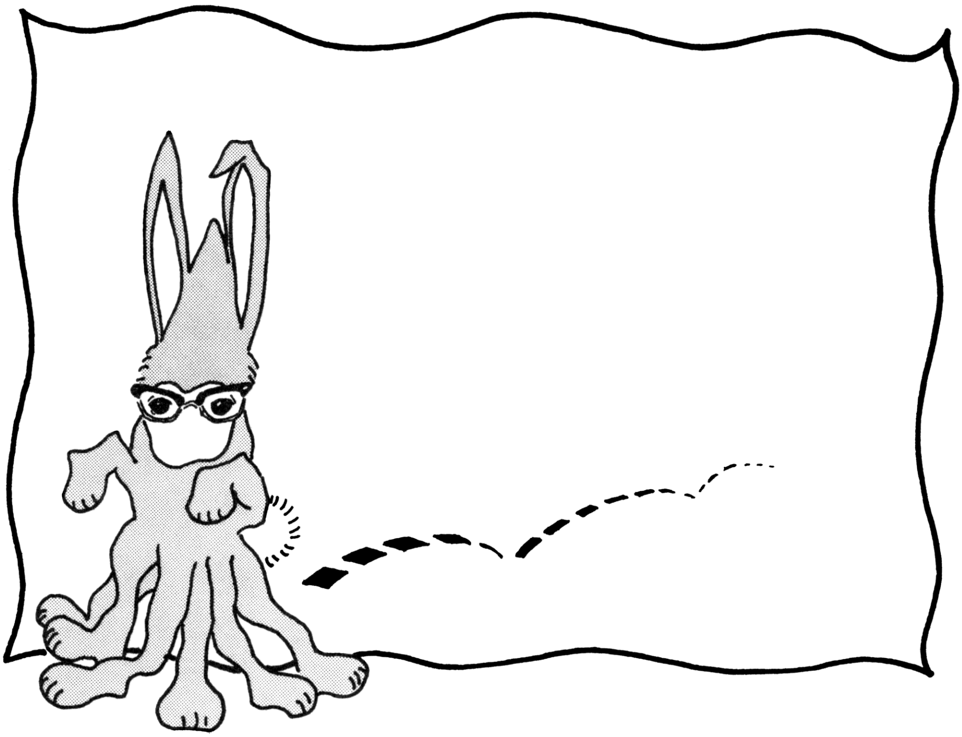
Now that the routine is written the FAT must be updated. The first executable instruction, Read 13(c), is at address 809A. So the update of the FAT would be:

Address	Hexcode	Meaning
8000	001	XROM number
8001	003	This was increased to 3. This is the number of functions in our sample ROM.
8002	000	First ROM function. SKWID 1A header.
8003	08C	
8004	000	Address of Y<>Z function.
8005	091	
8006	000	Address of GE function.
8007	09A	

That's what the FAT should now look like. These two functions we've just created may be used in programs and from the keyboard just like any of the functions that are built into the calculator. However, the MLDL box they are in must be plugged into your calculator at the time they are executed or you will get NONEXISTENT in the display.

JUMPS and JUMPING

Okay everyone, now it is time for you to put on your bunny suits (in Australia you may substitute Kangaroo suits), as we are going to introduce jumping. There are two kinds of jumps. For those of you who like to travel light, there is the Jump No Carry (JNC). Or, if you like to bring along the kitchen sink, there is the Jump on Carry (JC). The length of the jump may be up to 63 (3F in hex) steps forward (+) or 64 (40 in hex) steps backwards (-). The Jump on Carry instruction will only jump if the step preceding it sets the carry bit. Otherwise, the Jump on Carry instruction will be treated as if it were a NOP. The same is true for the Jump No Carry, except that the carry bit must not be set for the jump to occur. If the carry bit is set, the JNC instruction will be treated as a NOP. Table 4 shows the hexcodes for the JC and JNC instructions.



SKWID practicing his jumps.

DIST	JNC	JC	JNC	JC	DIST	JNC	JC	JNC	JC
ANCE	-	-	+	+	ANCE	-	-	+	+
01	3FB	3FF	00B	00F	02	3F3	3F7	013	017
03	3EB	3EF	01B	01F	04	3E3	3E7	023	027
05	3DB	3DF	02B	02F	06	3D3	3D7	033	037
07	3CB	3CF	03B	03F	08	3C3	3C7	043	047
09	3BB	3BF	04B	04F	0A	3B3	3B7	053	057
0B	3AB	3AF	05B	05F	0C	3A3	3A7	063	067
0D	39B	39F	06B	06F	0E	393	397	073	077
0F	38B	38F	07B	07F	10	383	387	083	087
11	37B	37F	08B	08F	12	373	377	093	097
13	36B	36F	09B	09F	14	363	367	0A3	0A7
15	35B	35F	0AB	0AF	16	353	357	0B3	0B7
17	34B	34F	0BB	0BF	18	343	347	0C3	0C7
19	33B	33F	0CB	0CF	1A	333	337	0D3	0D7
1B	32B	32F	0DB	0DF	1C	323	327	0E3	0E7
1D	31B	31F	0EB	0EF	1E	313	317	0F3	0F7
1F	30B	30F	0FB	0FF	20	303	307	103	107
21	2FB	2FF	10B	10F	22	2F3	2F7	113	117
23	2EB	2EF	11B	11F	24	2E3	2E7	123	127
25	2DB	2DF	12B	12F	26	2D3	2D7	133	137
27	2CB	2CF	13B	13F	28	2C3	2C7	143	147
29	2BB	2BF	14B	14F	2A	2B3	2B7	153	157
2B	2AB	2AF	15B	15F	2C	2A3	2A7	163	167
2D	29B	29F	16B	16F	2E	293	297	173	177
2F	28B	28F	17B	17F	30	283	287	183	187
31	27B	27F	18B	18F	32	273	277	193	197
33	26B	26F	19B	19F	34	263	267	1A3	1A7
35	25B	25F	1AB	1AF	36	253	257	1B3	1B7
37	24B	24F	1BB	1BF	38	243	247	1C3	1C7
39	23B	23F	1CB	1CF	3A	233	237	1D3	1D7
3B	22B	22F	1DB	1DF	3C	223	227	1E3	1E7
3D	21B	21F	1EB	1EF	3E	213	217	1F3	1F7
3F	20B	20F	17F	1FF	40	203	207	XXX	XXX

TABLE 4

To use Table 4 the jump distance must be known. This is the 2-digit hex number listed under distance. Next, you must decide whether the jump is a JNC or a JC. Then look down the appropriate column and use the ones with the + for forward jumps and the columns with the - for backward jumps.

Now we will introduce a few miscellaneous instructions. A table of their hex codes and mnemonics is given below.

ST=0	3C4	XQ>GO	020	N=C	070
CLRKEY	3C8	POWOFF	060	C=N	0B0
?KEY	3CC	SLCT P	0A0	C<>N	0F0
R=R-1	3D4	SLCT Q	0E0	LDI S&X	130
R=R+1	3DC	?P=Q	120	PUSH ADR	170
G=C	058	?LOWBAT	160	POP ADR	1B0
C=G	098	A=B=C=0	1A0	GTO KEY	230
C<>G	0D8	GOTO ADR	1E0	RAMSLCT	270
M=C	158	C=KEY	220	WRITE DATA	2F0
C=M	198	SETHEX	260	READ DATA	038
C<>M	1D8	SETDEC	2A0	FETCH S&X	330
T=ST	258	DSPOFF	2E0	C=C OR A	370
ST=T	298	DSPTOG	320	C=C AND A	3B0
ST<>T	2D8	?C RTN	360	PRPH SLCT	3F0
ST=C	358	?NC RTN	3A0	RTN	3E0
C=ST	398	C<>ST	3D8		

TABLE 5

Explanations on how most of these instructions operate follows.

Instruction	Description
ST=0	Clears the ST register (flags 0 through 7).
CLRKEY	Clears the KY register. Usually followed by ?KEY. If a key is still down then the keyboard flag will be immediately reset.

If no key is being pressed the key flag will stay clear. An example will be shown in the next program.

?KEY	Sets the carry bit if there is anything in the KY register; i.e., if a key has been pressed.
R=R-1	Decrements the active pointer by one.
R=R+1	Increments the active pointer by one.
XQ>GO	Deletes the next return on the return stack and pushes the other returns down one notch. i.e. the second becomes the first return and the third becomes the second return. A 0000 is put in for the fourth return spot.
POWOFF	This instruction places the calculator into standby mode or deep sleep depending on whether the display is on or off. If the display is on then we go into standby mode, in which the calculator is on and just sitting there doing nothing. If the display is off then the result is the same as if we turn the calculator off using the ON button. This instruction must be followed by the 000 instruction. The PC register is reset to 0000 and the CPU stops there waiting for a key to be pressed.
SLCT P	Selects register P as the active pointer. Does not change the value of either of the pointer registers.
SLCT Q	As above but selects the Q register.
?P=Q	Sets the carry bit if the values of the P and Q registers are the same.
?LOWBAT	Sets the carry bit if the battery voltage is low.
A=B=C=0	Sets the A, B, and C registers equal to zero.

GOTO ADR	Replaces the program counter (PC) register with nybbles three through six of the C register.
C=KEY	Places the contents of the KY register into nybbles 3 and 4 of the C register.
SETHEX	Puts the CPU into hexadecimal mode. All calculations are now done using the digits 0 to F.
SETDEC	Puts the CPU into decimal mode. All calculations are done using the digits 0 to 9. However, register exchanges may still be done with hex numbers while in this mode.
DSPOFF	Turns off the display.
DSPTOG	Toggles the display between on and off. This switches it to which ever state it was not in before the instruction was executed.
?C RTN	Return if the carry bit was set by the preceding instruction.
?NC RTN	Return if the carry bit was not set by the preceding instruction.
LDI S&X	This instruction places the hexcode of the next ROM word into the S&X field of the C register.
PUSH ADR	Places nybbles 3 - 6 of the C register onto the subroutine stack. All pending returns are moved up one. The C register is not changed.
POP ADR	Takes the 1st return from the subroutine stack and places it at digits 3 - 6 of the C register. All of the remaining returns are moved down one and 0000 is placed into the fourth return

position on the stack.

GTO KEY	Places the contents of the KY register into the last two nybbles of the program counter (PC) register.
FETCH S&X	Uses the address in nybbles 3 - 6 of the C register to copy the ROM word at that location into the S&X field of the C register.
C=C OR A	Performs a logical OR of the A and C registers and puts the answer in C. Looks at each bit position in both registers and sets the corresponding bit in the C register result if it is set in either the original C register or the A register.
C=C AND A	Same as above except that both matching bits in the A and C registers must be set in order for that bit to be set in the C register. Neither of these functions disturb the A register.
PRPH SLCT	Uses digits 1 and 0 of register C as the number of the peripheral to select.

As an example, the program below is a counting program. It will count by ones (in MCODE of course) from the moment the program is executed until a key on the keyboard is pressed. We shall input the program to show the use of some of the functions that are described above, and also to show how the JC and JNC instructions work.

"COUNT"

Address	Hexcode	Mnemonic	Description
80A2	094	"T"	Last letter of the name of the routine COUNT. Hex 080 is added to the hex code for T.
80A3	00E	"N"	The next four words are the rest of the name.

80A4	015	"U"	
80A5	00F	"O	
80A6	003	"C"	
80A7	2A0	SETDEC	Set the CPU so that counting will be in decimal mode.
80A8	04E	C=0 ALL	Zero C so that counting will start at zero.
80A9	23A	C=C+1 M	Add one to the Mantissa of C. This is the start of the counting loop.
80AA	3CC	?KEY	If a key is pressed the carry bit will be
80AB	3F3	JNC -02	set, and the JNC instruction will act as a NOP. If no key is pressed, the carry will not be set and we jump back to the beginning of the loop.
80AC	130	LDI S&X	The largest exponent a 10 digit number may
80AD	009	HEX: 009	may have is nine. This is loaded into the exponent field. The number that we counted up to is right justified in the mantissa of C. If this number is not 10 digits long, we will decrement the exponent.
80AE	35C	R= 12	Set the active pointer to the leftmost
80AF	11A	A=C M	nybble of the mantissa. This allows us
80B0	342	?A≠0 @R	to check if this digit is zero. If it is,
80B1	027	JC +04	we shift the whole mantissa left one and
80B2	266	C=C-1 S&X	subtract one from the exponent. If it is
80B3	3FA	LSHFA M	not zero, the carry will be set and we
80B4	3E3	JNC -04	jump out (JC) to the rest of the routine. The reason we check for leading zeros, that is, the zeros in the leftmost nybbles of the mantissa, is because the number we counted up to is right justified in the mantissa of C. We shift this left to remove these leading zeros, if necessary. If there are leading zeros, we

			loop around to check for more leading zeros again.
80B5	3C8	CLRKEY	Loop to check if the key that stopped the counting has been released. If it is still down, the carry will be set during the ?KEY step. If it is not down, the ?KEY will not set the carry, and the JC instruction will not be executed.
80B6	3CC	?KEY	
80B7	3F7	JC -02	
80B8	0BA	A<>C M	Get back the mantissa and write out the number to X. The exponent is in C so we only need to retrieve the mantissa from A.
80B9	0E8	WRIT 3(X)	
80BA	3E0	RTN	Return.

To update the FAT you should increase the number at address 8001 from 003 to 004. The rest of the FAT update looks like the following:

Address	Hexcode	Description
8001	004	Number of functions in our sample ROM.
8008	000	First word of the address of the COUNT function.
8009	0A7	Second word of the FAT Address for COUNT.

Running this program on one calculator for 60 seconds produced an answer of 129,686. Compare this with 1,056 for a User code version of the same program and the MCODE version is about 120 times as fast. This program really shows you what kind of speed advantage can be enjoyed using MCODE.

We will now write another program, using jumps, that introduces a few more instructions to your vocabulary. We shall introduce the RAMSLCT, WRITE DATA, and READ DATA instructions.

The RAMSLCT instruction uses the S&X field of register C for the number of the RAM register to be selected. The number in the S&X field of C is interpreted as a hex number, not a decimal number. First, some explanation on how the User RAM is set up from the CPU's point of view. RAM is divided into 16 register blocks, or chips, as they are known. The addresses of chip xy are xy0 to xyF; xy may be from 00 to 3F (0 to 63 in decimal). Each of these chips may only be accessed if a register in that chip has been selected using the RAMSLCT instruction. The RAMSLCT instruction selects both a chip and a register within that chip. If S&X of C is xyz, RAMSLCT selects chip xy and register xyz. The 15 read/write instructions introduced earlier will only operate on a register within the selected chip. In addition, the read and write instructions change the RAMSLCT pointer to the designated register within the selected chip. Thus if chip xy is selected, READ n or WRIT n will address register xyn and change the RAMSLCT pointer to register xyn. Here's an example to clarify this mess.

Hexcode	Mnemonic	Description
130	LDI S&X	Load hex 0C0 into C register S&X field. The RAMSLCT
0C0	HEX: 0C0	instruction will then select this register (number
270	RAMSLCT	192). This is register zero of the selected chip (the last digit in the hex number is the register number in the chip that is selected).
0F8	READ 3(X)	Reads the fourth register in this chip (decimal 195) into the C register. The selected RAM register is now 0C3. This would be the same if we used a write instead of a read.

Sometimes we don't know exactly where in a RAM chip we will be, and we can't have the RAMSLCT pointer being moved on us. How do we read or write to the selected RAM register without moving the RAMSLCT pointer? We use the READ DATA and WRITE DATA instructions. These instructions read and write data between the C register and RAM without modifying the RAMSLCT pointer.

The READ DATA instruction is sometimes listed as READ 0 by some disassemblers. THIS IS INCORRECT! There is no such thing as a READ 0 instruction. This was a mistake made by some of the early pioneers in the MCODE field, working without factory documentation that appeared later.

Disassemblers typically place a letter after the register number of each read/write instruction. These letters correspond to the status registers, and only apply if chip 0 is selected.

Next we will write a combination Alpha-to-Memory and Memory-to-Alpha routine. These programs will take the four registers that comprise the Alpha register and put them into User data registers. This data can not be safely recalled from the data registers using the RCL function.

These routines are good for storing the contents of Alpha and then retrieving the Alpha register unaltered. The routine will use four data registers starting with data register 0. The next 3 data registers will also be used. Fill the Alpha register with the desired characters. You now can execute the AM (Alpha to Memory) function. Next, clear the Alpha register. Then execute the MA (Memory to Alpha) function. The old Alpha data reappears. That was pretty fast wasn't it? One other note: this routine assumes that you have a HP-41CX, HP-41CV, or a HP-41C with a quad memory module. Now here's the routine:

"AM & MA"

Address	Hexcode	Mnemonic	Description
80BB	081	"A"	Second letter of the Memory to Alpha name.
80BC	00D	"M"	First letter of the name.
80BD	248	SETF 9	We set this flag to tell which routine we are executing. If it is set we are using MA. If it is clear we are using AM.
80BE	023	JNC +04	Jump to READ 3(X) instruction. We do this so that the AM name is not executed as MCODE instructions.
80BF	08D	"M"	Name for Alpha to Memory routine.
80C0	001	"A"	
80C1	244	CLRF 9	Clearing flag nine means we are in AM routine (see address 80BD).

80C2	378	READ 13(c)	Get the absolute address of data register zero. It is in nybbles 3, 4, and 5 of status register c.
80C3	03C	RCR 3	Rotate the address of data register zero into the S&X field of the C register.
80C4	106	A=C S&X	Save the address of data register zero in A. Load the highest absolute address that can be used without overflowing main memory.
80C5	130	LDI S&X	
80C6	1FD	HEX: 1FD	
80C7	306	?A<C S&X	If A is less than C, then the registers wanted will not overflow into extended memory. The carry bit will be set and we will jump out. Otherwise we will zero the C register and write it out to X, so X will be zero if we error. We then return to the calling program without finishing the routine.
80C8	027	JC +04	
80C9	04E	C=0 ALL	
80CA	0E8	WRIT 3(X)	
80CB	3E0	RTN	
80CC	39C	R= 0	Set active pointer to zero for use as a counter.
80CD	130	LDI S&X	Load the absolute address of the start of the Alpha register. This is the M register. As you remember, the other three registers that comprise the Alpha register are numbered 6, 7, and 8 (for N, O, and P).
80CE	005	HEX: 005	
80CF	24C	?FSET 9	Check which of the two routines is being run. Right now the address pointer to the the Alpha registers is in C and the data register pointer is in A. If we are running the MA routine then we want to reverse this and not jump over the A<>C S&X instruction. The register pointer in C after this will be the one from which the data is transferred.
80D0	013	JNC +02	
80D1	0A6	A<>C S&X	
80D2	270	RAMSLCT	Select the RAM register of the pointer in

			C. This is the beginning of the loop.
80D3	226	C=C+1 S&X	Increment the register pointer of the RAM register from which the data is being transferred.
80D4	0E6	C<>B S&X	Save the RAM register pointer in B.
80D5	038	READ DATA	Read the selected RAM register into C.
80D6	0AE	A<>C ALL	Exchange the data with the other RAM pointer.
80D7	270	RAMSLCT	Select the other set of RAM registers.
80D8	0AE	A<>C ALL	Get the data back and put the second RAM pointer back into A.
80D9	2F0	WRITE DATA	Write out the data to the selected register.
80DA	166	A=A+1 S&X	Increment the second RAM pointer.
80DB	3DC	R=R+1	Increment the active pointer.
80DC	0E6	C<>B S&X	Put the first RAM pointer back into C.
80DD	054	?R= 4	Have we been through the loop 4 times?
80DE	360	?C RTN	Remember there are 4 registers that make
80DF	39B	JNC -0D	up the Alpha register. If so, the carry will be set and we return. Otherwise, jump back to the beginning of the loop.

Well, that's the end of the routine. Hope you liked it and learned how the RAM registers may be selected and written to. For these routines there are 2 entries in the FAT. One for the MA routine and one for the AM routine. It does not matter that the two routines are combined. The names must still have an address in the FAT in order to show up in Catalog 2. The entries into the FAT are shown below. The number at address 8001 should be increased by 2 from 004 to 006 since we are adding two routines to the FAT.

Address	Hexcode	Description
8001	006	This is the number of functions in our sample ROM. Notice it has been increased by 2 since the last time we modified the FAT since we have two new routines.

800A	000	First word of the address of the MA routine.
800B	0BD	Second word of the address of the MA routine.
800C	000	First word of the address of the AM routine.
800D	0C1	Second word of the address of the AM routine.

Before we demonstrate the use of any more instructions, we need to introduce a new subject area which will make our programming easier and far more versatile.

ABSOLUTE EXECUTEs AND GOTOs

There are 4 different types of instructions in this group. If the last two bits of the first word of an instruction are 01 then they fall into this category. These instructions all use two words to form one instruction. They differ based on how the last two bits in the second word are set. The 4 types of instructions are:

Instruction Mnemonic	How it Works
?NC XQ ----	This is the No Carry EXecute. This instruction will only jump to the specified address if the carry bit is not set when the instruction is executed. If the carry bit is set the instruction is treated as a NOP.
?C XQ ----	This is the EXecute on Carry. This instruction is the same as the one above except the carry must be set for it to jump.
?NC GO ----	This is the No Carry GOTO instruction. It will go to the specified address only if the carry bit is not set when the instruction is executed. If the carry is set it is treated as a NOP instruction.
?C GO ----	Here is the GOTO on Carry. This is the opposite of the above instruction. If the carry bit is set the instruction will go to the specified address.

Don't forget, the Carry bit is cleared by any instruction. To use a jump on Carry, the Carry bit must be set by the instruction immediately preceding the jump instruction.

The dash after each instruction is the address you want to GOTO/EXECUTE, when the instruction is displayed as a mnemonic.

An EXECUTE is a subroutine call: it loads a return address onto the subroutine return stack. A GOTO is merely an exit to a specified address.

If the first word that an EXECUTE branches to is the NOP 000, then that instruction produces an immediate return. This feature of the EXECUTE instructions allows calls to possibly nonexistent ROMs.

Now we will show you how these 4 instructions are put into hexcodes. The way the CPU tells that the instruction is either a GOTO or an EXECUTE is by the last two bits in the first word. If these are set to 01 the next word is interpreted as the second half of a GOTO or EXECUTE instruction. The way it differentiates between these is by the last two bits of the second word. A table for the interpretation of these two bits is given below.

Instruction	Value of bit from 2nd word	1	0
?NC XQ		0	0
?C XQ		0	1
?NC GO		1	0
?C GO		1	1

Note that the 0 bit corresponds to the setting of the Carry flag (1 for Carry set, 0 for Carry clear).

The numbers are the values of the last two bits of the second word of the instruction, the two least significant bits. Now we will show you how the rest of the instruction is formatted.

Instruction	Bit number	9 8 7 6	5 4 3 2	1 0
Value of the four bits		3	2	
?NC XQ 8432	first word	0 0 1 1	0 0 1 0	0 1
Value of the four bits		8	4	
	second word	1 0 0 0	0 1 0 0	0 0

You will notice that after taking away the 0 and 1 bits we are left with the digits from the address that we want in the remaining 4 nybbles. The first hex digit of the address is in the 4 most significant bits (6 to 9) of the second word. The second digit of the address is in the next 4 bits (2 to 5) of the second word. Then we jump up to bits 6 to 9 of the first word for the third digit of the address. That leaves bits 2 through 5 of the first word for the last digit in the address.

Again, notice that bit 1 is zero and bit 0 is equal to one in the first word. This signals to the CPU that the instruction is a GOTO/EXECUTE instruction. Since both bits 0 and 1 are zero in the second word, the CPU knows that it is a ?NC XQ instruction. For a ?C XQ to the same location only bit zero of the second word would have to be changed, since the address information is coded in the same way for all 4 types of instructions. In order to make the input of these instructions into your MLDL box easier, it is recommended that you use an assembler to figure out the details of the hexcode. This way, all you have to do is input the mnemonic, such as ?C GO 14E2. The assembler program does the rest.

These instructions are usually not used to EXECUTE or GOTO another part of a routine that you are writing in MCODE. This is because if we put a ?NC XQ 8432 in our example ROM page and then move the page to another port, the code we wish to execute will no longer be at address 8432. However, the EXECUTE may still end up going there, sometimes with fatal results. There is another kind of EXECUTE and GOTO for use within a 4K page, which will be discussed later.

The absolute EXECUTEs and GOTOs are used for accessing code in the mainframe ROMs. These are the 12K of ROM that contain the code for controlling the User portion of the calculator. They contain many useful routines that may be used as subroutines in our programs.

If you remember, the MA and AM routines that we programmed earlier could only save data in registers 0 to 3. Now we shall rewrite them to use some entry points in the mainframe ROMs so that you can specify the first data register to be used by entering its number in the X register.

We shall use two entry points, one to convert the number in X to a hexadecimal number in the S&X field of C, and another entry point for the NONEXISTENT error routine in case the registers that would be used are not part of the calculator's RAM memory. We still assume that you have a 41CX, 41CV, or 41C. So let's rewrite the routine.

"AM & MA" revised

Address	Hexcode	Mnemonic	Description
80BB	081	"A"	Name for the MA routine. Notice that the address of the first executable instruction for each routine has not changed. The first seven instructions are exactly the same.
80BC	00D	"M"	
80BD	248	SETF 9	
80BE	023	JNC +04	
80BF	08D	"M"	
80C0	001	"A"	
80C1	244	CLRF 9	
80C2	0F8	READ 3(X)	
80C3	38D	?NC XQ	This execute instruction accesses a subroutine that takes the number in C and converts the number to its hexadecimal equivalent in the S&X field of C. For example, the conversion for 999 decimal
80C4	008	02E3 [BCDBIN]	

			would be 3E7. This mainframe entry point is called BCDBIN (BCD to binary) in HP's annotated VASM listings for the operating system of the 41.
80C5	106	A=C S&X	We save the result in A and get the absolute address of data register zero from the c register and rotate it into the S&X field of C. We then add these two to get the absolute address of the first data register to which we will write.
80C6	378	READ 13	
80C7	03C	RCR 3	
80C8	146	A=A+C S&X	
80C9	130	LDI S&X	Load the largest absolute address that can be used without overflowing main memory when we store data in the following 3 registers.
80CA	1FD	HEX: 1FD	
80CB	306	?A<C S&X	If A is less than C, the registers used by the routine will not be NONEXISTENT, so the carry will be set and the ?NC GO instruction will be ignored. If A is greater than or equal to C, we go to the entry point at 02E0, called ERRNE (error - NONEXISTENT), which is the NONEXISTENT error message routine.
80CC	381	?NC GO	
80CD	00A	02E0 [ERRNE]	

The instructions from 80CC to 80DF have been moved down to 80CE through 80DF. This routine is much more versatile. In order to use it you just place the number of the data register where you want to start saving data into X, and place the Alpha characters to be saved into Alpha. Then just execute the revised routine, and bingo, it's all done.

THE NORMAL FUNCTION RETURN

Before a function is executed, a special return address called the Normal Function Return is loaded into the CPU subroutine return stack; this is address 00F0. The code at this address does the necessary processing that is required after any function is executed. If you use all four levels of the subroutine return stack, this address will have been pushed off and you will have to end your program by exiting to address 00F0. Otherwise, the pending return will be 0000 if you try to finish with a RTN, and you will end up at that address of the mainframe. This sends the 41 directly into standby mode whether you should be there or not, and fails to do the necessary processing that follows function execution. When this happens, the calculator appears to have crashed, because the display freezes instead of reverting to a default display such as the X-register. However, unlike an ordinary crash, the calculator will respond to keystrokes, and you can then conclude that your routine has not exited through the Normal Function Return. You should place an ?NC GO 00F0 as the ending instruction of your program instead of a return. If the calculator does not respond to keystrokes, then you are in an infinite loop and something else is wrong with your program.

Another interesting routine that we have provided for your programming pleasure is an Invert Flag routine. This routine takes the number in X to be the flag that you wish to invert. Invert means that if the flag was set the routine will clear it; and if the flag was clear, the routine will set it. The routine may be used with all 56 User flags (0 to 55).

This routine utilizes three mainframe ROM entry points. These are: BCDBIN at address 02E3 (converts a decimal number into hexidecimal in S&X of C), the clear flag routine at address 164D, and the set flag routine at address 164A. This program also introduces some other interesting tricks. It uses the C=C+C ALL instruction to shift the C register left by only one bit at a time. The other instruction that will be introduced is the C=C AND A instruction. Its use will be explained with the routine.

"IF"

Address	Hexcode	Mnemonic	Description
80E2	086	"F"	Name of routine.
80E3	009	"I"	

80E4	0F8	READ 3(X)	Get the flag number from the X register
80E5	38D	?NC XQ	and convert it to binary in the S&X field
80E6	008	02E3	of C. This is the hex representation of
		[BCDBIN]	the decimal number that is in X (46
			decimal would be 02E in hex).
80E7	10E	A=C ALL	Save the answer in A. Load S&X of C with
80E8	130	LDI S&X	the largest value the number may have (55
80E9	037	HEX: 037	decimal) because there are only flags 0
80EA	0AE	A<>C ALL	to 55 and for numbers over 55 the flag is
80EB	1C6	A=A-C S&X	NONEXISTENT. Exchange the two numbers
80EC	381	?C GO	and then subtract them. If the carry is
80ED	00B	02E0	set, there was an underflow during the
			subtraction and the number in X was
			greater than 55. This causes us to go to
			the NONEXISTENT error routine at 02E0 in
			the mainframe ROMs. Otherwise, we
			continue on with the routine.
80EE	04E	C=0 ALL	We now have 55 minus the original flag
80EF	226	C=C+1 S&X	number in S&X of A. We zero C and then
80F0	1A6	A=A-1 S&X	add one to it. This sets only the least
80F1	01F	JC +03	significant bit of register C. Then one
80F2	1EE	C=C+C ALL	is subtracted from S&X of A. This serves
80F3	3EB	JNC -03	as a counter for the number of times we
			must go through the bit shifting loop. If
			we have an underflow (0 minus 1) then the
			carry will be set and we jump out of the
			loop. The next step shifts the bit in C
			one to the left and the following step
			jumps back to the start of the loop.
80F4	0EE	C<>B ALL	In order to use the set flag and clear
80F5	3B8	READ 14(d)	flag entry points you need a mask with the
80F6	10E	A=C ALL	bit set corresponding to the flag that you
80F7	0CE	C=B ALL	want to manipulate. This mask must be put
80F8	3B0	C=C AND A	into B. Register A must contain the flag
80F9	2EE	?C≠0 ALL	register, which is register d of the RAM

80FA	135	?C GO
80FB	05B	164D [XCF]
80FC	129	?NC GO
80FD	05A	164A [XSF]

status registers. These conditions are met and then the mask is put back into C. We next AND it with the flag register which is now in A. If the bit in the flag register that corresponds to the bit set in the mask is also set, then this bit will be set. All other bits in the mask are 0 so the answer when these are AND'ed will always be 0. If the corresponding bit is not set in A, then C will be zeroed. We then check whether or not C is 0. If not, the carry will be set and we want to go to entry point XCF (execute CF), the clear flag routine (164D). If C is 0 the flag was clear and we want to set it; so, we go to XSF (execute SF), the set flag routine (164A). The routine returns through one of the mainframe flag routines.

Remember to update the FAT. We now have seven functions. The address of the first executable instruction in this routine is at 80E4.

The next routine has a pair of functions HP should have built as standard functions into the calculator. These are the FS?S and FC?S functions. These functions are analogous to the FS?C and FC?C functions built into the calculator. They leave the specified flag set and check to see whether the test is true or not. If it is not true, one step is skipped in a running program. A YES or NO will appear in the display if they are executed from the keyboard.

We have another one of those handy entry points to help in these functions. The only difference is that our routines take the flag number from X, while the HP routines prompt for the flag number. One advantage of our routines is that they work on all 56 flags. HP's only work for flags 0 to 29. These programs use the FS? and FC? routines in the mainframe ROMs. They test the flag and automatically skip a step in a program if the test is false. They share a lot of code with the IF routine as well. We will leave the combining of these two routines as an exercise for you to do. The combination takes a total of 60 words. See if you can match this. For now, here are the FS?S and FC?S routines.

"FS?S & FC?S"

Address	Hexcode	Mnemonic	Description
80FE	093	"S"	Name for the FS?S routine.
80FF	03F	"?"	
8100	013	"S"	
8101	006	"F"	
8102	244	CLRF 9	This flag is used to tell which routine is being executed. Clear is the FS?S routine and with flag nine set the FC?S routine as being executed. This flag is used later in the routine to figure out which routine was executed.
8103	033	JNC +06	Jump over the FC?S name to the READ 3(X) instruction.
8104	093	"S"	Name for the FC?S routine.
8105	03F	"?"	
8106	003	"C"	
8107	006	"F"	
8108	248	SETF 9	See the description for the CLRF 9 instruction.
8109	0F8	READ 3(X)	Get the flag number from the X register.
810A	38D	?NC XQ	Convert the flag number to hex in S&X of
810B	008	02E3 [BCDBIN]	C.
810C	106	A=C S&X	Save this in A. Then load the largest

810D	130	LDI S&X	possible flag number (55) into S&X of C.
810E	037	HEX: 037	Exchange the number of the flag to be
810F	0AE	A<>C ALL	tested and the highest possible flag num-
8110	1C6	A=A-C S&X	ber. These are then subtracted. If the
8111	381	?C GO	carry is set, we will have an underflow
8112	00B	02E0	since the flag number to be tested is
		[ERRNE]	greater than 55 (037 hex) and we will go
8113	04E	C=0 ALL	to the NONEXISTENT routine. Otherwise,
8114	226	C=C+1 S&X	we have the number of times we wish to go
			through the bit shifting loop in the S&X
			field of A. We now have a counter for the
			number of times we wish to move the bit in
			the mask over from the rightmost position.
			We first zero C and set the rightmost bit
			using the C=C+1 instruction.
8115	1A6	A=A-1 S&X	This is the mask making loop. We want to
8116	01F	JC +03	set the bit that corresponds to the number
8117	1EE	C=C+C ALL	in X. If A is zero (55 minus 55), then an
8118	3EB	JNC -03	underflow will occur and the carry will be
			set and we jump out of the loop. If
			there is no underflow, we shift the bit
			left by one and jump back to the start of
			the loop to try again.
8119	10E	A=C ALL	Save the mask in A. Then put it into B
811A	0EE	C<>B ALL	for later use by the mainframe routines.
811B	3B8	READ 14(d)	Get the flag register. We save this in N
811C	070	N=C	for later use. The flag register and mask
811D	370	C=C OR A	are ORed so that the mask bit will be set
811E	3A8	WRIT 14(d)	in the flag register. This is then writ-
			ten out to the flag register.
811F	0B0	C=N	Get back the original flag register con-
8120	10E	A=C ALL	tents and place them into A for use with
8121	24C	?FSET 9	the mainframe routines. Check to see
8122	169	?C GO	which routine is being executed. These
8123	047	115A	routines require that the flag register is

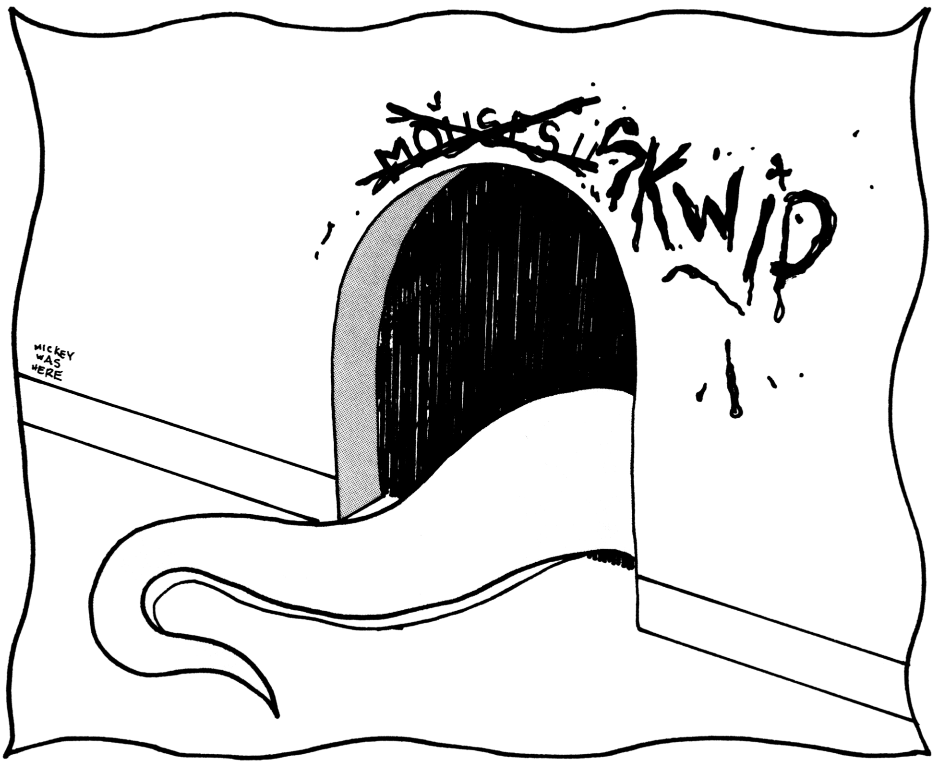
		[FC?]	in A and that the mask is in B upon entry
8124	115	?NC GO	to them. If the carry is set we GOTO the
8125	05A	1645	FC? routine (115A). Otherwise we GOTO the
		[XFS?]	XFS? (eXecute FS?) entry point (1645). The programs return through these mainframe routines.

Don't forget to update the FAT. These two programs are combined into one. But we still need two entries in the FAT to be able to access both of the routines. Here is what the FAT should look like.

Address	Hexcode	Description
8010	001	Since the third digit from the right of the address of the FS?S routine is not zero we have to put the number of this digit into the rightmost digit of the first word of the two word FAT entry (see page 20). The starting address for this routine is 8102.
8011	002	The last two digits of this word are the last two digits of the address of the FS?S routine. This is no different than the entries we did before.
8012	001	The purpose of this word is the same as the one at address 8010 except that the second word of the two word FAT set will be different. It will be the starting address of the FC?S routine.
8013	008	These are the two rightmost digits of the first executable instruction in the FC?S routine.

Remember to update the word at address 8001. This tells the calculator the number of entries in the FAT. It is now 009.

The next routine uses an entry point called GENNUM (generate number) in the mainframe ROMs to decode a 3 digit hex number into decimal. This entry point is at address 05E8 in the mainframe. This routine takes a binary number in the S&X field of the A register and converts it to a decimal number. The answer ends up in the mantissa of the A register. However, things are never simple and this routine is no exception. It does not place an exponent on the decimal number, and in addition leaves garbage in the rest of A. Since the mainframe routine assumes that the display is selected, a nonexistent chip must be selected in order to keep the mainframe routine from writing to RAM registers. The number of digits output by the routine can be from 1 to 4. In order to guarantee a fixed number of output digits, a number from 1 to 4 is placed in the mantissa sign of A as an input to the routine. We shall use the number 4 to provide a 4-digit result (possibly with leading zeros). Basically, that is all there is to the routine; it is called BIN-BCD (binary to binary coded decimal).



"BIN-BCD"

Address	Hexcode	Mnemonic	Description
8126	084	"D"	Last letter of the routine name with hex 080 added to its hexcode.
8127	003	"C"	The next six words are the rest of the routine name.
8128	002	"B"	
8129	02D	"-"	
812A	00E	"N"	
812B	009	"I"	
812C	002	"B"	
812D	0F8	READ 3(X)	Get the number to be decoded from the X register.
812E	106	A=C S&X	Put the number into the A register.
812F	130	LDI S&X	Load the address of a nonexistent RAM chip
8130	010	HEX: 010	into the S&X field and RAMSLCT it.
8131	270	RAMSLCT	
8132	2DC	R= 13	Set the pointer to the mantissa sign so
8133	110	LD@R 4	that a 4 may be loaded. This number will
8134	11E	A=C MS	be put into the A register. The mainframe routine uses this number to set the number of output digits. If the number output is not 4 digits, leading zeros are inserted.
8135	3A1	?NC XQ	Execute the mainframe routine to do most
8136	014	05E8 [GENNUM]	of the dirty work. The result is in the mantissa of A.
8137	0AE	A<>C ALL	Put the answer into C. Set the pointer to
8138	11C	R= 8	8. The least significant digit of the
8139	04A	C=0 R<	mantissa of the answer will be in nybble 9. Zero register C from digit 8, the digit pointed to by the pointer, to digit 0.
813A	270	RAMSLCT	Select the RAM status registers, chip 0.

			The S&X field of C was zeroed by the previous instruction.
813B 813C 813D	39C 0D0 010	R= 0 LD@R 3 LD@R 0	Set the pointer equal to 0 so that we may load in the exponent. Remember the main-frame routine does not provide this. The largest exponent possible is 3. Four decimal digits are i.jkl * 10^3. The mantissa sign is then zeroed because garbage is left there by the routine. Remember that LD@R decrements the pointer by one. After loading the value 3 in nybble zero, we wrapped back around to nybble 13, the mantissa sign digit.
813E 813F 8140 8141 8142 8143	0AE 342 027 3FA 1A6 3E3	A<>C ALL ?A≠0 @R JC +04 LSHFA M A=A-1 S&X JNC -04	Put everything back into A. Check to see if there are any leading zeros in the mantissa of A. If there are no leading zeros, jump out (the carry will be set). Otherwise, we can shift out any leading zeros in the mantissa (nybble 12 will be zero) using the LSHFA M instruction. We decrement the exponent by one since there is one less digit in the mantissa than before. We loop around again to check for more leading zeros in the mantissa.
8144 8145 8146 8147 8148 8149	0AE 2FA 017 04E 0E8 3E0	A<>C ALL ?C≠0 M JC +02 C=0 ALL WRIT 3(X) RTN	Put the final answer into the C register. Check to see if the mantissa is zero. If it is the exponent will be FFF. If not zero, the carry will be set and we jump to the WRIT 3(X) instruction and return. If the mantissa is zero, then zero the whole C register, write it out to X, and return.

Don't forget to update the FAT. We now have ten functions. The hexcode at address 8001 would be 00A (ten in hex), not 010 (which is sixteen).

The way you may use the above routine, in a program or from the keyboard, is to put the number you want to decode into X. The last three nybbles of whatever is in X will be decoded and placed into X. For example, if the number in X is 987234.92 the BIN-BCD routine will give an answer of 5. This is because the exponent of this number is 5 and the exponent sign is zero. The S&X field of X upon entry would be 005 in hex.

However, the real use of this routine is as a subroutine to decode binary numbers that we get as results in MCODE routines that we write. Our next routine is a Free Register Finder routine. It finds the number of empty registers below the permanent .END.. This result is the same number you see after you key GTO .. in program mode. The routine is very short (only 3 words long) and shows the power of MCODE. In particular, it illustrates how useful the BIN-BCD routine can be.

"F?"

Address	Hexcode	Mnemonic	Description
814A	0BF	"?"	Name
814B	006	"F"	
814C	285	?NC XQ	This routine in the mainframe calculates
814D	014	05A1 [MEMLFT]	the number of free registers left (MEMory LeFT). No inputs are needed. The answer is given in binary form in the S&X field of C.
814E	303	JNC -20	This jump goes back to the A=C S&X instruction at address 812E of the BIN-BCD routine. This routine will decode the contents of the S&X field of C and put the answer into the X register.

Many of the outputs from routines in the mainframe ROMs are in binary format. We need this routine, or one like it, to decode the binary form to decimal so we can output it to the X register for use in our programs. Don't forget to update the FAT. We now have 11 functions in our sample ROM.

Now, what about taking decimal numbers from the X register and converting them to binary? This can be done in 2 ways. The easiest way, as we have seen is to execute the routine in the mainframe ROMs at address 02E3. But what if we want to code a number greater than 999 into the S&X field of X? After all, 3 hex digits may be a number as large as 4,095 (FFF). To do so we must write our own routine to decode numbers greater than 999. This routine will decode numbers from 0 to 9,999. For numbers greater than 4,095 the answer will be the remainder of the original number divided by 4,096. This conversion routine is called BCD-BIN.

"BCD-BIN"

Address	Hexcode	Mnemonic	Description
814F	08E	"N"	Last letter of the name. Notice that hex 080 is added to the hexcode for "N".
8150	009	"I"	Now come the next six letters.
8151	002	"B"	
8152	02D	"-"	
8153	004	"D"	
8154	003	"C"	
8155	002	"B"	
8156	0F8	READ 3(X)	Get the decimal number to be converted and put it into C.
8157	10E	A=C ALL	Save the integer number in A for later
8158	1BE	A=A-1 MS	use. Check for alpha data. If the number
8159	1BE	A=A-1 MS	is alpha data, then the mantissa sign will
815A	389	?C GO	be 1. By subtracting 1 twice, we first
815B	053	14E2	hit zero then create an underflow (sub-

		[ERRAD]	tract 1 from 0) which will set the carry if the mantissa sign is 1. The GOTO is to the ALPHA DATA error message (ERRAD = ERRor - Alpha Data) only if the carry is set.
815C	130	LDI S&X	Load the exponent that the number cannot be greater than or equal to (exponent for 10,000). Then check to see if the exponent of the decimal number is less than this number. If it is less, the carry will be set and the next instruction will not be executed. However, if the carry is not set, the instruction <u>will</u> be executed. This instruction is a GOTO to the OUT OF RANGE error message (ERROF = ERRor - OverFlow).
815D	004	HEX: 004	
815E	306	?A<C S&X	
815F	289	?NC GO	
8160	002	00A2	
		[ERROF]	
8161	266	C=C-1 S&X	Now we check if the number is less than 1,000 (the exponent is 2 or less). If the exponents are not equal (3) then the number is less than 1,000 (the exponent will be 0, 1, or 2). The carry will be set and the JNC is treated as a NOP.
8162	0AE	A<>C ALL	
8163	366	?A≠C S&X	
8164	023	JNC +04	
8165	38D	?NC XQ	If the carry is set we end up here. We execute the BCDBIN routine in the mainframe and then jump to the spot in our routine that clears the rest of C and writes it to X.
8166	008	02E3	
		[BCDBIN]	
8167	07B	JNC +0F	
8168	27C	RCR 9	If we got this far we know that the number is between 1,000 and 9,999; i.e., it is 4 digits long. The mainframe subroutine will only take up to 3 digits. So we peel off the 1000's digit and save it in the last nybble of the mantissa of A by roating it to nybble three of C and then saving it in A. We must then zero the
8169	11A	A=C M	
816A	05A	C=0 M	
816B	3E1	?NC XQ	
816C	008	02F8	
		[GOTINT]	

			mantissa of C because the subroutine at 02F8 requires this. The last three digits of the original decimal number are now in the S&X field of C. The GOTINT subroutine then converts them to binary in the S&X field of C.
816D	106	A=C S&X	Save the binary equivalent of the last 3 digits in A. The number of 1000's to add to this number is in nybble 3 of A. We load 1,000 into the S&X of C. We subtract 1 from the 1000's counter and add 1,000 to the answer in A. If there are no more 1,000's to add, the carry will be set (there will be an underflow) and we will not jump back to add more 1,000's. If the carry is not set we will loop around to add more 1000's until it does get set. We then place the answer in the S&X of C so that it may be written out to X. The mantissa and its sign are cleared to get rid of extraneous digits. We then write the answer out to X so we it may be used in some way by one of our User code programs.
816E	01C	R= 3	
816F	130	LDI S&X	
8170	3E8	HEX: 3E8	
8171	1A2	A=A-1 @R	
8172	146	A=A+C S&X	
8173	1A2	A=A-1 @R	
8174	3F3	JNC -02	
8175	0A6	A<>C S&X	
8176	05E	C=0 MS	
8177	05A	C=0 M	
8178	0E8	WRIT 3(X)	
8179	3E0	RTN	

Make sure that you update the FAT. There are now 12 functions. The last entry in the FAT should look like this:

Address	Hexcode	Description
8018	001	The first word of the FAT entry for BCD-BIN. The number is one because we have now reached the portion of RAM where there is not a zero in the third digit from the right in the starting address of the routine.
8019	056	This is the 2 least significant digits of the address.

Now let us go on to another subject: how to call a routine as a subroutine from another program in our example ROM.

RELATIVE EXECUTEs and GOTOs

In order to call any program as a subroutine from another MCODE routine in our example ROM, you must use a 3-word execute instruction. These instructions are known as relative executes. This is because it does not matter in which page the MCODE routine resides; the execute statement will always jump the same number of steps ahead or back and then return. The absolute executes that we described before always jumped to the same place regardless of the location of these instructions. These relative execute's and goto's are usually referred to as Port Dependent Execute's and Goto's. A drawback to this type of execute is that the C register is used by the routine that computes the branching address. Now for an explanation on how these three words are coded. The CPU of the 41 does not contain any three word instructions, so we shall describe how we come up with the mnemonics for them.

First, a discussion of how ROMs are divided up by these instructions. The 4K ROM page is divided into four blocks of 1024 words each. These 1024 word blocks are known as quads. The beginning addresses of each of the quads are at P000, P400, P800, and PC00 (in our example P = 8). The quads are numbered from zero to three. The first two words of the instruction is a subroutine call to a routine in the mainframe. There are 5 such routines. The first four handle subroutine calls to a specific quad. They take the third word of the execute instruction and add it to the number that is the start of their quad. The fifth entry point is used only when the subroutine being executed is in the same quad as the execute instruction. All five of these executes may only be of the No Carry execute variety. The hexcodes of these five entry points are given below.

In order for these relative execute's and goto's to properly function, the CPU <u>must</u> be in HEX mode, or you <u>WILL</u> end up at the wrong spot.

Hexcodes	Description
349 08C	This is the routine you call when you want to use an execute statement to access code in quad 0. This is at addresses 8000 to 83FF. The third word would be the 3 least significant digits of the address being called. For example, on a call to 8291, the third word would be 291.
36D 08C	This is the code for the first two words of a call to quad 1, which is at addresses 8400 to 87FF. The third word is the number of words after address 8400 at which you want to start executing the code. An example: for an execute to 8567 the third word would be 167 (167 + 8400 = 8567 in hex).
391 08C	These are the hexcodes for the first two words of an execute statement that calls a subroutine in quad 2. These are at addresses 8800 to 8BFF. The third word is added to 8800 to get the starting address of the subroutine that is being called. Therefore, to call a subroutine at address 8BFE, hex 3FE would be the third word of the instruction (3FE + 8800 = 8BFE).
3B5 08C	These are the hexcodes for subroutine calls to quad 3, at addresses 8C00 to 8FFF. The third word is added to 8C00 and the value for the starting address of the subroutine is obtained. For example, to execute code at 8E34, the third word would be 234 (234 + 8C00 = 8E34).

These instructions are subroutine calls themselves, and each uses an additional subroutine call of its own. They can therefore only be called when there are no more than two pending returns in the subroutine return stack. Otherwise the third and fourth subroutine returns, if any, will be lost. Don't confuse this with the User subroutine stack of the calculator. This is the CPU subroutine return stack, and may only have four pending returns, not six like the User subroutine stack.

The fifth set of hexcodes has the advantage of not using the additonal subroutine level required for each of the above types. This means that you can have three pending returns on the subroutine stack. However, its use is restricted to branches within the same quad. Also, it destroys the C register just like the other four types of calls. Here are the hexcodes and a description of them.

Hexcodes	Description
379 03C	This pair of words is always the same regardless of which quad is involved. The third word is the difference between the address of the first word in the quad you are in, and the address of the subroutine you are calling. For example, if you are in quad 2 (8800-8BFF), and the subroutine is at 8964 then the third word would be 164 (8964 - 8800 = 164). A call to a subroutine outside of quad 2 if the subroutine call originates from inside quad 2 would have to use one of the instruction hexcodes described above.

All addresses have been given with the most significant digit being 8 since our sample ROM is in page 8. However, this digit may be changed to any other page without affecting any of the values of the hexcodes.

If you want a relative GOTO instruction, then subtract hex 008 from the first word of the three word instruction. This only applies to the first four hexcode sets. For the last one given subtract hex 010 from the first of the three words. The interpretation of the third word is the same as for the execute instructions. These relative GOTO's use only one subroutine level, so each allows three pending returns on the stack. Again, to make things easy on yourself, it is highly recommended that you get an assembler.

There are actually no three word instructions in the instruction set of the 41 CPU. The relative execute's and goto's are disassembled correctly by most dissassemblers since whomever wrote the dissassembler knew that the five entry points mentioned above would use the ROM word directly after them to form a relative jump instruction. This type of dissassembly is called a MACRO. The actual instruction dissassembled is a combination of two or more instructions. The HP mainframe ROM listings use C=A even though there is no such instruction in the CPU instruction set. The actual dissassembly is A<>C, A=C.

Now we shall use one of these execute instructions to modify the BCD-BIN routine that we just wrote so that it may be used as a subroutine by other programs in our sample ROM. It may be called as a subroutine right now as is, except that it overwrites the decimal number in the X register with the hex equivalent of the original number. Since it would be nice to leave the X register alone as much as possible, we will modify the routine so this won't happen.

"BCD-BIN" revised

Address	Hexcode	Mnemonic	Description
814F	08E	"N"	Name of the routine.
8150	009	"I"	
8151	002	"B"	
8152	02D	"-"	
8153	004	"D"	
8154	003	"C"	
8155	002	"B"	
8156	379		This is the call to the entry point in our
8157	03C	GOSUB	ROM which is at 815B. This is just the
8158	15B	815B	BCD-BIN routine without the WRIT 3(X)
8159	0E8	WRIT 3(X)	instruction as the second to last step.
815A	3E0	RTN	Instead, this step is placed after the
			subroutine call and will be executed when
			the routine returns.
815B	0F8	READ 3(X)	**This is the entry point to be used by**
815C	10E	A=C ALL	**other programs in our ROM. The rest of**
815D	1BE	A=A-1 MS	**the routine is the same from this point**
815E	1BE	A=A-1 MS	**on until we get to the second to last**
815F	389	?C GO	**step of the original routine. The WRIT**
8160	053	14E2	**3(X) instruction should be removed and the**

[ERRAD] **RTN instruction should be moved up 1 word So essentially the rest of the routine is just moved down by 5 words.**

SKWID relaxing after a hard day of **MCODE**

TIPS, SHORT ROUTINES, and OTHER LITTLE GOODIES

This section will cover some exciting ways of programming useful functions that HP did not provide in the calculator. We will discuss how to shift bits right in the C register (you already learned how to shift bits left in the IF routine) and some other interesting tidbits.

In our first tip we will shift the C register right by one bit. In order to do this the following sequence of instructions are used.

Mnemonic	Description
C=C+C ALL C=C+C ALL C=C+C ALL RSHFC ALL	We shift the C register left by three bits (use C=C+C three times) and then shift right by one nybble. The end result is that the bit(s) are shifted right one. However, this does have its drawbacks. If there is a bit that is within the last three bits of the left side of register C when we start this sequence, then that bit will be lost (because it will cause an overflow when you do C=C+C with the leftmost bit set). So this routine does not work for the three leftmost bits of C.

The above sequence can be done on all or part of the C register. The same rules apply. The three leftmost bits of the field should be zero.

Some of you computer scientists will appreciate this next short routine. It is an XOR routine. HP gave us functions for AND and OR, so why not make one for EXCLUSIVE OR? The XOR function is a bit flipping function. We synthesized this in the IF flag routine by using calls to the mainframe ROMs. However, what if you want to do an EXCLUSIVE OR on the whole 56 bits of two registers? You should use the eight word routine below. This routine uses the A, B, and C registers. There are two inputs: the number to be changed, and the mask against which it will be compared. At the start the mask is in C and the number to be changed is in A. The way this routine works can best be illustrated by an example. For this example let's use just eight bits. The number to be changed will be 01001110 and the mask will be 00111011. The only bits that get inverted from their original position will be the ones that correspond to a bit in the mask that is equal to one.

bit number	7 6 5 4 3 2 1 0
Mask	0 0 1 1 1 0 1 1
Number	0 1 0 0 1 1 1 0

Since bits 0, 1, 3, 4, and 5 are one in the mask, these bits will be inverted in the original number; all of the other bits in the original number are left unchanged. Therefore, the final answer is 01110101. We assume the CPU is set to hex mode upon entry to this routine. The routine is given below.

Hexcode	Mnemonic	Description
0EE	C<>B ALL	Save the mask in B for later use, and get it back into C. B was picked because register A will be used for something else and we need to have a register that can interact with A. B meets all of these requirments.
0CE	C=B ALL	
370	C=C OR A	Set all of the bits in the C register which are set in either the A or C registers. Then exchange this result with the original mask value.
0EE	C<>B ALL	
3B0	C=C AND A	Set all of the corresponding bits in register C that are set in C and A. This tells us which bits must be cleared. The next instruction inverts every bit in the whole register. We now have set all of the bits that were not set in both registers.
2AE	C=-C-1 ALL	

06E	A<>B ALL	Get back the answer from the OR instruction.
3B0	C=C AND A	Since we have zeroed all of the bits that were set in the previous AND instruction, these bits will now be cleared. The bits set by the OR instruction and C=-C-1 will now be set.

Well, that's the routine. There is no entry in the FAT for this routine. It is just a sample of how short instruction sequences may be used to form instructions that are not in the CPU chip. The answer is left in the C register. Maybe you can find a place to put it in one of your programs.

You may wonder how it's possible to save four nybbles away someplace without altering the contents of the C register or any of the other 56-bit registers. There are many places that you could use for storage, but the following procedure is used in several mainframe routines. If you are not using the G register or any of the flags in ST, you can rotate the desired nybbles until they are right justified in the C register (in positions 0 thorugh 3). Then you can put 2 nybbles in ST and the other 2 nybbles in G. When you need the data again, the reverse of this procedure brings the 4 nybbles back into C. Here are the instructions you need:

Hexcode	Mnemonic	Description
	RCR n	Rotate C right by n nybbles so that the nybbles you want to store are in positions 0 through 3. The value of n depends on which nybbles are to be saved.
358	ST=C	Copy nybbles 0 and 1 into the ST register.
21C	R= 2	Set the pointer to 2.
058	G=C	Copy nybbles 2 and 3 into G.
	RCR m	Rotate C right by m = 14-n nybbles so that the four nybbles you stored away are put back in their original positions.
.		This represents the rest of the routine before you
.		bring back the four saved nybbles. This section
.		should not use G or ST. To recover the data, use:
398	C=ST	Copy ST into nybbles 0 and 1 of C.

21C	R= 2	Set the pointer to 2.
098	C=G	Copy G into nybbles 2 and 3 of C.

Our next routine will be very helpful to some of you. It is a routine to check if a RAM register exists. If you remember, when we wrote our AM and MA routines, we assumed that you had a 41CV, 41CX, or 41C with a Quad memory module. With the following routine you can find out whether or not a RAM register actually exists without putting any constraints on the user of the program. The routine assumes that the register to be checked has been selected using the RAMSLCT instruction and that the CPU is in hex mode.

Hexcode	Mnemonic	Description
038	READ DATA	Reads the contents of the selected RAM register into C; remember the register to be tested must be selected before starting this routine.
2A6 10E	C=-C-1 S&X A=C ALL	This instruction inverts all of the bits in S&X of C. All of the 1 bits, in the sign and exponent, become 0's, and all of the 0 bits become 1's. This result is then stored there because we will later test the A and C registers to see if they are not equal. These are the only two CPU registers that may be used if a not equal test is wanted between registers.
2F0 038 36E 381 00B	WRIT DATA READ DATA ?A≠C ALL ?C GO 02E0 [ERRNE]	We write the results of the bit inversion out to the RAM register we are checking for existence. We immediately read back this same register. If the register exists then the data will not change; the test will not be true, and we skip the GOTO to the NONEXISTENT error routine. If the register does not exist then the data we stored there will not be the same since there is no RAM in which to save it. Therefore the two values will test unequal so we exit to the NONEXISTENT

		error message.
2A6	C=-C-1 S&X	If we get this far, then A and C are equal so we invert C back to what was originally read from the RAM register. If you do C=-C-1 twice, each logic 1 bit will have been inverted to zero and then back to 1, so, we should get the same answer returned. The same applies for the 0 bits. We then write the result out to the RAM register and then return. The contents of the register that is selected are in C at the end of this routine. The RAM select pointer is not changed.
2F0	WRIT DATA	
3E0	RTN	

Ten bonus points for anyone who figures out how to integrate this routine into the AM/MA routine combination. This way we don't have to put any constraints on the user of the routine.

Now we will place this routine into our sample ROM and write a program to use it. The routine we shall write will be a Non-normalized Recall routine. By using it we shall be able to recall the contents of any RAM register in the calculator. The number input into the X register before this function is executed is the absolute address of the register you wish to recall. If 192 is in X, then the bottom register of Main Memory will be recalled (see page 32 for an explanation on this subject). If a register is recalled that does not exist, then the NONEXISTENT error message will be displayed. Non-normalization means recalling the contents of a register without modifying it. When you use the RCL function on a register which does not contain ALPHA DATA and there are hex digits greater than 9 in the register, then those digits are converted to BCD values.

"NR"

Address	Hexcode	Mnemonic	Description
817F	092	"R"	Second letter of the routine name.
8180	00E	"N"	First letter of the name.

8181	0F8	READ 3(X)	Get the contents of the X register, then
8182	128	WRIT 4(L)	save X in the LASTX register.
8183	379		This subroutine call is to our entry point
8184	03C	GOSUB	to convert decimal numbers to hexadecimal
8185	15B	815B	numbers (see page 78). We need this in
8186	270	RAMSLCT	hex so that we may use RAMSLCT to
			select the desired RAM register.
8187	379		This is a call to another entry point in
8188	03C	GOSUB	our sample ROM. It is at 8190. It is
8189	190	8190	the routine we wrote to tell whether or
818A	10E	A=C ALL	not a RAM register exists. Upon retur-
818B	04E	C=0 ALL	ning from the subroutine, the contents of
818C	270	RAMSLCT	the desired register are in C. We need to
			select chip 0 so we may write the answer
			out to the X register. Remember, the
			tested register must be selected upon
			entry to our subroutine and our subroutine
			does not change this. We save C in A and
			then zero C so the RAMSLCT instruction
			will select chip 0.
818D	0AE	A<>C ALL	We now retrieve the contents of the
818E	0E8	WRIT 3(X)	recalled register from A. This value is
818F	3E0	RTN	then written out to the X register. Then
			we return.
8190	038	READ DATA	This is the start of our routine to find
8191	2A6	C=-C-1 S&X	out if the register we want to access
8192	10E	A=C ALL	exists. 8190 is the address which you
8193	2F0	WRITE DATA	call if you want to execute this as a
8194	038	READ DATA	subroutine. For an explanation of how this
8195	36E	?A≠C ALL	routine works see page 83.
8196	381	?C GO	
8197	00B	02E0	
		[ERRNE]	
8198	2A6	C=-C-1 S&X	
8199	2F0	WRITE DATA	

819A 3E0 RTN

Don't forget to update the FAT. We now have 13 functions in the FAT. Therefore, 00D would be placed at address 8001 of our ROM. We would not put 013. The number of functions is in hex and 00D is 13 in hex.

What's this you are saying? You think the NR routine is a complete waste and want to get rid of it but you say you like the routine to tell if RAM registers exist. Well, not everyone is perfect. You can't just delete the routine, you must also delete the FAT entry for this routine. We'll show you how to do this now. First, let's see how the whole FAT currently looks.

Address	Hexcode	Description
8000	001	XROM number of our ROM.
8001	00D	This is the number of entries in the FAT, in hex.
8002	000	These two words are the address of the first executable
8003	08C	instruction of the ROM header SKWID 1A. All of the rest of the FAT will be grouped into sets of two words which are the three rightmost digits of the first executable instruction of each function (see page 20).
8004	000	Address of first executable instruction of Y<>Z.
8005	091	
8006	000	Address of first executable instruction of GE.
8007	09A	
8008	000	Address of first executable instruction of COUNT.
8009	0A7	
800A	000	Address of first executable instruction of MA.
800B	0BD	
800C	000	Address of first executable instruction of AM.
800D	0C1	

800E	000	Address of first executable instruction of IF.
800F	0E4	
8010	001	Address of first executable instruction of FS?S.
8011	002	
8012	001	Address of first executable instruction of FC?S.
8013	008	
8014	001	Address of first executable instruction of BIN-BCD.
8015	02D	
8016	001	Address of first executable instruction of F?.
8017	04C	
8018	001	Address of first executable instruction of BCD-BIN.
8019	056	
801A	001	Address of first executable instruction of NR.
801B	081	

Well, there's what the FAT should look like. The rest of the FAT words are 000 instructions since we haven't put anything in them. If it doesn't look like this something went wrong somewhere. The problem is probably that you forgot to add one of the entries into the FAT.

If you want to delete the last entry in the FAT, you must decrease the number at address 8001 by one. Then you may put a 000 hexcode at addresses 801A and 801B since that is where the last FAT entry is in our ROM. Now you may delete the NR routine from your ROM starting with address 817F, the address of the last letter of the NR name, until 819A, the last instruction in routine. Or you could leave the routine in place and just delete the FAT entry. The calculator will think that the routine has been deleted and you will still have the entry point at 8190 for checking if RAM registers exist.

Now suppose you want to delete the IF routine from the FAT. That is a little harder. For starters, you can't just delete the two words that point to the first executable instruction of IF. This would leave a void of two 000 words in the middle of the FAT. These would tell the calculator that the first executable instruction of some routine is at 8000. Also, when you decrease the number at address 8001 by one you are making the last routine in the FAT (NR), inaccessible.

The best way to illustrate this is for you to try it out. Set the two words at addresses 800E and 800F to 000. Now do a CATALOG 2. The calculator starts through the catalog correctly, until the place where the IF function was. At this point the calculator should lock up with "@" in all twelve positions of the display. The calculator is looking for a routine that begins at 8000. It is trying to read the function name from the last few words of page 7, which immediately precedes address 8000.

To get out of this lockup condition pull the batteries out of the calculator and put them back in after about 5 seconds. You may be able to use a simpler method as well. HP-41's manufactured since the introduction of the HP-41CV incorporate two hardware reset sequences that permit recovery from most crashes. To use the first reset method press and hold the ENTER key while turning the calculator off and on. Then release the ENTER key. The second method is to hold the backarrow key down while turning the calculator off and on. Then release the backarrow key. If you have an earlier HP-41, the only way to recover from a microcode "infinite loop" involves removal of the batteries and possibly additional steps. See page 214 of "HP-41 Extended Functions Made Easy" for more crash recovery tips applicable to older machines.

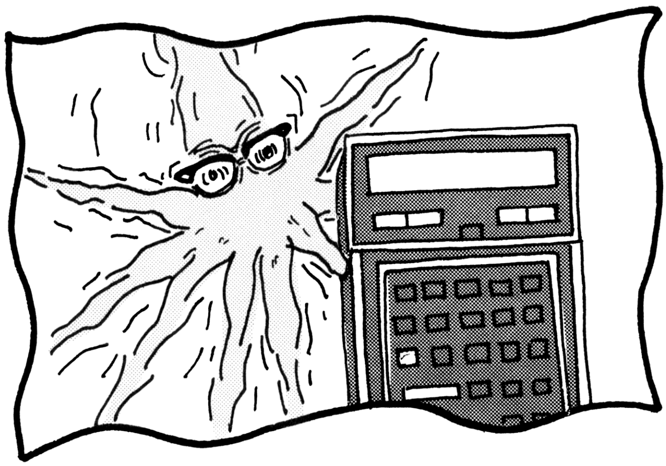
Now decrease the number at address 8001. Do a CATALOG 2 and the same lockup will occur. What you have to do to fix this situation is to fill the gap left in the FAT by the absence of the IF function. One way to fill the gap is to move all of the FAT entries after the IF function up by two words. Another way is to just MOVE the FAT entry for NR to the position that was occupied by IF. This second approach will naturally change the order of functions displayed in Catalog 2.

After you have removed the gap in the FAT, decrease the number at address 8001 by one. The FAT should now look like the lisiting that follows. We will just put the routine name next to the first of the two words that tell where the first executable instruction is located.

Address	Hexcode	Description
8000	001	XROM number.
8001	00C	Number of functions in the FAT. This is decreased by one from what it was before.
8002	000	SKWID 1A
8003	08C	
8004	000	Y<>Z
8005	091	
8006	000	GE
8007	09A	
8008	000	COUNT
8009	0A7	
800A	000	MA
800B	0BD	
800C	000	AM
800D	0C1	
800E	001	FS?S. This is where the address for the IF function was.
800F	002	The rest of the function addresses are moved up by two words from where they were before.
8010	001	FC?S
8011	008	
8012	001	BIN-BCD
8013	02D	
8014	001	F?
8015	04C	
8016	001	BCD-BIN

8017	056	
8018	001	NR
8019	081	

The words at 801A and 801B should now be set to 000. This will signal to the calculator that the FAT has ended (see page 20). Now you may do a CATALOG 2; the IF function will be gone and the calculator will no longer lock up. You may also use the space where the IF routine resides, addresses 80E2 through 80FD, for some other program. However, the new program must fit completely into the space left by the IF routine.



SKWID really gets into his programming.

You say that you like math functions. We've come up with a neat little routine for you. It is a Quotient Remainder routine. This routine will place Y modulo X (integer number of times that the X register will divide into the original number in the Y register) into the Y register. It places the remainder in the X register. The formulas used are:

Input	Output
X: x	X: y MOD x
Y: y	Y: (y - y MOD x)/x

The Z and T stack registers are left undisturbed. The old X register is saved in LASTX. The routine checks for Alpha Data and also if X is zero since we can't divide by zero. Just in case you are not familiar with the MOD function in the calculator we shall explain its use. The MOD function uses both the X and Y registers. The formula is the following: Y - [Y/X]*X, where the brackets denote "integer part". What this gives us is the remainder of a division represented as a whole number instead of a decimal number less than 1. It is represented as Y MOD X.

As an example, if Y equals 5 and X is 2 then 5 MOD 2 is 1. Our program will call the MOD routine in the mainframe as a subroutine. There are many other useful math subroutines used in this program. Our program shall be called QR and will be placed in the vacant space left by the IF program. We will start QR at address 80E2, the same place where IF started.

"QR"

Address	Hexcode	Mnemonic	Description
80E2	092	"R"	Last letter of the routine name; hex 080 has been added to its hexcode.
80E3	011	"Q"	First letter of routine name.
80E4	0F8	READ 3(X)	Get the X register and put it into C. We
80E5	128	WRIT 4(L)	then write it out to the LASTX register.
80E6	10E	A=C ALL	We now save the X register, which was in
80E7	0B8	READ 2(Y)	C, into A and put the Y register into C.
80E8	355	?NC XQ	The call to the mainframe subroutine at
80E9	050	14D5 [unlabeled]	14D5 checks both the A and C registers, X and Y, to see if they contain Alpha data.
80EA	070	N=C	If either of them do, then the mainframe

			routine exits to the ALPHA DATA error message. If neither of the registers contain Alpha data, the routine returns with the A and C registers exchanged and with the CPU in decimal mode. This does exactly what we want for the next steps. We must then save C in N to satisfy the requirements of the MOD routine.
80EB 80EC	171 064	?NC XQ 195C [MOD10]	This is a call to the MOD routine. It requires that the CPU be in decimal mode. Notice that the call to the routine at 14D5 made sure of that. The MOD10 (modulo in base 10) routine takes A MOD C. We want Y to be in the A register and X to be in C. Also notice that Y was put into A and X was switched into C by the last mainframe subroutine.
80ED 80EE 80EF 80F0 80F1 80F2	070 2BE 10E 0B8 01D 060	N=C C=-C-1 MS A=C ALL READ 2(Y) ?NC XQ 1807 [AD2-10]	We now have the answer for the X register, Y MOD X, but we can't put it there yet, so we save it in N. We then invert the sign of the mantissa. In order to subtract using the mainframe routine you change the sign and add. We then save this in A and get the Y register again. The mainframe subroutine AD2-10 at 1807 performs C=A+C on two normalized decimal numbers. The answer will end up in C.
80F3 80F4 80F5 80F6 80F7 80F8	10E 0F8 261 060 0A8 0B0	A=C ALL READ 3(X) ?NC XQ 1898 [DV2-10] WRIT 2(Y) C=N	We now have Y - (Y MOD X) in C. We place this in A so we may call the X register into C for the last step. We must now divide A by C. Fortunately there is a routine at address 1898 of the mainframe ROMs where this is done. It even checks for division by zero. After the routine

80F9	0E8	WRIT 3(X)	is done we have (Y- Y MOD X)/X in C and Y MOD X in the N register. So now we write C out to Y. Then we retrieve Y MOD X from N and write this out to X before returning.
80FA	3E0	RTN	

You will notice that this routine barely fits into the space left by IF. There are only three words left unused. Now we must update the FAT. We do not have to open up the place where the address for the IF routine was and place the address of the first executable instruction of QR in its place. Instead, we may place the FAT entry for QR after the last address now in the FAT. The calculator does not care whether or not the addresses for the functions are in sequencial order. They may be put into any order you choose as long as each set of two words points to the first executable instruction of a routine. There are now 13 functions in our ROM. (We left the NR routine in and only deleted the IF routine.)

This next routine will be a welcome relief to those of you who need to see all ten digits of a number but find that the exponent keeps getting in the way. It is a View Mantissa routine. This routine allows you to view all ten digits of the mantissa of a number without changing the setting of the display or getting rid of the exponent of the number. This routine only views the mantissa and does not change any RAM registers. The way this is done is to put the value to be displayed into C and execute the mainframe entry point that places the contents of C into the display. A few other things must be done so everything will work right. These are explained in the listing below. This routine will allow you to view all ten mantissa digits of whatever number is in the X register.

"VM"

Address	Hexcode	Mnemonic	Description
819B	08D	"M"	Last letter of the routine's name.
819C	016	"V"	First letter of the routine's name.

819D	0F8	READ 3(X)	First we check X to make sure it is not alpha data. We read in X and then we use an entry point that checks the C register for alpha data. If there is alpha data we exit to the ALPHA DATA error message. Otherwise the routine returns with the original contents of C intact and the CPU in decimal mode. We want to be in hex mode so we reselect it.
819E	361	?NC XQ	
819F	050	14D8 [CHK#S]	
81A0	260	SETHEX	
81A1	3B8	READ 14(d)	In order to fool the calculator into thinking that we are in FIX 9 mode, we must modify the flag register so that the mainframe view routine will think we are in FIX 9. The bits that determine how many digits are to be displayed are in nybble 4. To get a setting of 9, we load a 9 into this spot. The bits for the current display mode, FIX, SCI, or ENG, are in nybble 3. In order to set FIX notation we must clear bit 2 of this nybble and set bit 3. We do this by loading eight into this nybble. Before we do all of this we save the original contents of the flag register so that they may be restored.
81A2	158	M=C	
81A3	05C	R= 4	
81A4	250	LD@R 9	
81A5	210	LD@R 8	
81A6	3A8	WRIT 14(d)	We now write this modified register out to the flag register. The calculator now thinks that it is in FIX 9 mode.
81A7	0F8	READ 3(X)	Get the contents of the X register.
81A8	046	C=0 S&X	We then zero the exponent and its sign since we only want to view the mantissa.
81A9	099	?NC XQ	Now we can execute the mainframe view routine called DSPCRG (DiSPlay C ReGister.) What is to be viewed is in C upon entry to this routine. It sends this
81AA	02C	0B26 [DSPCRG]	

			to the display and does not overwrite the X register.
81AB	198	C=M	We now retrieve the old flag register back from M. Then we must set flag 50, the message flag; the purpose of this flag is to tell the calculator to preserve the contents of the display when we go into standby mode. Otherwise the 41 defaults to the display corresponding to the current mode. The three modes are RUN, ALPHA, and PRGM. Fortunately there is a routine in the mainframe to do this. Actually we enter three words into the routine since we are restoring the old contents of the flag register which were saved in M.
81AC	205	?NC GO	
81AD	00E	0381 [unlabeled]	

Upon execution of this routine you will notice that the status of the decimal point does not change. If you normally use the comma as the decimal point then this is what will be used; if you use the period as the decimal point the answer will show up in that format. Now execute the routine and hit the backarrow key. The displayed answer went away but the X register stayed the same, just like HP's VIEW functions. Remember to update the FAT. We now have 14 functions, 00E in hex.

To skip, or not to skip, that is the question. Our next routine will show you the sequence used for skipping lines in a User code program. This is the same sequence that all of the functions in the calculator that have a "?" use. If the "?" is false they skip a step in your program. The function we will write is a multiple compare function. It shall be called X=Y? Z?. It will first check to see if X is equal to Y. If this is true we will end the routine and the program will continue at the next step. However, if X does not equal Y, then our routine will cause the User code program to skip either one or two steps, depending whether X equals Z. So at this point in the routine, just after we find out that X does not equal

Y, we skip one User program step. Next we compare the X and Z registers. If they are equal we exit our routine having skipped only one program line. If X does not equal Z we skip another program line and then end our routine. This routine illustrates the sequence of instructions you use to tell the calculator to skip a User code program line.

"X=Y? Z?"

Address	Hexcode	Mnemonic	Description
81B2	0BF	"?"	This is the last letter of the name of our routine. Notice that a space separates the two words. This space must be keyed in when executing the routine.
81B3	01A	"Z"	
81B4	020	" "	
81B5	03F	"?"	
81B6	019	"Y"	
81B7	03D	"="	
81B8	018	"X"	
81B9	244	CLRF 9	This flag is used to tell if we have reached the X=Z part of the routine. If it is clear we are doing the X=Y part of the routine. If it is set then we are in the X=Z part.
81BA	0F8	READ 3(X)	Put the X register into C and then save it in A. We choose A so that we may use the ?A≠C instruction to compare these two registers later in the routine. Then we retrieve the Y register and compare it with X. If X=Y the carry will not be set and we can return. If X≠Y the carry will be set and we go to the section of our routine that has the instructions for skipping a program line.
81BB	10E	A=C ALL	
81BC	0B8	READ 2(Y)	
81BD	36E	A≠C ALL	
81BE	3A0	?NC RTN	
81BF	03B	JNC +07	
81C0	248	SETF 9	Setting this flag tells the routine that we have reached the X=Z portion of our routine. We then get X and put it into A
81C1	0F8	READ 3(X)	
81C2	10E	A=C ALL	

81C3	078	READ 1(Z)	so we may go through the same sequence of steps as at addresses 81B8-81BC except we use Z in place of Y. This is the start of the sequence for skipping one line of a User code program. First ?NC XQ 2950 GETs the Program Counter in the format required by other mainframe ROM routines. This format is called "MM form", and entails doubling the byte digit of the User code program counter when the pointer is in RAM. Then we increment this pointer by the number of bytes in the next program line using the mainframe SKPLIN (skip line) routine at 2AF9. There may be anywhere from one to sixteen bytes in a program line. Then we update the User program pointer by storing the new value in register b (using the routine at 232F) so that the program has now skipped a program line without executing it. PUTPCX is one of the PUT Program Counter entry points.
81C4	36E	?A≠C ALL	
81C5	3A0	?NC RTN	
81C6	141	?NC XQ	
81C7	0A4	2950 [GETPC]	
81C8	3E5	?NC XQ	
81C9	0A8	2AF9 [SKPLIN]	
81CA	0BD	?NC XQ	
81CB	08C	232F [PUTPCX]	
81CC	24C	?FSET 9	Now we check to see if this is the first time through the line skipping loop. If it is, flag 9 will be clear and the carry will not be set, so the ?C RTN instruction will not be executed. Since we have not yet gone through the X=Z section of our routine we will jump back to this section (at 81C0) if flag 9 is clear. If flag 9 is set, the carry will be set and the ?C RTN instruction will be executed. This tells us that we have been through the loop to skip a program line twice, once for the X=Y part and once for the X=Z
81CD	360	?C RTN	
81CE	393	JNC -0E	

	part. Since we have asked both questions we may return.

Try out this function in one of your programs. A sample setup could be as follows:

Instruction	Description
.	Steps preceding the X=Y? Z? instruction.
.	
X=Y? Z?	
GTO 01	Go to label 01 if X is equal to Y.
GTO 02	Go to label 02 if X is equal to Z but is not equal to Y.
.	Continue on with the program if X does not equal to either Y
.	or Z.

Remember to update the FAT. You should get into the habit of doing this right after you finish writing a routine. We now have 15 functions in our sample ROM (00F in hex).

The next routine is an Alpha View routine that will never stop a program. The AVIEW function will stop a program for no apparent reason if flag 21 is set and there is no printer plugged into the calculator. This routine allows you to view Alpha without sending anything to the printer as does AVIEW. It is an excellent example of the power of using the mainframe ROM entry points. The routine is five words long and four of these words are used to call mainframe entry points. This is very efficient. The routine is called VA.

"VA"

Address	Hexcode	Mnemonic	Description
81CF	081	"A"	Routine name.
81D0	016	"V"	

81D1	104	CLRF 8	The first mainframe entry point at address 2C10, ARGOUT = Alpha ReGister OUT, outputs the Alpha register to the display. Clearing flag 8 tells the routine not to treat this as a prompt, as this would stop the routine. The GOTO instruction to address 0380 recalls the contents of the flag register and then sets the message flag (50) and restores the flag register with the message flag set (see page 95). Our routine returns through this mainframe routine.
81D2	041	?NC XQ	
81D3	0B0	2C10 [ARGOUT]	
81D4	201	?NC GO	
81D5	00E	0380 [unlabeled]	

All these addresses for the mainframe entry points we are using came from HP's documented listings of the 12K of mainframe ROMs. These listings are partially annotated by the programmers who developed the HP-41. The entry points are usually very well described with the kind of setup your routine needs to do before calling on one of these entry points. They also tell what the output should be.

One of the drawbacks of these documents is that they are listed in octal, not hexadecimal. So you need some way of converting from octal to hex. This little problem should not stop you from getting these documents. They are much too valuable a tool to let such a little thing like this interfere. How do you get hold of one of these documents? Well, for starters, don't call HP, they will refuse to answer any questions regarding MCODE programming on the 41. In fact, that is one of the reasons for this book. The place to get these listings, or VASM as they are called, is from a worldwide HP calculator user's group called PPC. PPC's address is given in Appendix A.

Since seeing the examples of how entry points can be used, you have probably ordered your VASM listings and are anxiously awaiting their arrival. But for now let's get on with some more examples.

This next routine is a Random Number generator program. There is nothing fancy about this program. We use the brute force method on this one. Just load in the numbers and crank away. This algorithm has been used in the HP-34C Applications book and the 41C Standard Applications Pac. The input for the program is in the X register. It can be any number; just don't make it too big. This input is the seed for the algorithm. The program takes this seed and then multiplies it by 9,821, adds 0.211327, then takes the fractional portion. The answer is output to X. The old X is saved in LASTX. This program is just over 7 times as fast as a User code program that performs the same calculations. Arithmetic operations are already relatively efficient in User code, because most of the work is done within highly optimized mainframe MCODE routines. The overhead of going to the User level (approximately 10 milliseconds per instruction) is less on a percentage basis for the more complicated User code instructions. Guess we can't always be 100 times faster.

"RN"

Address	Hexcode	Mnemonic	Description
81D6	08E	"N"	Routine name.
81D7	012	"R"	
81D8	00E	A=0 ALL	First we zero A and get the Random number
81D9	0F8	READ 3(X)	seed. Then we save the seed in the LASTX
81DA	128	WRIT 4(L)	register.
81DB	355	?NC XQ	The reason we zeroed A was so that there
81DC	050	14D5	would not be Alpha data there when we executed the mainframe routine at address 14D5. This routine checks A and C for Alpha data and sets the CPU to decimal mode. It then exchanges A and C from what they were originally.
81DD	35C	R= 12	We now set the pointer to the first digit
81DE	250	LD@R 9	of the mantissa so we can load in our
81DF	210	LD@R 8	first constant. It is 9,821. We load in

81E0	090	LD@R 2	the mantissa and also the exponent (003).
81E1	050	LD@R 1	We are now set up to do the multiplication
81E2	130	LDI S&X	of these two numbers. Mainframe routine
81E3	003	HEX: 003	MP2-10 at 184D multiplies A times C.
81E4	135	?NC XQ	The answer is left in C.
81E5	060	184D	
		[MP2-10]	
81E6	10E	A=C ALL	We save the answer from the multiplica-
81E7	35C	R= 12	tion in A so we may load C with the next
81E8	04E	C=0 ALL	constant. Before we start to load C with
81E9	090	LD@R 2	the constant, we zero it so that we start
81EA	050	LD@R 1	with a clean slate. We set the pointer to
81EB	050	LD@R 1	the first digit of the mantissa and start
81EC	0D0	LD@R 3	to load the mantissa of the constant. We
81ED	090	LD@R 2	set the pointer to the first digit of the
81EE	1D0	LD@R 7	exponent sign. The exponent sign is 9
81EF	21C	R= 2	since the exponent is negative (see page
81F0	250	LD@R 9	5). Why is the exponent 99 instead of
81F1	250	LD@R 9	01? The calculator represents negative
81F2	250	LD@R 9	exponents by subtracting them from 100
			(100-1=99) so for a number with a negative
			exponent of 3 the exponent would be 97
			(100-3). Another way to accomplish the
			last four instructions is to use a C=C-1
			S&X.
81F3	01D	?NC XQ	Now that we have the two numbers all set
81F4	060	1807	up, we call on the mainframe routine that
		[AD2-10]	will add the normalized values in the A
81F5	084	CLRF 5	and C registers. The answer from this is
81F6	0ED	?NC XQ	left in C. The routine at address 193B is
81F7	064	193B	a dual-purpose integer/fraction routine.
		[INTFRC]	Here we use it as a fraction routine by
81F8	0E8	WRIT 3(X)	clearing flag 5. (Setting flag 5 gives
81F9	3E0	RTN	the integer routine.) ?NC XQ 193B takes
			the fractional portion of the number in C

and outputs it back to C. We then write our answer out to X and return.

Don't forget to update the FAT. There are now seventeen functions in our ROM. Therefore you would put 011 hex at address 8001.

The next routine sounds like it will be very easy to program. However, this is deceiving. It is a SIZE-finder routine. It will give the number of RAM registers that are allocated for data storage. This number will be put into the X register. This routine will work on any 41 Calculator with any amount of memory. The object of this routine is to find the largest existent RAM register in the calculator. Since RAM may be added in blocks of 64 (one memory module for the 41C) we start at the highest possible RAM address and check to see if it exists. If the register exists we've found the top of RAM. This is why we start from the highest possible address and work our way down. We do some manipulations before calling on the BIN-BCD routine that we wrote earlier. The routine will be called "S?".

"S?"

Address	Hexcode	Mnemonic	Description
81FA	0BF	"?"	Second letter of name.
81FB	013	"S"	First letter of name.
81FC	130	LDI S&X	We load into C the highest possible
81FD	1FF	HEX: 1FF	address of an existent RAM register. If you have the full 320 RAM registers in your calculator the top address will be 1FF.
81FE	158	M=C	This is the start of the loop to find out
81FF	270	RAMSLCT	the address of the topmost RAM register. We first save the RAM register pointer in M and then select that register. Now we will check to see if the register exists.

8200	038	READ DATA	This is the start of the section that figures out whether or not the RAM register exists. You are probably wondering why we did not jump to the entry point in our ROM that does this. The only problem with that approach is that if the RAM register does not exist we would go to the NONEXISTENT error message. In this routine if the register does not exist then we decrement the RAM register pointer by 64 and check again. We do this until we find a register that exists. This section is exactly like the entry point in our ROM except that instead of going to the NON-EXISTENT error message we jump to another part of the routine (JC +0E to 8214). For an explanation of this routine see page 83.
8201	2A6	C=-C-1 S&X	
8202	10E	A=C ALL	
8203	2F0	WRITE DATA	
8204	038	READ DATA	
8205	36E	?A≠C ALL	
8206	077	JC +0E	
8207	2A6	C=-C-1 S&X	
8208	2F0	WRITE DATA	
8209	198	C=M	We now retrieve the RAM register pointer into C and save it in A for later use. This pointer is the address of the top-most existent RAM data register. Chip 0 is then selected (remember the last register selected was the topmost register of RAM) and the address of data register 0 is obtained from nybbles 3, 4, and 5 of status register c (see page 35). In order to put this into the S&X field of C, we must rotate right 3 nybbles. We then add one to the address of the topmost existent RAM register. This is because the actual top address is one more than the highest register we can address. These two numbers are then subtracted and the answer is left in A. This is done because the GOTO
820A	106	A=C S&X	
820B	046	C=0 S&X	
820C	270	RAMSLCT	
820D	378	READ 13(c)	
820E	03C	RCR 3	
820F	166	A=A+1 S&X	
8210	1C6	A=A-C S&X	
8211	369		
8212	03C	GOTO	
8213	12F	812F	

			812F statement uses the C register. This is a GOTO to the BIN-BCD routine that we wrote earlier. The answer is placed into X.
8214	198	C=M	This section of our routine gets the RAM register pointer from M and then puts it into A. We then load 040 (64 decimal) into C. Since the calculator memory is arranged into blocks of 64, the next try will be a register that is 64 less than the previous one. This is subtracted from the current RAM register pointer. Then we go back to the start of the loop at address 81FE.
8215	106	A=C S&X	
8216	130	LDI S&X	
8217	040	HEX: 040	
8218	246	C=A-C S&X	
8219	32B	JNC -1B	

Remember to update the FAT. There are now 18 functions in our ROM. The number at address 8001 should be 012. The last entry in the FAT should look like this:

Address	Hexcode	Description
8024	001	The 1 is the third digit from the right in the address of the first executable instruction of the "S?" routine. It has the two leading zeros like all of the other functions.
8025	0FC	This is the two rightmost digits of the address of the first executable instruction. As always, the leading 0 has been placed in front.

The next routine will be one of the comparison functions that HP left out of the calculator mainframe. It is the "X>=Y?" function. This routine is rather short and is an excellent routine to show how a good knowledge of the mainframe entry points can be put to use. In this routine we shall use two such entry points. The first will be at address 1619. This will tell the calculator not to skip a line if we are running or single-stepping a program. If we execute it from the keyboard then a YES is put into the display. The other entry point is to address 15F8. This is just the routine to see if X is greater than Y. The necessary setup must be done before either routine can be executed.

"X>=Y?"

Address	Hexcode	Mnemomic	Description
821A	0BF	"?"	Routine name.
821B	019	"Y"	
821C	03D	"="	
821D	03E	">"	
821E	018	"X"	
821F	0B8	READ 2(Y)	We put the Y register into C and then save it in A. Then we get the X register into C and place it into N. These two conditions must be met because the entry point at address 15F8 must have X in N and Y in A in order to correctly perform its duties.
8220	10E	A=C ALL	
8221	0F8	READ 3(X)	
8222	070	N=C	
8223	36E	?A≠C ALL	We now check to see if X (C) is equal to Y (A). If it is, the carry will not be set and we will not want to skip a step if a program is running. The NOSKP routine at 1619 does this and will put YES into the display if the function is executed from the keyboard.
8224	065	?NC GO	
8225	05A	1619 [NOSKP]	
8226	3E1	?NC GO	This is the call to the routine to check if X is greater than Y. Since we know that they are not equal (if we get this far) X is either greater or less than Y. The XX>Y? routine (eXecute X>Y?) will figure out which is true and skip a program step if X is less than Y or put a NO into the display if it was executed from
8227	056	15F8 [XX>Y?]	

the keyboard. If X is greater than Y a program step will not be skipped or a YES will be placed into the display.

Remember to update the FAT. You can program the X>=0? function by just replacing the READ 2(Y) statement with a C=0 ALL instruction. This will compare X with zero instead of Y.

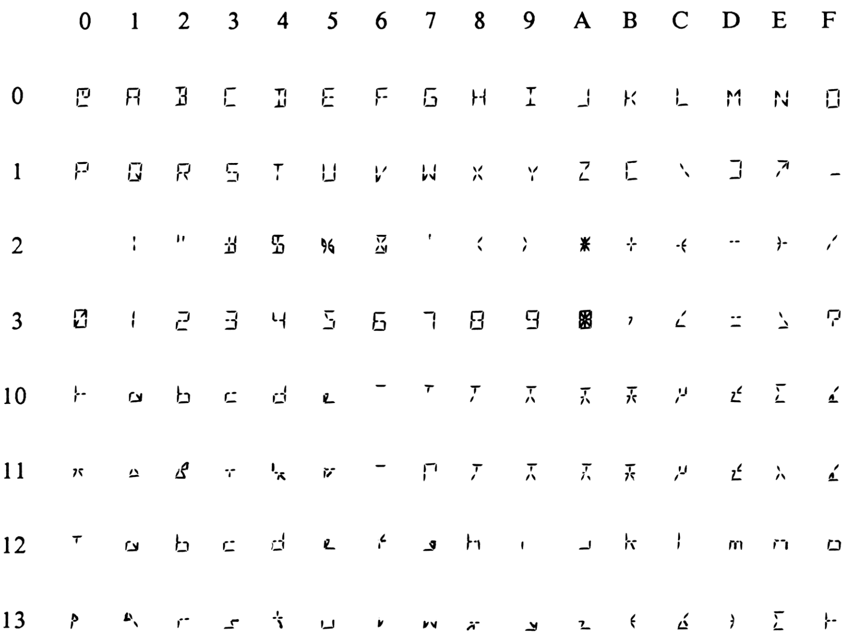
THE VISUALS

ACCESSING THE DISPLAY

The display is treated by the CPU as a peripheral. In order to access the display you must select it using the PRPH SLCT command. This instruction uses digits 1 and 0 of C to specify the peripheral to be selected. This is much like the RAMSLCT instruction, except that in order to select the display you must always use the same value in digits 1 and 0 of C. This number is FD. Once the display is selected it may be read from and written to. To do this you use the READ/WRIT instructions. If we write to the display using these functions and RAM registers are selected that exist, then these registers will also be written to. Therefore we should select a nonexistent chip whenever we select a peripheral. The nonexistent RAM chip that is usually used is chip 1 which starts at address 010 and goes through address 01F. To select this chip we must put 010 into the S&X field of C and use the RAMSLCT instruction to select the nonexistent RAM at this address.

There have been three different displays in the life of the 41. The first appeared in 41C's manufactured before 1981. The second display appeared in 1981 and has been in all HP-41 calculators manufactured up until about the time this book came out. These two displays cannot access the last three rows of the LCD character table (see next page). If a hexcode from these last three rows is used, a space will be displayed. The third display can access the entire LCD table and also allows you to change the contrast (viewing angle).

LCD CHARACTER TABLE



The display is divided into three registers. They are called the A, B, and C registers. These are not the same as the main CPU registers and should not be confused with them. The A register contains the lower four bits of each character, the B register contains bits four to seven of each character, and the C register holds bit 8 of each character.

The display READ/WRIT functions each have certain, well-defined, tasks that they perform. Data transfers can be in 1, 4, 8, or 9 bit format. These may be transferred one character at a time, or in multicharacter formats, depending on the instruction. The READ instructions give varied outputs depending upon which display your calculator has. These variations only apply to the bits and nybbles which are not the recipient of the data obtained during a READ instruction. The scope of these output variations will not be covered in this book, so your programs should not depend on getting particular values in these "unused" bits or nybbles.

The display is set up so that each of the 12 character positions in the display uses 9 bits (4 bits from A, 4 bits from B, and 1 bit from C). Bits 0 through 5 specify a character from rows 0 to 3 of the LCD character table. Bits six and seven are the punctuation field. The table below shows how to set/clear bits 6 and 7 for various punctuation symbols.

bit 7	6	punctuation symbol
0	0	no punctuation symbol
0	1	period
1	0	colon
1	1	comma

Here is the table of all of the HP display mnemonics which correspond to the READ/WRIT instructions. These instructions, which appear in the HP documentation for the display and mainframe, are not correctly dissassembled by any of the currently available dissassemblers.

	READ	WRIT	
15	FLSABC*	SLSABC	
14	FRSABC**	SRSABC	
13	FLSDAB	SLSDAB	
12	FRSDAB	SRSDAB	
11	FLSDB	SLSDB	READ DATA : FLLDA
10	FLSDA	SLSDA	WRITE DATA : WRTEN
9	FRSDC	SRSDC	
8	FRSDB	SRSDB	
7	FRSDA	SRSDA	
6	FLSDC	SLLABC	
5	READEN	SLLDAB	
4	FLLABC	SRLABC	
3	FLLDAB	SRLDAB	
2	FLLDC	SRLDC	
1	FLLDB	SRLDB	*appears as RABCL in HP listings
0	-------	SRLDA	**also given as RABCR in HP listings

Now we shall describe how to decipher these mnemonics.

The first character is either F or S corresponding to FETCH or SHIFT. The second letter is an L or R for LEFT or RIGHT. The third character is an S or L for SHORT or LONG. The remaining characters identify the registers on which the operation is to be performed: A, B, C, AB, or ABC. All one-or two-letter suffixes are preceded by the character D (display), which has no significance other than its value as a mnemonic.

FETCH reads data from the display into the C register. SHIFT pushes data from the C register into the display. LEFT or RIGHT specifies which direction the designated fields rotate within the display. (Rotation only occurs for the specified register or registers.) SHORT or LONG specifies the number of character positions which are to be read from or written to. SHORT means a single character position. LONG is the maximum number of character positions for which the corresponding data can fit in 12 nybbles. This is 4 positions for ABC, 6 for AB, and 12 for A, B, or C.

For example, consider SLSABC. This instruction writes data to the display (SHIFT), shifting in a single character (SHORT) in from the right (forcing a shift to the LEFT). The data written is 9 bits (ABC), which completely defines the character and punctuation.

Next consider FRLDC. This instruction FETCHes data from the right side of the display (forcing rotation to the RIGHT). The rightmost bit is placed into bit zero of nybble 0 of C and the second bit is put into nybble two and so on until the last bit is placed into nybble 11 of C. The display is not affected by this instruction since twelve characters are involved and the display will be rotated all the way around.

What follows are descriptions of the display instructions that are most commonly used within the HP-41's operating system ROMs. They are all 9 bit transfers, operating simultaneously on A, B, and C.

Instruction	Description
READ 14(d) (RABCR or FRSABC)	Reads the rightmost character in the display into the S&X of C. All characters are rotated right by one.
READ 15(e) (RABCL or FRSABC)	Reads the leftmost character in the display into the S&X of C and rotates the display left by one character.
WRIT 14(d) (SRSABC)	Takes the rightmost 9 bits of the S&X of C and pushes them into the leftmost position of the display. All of the existing characters are shifted right by one.
WRIT 15(e) (SLSABC)	Takes a single nine-bit character from S&X of C and writes it to the rightmost character of the display. The characters in the display are shifted left by one.
WRIT 4(L) (SRLABC)	Writes four characters from C to the left of the display. The characters that were in the display are shifted right by four. The first character is in digits 0 to 2 of C, the second is in digits 3 to 5 and so on. The character in digits 0 to 2 is pushed onto the left of the display first then the character in digits 3 to 5 is pushed to the left of that character and so on.

Now that we have gone through the instructions for writing and reading the display characters, we still have to deal with the annunciators at the bottom of the display. The status of these 12 annunciators is kept in a fourth display register, called E. Annunciators are set using the WRITE DATA (WRTEN) instruction. They may be read by using READ 5(M) (READEN). The transfer is to and from the S&X field of C. Below is a list of the bit in the S&X field of C which corresponds to each annunciator.

bit	Annunciator	bit	Annunciator
0	ALPHA	3	Flag 3
1	PRGM	4	Flag 2
2	Flag 4	5	Flag 1

6	Flag 0	9	G (for GRAD)
7	SHIFT	10	USER
8	RAD	11	BAT

As can be seen, the leftmost bits are for the leftmost annunciators. In normal operation, these annunciators do not stay on unless the corresponding condition is actually in effect. For instance, if you write a program that turns the ALPHA annunciator on and makes the standard exit to the normal function return, then you must be in Alpha mode or the annunciator will turn off.

Now let's have some fun and write a routine using some of these display instructions. We shall write a display test routine. This routine first displays twelve commas and pauses for a second or so. Then there are twelve starbursts in the display. Each of these is followed by a colon. The annunciators at the bottom of the display are also lit up. Now every display segment is on except the comma tails, which is why we viewed them first. This routine does not use any RAM registers, only the display. Ah, the beauty of MCODE. We shall call the routine DISTEST.

"DISTEST"

Address	Hexcode	Mnemonic	Description
8228	094	"T"	Routine name.
8229	013	"S"	
822A	005	"E"	
822B	014	"T"	
822C	013	"S"	
822D	009	"I"	
822E	004	"D"	
822F	130	LDI S&X	First we shall enable the display by
8230	0FD	HEX: 0FD	selecting peripheral FD. We must then
8231	3F0	PRPH SLCT	disable the RAM. Since we will be using
8232	130	LDI S&X	WRIT instructions we must choose a non-

```
8233    010    HEX: 010
8234    270    RAMSLCT
```

existent RAM chip so that RAM won't be written to.

```
8235    130    LDI S&X
8236    00B    HEX: 00B
8237    106    A=C  S&X
8238    130    LDI S&X
8239    020    HEX: 020
823A    3A8    WRIT 14(d)
823B    1A6    A=A-1  S&X
823C    3F3    JNC  -02
```

We shall now fill the display with spaces. This is what the calculator places into the display when it clears it. It is good practice to clear the display at the beginning of any routine that directly accesses the display. First we load a counter into C and save it in A. This will be decremented, and when underflow occurs, we jump out of the loop that fills the display with spaces. The hexcode for a space is 020. We load this into the S&X field of C and write it out to the display using the nine bit transfer instruction WRIT 14(d). This places a space into the left of the display and shifts all of the other characters right by one. The counter in A is then decremented and we jump back to the WRIT instruction and write another space to the display. When the counter underflows we drop out of the loop.

```
823D    19C    R=  11
823E    390    LD@R  E
823F    010    LD@R  0
8240    2D4    ?R=  13
8241    3EB    JNC  -03
```

The pointer is set to 11, the largest digit used when six characters (12 nybbles of data), are sent to the display using an eight bit transfer instruction. We load up each eight bits with the value E0 = 1110 0000. Bits six and seven are set to signify a comma. The lower six bits are set to the hexcode for a space (20 in hex or 100000 in binary). The character-loading loop is cycled 6 times. After the sixth time through, the pointer will equal thirteen since we just loaded a number

			into nybble zero. (The pointer decrements when we use the LD@R instruction.) When this happens, the carry will be set and we will not jump back to load more digits.
8242	0E8	WRIT 3(X)	These two instructions fill the display
8243	0E8	WRIT 3(X)	with commas. The first puts six commas into the display. There are spaces between the commas. The spaces we originally put into the display are shifted to the right by six characters. The second WRIT instruction finishes filling the display with commas.
8244	046	C=0 S&X	This is the delay loop so that you can see
8245	2A6	C=-C-1 S&X	the twelve commas in the display. First C
8246	266	C=C-1 S&X	is zeroed and then all twelve bits are
8247	3FB	JNC -01	inverted to ones using the C=-C-1 instruction. Then we subtract one from the S&X field until the carry is set. The carry will be set when we subtract 1 from 0. When this happens we will not jump back and the pause will be over.
8248	19C	R= 11	This is the loop to fill the display with
8249	2D0	LD@R B	the starburst character and the colon.
824A	290	LD@R A	The LD@R B instruction sets bit 7 which is
824B	2D4	?R= 13	the colon if bit 6 is not set. The other
824C	3EB	JNC -03	six bits are set so that the starburst character (hex 3A) is put into the display. The logic behind the loop is the same as for the steps at 823D to 8241.
824D	0E8	WRIT 3(X)	These two steps write six starbursts each
824E	0E8	WRIT 3(X)	out to the display. The commas are shifted off the display after the second instruction.
824F	046	C=0 S&X	First we zero the S&X field of C so that
8250	2A6	C=-C-1 S&X	when we invert all the bits, using C=-C-1,

8251	2F0	WRITE DATA	they will all go to one. Then we use the WRITE DATA instruction to turn on all of the annunciators at the bottom of the display.
8252	046	C=0 S&X	Now the message flag is set only to keep
8253	3F0	PRPH SLCT	the X register from being cleared when the
8254	1FD	?NC XQ	user presses the backarrow key to clear
8255	00C	037F	the display. Normally the message flag is set for the main purpose of preventing the display from being altered upon return of control to the operating system. Here we are not returning control to the operating system, but we still need to set the message flag. First we must deselect the display as a peripheral and then we enter the mainframe routine at a spot which selects chip 0 and sets the message flag.
8256	060	POWOFF	Since we want the display to stay as it is
8257	000	NOP	we go directly into standby mode so as to skip over the processing normally done after a function is executed in order to avoid having the annunciators updated. Remember that a NOP is required after the POWOFF instruction.

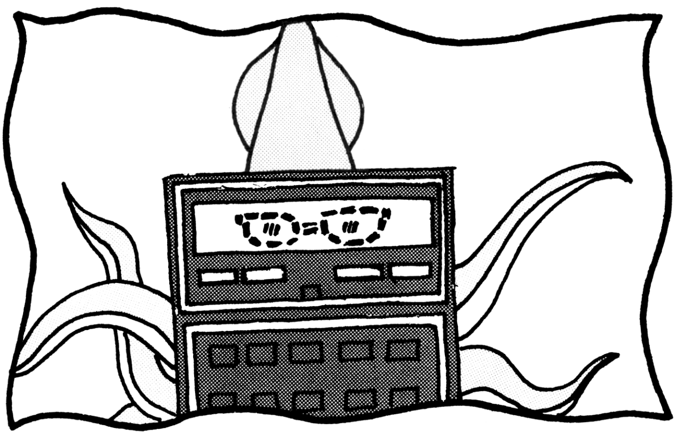
When the DISTEST routine is executed every display segment will have been lit up. You can amaze your friends with this little routine.

For those of you with the new display (the one with rounded edges) HP has added a new peripheral address, hex 10. This allows you to make use of six new READ/WRIT commands. Two of these, READ 5(M) and WRIT 5(M), are extremely useful. When peripheral 10 is selected these instructions read or write the contrast nybble of the display to or from digit zero of C. This allows you to control the contrast of the display. The default setting is 5. Here's an example of how to change the contrast setting.

Hexcode	Mnemonic	Description
130	LDI S&X	Load the address of a nonexistent RAM chip and
010	HEX: 010	the new peripheral.
270	RAMSLCT	Deselect RAM and Select the peripheral.
3F0	PRPH SLCT	
130	LDI S&X	Load in a value for the contrast. Let's try 0.
000	HEX: 000	
168	WRIT 5(M)	Write the zero to the contrast nybble.
3E0	RTN	Return.

The display should become very dim, except when viewed from a shallow angle. Place 00F in place of the 000 and see what happens. The display should become very dark. If nothing happens when you execute this routine, then you have an older display that does not have this feature.

The other READ/WRIT commands are not fully understood at this time. However it is known that the WRIT 15(e) instruction with this peripheral selected will crash the display, simultaneously lighting all segments, including the comma tails. The only way to recover from this particular crash is to remove the batteries for about one minute and then replace them.



A **SKWID** display test.

Our next routine will be a little more useful. It's a base conversion routine. This little beauty will convert a decimal number in X into a number of base b. The answer for the base b will end up in the display. Any base from two to thirty-six may be used. Sorry, but for bases over thirty-six we run out of letters in the alphabet. This base number is put into Y and the decimal number to be converted is put into X. Since the answer comes out in the display it will be lost if you clear the display.

The algorithm for this routine is taken from the PPC ROM routine "TB". This routine converts base ten to base b. First we compute X MOD Y. This gives us the value of the rightmost digit of the base b number. This number is then output to the display. Then we divide X, the decimal number, by Y, the base b, and take the integer of the result to get rid of the remainder that we already stripped off using the MOD function. We then check to see if we are at zero and jump back to the beginning of the loop if zero has not been reached. The routine is called 10-BASE.

"10-BASE"

Address	Hexcode	Mnemonic	Description
8258	085	"E"	Routine name.
8259	013	"S"	
825A	001	"A"	
825B	002	"B"	
825C	02D	"-"	
825D	030	"0"	
825E	031	"1"	
825F	0B8	READ 2(Y)	First we read Y and place it into A and
8260	10E	A=C ALL	then get X and put it into C. We then
8261	0F8	READ 3(X)	check to see if either of them contain
8262	355	?NC XQ	Alpha data (call to 14D5). If so, the
8263	050	14D5	mainframe call will exit to the ALPHA DATA
		[unlabeled]	error message. At the end of this routine
			Y is in C and X is in A. This routine

			also sets decimal mode so that we may do decimal number manipulations.
8264 8265 8266 8267 8268	088 0ED 064 0A8 260	SETF 5 ?NC XQ 193B [INTFRC] WRIT 2(Y) SETHEX	Y is left in C by the routine at address 14D5. So then we take the integer of this and write it out to Y. This ensures that this number will be an integer. If it is not an integer the rest of the routine will not work correctly. The ?NC XQ 193B calls the integer/fraction routine in the mainframe ROMs. Flag 5 must be set to get the integer portion of the number in C. (the fractional part is taken when flag 5 is clear.) Hex mode is then selected.
8269 826A 826B 826C 826D 826E 826F 8270 8271 8272 8273 8274	38D 008 266 2E6 0B5 0A2 106 130 024 306 0B5 0A2	?NC XQ 02E3 [BCDBIN] C=C-1 S&X ?C≠0 S&X ?NC GO 282D [ERRDE] A=C S&X LDI S&X HEX: 024 ?A<C S&X ?NC GO 282D [ERRDE]	Since we have the base number in C, we can convert it to binary in S&X of C. Then one is subtracted and we see if the S&X field of C is equal to zero. If it is, the carry will not be set and we go to the DATA ERROR message since a base of one is not valid. If we get through this we save the base b-1 number in A. We then load one greater than the highest allowable base minus one (37-1 dec. or 25-1 in hex). Then we compare these two numbers to see if the base b number is greater than 36. If it is, the carry will not be set and we go to the DATA ERROR error message.
8275 8276 8277	130 00C 268	LDI S&X HEX: 00C WRIT 9(Q)	Now we load the digit counter into the Q register. If you remember, this register is used as a scratch register by the mainframe. All we have to do is make sure that none of the routines we call use this register for scratch. The hex number 00C is loaded into Q to count the number of

			characters loaded into the display. It is decremented each time a number is loaded into the display.
8278	3C1	?NC XQ	This call to the mainframe enables the display and then clears it (fills it with spaces). This does the same thing that we did at addresses 822F to 823C of the DIS-TEST routine. The only difference is that this only takes two words instead of fourteen.
8279	0B0	2CF0 [CLLCDE]	
827A	149	?NC XQ	This call to the mainframe ROMs disables the display and selects chip 0.
827B	024	0952 [ENCP00]	
827C	0F8	READ 3(X)	We retrieve X and set the CPU to decimal mode as required by the next steps.
827D	2A0	SETDEC	
827E	088	SETF 5	This is the beginning of the loop to convert the decimal number to base b. The first thing we do is take the integer of the number in C. The first time through this is done to make sure the number in X is an integer. The next time through, when we loop back, we get rid of the fractional portion of the number in C.
827F	0ED	?NC XQ	
8280	064	193B [INTFRC]	
8281	2FA	?C≠0 M	The mantissa is checked to see if it is zero. If it is not zero we skip over the mainframe GOTO so we may continue on with the routine. Otherwise, we go to the subroutine in the mainframe that sets the message flag (User flag 50, see page 95 for full details).
8282	201	?NC GO	
8283	00E	0380 [unlabeled]	
8284	158	M=C	First we save the decimal number in M so that we may use it later. Now we set up for the MOD function. We do a decimal MOD base b. To do this we put the decimal number into A and get the base b from Y.
8285	10E	A=C ALL	
8286	0B8	READ 2(Y)	
8287	070	N=C	
8288	171	?NC XQ	

8289	064	195C	This must be copied into N before entry
		[MOD10]	into the MOD routine.
828A	260	SETHEX	We now have the remainder of the decimal
828B	38D	?NC XQ	number in C. This is the number we want
828C	008	02E3	to convert to an LCD display character.
		[BCDBIN]	The representation of these characters are
828D	106	A=C S&X	the same as for the characters that you
828E	130	LDI S&X	use for the names of your functions. We
828F	030	HEX: 030	SETHEX since the BCD-BIN routine requires
8290	146	A=A+C S&X	this setting. Then we convert the decimal
			remainder to hex in S&X of C. This is
			saved in A so we may add 030 hex to it to
			get the LCD character representation of
			this number. The numbers are in row three
			and start at zero and work up to nine.
			This result ends up in A.
8291	130	LDI S&X	Now we will check to see if the number we
8292	03A	HEX: 03A	want to display is greater than 9. This
8293	306	?A<C S&X	would mean that the hexcode in A would be
8294	01F	JC +03	03A or greater. We load 03A into C and
8295	1C6	A=A-C S&X	check to see if A is less than C. If it
8296	166	A=A+1 S&X	is, we want to display a decimal number
			and skip the next two steps. If the num-
			ber we want is greater than 9, i.e. an
			Alpha character, we subtract 03A from it
			and add one to get the Alpha LCD repre-
			sentation of the number.
8297	3D9	?NC XQ	Now we enable the display but do not clear
8298	01C	07F6	it. We get the LCD character we want to
		[ENLCD]	write to the display into the S&X of C so
8299	0A6	A<>C S&X	that it may be written out to the left
829A	328	WRIT 12(b)	side of the display using the WRIT 12(b)
829B	149	?NC XQ	instruction. We then call the mainframe
829C	024	0952	routine to disable the display and select
		[ENCP00]	chip 0.

829D	278	READ 9(Q)	Now we shall decrement the display counter number that is kept in Q. If this number should reach zero we have twelve digits in the display. If we go through the loop again we will push the rightmost digit off the display. To prevent this we put a call to the OUT OF RANGE error message. This tells us that the number of digits wanted was larger than the display could hold. The carry will be set on the thirteenth time through the loop since we will be subtracting one from zero. Then we shall go to the error message. If we make it past the error message the decremented counter is restored to Q.
829E	266	C=C-1 S&X	
829F	289	?C GO	
82A0	003	00A2 [ERROF]	
82A1	268	WRIT 9(Q)	
82A2	2A0	SETDEC	Now we shall divide the decimal number by the base b number. This puts the remainder into the fractional portion of the number which is removed when we loop back. First we must set the CPU back to decimal mode so we may do a decimal divide. We get the decimal number from M and put it into A and put the base b into C. Then the divide routine in the mainframe ROMs is executed and we loop back to the start of the loop at address 827E.
82A3	198	C=M	
82A4	10E	A=C ALL	
82A5	0B8	READ 2(Y)	
82A6	261	?NC XQ	
82A7	060	1898 [DV2-10]	
82A8	2B3	JNC -2A	

Try this routine a few times. Place sixteen into Y and 999 into X. Then execute 10-BASE. The result in the display will be 3E7 pushed to the left of the display. Now if you hit the CLX button the characters in the display will be erased. The number in the X register will not be changed. If you hit the CLX button again then the number in the X register will be cleared. This routine does not provide for an input of zero in the X register. Don't forget to update the FAT before you try to execute this routine or you will get NONEXISTENT.

WRITING CUSTOM ERROR MESSAGES

This section will deal with how to place your own error messages into the display. For example, if the base b in the last routine is greater than 36, you might want to display the error message BASE > 36. This would be much better than using the DATA ERROR message, which is used for many other purposes by the HP-41 system. A customized message would also give you the exact problem with your inputs to the routine. In order to do this we will show you how to program a routine that will output a message of up to twelve characters to the display. Three instructions will be introduced. They are FETCH S&X, POP ADR, and GOTO ADR. First we will show you a sample of what you would have to do for setup to use the routine that displays the message for you. We will use the addresses starting at 8400 for our example. The message we will display in our example will be BASE > 36.

Address	Hexcode	Mnemonic	Description
8400	3A1	?NC XQ	This routine checks if user flag 25 is
8401	088	22E8	set; if this is the case we exit to a
		[ERRSUB]	Normal Function Return, otherwise we return and continue on with this error processing.
8402	379		This is the call to our subroutine that
8403	03C	GOSUB	will output the characters in the message
8404	020	8420	we wish to display. The characters are input immediately after the subroutine call.
8405	002	"B"	This is the first letter in the message we will display. Notice that the message is not in reverse order like the names of our routines.
8406	001	"A"	These are the second through the next to
8407	013	"S"	last letters. The hexcodes are just the

8408	005	"E"	LCD representation of the characters as presented on page 108.
8409	020	" "	
840A	03E	">"	
840B	020	" "	
840C	033	"3"	
840D	236	"6"	This is the last letter of our message. Notice that the leftmost digit in the hexcode has been set to 2. In our routine when bit nine is set, the leftmost hexcode digit is either 2 or 3. This signals to the routine that this word contains the last character to be displayed.
840E	201	?NC XQ	This mainframe routine enables chip 0, sets the message flag, and prints the message if the printer is in trace mode.
840F	070	1C80 [MSG105]	
8410	3ED	?NC GO	This routine checks if we need to back-step, due to an error while we were single-stepping or running a program, stops a running program, and computes a valid line number. It then exits to a Normal Function Return.
8411	08A	22FB [ERR110]	

Now we know how to set up for the routine but don't know how to get the message out to the display. This next little routine will send the characters out to the display and then left justify them.

Address	Hexcode	Mnemonic	Description
8420	3C1	?NC XQ	This is a call to the mainframe routine that enables the display and then clears it (fills it with spaces).
8421	0B0	2CF0 [CLLCDE]	
8422	1B0	POP ADR	This instruction places the return address from the GOSUB statement into nybbles 3 to 6 of C. This is the address of the first
8423	330	FETCH S&X	

			instruction after the GOSUB statement. This would be the "B" character. We then use the FETCH S&X instruction to get the hexcode of the instruction at the address in nybbles 3 to 6 of C. The hexcode for this instruction is placed into the S&X field of C. The FETCH S&X instruction is the beginning of the loop to output the characters to the display.
8424	23A	C=C+1 M	We now increment the return address by one so that we may get the next instruction if we loop back again to the FETCH S&X instruction.
8425	3E8	WRIT 15(e)	Now the character in the S&X of C is written out to the display using a nine bit transfer instruction. We then subtract one from the exponent sign. If the exponent sign is zero we get an underflow which sets the carry and we jump back. We subtract one again to see if the exponent sign was one. If this was the case then we will get an underflow which sets the carry and we jump back. If the carry still has not been set then we know the 9th bit was logic one and the character is the last in the message.
8426	276	C=C-1 XS	
8427	3E7	JC -04	
8428	276	C=C-1 XS	
8429	3D7	JC -06	
842A	130	LDI S&X	This loads the hexcode for the space character into C and then it is saved in A. This part of the routine will strip off the spaces to the left of the message if there are any. The contents of the ADR field of C is also saved in A.
842B	020	HEX: 020	
842C	10E	A=C ALL	
842D	31C	R= 1	Set pointer to 1 so we may compare digits 0 and 1 of A and C.
842E	3F8	READ 15(e)	This instruction reads the leftmost char-

			acter in the display into S&X of C and rotates the display left by one character. The character just read in becomes the rightmost character in the display.
842F	36A	A≠C R<	This is now compared to the hexcode for a space. If the two are equal we want to rotate the display so that the message will be moved toward the left and a space will be put at the right. Then we jump back to the READ instruction to get the next character. If A and C are not equal, we have hit a character that is not a space, i.e., the beginning of our message; we don't want to rotate this to the left of the display so we use the WRIT 14(d) instruction. This will write out the hexcode to the left of the display and shift all of the other characters right by one.
8430	3F3	JNC -02	
8431	3A8	WRIT 14(d)	
8432	0AE	A<>C ALL	Now we get the address of the next instruction, which we saved in nybbles 3 to 6 of register A, and push it into the PC register using the GOTO ADR instruction.
8433	1E0	GOTO ADR	

If you want to use this routine, you must change the call to the DATA ERROR message at address 8273. The new sequence should be put into the place of this call.

Address	Hexcode	Mnemonic	Description
8273	027	JC +04	If the carry is set by the preceding instruction (?A<C), we don't want to go to the error message. We jump over the error exit because the calculator will interpret
8274	365		
8275	08C	GOTO	
8276	000	8400	

the first two words as a ?NC XQ. If the carry is set, then this instruction will be skipped, but the third word of the relative GOTO will then be executed as an instruction. If the carry is not set, the JC instruction will be skipped and we shall go to the error message. **The rest of the routine must be moved down by two words. None of the instructions after the GOTO change, they are just moved down.**

Now try the 10-BASE routine with a base greater than 36 and the error message BASE > 36 should come into the display.

The mainframe ROMs have a routine that does almost the same thing as the routine that we wrote to display messages. There is one main difference between the routine we wrote and the one in the mainframe. With ours you may put characters from rows 10-13 of the LCD character table into the message at any point. With the one in the mainframe ROMs you may only have the last letter of the message from rows 10-13 of the LCD table. This is because the mainframe ROM routine only checks to see if the exponent sign (bits 8 and 9) of the character is not equal to zero. If it does not equal zero then the end of the message is reached. In our routine we check to see if bit 9 is set before we end our message. If bit eight is set and the middle digit is zero, then the character to be displayed will be from row 10 of the LCD table. This only occurs if we are using nine bit transfers. The character "a" would have the hexcode 101. Our routine also left justifies the message in the display. The mainframe routine at address 07EF leaves the message right justified. In order to use the routine at 07EF you just replace the GOSUB 8420 statement in the error message with the ?NC XQ 07EF instruction.

Well, that's all folks. I hope this book has helped to give you an insight into how to program in the native language of the 41. There are many routines that need to be programmed using MCODE because of the speed advantage or just because the desired result cannot be achieved using User code programming.

THE END

APPENDIX A-List of suppliers

You may obtain MCODE storage devices (MLDL) from the following organizations.

ERAMCO MLDL - ERAMCO Systems, Valentynkade 27-11,
NL-1094 SR Amsterdam, The Netherlands.

In the U.S.A. contact: PPC, P.O. Box 9599
Fountain Valley CA 92728-9599 USA.
phone 714-754-6226

or EduCalc Mail Store, 27953 Cabot Road,
Laguna Niguel CA 92677 USA.
phone 714-831-2637

PROTOCODER II - ProtoTECH Inc., P.O. Box 12104 Boulder, CO 80303 USA
Phone 303-499-5541

For the annotated listing of the HP-41 mainframe ROMs contact:

PPC, P.O. Box 9599
Fountain Valley, CA 92728-9599 USA.
phone 714-754-6226

or Zengrange LTD., Greenfield road,
GB-Leeds, WYORKS LS9 8DB, England.
phone 0532 489048

or Editions de Cagire, 77 rue de Cagire,
F-31100 Toulouse, France.

ZENROM: The ZENROM is a custom programmers module manufactured by Hewlett-Packard for Zengrange Ltd. It has the best dissassembler for MCODE to date. With this module you can key in any synthetic instructions from the keyboard without the help of key assignments. To obtain the ZENROM write to:

Zengrange Ltd., Greenfield Road,
GB-Leeds, WYORKS, LS9 8DB, England
Phone 0532 489048

In the United States: EduCalc Mail Store, 27953 Cabot Road,
Laguna Nigel CA 92667 USA.
phone 714-831-2637

or PPC, P. O. Box 9599,
Fountain Valley CA 92728-9599 USA.
phone 714-754-6226.

Information on EPROM boxes may be obtained from the following sources. Contact them for the dealer nearest you.

Corvallis MicroTechnology, Inc. 33815 Eastgate Circle, Corvallis OR 97333 USA. phone 503-752-5456

Hand Held Products, P.O. Box 2388, Charlotte, North Carolina 28211 USA
Phone 704-541-1380

Prototech Inc., P. O. Box 12104, Boulder, CO 80303 USA. Phone 303-499-5541

The ASSEMBLER 3 EPROM may be obtained from:
Deep Thinking Software C/O Michael Thompson, 24 Canterbury Road, Camberwell, Victoria 3124, Australia.

The DAVID ASSEMBLER EPROM may be obtained from:
ERAMCO Systems, Kromboomsloot 16-3
1011 GW Amsterdam, The Netherlands

Phi Trinh's LOADP software package may be obtained from:

Phi Trinh, P.O. Box 184, Rockport WA 98283 USA

Here are the two major English language Users' Groups that support the HP-41. For information on either one, send $1 or a self-addressed envelope with 3 ounces of postage to:

Club for HP Handheld Users, 2545 W. Camden Pl., Santa Ana, CA 92704 USA. Phone 714-754-7757, noon to 4 AM Pacific time. Publishes the CHHU Chronicle.

PPC, P.O. Box 9599, Fountain Valley, CA 92728-9599 USA. Phone 714-754-6226. Publishes the PPC Journal.

Other HP-41 Users' Groups include:

CCD (ComputerClub Deutschland),
Postfach 2129, D - 6242 Kronberg 2, West Germany.
Publishes PRISMA (German) supporting synthetic programming and MCODE.

PPC-Holland, c/o TH Boekhandel Prins, Binnenwatersloot 30, NL-2611 BK Delft, The Netherlands.

PPC-Melbourne, P.O. Box 512, Ringwood, Victoria 3134, Australia.
Membership enquiries: Editions du Cagire, 77 rue du Cagire, F-31100 Toulouse, France. Publishes PPC Technical Notes, supporting advanced synthetic programming and MCODE.

PPC-Toulouse, 77 rue du Cagire, F-31100 Toulouse, France.
Publishes PPC-T (French) supporting synthetic programming and MCODE.

PPC-UK, c/o Astage, Rectory Lane, GB - Windlesham, Surrey, GU20 6BW, England. Membership enquiries: c/o Dave Bundy, 9 Kings Court, Kings Avenue, GB - Buckhurst Hill, Essex, IG9 5LP, England. Publishes "Datafile" (English) supporting synthetic programming and beginning MCODE.

APPENDIX B - What's up on entry to an MCODE routine

Here we shall explain the status of the CPU upon entry to an XROM function. Here's the low down on what's up:

1.) CPU is set to hex mode.
2.) Pointer P is selected and set to 1. The value of Q is variable.
3.) Flags 48 to 55 of the user flag register are placed into ST. CPU flag 7 corresponds to user flag 48 and 0 to 55. This is called Status Set 0 (SS0). When this is contained in ST the User flag number may be calculated from a bit in ST by subtracting its number from 55 (i.e. status bit 5 is the message flag (50) since 50 = 55 - 5). Flags 1 and 2 can be assumed to be clear upon entry to an XROM function since they correspond to the pause and I/O flags (the pause flag is cleared whenever any function is executed).
4.) RAM chip zero is selected.
5.) G is equivalent to the first byte of the XROM instruction. This is Aj in hex, where j may range from 0 to 7. Therefore bit three is always clear upon entry to an XROM function. This is useful for partial key sequencing which will be explained in detail later.
6.) The address of the first line of the MCODE program is in nybbles 3 to 6 of C. Nybbles 12 and 13 are always zero.

If your function is executed as a global execute in a program (XEQ "ABCDEFG"), then some of the above are different. In particular, the pointer is set to 3 instead of 1, register G contains the ROM ID number (1 to 31), and it cannot be assumed that nybbles 12 and 13 of C are zero. You will not normally encounter this situation, because the instruction will change to an XROM when it is keyed into the program, unless the corresponding module is not present at that time.

APPENDIX ZZZzzz... - The 3 CPU modes

There are three principal CPU modes. They are Deep sleep (calculator is off), Light sleep (41 on but CPU not running; also known as standby mode.), and Running (41 is executing code). If the CPU PC is at address 0000 as the result of a POWOFF instruction, it is fixed there and the 41 is in light sleep or deep sleep, waiting for a key to be pressed. If the ON key is pressed while in deep sleep, the carry is set, providing for a branch to the deep sleep wakeup routine at 01AD. If any key is pressed while in standby mode, the carry flag is clear and the light sleep wakeup routine at 0180 is executed.

APPENDIX C - Other Advanced Stuff

In this section we shall cover the various keycodes used by the mainframe, and how to make your MCODE programs nonprogrammable and/or prompting. First we cover the special key tables.

The mainframe has three tables listed in its coding that define keycodes for different keyboards. They are the default function keyboard (this is used when an unassigned key is pressed), the ALPHA keyboard (used when we are in alpha mode), and the partial key table, which is consulted during a partial key sequence. There is also a table contained in the hardware of the microprocessor. Its values are placed into the KY register whenever a key is pressed. From these values two more key tables are computed. They are the logical key table and the assignment key table. The tables are shown on pages 149-150.

In order to make a MCODE function nonprogrammable (so the function will run instead of being inserted when executed in program mode), just make the first executable instruction of the function a NOP. For example, if the first line of the GE routine were a NOP and all of the rest of the code was pushed down by one word, you could execute "GE" in program mode and you would end up at line 000 of the last program in memory. It would not be inserted as a program line. We shall rename the routine and make it nonprogrammable. The new name is GEE.

"GEE"

Address	Hexcode	Mnemonic	Description
82AB	085	"E"	Name for GEE function.
82AC	005	"E"	
82AD	007	"G"	
82AE	000	NOP	This is the start of the routine. The address in the FAT points to this instruction.

82AF	378	READ 13(c)	This was the first instruction in the old routine. The rest of the routine is the same as before.
82B0	05A	C=0 M	
82B1	01C	R= 3	
82B2	0D0	LD@R 3	
82B3	0C4	CLRF 10	
82B4	2C8	SETF 13	
82B5	328	WRIT 12(b)	
82B6	3E0	RTN	

The address in the FAT points to the NOP instruction, not the READ 13(c) instruction. Now if you execute "GEE" in program mode you will end up at line 000 of the last program in memory; the instruction will not be inserted as a program line.

In order to allow a function to become prompting, the first and second letters of the program name have the leftmost digit of their hexcode set to something other than zero. For example, here is what the name for the COPY function in the calculator looks like.

Hexcode	Letter
099	"Y"
010	"P"
00F	"O"
103	"C"
first executable instruction	

Notice that leftmost digit of the hexcode of "C" is a one. This signals to the calculator that some kind of prompt is needed. This digit may also be a two or three. The leftmost digit in the second letter of the function can range from zero to three. Here is a chart of the different combinations that produce prompts.

Example	Leftmost digit of 1st Chr	2nd Chr	Type of prompt
SIN	0	-	If the leftmost digit of the first character of the name is zero, the second character is not looked at.
COPY	1	0	Alpha input only (null input okay).
DEL	1	1	Three digits or four by pressing EEX.
	1	2	Same as for COPY except null input is not accepted (hitting the ALPHA key twice while entering no letters).
FIX	1	3	Allows entry of a single digit, an indirect register, or indirect stack.
STO	2	0	Accepts two digit entries, indirect, indirect stack, and stack. When the +, -, *, or / key is pressed at the double prompt the function defaults to the storage arithmetic function.
ASTO	2	1	Same as above except the storage arithmetic part does not work.
FS?C	2	2	Allows two digit entries, indirect, or indirect stack.
	2	3	Same as above.
LBL	3	0	Allows non-null alpha input or two digit numbers.
XEQ	3	1	Accepts non-null alpha, indirect stack, stack, or two digits inputs.
	3	2	Allows two digit input or non-null alpha.
GTO	3	3	Accept two digit entries, non-null alpha, indirect, indirect stack. If the decimal key is pressed while there are two prompts showing, the function changes to GTO . _ _ _

For numeric entries the hex equivalent of the number entered is put into the S&X field of CPU register A. For example, if you entered 46 at the double prompt, then 02E would end up in S&X of A. For indirect inputs just add hex 80 to the hex value of the number entered. NOTE: Stack suffixes (the ones that appear in the display as ST _) apply only to mainframe functions. These suffixes will not operate as might be expected in your XROM functions.

Alpha entries are placed into register Q of the status registers. They are put there in reverse order and right justified with unfilled places being filled with 00 bytes . For example, if you filled in "QWERTY" at the prompt the Q register would look like the following: 00 Y T R E W Q. The 00 is the filler byte since there were only six letters entered.

Any function that uses one of these prompts should also be made nonprogrammable. If it is executed in program mode the function will be inserted as a program line, and the value keyed in at the prompt will be lost. Only mainframe functions can use that value when inserted in a program.

The prompts for the above functions are dictated by a process called partial key sequencing. This is an esoteric procedure that has not previously been documented. Very few people fully understand its intricacies. The leftmost hex digit of the first two characters of the name in these MCODE functions are called op bits. These are used by the mainframe to tell what kind of a prompting function is being executed. The op bits for the first character are called op1, and the bits for the second character are called op2 (these are the leftmost hex digits in the first two characters of the name as previously described).

These op bits form part of a special pair of status bytes called PTEMP1 and PTEMP2. PTEMP2 is saved in register G during partial key sequence processing and in nybbles 3 and 4 of status register e during standby mode while in a partial key sequence. The eight bits of PTEMP2 are designated as follows:

Bit Description

0 Bit 0 of op2 (bit 8 of the second character of the function name).
1 Bit 1 of op2 (bit 9 of the second character of the function name).
2 Bit 0 of op1 (bit 8 of the first character of the function name).
3 This bit is always zero. Bit 1 of op1 initially accompanies the preceding 3 bits, but it is left in bit 3 of ST, before PTEMP2 is fully formed. Bit 1 of op1 is tested at that point, then it is no longer needed.
4 If this bit is set the function will be inserted as a line in a program. This is called the INSERT bit. Before setting this bit, the mainframe checks that you are in program mode and that the function is programmable.
5 This is the XROM bit indicating the function resides in a non-mainframe ROM. This bit only affects numeric entries. When clear, it indicates that the numeric entry value from the S&X field of A is to be merged with the function code as the postfix of a mainframe function. When the XROM bit is set, the value is left in S&X of A for use by the XROM program.
6 This is the IND bit. When set, hex 80 is added to the number in S&X of A. This bit's use is associated with the partial key sequencing of mainframe functions using an indirect operand.
7 This bit is unused by PTEMP2.

PTEMP1 is formed by setting aside the rightmost digit of the corresponding key from the partial key table, and multiplying the two leftmost digits by 4. Bits 0 to 3 of PTEMP2 are then added to this value. Note that there is no overlap in this addition, since the middle digit of the key table entry is always divisible by two, and since bit 3 of PTEMP2 is always zero. From this we get the following definitions for the 8 bits of PTEMP1:

Bit Description

0 This is bit 0 from PTEMP2 (bit 0 of op2).
1 Bit 1 of PTEMP2 (bit 1 of op2).

2 Bit 2 of PTEMP2 (bit 0 of op1).
3 If a digit key was pressed then this bit will be set. This is for digits 0 to 9.
4 If a key from row one or two of the keyboard (A through J) was pushed then this bit will be set.
5 When the ALPHA mode key is pressed this bit is set.
6 This bit is set when the SHIFT key is pushed.
7 When the decimal point is pressed this bit is set.

Upon return from a partial key sequence keystroke, PTEMP1 is in register ST, PTEMP2 is in register G, the rightmost digit of the keycode from the partial key table is in the mantissa sign of A, and the keycode from the logical key table is in nybbles 1 and 2 of register N.

In order to write your own partial key sequencing routine you must merely ensure that bit three of register G is zero upon entry. The rest of PTEMP2 is generally meaningful only for functions whose prompting is dictated by the op bits in its name, and can usually be ignored when setting up partial key sequences in the coding of an MCODE program. There are four entry points used for this purpose. They are at 0E45, 0E48, 0E4B, and 0E50. Upon entry to these locations the display must be enabled. These addresses must be called as a subroutine so control can be returned to your program once a key has been pressed. Now we shall describe each entry point.

Address	Description
0E45 [NEXT1]	This entry appends a single underscore to the display. The display is then left justified. The FIX instruction is an example of a single underscore function.
0E48 [NEXT2]	Here two underscores are appended to the display before left justification takes place. The STO function is an example of this type of prompt.
0E4B [NEXT3]	Three underscores are placed into the display by this entry point. The display is then left justified. The DEL instruction is an example of this type of prompt.

0E50 [NEXT] This entry point does not append an underscore to the display. The display must have at least one character present which is not a space, otherwise the left justify routine will go into an infinite loop since it looks for a non-space character.

These routines set the partial key (46) flag and the message flag (50). (Setting the message flag turns out to be unnecessary in this particular case.) They then update the annunciators in case the ALPHA key was pressed in preparation for entry of a function name or the SHIFT key was pressed during entry of the characters of a function name. Finally the keyboard is reset, and we go into standby mode.

When a key is pressed, the calculator starts executing code and figures out that we are in the middle of a partial key sequence (the partial key flag is set). The partial key table is then consulted in order to construct PTEMP1. Then the display is right-justified and all of the prompts (underscores) are removed. Finally a check is made to see if the backarrow key was pressed. If it was, a return is made to the step immediately following the execute statement of the partial key sequence routine. If some other key is pressed, the step immediately after the execute statement is skipped. Your program may now use PTEMP1 and the contents of the mantissa sign of A (and/or the logical keycode in nybbles 1 and 2 of register N), to figure out which key was pressed and go off and do the appropriate stuff. If you have a multiple prompt you will want to place the pertinent character into the display and call one of the above routines which appends one less prompt than was previously in the display. When you are finished prompting for input you should execute the routine at 0385 to clear the message flag (50) and the partial key flag (46) in order to tell the calculator you are no longer in a partial key sequence.

We now introduce a program which uses one of the partial key sequence entry points. It is a routine for entering non-normalized numbers directly from the keyboard. The 0-9 and A-F keys are reassigned to allow them to be executed from an unshifted keyboard. The routine places the ASCII digits into alpha and then codes the rightmost fourteen characters into X upon exit. This routine was written by Clifford Stern. It is called HXENTRY.

"HXENTRY"

Address	Hexcode	Mnemonic	Description
82B7	099	"Y"	Routine name
82B8	012	"R"	
82B9	014	"T"	
82BA	00E	"N"	
82BB	005	"E"	
82BC	018	"X"	
82BD	008	"H"	
82BE	345	?NC XQ	These first two executes clear the alpha register (10D1) and clear and enable the display (2CF0).
82BF	040	10D1 [CLA]	
82C0	3C1	?NC XQ	
82C1	0B0	2CF0 [CLLCDE]	
82C2	115	?NC XQ	Next a single underscore is pushed into the right of the display which is then left justified. Chip 0 is then enabled so the partial key sequence flag (46) and the message flag (50) can be set. The keyboard is then reset and we go into standby mode.
82C3	038	0E45 [NEXT1]	
82C4	07B	JNC +0F	If the backarrow key is pressed we return here and jump to a routine which deletes the rightmost character from both the display and the alpha register.
82C5	04C	?FSET 4	If flag 4 is set, a key from row 1 or 2 has been pressed. We jump to another flag test if the flag is clear.
82C6	11B	JNC +23	
82C7	35E	?A≠0 MS	If we make it to here a row 1 or 2 key has been pressed. The least significant digit
82C8	3D3	JNC -06	

			of the keycode (see partial key table on page 150) is placed into the mantissa sign of A. If it is zero, the J key has been pressed. Since this is not a hex digit we ignore the key and jump back to 82C2.
82C9	130	LDI S&X	Now we load a seven and rotate it into the mantissa sign of C so we may compare it to the number in the mantissa sign of A. This has the additional feature of clearing what is now digits zero and one of C.
82CA	007	HEX: 007	
82CB	33C	RCR 1	
82CC	31E	?A<C MS	If the key pressed is not less than G (7) then we ignore it and jump back to 82C2.
82CD	3AB	JNC -0B	
82CE	0BE	A<>C MS	If we get to here we know that a key from A to F has been pressed. First we place the least significant digit of the keycode from the partial key table into nybble 0 of C. Then we send it to the right end of the display. The partial key sequence routine leaves the pointer set to one so we may load a 4 to obtain the ASCII equivalent. We then jump to the code that appends this to alpha.
82CF	2FC	RCR 13	
82D0	3E8	WRIT 15(e)	
82D1	110	LD@R 4	
82D2	0EB	JNC +1D	
82D3	3B8	READ 14(d)	This is where we jump to if the backarrow key was pressed. Upon return from a partial key sequence the display is right justified and the prompts are deleted. Therefore the character we want to remove is the rightmost in the display. The READ 14(d) instruction rotates the display right by one character. When we return to 82C2 a prompt is pushed into the right of the display and the character to be deleted is shifted off the display.
82D4	149	?NC XQ	First chip 0 is enabled and the display is

82D5	024	0952 [ENCP00]	disabled. The pointer has been left at one upon exit from the partial key sequence routine. What is now done is to delete the rightmost character from the alpha register. This is done by successive manipulation of the first and last digits of each register of alpha. We then jump down to a point that enables the display and goes back to 82C2.
82D6	238	READ 8(P)	
82D7	10E	A=C ALL	
82D8	1F8	READ 7(O)	
82D9	0AA	A<>C R<	
82DA	23C	RCR 2	
82DB	2F0	WRITE DATA	
82DC	1B8	READ 6(N)	
82DD	0AA	A<>C R<	
82DE	23C	RCR 2	
82DF	2F0	WRITE DATA	
82E0	178	READ 5(M)	
82E1	04A	C=0 R<	
82E2	0AA	A<>C R<	
82E3	23C	RCR 2	
82E4	2F0	WRITE DATA	
82E5	0AE	A<>C ALL	
82E6	23C	RCR 2	
82E7	228	WRIT 8(P)	
82E8	073	JNC +0E	
82E9	00C	?FSET 3	This is where we end up if the key that is pressed is not a key from row 1 or 2. If flag 3 is set then a numeric key was pressed. If a numeric key was not pressed then we go to a point to check if the decimal point was pressed.
82EA	07B	JNC +0F	
82EB	0BE	A<>C MS	Now we know a numeric key has been pressed. The number is retrieved from the mantissa sign of A and rotated into nybble zero of C and a three is loaded into nybble 1. This is then written out to the right of the display. We use an eight bit display transfer since we can't depend on nybble 2 being even.
82EC	2FC	RCR 13	
82ED	0D0	LD@R 3	
82EE	368	WRIT 13(c)	

82EF	058	G=C	This is the place we enter to append characters to alpha. The pointer is now zero so nybbles zero and one of C are saved in G. We then enable chip 0 and disable the display (0952). The append routine (2D14) takes the contents of G and places it as the last character in alpha.
82F0	149	?NC XQ	
82F1	024	0952 [ENCP00]	
82F2	051	?NC XQ	
82F3	0B4	2D14 [APNDNW]	
82F4	042	C=0 @R	The purpose of this pair of instructions is to clear bit 3 of register G. This will provide for PTEMP1 to be correct upon return from the next execution of partial key sequencing.
82F5	058	G=C	
82F6	3D9	?NC XQ	We now enable the display so that we may return to address 82C2.
82F7	01C	07F6 [ENLCD]	
82F8	253	JNC -36	
82F9	28C	?FSET 7	This routine may be inserted as a line in a program. If we are in a running program the R/S key will halt digit entry and the program will continue. However if the decimal key is pressed the program will be terminated. If flag 7 is set the decimal key was pressed. CPU flag 13 is cleared in order to halt a running program. We then go on to finish the routine.
82FA	01B	JNC +03	
82FB	2C4	CLRF 13	
82FC	03B	JNC +07	
82FD	130	LDI S&X	If flag 7 is not set then a key other than a hex entry or the decimal point has been pressed. We shall now check if the R/S key was pushed. We load the logical keycode of R/S into nybbles one and two of C then transfer this to A. The logical keycode for the key that was pressed is in nybbles one and two of N. We retrieve this into C and they are compared. If the R/S key was pressed we continue on with
82FE	370	HEX: 370	
82FF	106	A=C S&X	
8300	0B0	C=N	
8301	366	?A≠C S&X	
8302	207	JC -40	

			the routine. Otherwise, we ignore the key
			and jump back to 82C2.
8303	3D9	?NC XQ	The display is cleared (2CF6) to clean it
8304	0B0	2CF6	up. The keyboard is then reset (0098).
		[CLRLCD]	This is just waiting for the release of
8305	261	?NC XQ	the key. If this is not done the routine
8306	000	0098	could finish and the function on the
		[RSTKB]	depressed key would be executed.
8307	149	?NC XQ	Chip 0 is now enabled and the display is
8308	024	0952	disabled (0952). The message (50), and
		[ENCP00]	the partial key sequence (46) flags are
8309	215	?NC XQ	cleared (0385). User flags 48 to 55 are
830A	00C	0385	loaded into register ST.
		[RSTSQ]	
830B	130	LDI S&X	This value is used to CODE the rightmost
830C	049	HEX: 049	fourteen digits of alpha. We shall now
830D	23C	RCR 2	rotate these digits into nybbles 12 and 13
830E	0EE	C<>B ALL	of register C. They are then transferred
			to register B so we may do a series of
			comparisons and possible additions with
			register A.
830F	35C	R= 12	The pointer is set to 12 so we may add the
			two nybbles in A and B when an alphabetic
			character is processed.
8310	00E	A=0 ALL	Clear what will become the accumulator
			register. If there are fewer than eight
			characters in alpha the inner loop won't
			be executed 14 times so we must have
			leading zeros in C to account for this.
8311	1B8	READ 6(N)	Characters eight through fourteen are
			placed into C so we may begin coding them.
8312	0AE	A<>C ALL	This is the beginning of the outer loop.
			The contents of C are either status regis-
			ter M or N.
8313	33E	?A<B MS	We now check to see if we have an alpha

8314	017	JC +02	character, or instead a digit character or a null byte. If the mantissa sign of A is less than four the latter is the case (the most significant hex digit of an alpha character is four). If that is true then we skip the addition step because the least significant digit of that byte is the correct hex equivalent. For alpha hex numbers a nine must be added to this digit to correct it (i.e. A is 41 in ASCII and we add 9 to get 4A which sets the right-most digit to the character it represents). This is the start of the inner loop.
8315	122	A=A+B @R	
8316	3EE	LSHFA ALL	The A register is shifted left to discard the left nybble of the character just examined. This places the desired digit in the mantissa sign of A. We now place this into C and shift A left again to bring up the next character to be coded. The digit in the mantissa sign of C is now rotated to the right end.
8317	0BE	A<>C MS	
8318	3EE	LSHFA ALL	
8319	2FC	RCR 13	
831A	34E	?A≠0 ALL	If there are more characters to be coded, A will not be equal to zero and we jump back to the start of the inner loop at address 8312.
831B	3C7	JC -08	
831C	30C	?FSET 1	If this is the first time through the loop this flag will be clear. We know this because status set zero was placed in register ST. Status set 0 is in ST as a result of the call to 0385, and flag 1 corresponds to the pause flag which is cleared by that routine. If it is set we jump to the end of the routine and finish up.
831D	02F	JC +05	

831E	308	SETF 1	Setting this flag tells us that this is the second time through the inner loop.
831F	10E	A=C ALL	The result from the first execution of the
8320	178	READ 5(M)	inner loop is temporarily saved in A so we
8321	38B	JNC -0F	may fetch the rightmost seven characters of alpha. We then jump back to the beginning of the outer loop at address 8312.
8322	0EE	C<>B ALL	The final value is in C and we save it in
8323	0B9	?NC GO	B as required by the routine at address
8324	04A	122E [RCL]	122E, which sends register B to X according to the status of the stack enable flag (CPU flag 11).

To use this routine execute HXENTRY. The program will place a single prompt in the left of the display. You may now press any key, with only the 0 to F keys entering digits into the display. The ON, R/S, and Decimal Point keys will terminate the routine. If the R/S key is pressed when the function was executed in a running program the program resumes running. With the decimal point the program is terminated. The backarrow key deletes the rightmost character in the display and alpha. All other keys are ignored.

We are providing another routine that executes just the CODE section of HXENTRY; the contents of alpha are coded into X. However, you must enter the alpha characters manually (or from a program) and then execute CODE. Here is the routine. It simply uses the CODE portion of HXENTRY to do all of the dirty work.

CODE

Address	Hexcode	Mnemonic	Description
8325	085	"E"	Routine name.
8326	004	"D"	
8327	00F	"O"	
8328	003	"C"	
8329	313	JNC -1E	This is a jump back to the CODE section of HXENTRY.

MAINFRAME KEY TABLES

Alpha Keyboard

	10C		10C	10C
61	62	63	64	65
41	42	43	44	45
7E	25	1D	3C	3E
46	47	48	49	4A
10E	7F	19A	19B	207
10E	4B	4C	4D	108
5E		D	24	187
4E		4F	50	0
2D	37	38	39	
51	52	53	54	
2B	34	35	36	
55	56	57	58	
2A	31	32	33	
59	5A	3D	3F	
2F	30	2E	17E	
3A	20	2C	105	

Default Function Table

	10C		10C	10C
148	153	151	157	155
147	160	152	156	150
170	14C	15C	15D	15E
171	175	159	15A	15B
10E	10F	1CF	1D0	107
10E	1E0	191	190	108
100		196	185	177
183		1C	1B	0
178	1A8	1A9	1AC	
141	17	18	19	
146	186	14E	14F	
140	14	15	16	
145	19C	19D	19E	
142	11	12	13	
167	172	176	198	
143	10	1A	105	

Logical Keycodes

	46		45	44
08	18	28	38	48
00	10	20	30	40
09	19	29	39	49
01	11	21	31	41
0A	1A	2A	3A	4A
02	12	22	32	42
0B		2B	3B	4B
03		23	33	43
0C	1C	2C	3C	
04	14	24	34	
0D	1D	2D	3D	
05	15	25	35	
0E	1E	2E	3E	
06	16	26	36	
0F	1F	2F	3F	
07	17	27	37	

MORE MAINFRAME KEY TABLES

PARTIAL KEY TABLE

	000		000	080
041	042	043	044	045
046	047	048	049	040
100	000	000	000	000
000		000	000	00F
002	027	028		029
001	024	025		026
003	021	022		023
004	020	200		000

KEYCODES from KY

18	C6		C5	C4
10	30	70	80	C0
11	31	71	81	C1
12	32	72	82	C2
13		73	83	C3
14	34		74	84
15	35		75	85
16	36		76	86
17	37		77	87

ASSIGNMENT KEY TABLE

(top keys not assignable)

09	19	29	39	49
01	11	21	31	41
0A	1A	2A	3A	4A
02	12	22	32	42
0B	1B	2B	3B	4B
03	13	23	33	43
0C		2C	3C	4C
04		24	34	44
0D	1D	2D	3D	
05	15	25	35	
0E	1E	2E	3E	
06	16	26	36	
0F	1F	2F	3F	
07	17	27	37	
10	20	30	40	
08	18	28	38	

APPENDIX D - Using the Polling Points

You may remember when we were describing which words in a 4K page had been set aside for specific purposes, the words from addresses PFF4 to PFFA were off limits unless you knew exactly what you were doing. During certain specific times the 41 conducts a process called polling. This entails checking a fixed polling point in all ROMs from page 5 to F. In order to use these points several conditions must be observed. We shall now describe how these may be used. First, if there is any nonzero word in one of the polling point addresses and the calculator polls that address then it will branch there and start executing code. Usually we put a JNC that jumps to the start of the routine we wish to execute. The seven different polling points are polled at specified times. These times are given below.

Address	Description of poll
PFF4	This is the pause loop interrupt. Any time the calculator executes the PSE instruction this address is polled.
PFF5	This address is polled after any RPN function is executed, if user flag 53 or peripheral flag 13 is set. This includes execution of functions during a User code program, and is called the main running loop interrupt.
PFF6	This is polled when the calculator is turned on by something other than the ON button (for example, an alarm).
PFF7	This location is polled when the calculator is being turned off.
PFF8	This is polled whenever the calculator goes into standby mode, and is called the I/O interrupt.
PFF9	The calculator polls this address when it is turned on using the ON button.
PFFA	Whenever there is a MEMORY LOST this location is polled.

Once you have taken control by using one of these interrupts you MUST observe some rules.

Your routine must exit with the following intact:

1.) Restore nybbles 10 through 3 of register C to what they were when you took control at the interrupt.
2.) Have P as the selected pointer.
3.) Load flags 48 to 55 of the user flag register into CPU register ST. This set of flags is called status set zero (SS0).
4.) Have chip 0 (the status registers) selected.
5.) The CPU must be in HEX mode.
6.) You must do a GOTO to 27F3 to end the interrupt and give control back to the calculator so that it may continue polling.

If any of these rules are not observed the calculator could end up doing some strange things (like locking up the keyboard). To clarify this mess we shall do an example. In our example we shall use the MEMORY LOST interrupt. Whenever a MEMORY LOST occurs we shall resize the calculator to a size of 25 instead of the normal 273 (CV) or 100 (CX). Here is the routine.

Address	Hexcode	Mnemonic	Description
8FE8	268	WRIT 9(Q)	This is the entry to our routine. The first thing we do is save register C in Q so that we may retrieve it later as required.
8FE9	130	LDI S&X	We shall now load the size (25 in decimal)
8FEA	019	HEX: 019	into S&X of C and then transfer it to A.
8FEB	106	A=C S&X	This is done because the size routine requires the specified size to be in A (remember SIZE is a prompting function).
8FEC	244	CLRF 9	We shall now call the routine in the main-
8FED	259	?NC XQ	frame that changes the size. Flag 9 is
8FEE	05C	1796	cleared in case we should get an error. If we get an error, the routine will just return and do nothing if flag 9 is cleared. If it were set we would go to the

			PACKING error message and would not be able to return control to the polling process.
8FEF	25D	?NC XQ	This entry point selects chip zero, and
8FF0	01C	0797	then places the user flag register into C.
		[LDSST0]	Flags 48 to 55 are then placed into the ST register.
8FF1	278	READ 9(Q)	Now we retrieve the original contents of C upon entry to the poll.
8FF2	3CD	?NC GO	We then exit back to the mainframe after
8FF3	09E	27F3	having satisfied all of the described conditions. The size routine does not change the selected pointer so we didn't have to do anything about that.

Now we shall place the jump from the MEMORY LOST interrupt location at 8FFA to the beginning of our routine which is at 8FE8, by using a JNC -12 (hexcode 373). Always remove the word at the interrupt location before you modify the routine that uses the interrupt. After you have updated the routine make sure that the interrupt jumps back to the correct place or you could lose control of the calculator when the interrupt is polled.

If you happen to place the jump to a wrong location and the calculator goes crazy, try the following: unplug you MLDL and regain control of the calculator. Now change the selected page of your MLDL to page 2. Then write NOPs (000) to all of the interrupt locations (2FF4-2FFA). You may now place your MLDL back to the original page.

APPENDIX E - MCODE Debugging Program

Clifford Stern has written a program to allow you to interrupt your MCODE routine. This routine saves the contents of all the CPU registers at the point of interruption in the RAM of the calculator. The 16 status registers are also saved away. The name of the routine is BREAK.

To use BREAK you must have the address of the point you wish to insert the breakpoint in X. Place it there using HXENTRY (example, for address 8967 press the 8, 9, 6, and 7 keys at the prompt and then press R/S). Then execute BREAK. The breakpoint is inserted automatically by the program and user flag 1 will be set. Flag 1 should be cleared before you execute BREAK. You must be sure that the carry is not set by the instruction immediately preceding the breakpoint. This is because the BREAK routine writes an ?NC GO to the debugging routine. Now load the appropriate data and execute the function to be debugged. When the breakpoint is reached during execution of your function, the CPU and status registers are written into the last 25 data registers of the calculator RAM (1E7-1FF), the original program bytes are restored, and flag 1 is cleared. The routine assumes that you have a 41CX, 41CV, or a 41C with a quad memory module. If the number of data registers available is less than 25 then BREAK exits to the NONEXISTENT error message. If flag 1 is still set when the routine finishes (crashes?) the breakpoint was not reached. To restore the original bytes just clear X and execute BREAK. Registers 1FE and 1FF are reserved for use by the BREAK program, and must not be altered by the routine being debugged.

The Data is saved in the RAM registers in the order shown on the next page.

```
abs.              Contents of register by nybble
reg.#    13   12   11   10    9    8    7    6    5    4    3    2    1    0

487       0    0   <---RTN #3--->  <---RTN #2--->    <---RTN #1--->

488      <-KY->  <---RTN #4--->  <-XY->   P    Q   <-G->    <-ST->
  Detail of XY:
  BIT #     7   6   5   4   3   2   1   0   v=0/1 denotes hex/dec mode
  FLAG # 13  12  11  10   9   8   v   w   w=0/1 denotes SLCT P/Q

489      <------------------ CPU REGISTER C --------------------->
490      <------------------ CPU REGISTER A --------------------->
491      <------------------ CPU REGISTER B --------------------->
492      <------------------ CPU REGISTER M --------------------->
493      <------------------ CPU REGISTER N --------------------->

494      <----------------- STATUS REGISTER T ------------------>
495      <----------------- STATUS REGISTER Z ------------------>
496      <----------------- STATUS REGISTER Y ------------------>
497      <----------------- STATUS REGISTER X ------------------>
498      <----------------- STATUS REGISTER L ------------------>
499      <----------------- STATUS REGISTER M ------------------>
500      <----------------- STATUS REGISTER N ------------------>
501      <----------------- STATUS REGISTER O ------------------>
502      <----------------- STATUS REGISTER P ------------------>
503      <----------------- STATUS REGISTER Q ------------------>
504      <----------------- STATUS REGISTER ⊦ ------------------>
505      <----------------- STATUS REGISTER a ------------------>
506      <----------------- STATUS REGISTER b ------------------>
507      <----------------- STATUS REGISTER c ------------------>
508      <----------------- STATUS REGISTER d ------------------>
509      <----------------- STATUS REGISTER e ------------------>
510                               <breakpoint ADR> <break word>
511                               < break ADR +1 > <break word>
```

In order to examine this output use the following User-code routine. The DECODE function is given after the listing for BREAK. It decodes the contents of X into its hexidecimal representations and puts the result into alpha and the display. The program is called "RR". To view the contents of the desired register just place the absolute address in X and XEQ "RR". The hexidecimal representation of the contents of the desired register will be viewed, and printed if possible. Just hit R/S to examine each successive register.

```
LBL "RR"
NR              This is the non-normalized recall from our sample ROM.
DECODE          This routine is listed at the end of this appendix.
PROMPT
LASTX
1
+
GTO "RR"
END
```

In order to efficiently use BREAK you should use the following short User-code program.

```
LBL "?"
HXENTRY   Enter the address at which you wish to insert the breakpoint.
BREAK
  .
  .         This is where you place the steps to load the data for your
  .         function. Then place the function after the data is loaded.
  .
487       This number points to the lowest register in which data is saved
          by BREAK. It may be changed to start at any other register you
          wish to examine.
GTO "RR"
END
```

After assigning "?" to a key, this routine can be used to efficiently probe for errors in an MCODE program. To view the contents of the display at the breakpoint, set user flag 2 and place a STOP instruction before the 487 program line.

There are two values that the BREAK program does not give you. They are the value of the RAMSLCT pointer and the contents of register T. In order to obtain these values a second program was integrated into the BREAK program. It is called RSCLT. This routine uses the breakpoint location that was used by the last execution of BREAK. So BREAK must be executed before RSLCT is used. The results from RSLCT are placed in the X register. The RAMSLCT value is in the S&X field and the contents of register T are placed into nybbles 3 and 4. If the selected RAM register is nonexistent, the S&X field of X will be set to FFF. To use this function just execute RSLCT and then load the same data used for the previous execution of BREAK. Now execute the function you are debugging. To view the results of RSLCT just execute DECODE. The system RSLCT uses to compute the RAMSLCT value was pioneered by Paul Cooper.

Another routine we are providing for your programming pleasure is called LOOP. This function allows you to debug a loop within a program. You can execute the loop a specified number of times before the debugging routine dumps the CPU registers to RAM for inspection.

In order to use this routine you must be a genius on the order of Albert Einstein (just kidding). The number of times the breakpoint is bypassed is taken from the Y register. The address of the breakpoint is placed in X and is of the same format as for BREAK. The breakpoint location must be at a pair of NOPs since processing continues past the breakpoint a number of times. The LOOP routine uses one subroutine level and in addition utilizes the tone register (T) to store the loop counter. This precludes use of register T in your program and you cannot have more than three pending returns in the subroutine stack at the breakpoint. LOOP places the 41 into buzz mode (nonzero value in register T). If the debugging is not allowed to finish, the calculator can be removed from buzz mode by executing BEEP with flag 26 set.

LOOP requires two NOPs for its ?NC XQ to be inserted into your program. If this is not possible use the following procedure.

1.) Insert a jump to a location that contains the NOPs.
2.) Place the instruction that was replaced by the jump at the location to which you jump. Follow this instruction with two NOPs and then a jump to the step after the first jump instruction. Here's an example.

Address	Mnemonic	Description
Pabc	ABC	This is the instruction that was replaced by the first jump instruction.
XXXX	NOP	Here are the two NOPs.
XXXX	NOP	
XXXX	JNC +Pxyz	This is the second jump to the instruction after the first jump.
.		
.		
.		
Pxyy	JNC -Pabc	This is the spot where the first jump is placed and the jump goes to the spot where the instruction ABC is placed.
Pxyz	???	This is where the second jump goes to so the program may continue.

LOOP can be executed from the keyboard or a running program. An example of the later is given below.

LBL "??"	
RCL 00	This is the register containing the loop counter.
ISG 00	Increment the loop counter by one so the next time you execute this program the number of loops will be different.
NOP	Insert a NOP here. STO X for example.

"ABCD" This is the address where the LOOP breakpoint is to be placed.

CODE Code the address in the alpha register and push it onto the stack. The CODE routine is listed on page 148.

LOOP Execution of LOOP to insert the breakpoint and store the loop counter.

.

. As in BREAK this is where you place the steps to load data for

. your function. Then place the function after the appropriate

. data is loaded.

.

489 This number points to the first register you wish to view after the Nth iteration (N is in register 00) of the loop.

GTO "RR"

END

Simply assign "??" to a key and place a starting loop counter (such as zero) into register 00. Then press the assigned key repeatedly to obtain successive outputs from the loop.

LOOP and RSLCT are separable from the BREAK program, and can be omitted if desired. BREAK runs from 847A to 8545 in the following listing. The BREAK program must be present in order for RSLCT and LOOP to function.

Address	Hexcode	Mnemonic
8440	090	"P"
8441	00F	"O"
8442	00F	"O"
8443	00C	"L"
8444	0B8	READ 2(Y)
8445	38D	?NC XQ
8446	008	02E3 [BCDBIN]
8447	2F6	?C≠0 XS
8448	0B5	?C GO
8449	0A3	282D [ERRDE]
844A	358	ST=C
844B	258	T=ST
844C	308	SETF 1
844D	163	JNC +2C
844E	2D8	ST<>T<<<
844F	38C	?FSET 0
8450	01B	JNC +03
8451	384	CLRF 0
8452	12B	JNC +25
8453	388	SETF 0
8454	30C	?FSET 1
8555	01B	JNC +03
8556	304	CLRF 1
8457	103	JNC +20
8458	308	SETF 1
8459	20C	?FSET 2
845A	01B	JNC +03
845B	204	CLRF 2
845C	0DB	JNC +1B
845D	208	SETF 2
845E	00C	?FSET 3
845F	01B	JNC +03
8460	004	CLRF 3
8461	0B3	JNC +16
8462	008	SETF 3
8463	04C	?FSET 4
8464	01B	JNC +03
8465	044	CLRF 4
8466	08B	JNC +11
8467	048	SETF 4
8468	08C	?FSET 5
8469	01B	JNC +03
846A	084	CLRF 5
846B	063	JNC +0C
846C	088	SETF 5
846D	14C	?FSET 6
846E	01B	JNC +03
846F	144	CLRF 6
8470	03B	JNC +07
8471	148	SETF 6
8472	28C	?FSET 7
8473	01F	JC +03
8474	020	XQ>GO
8475	033	JNC +06
8476	284	CLRF 7
8477	2D8	ST<>T
8478	3E0	RTN
8479	16B	JNC +2D
847A	258	T=ST<<<
847B	3C4	ST=0
847C	3D8	C<>ST
847D	3F0	PRPH SLCT
847E	3D8	C<>ST
847F	308	SETF 1
8480	208	SETF 2
8481	008	SETF 3

Address	Hexcode	Mnemonic	Address	Hexcode	Mnemonic
8482	048	SETF 4	84A3	308	SETF 1
8483	33C	RCR 1	84A4	03C	RCR 3
8484	3D8	C<>ST	84A5	023	JNC +04
8485	2FC	RCR 13	84A6	173	JNC +2E
8486	270	RAMSLCT	84A7	23E	C=C+1 MS
8487	33C	RCR 1	84A8	3D4	R=R-1
8488	398	C=ST	84A9	394	?R= 0
8489	2FC	RCR 13	84AA	3EB	JNC -03
848A	268	WRIT 9(Q)	84AB	33C	RCR 1
848B	0AE	A<>C ALL	84AC	120	?P=Q
848C	2A8	WRIT 10(⊢)	84AD	03B	JNC +07
848D	0CE	C=B ALL	84AE	35C	R= 12
848E	2E8	WRIT 11(a)	84AF	0A0	SLCT P
848F	198	C=M	84B0	354	?R= 12
8490	328	WRIT 12(b)	84B1	06F	JC +0D
8491	0B0	C=N	84B2	388	SETF 0
8492	368	WRIT 13(c)	84B3	053	JNC +0A
8493	046	C=0 S&X	84B4	0E0	SLCT Q
8494	1B0	POP ADR	84B5	394	?R= 0
8495	07C	RCR 4	84B6	01B	JNC +03
8496	1B0	POP ADR	84B7	388	SETF 0
8497	07C	RCR 4	84B8	0A0	SLCT P
8498	1B0	POP ADR	84B9	23E	C=C+1 MS
8499	27C	RCR 9	84BA	3D4	R=R-1
849A	1E8	WRIT 7(Q)	84BB	394	?R= 0
849B	1A0	A=B=C=0	84BC	3EB	JNC -03
849C	298	ST=T	84BD	35C	R= 12
849D	3D8	C<>ST	84BE	0D8	C<>G
849E	258	T=ST	84BF	23C	RCR 2
849F	27E	C=C-1 MS	84CO	38C	?FSET 0
84A0	260	SETHEX	84C1	01F	JC +03
84A1	23E	C=C+1 MS	84C2	2DC	R= 13
84A2	017	JC +02	84C3	3D4	R=R-1

Address	Hexcode	Mnemonic
84C4	098	C=G
84C5	10C	?FSET 8
84C6	013	JNC +02
84C7	208	SETF 2
84C8	24C	?FSET 9
84C9	013	JNC +02
84CA	008	SETF 3
84CB	0CC	?FSET 10
84CC	013	JNC +02
84CD	048	SETF 4
84CE	18C	?FSET 11
84CF	013	JNC +02
84D0	088	SETF 5
84D1	34C	?FSET 12
84D2	023	JNC +04
84D3	013	JNC +02
84D4	1A3	JNC +34
84D5	148	SETF 6
84D6	2CC	?FSET 13
84D7	013	JNC +02
84D8	288	SETF 7
84D9	398	C=ST
84DA	2FC	RCR 13
84DB	1B0	POP ADR
84DC	07C	RCR 4
84DD	220	C=KEY
84DE	3C8	CLRKEY
84DF	0BC	RCR 5
84E0	228	WRIT 8(P)
84E1	130	LDI S&X
84E2	1EE	HEX: 1EE
84E3	0E6	C<>B S&X
84E4	39C	R= 0
84E5	0A6	A<>C S&X
84E6	270	RAMSLCT
84E7	106	A=C S&X
84E8	038	READ DATA
84E9	0EE	C<>B ALL
84EA	270	RAMSLCT
84EB	226	C=C+1 S&X
84EC	0EE	C<>B ALL
84ED	2F0	WRITE DATA
84EE	162	A=A+1 @R
84EF	3B3	JNC -0A
84F0	3F8	READ 15(e)
84F1	106	A=C S&X
84F2	330	FETCH S&X
84F3	0A6	A<>C S&X
84F4	040	WRIT S&X
84F5	0A6	A<>C S&X
84F6	3E8	WRIT 15(e)
84F7	3B8	READ 14(d)
84F8	106	A=C S&X
84F9	330	FETCH S&X
84FA	0A6	A<>C S&X
84FB	040	WRIT S&X
84FC	0A6	A<>C S&X
84FD	2F0	WRITE DATA
84FE	046	C=0 S&X
84FF	270	RAMSLCT
8500	215	?NC XQ
8501	00C	0385 [RSTSQ]
8502	2FC	RCR 13
8503	358	ST=C
8504	20C	?FSET 2
8505	027	JC +04

Address	Hexcode	Mnemonic
8506	208	SETF 2
8507	01B	JNC +03
8508	093	JNC +12
8509	204	CLRF 2
850A	398	C=ST
850B	33C	RCR 1
850C	2F0	WRITE DATA
850D	20C	?FSET 2
850E	027	JC +04
850F	30C	?FSET 1
8510	205	?C XQ
8511	00D	0381
8512	3C1	?NC GO
8513	002	00F0 [NFRPU]
8514	2F3	JNC -22
8515	08B	"K"
8516	001	"A"
8517	005	"E"
8518	012	"R"
8519	002	"B"
851A	130	LDI S&X
851B	1E7	HEX: 1E7
851C	106	A=C S&X
851D	378	READ 13(c)
851E	03C	RCR 3
851F	306	?A<C S&X
8520	381	?C GO
8521	00B	02E0 [ERRNE]
8522	0F8	READ 3(X)
8523	1BC	RCR 11
8524	130	LDI S&X
8525	1FE	HEX: 1FE
8526	270	RAMSLCT
8527	2FA	?C≠0 M
8528	243	JNC -38
8529	130	LDI S&X
852A	0B9	HEX: 0B9
852B	30C	?FSET 1
852C	01B	JNC +03
852D	130	LDI S&X
852E	0E5	HEX: 0E5
852F	286	C=0-C S&X
8530	10E	A=C ALL
8531	35D	?NC XQ
8532	000	00D7 [PCTOC]
8533	03C	RCR 3
8534	206	C=C+A S&X
8535	2FC	RCR 13
8536	3C6	RSHFC S&X
8537	1E6	C=C+C S&X
8538	1E6	C=C+C S&X
8539	226	C=C+1 S&X
853A	1FA	C=C+C M
853B	1FA	C=C+C M
853C	30C	?FSET 1
853D	01F	JC +03
853E	23A	C=C+1 M
853F	23A	C=C+1 M
8540	106	A=C S&X
8541	03C	RCR 3
8542	0AE	A<>C ALL
8543	2F0	WRITE DATA
8544	23A	C=C+1 M
8545	27B	JNC -31

Address	Hexcode	Mnemonic
8546	094	"T"
8547	003	"C"
8548	00C	"L"
8549	013	"S"
854A	012	"R"
854B	130	LDI S&X
854C	1FE	HEX: 1FE
854D	270	RAMSLCT
854E	038	READ DATA
854F	130	LDI S&X
8550	020	HEX: 020
8551	2FB	JNC -21
8552	293	JNC -2E
8553	038	READ DATA<<<
8554	158	M=C
8555	1A0	A=B=C=0
8556	3F0	PRPH SLCT
8557	21C	R= 2
8558	310	LD@R C
8559	0E6	C<>B S&X
855A	260	SETHEX
855B	26E	C=C-1 ALL
855C	29C	R= 7
855D	010	LD@R 0
855E	2F0	WRITE DATA
855F	10E	A=C ALL
8560	0C6	C=B S&X
8561	270	RAMSLCT
8562	226	C=C+1 S&X
8563	05F	JC +0B
8564	0E6	C<>B S&X
8565	038	READ DATA
8566	36E	?A≠C ALL
8567	3CF	JC -07
8568	198	C=M
8569	2F0	WRITE DATA
856A	130	LDI S&X
856B	3FF	HEX: 3FF
856C	06E	A<>B ALL
856D	3B0	C=C AND A
856E	266	C=C-1 S&X
856F	03C	RCR 3
8570	270	RAMSLCT
8571	3C4	ST=0
8572	2D8	ST<>T
8573	398	C=ST
8574	1BC	RCR 11
8575	0E8	WRIT 3(X)
8576	05A	C=0 M
8577	2DB	JNC -25

Here's the DECODE routine, written by Clifford Stern. It places the ASCII equivalent of the contents of X into ALPHA, and suppresses leading zeros. The routine ends by viewing alpha and printing if in RUN mode. The method used to convert hex digits to ASCII characters was invented by Michael Thompson.

Address	Hexcode	Mnemonic	Address	Hexcode	Mnemonic
8578	085	"E"	8590	308	SETF 1
8579	004	"D"	8591	30C	?FSET 1
857A	00F	"O"	8592	033	JNC +06
857B	003	"C"	8593	062	A<>B @R
857C	005	"E"	8594	206	C=C+A S&X
857D	004	"D"	8595	362	?A≠C @R
857E	0F8	READ 3(X)	8596	013	JNC +02
857F	0EE	C<>B ALL	8597	222	C=C+1 @R
8580	2A0	SETDEC	8598	1BA	A=A-1 M
8581	04E	C=0 ALL	8599	38B	JNC -0F
8582	228	WRIT 8(P)	859A	20C	?FSET 2
8583	1E8	WRIT 7(O)	859B	027	JC +04
8584	01C	R= 3	859C	208	SETF 2
8585	190	LD@R 6	859D	1A8	WRIT 6(N)
8586	31C	R= 1	859E	31B	JNC -1D
8587	0D0	LD@R 3	859F	30C	?FSET 1
8588	10E	A=C ALL	85A0	017	JC +02
8589	04E	C=0 ALL	85A1	0A6	A<>C S&X
858A	37C	RCR 12	85A2	168	WRIT 5(M)
858B	0EE	C<>B ALL	85A3	2CC	?FSET 13
858C	2FC	RCR 13	85A4	360	?C RTN
858D	0EE	C<>B ALL	85A5	260	SETHEX
858E	2C2	B≠0 @R	85A6	191	?NC GO
858F	013	JNC +02	85A7	00E	0364

APPENDIX V - OCTal-HEX Conversion Programs

OCTal - Hex

The following program converts mainframe addresses from the octal (base 8) form that appears in HP's documentation to hexadecimal (base 16), the form that you will need in constructing an MCODE execute or goto instruction. To use this program, just execute OCT-HEX. The program uses partial key sequencing to make your life easier.

The program comes back with the display

O _ .

The first number you should key in is the page number, which may be anywhere from 0 to 7. Other keys (except backarrow and R/S, as explained below) will be ignored. The number you select will appear in the display followed by a dash and another underscore prompt. Next key in the quad number, a digit from 0 to 3. The program will not accept any other values.

The program comes back with

O p-q-_ ,

where p and q are the page number and quad number, respectively. Now key in the four-digit octal address within the quad. The range of legal addresses is 0000 to 1777. Digits outside this range will not be accepted by the program. If the address is less than 1000, you must key in a leading zero. If you make a mistake (who me?) while keying in a number, you can use the backarrow key to remove digits. If there are no digits in the display and the backarrow key is pressed, the routine is terminated. This behavior of the backarrow key is consistent with mainframe functions, and you should strive for this kind of consistency in the behavior of your own programs.

To get the result, just press the R/S key. The hexadecimal equivalent of your octal address will be put into the display preceded by the word ADDRESS. Try the routine out a few times on addresses for which you know the hex equivalent so you can get the hang of it. Here is the listing for the routine.

Address	Hexcode	Mnemonic	Address	Hexcode	Mnemonic
85DD	130	LDI S&X	85FE	146	A=A+C S&X
85DE	370	HEX: 370	85FF	130	LDI S&X
85DF	106	A=C S&X	8600	03A	HEX: 03A
85E0	0B0	C=N	8601	306	?A<C S&X
85E1	366	A≠C S&X	8602	01F	JC +03
85E2	18F	JC +31	8603	266	C=C-1 S&X
85E3	3BD	?NC XQ	8604	1C6	A=A-C S&X
85E4	01C	07EF	8605	0A6	A<>C S&X
85E5	001	"A"	8606	3E8	WRIT 15(e)
85E6	004	"D"	8607	046	C=0 S&X
85E7	004	"D"	8608	2FC	RCR 13
85E8	012	"R"	8609	3D4	R=R-1
85E9	005	"E"	860A	394	?R= 0
85EA	013	"S"	860B	383	JNC -10
85EB	013	"S"	860C	261	?NC XQ
85EC	220	" "	860D	000	0098
85ED	149	?NC XQ	860E	046	C=0 S&X
85EE	024	0952	860F	3F0	PRPH SLCT
85EF	215	?NC XQ	8610	1FD	?NC GO
85F0	00C	0385	8611	00E	037E
85F1	278	READ 9(Q)	8612	25B	JNC -35
85F2	10E	A=C ALL	8613	183	JNC +30
85F3	3D9	?NC XQ	8614	149	?NC XQ
85F4	01C	07F6	8615	024	0952
85F5	04E	C=0 ALL	8616	278	READ 9(Q)
85F6	0BA	A<>C M	8617	0AE	A<>C ALL
85F7	33C	RCR 1	8618	1BE	A=A-1 MS
85F8	20E	C=C+A ALL	8619	049	?C GO
85F9	03C	RCR 3	861A	037	0D12
85FA	05C	R= 4	861B	35E	?A≠0 MS
85FB	106	A=C S&X	861C	0FB	JNC +1F
85FC	130	LDI S&X	861D	05E	C=0 MS
85FD	030	HEX: 030	861E	23E	C=C+1 MS

Address	Hexcode	Mnemonic
861F	3D9	?NC XQ
8620	01C	07F6
8621	37E	A≠C MS
8622	037	JC +06
8623	01C	R= 3
8624	002	A=0 @R
8625	130	LDI S&X
8626	020	HEX: 020
8627	3A8	WRIT 14(d)
8628	130	LDI S&X
8629	020	HEX: 020
862A	3A8	WRIT 14(d)
862B	149	?NC XQ
862C	024	0952
862D	0AE	A<>C ALL
862E	1E6	C=C+C S&X
862F	3C6	RSHFC S&X
8630	268	WRIT 9(Q)
8631	3D9	?NC XQ
8632	01C	07F6
8633	083	JNC +10
8634	098	"X"
8635	005	"E"
8636	008	"H"
8637	02D	"-"
8638	014	"T"
8639	003	"C"
863A	00F	"O"
863B	04E	C=0 ALL
863C	268	WRIT 9(Q)
863D	3C1	?NC XQ
863E	0B0	2CFO
863F	3BD	?NC XQ
8640	01C	07EF
8641	00F	"O"
8642	220	" "
8643	115	?NC XQ
8644	038	0E45
8645	27B	JNC -31
8646	00C	?FSET 3
8647	25B	JNC -35
8648	130	LDI S&X
8649	038	HEX: 038
864A	33C	RCR 1
864B	31E	?A<C MS
864C	3BB	JNC -09
864D	0BE	A<>C MS
864E	11E	A=C MS
864F	2FC	RCR 13
8650	3E8	WRIT 15(e)
8651	149	?NC XQ
8652	024	0952
8653	278	READ 9(Q)
8654	2FE	?C≠0 MS
8655	067	JC +0C
8656	23E	C=C+1 MS
8657	0BE	A<>C MS
8658	27C	RCR 9
8659	0BE	A<>C MS
865A	268	WRIT 9(Q)
865B	3D9	?NC XQ
865C	01C	07F6
865D	130	LDI S&X
865E	02D	HEX: 02D

Address	Hexcode	Mnemonic
865F	3E8	WRIT 15(e)
8660	31B	JNC -1D
8661	27E	C=C-1 MS
8662	2FE	?C≠0 MS
8663	0A7	JC +14
8664	2DC	R= 13
8665	110	LD@R 4
8666	31E	?A<C MS
8667	03F	JC +07
8668	3D9	?NC XQ
8669	01C	07F6
866A	130	LDI S&X
866B	020	HEX: 020
866C	3A8	WRIT 14(d)
866D	2B3	JNC -2A
866E	05E	C=0 MS
866F	07C	RCR 4
8670	0BE	A<>C MS
8671	1FE	C=C+C MS
8672	1FE	C=C+C MS
8673	0FC	RCR 10
8674	23E	C=C+1 MS
8675	23E	C=C+1 MS
8676	323	JNC -1C
8677	09E	B=A MS
8678	23E	C=C+1 MS
8679	23E	C=C+1 MS
867A	11E	A=C MS
867B	2DC	R= 13
867C	1D0	LD@R 7
867D	31E	?A<C MS

Address	Hexcode	Mnemonic
867E	353	JNC -16
867F	3DC	R=R+1
8680	0D0	LD@R 3
8681	37E	?A≠C MS
8682	077	JC +0E
8683	07E	A<>B MS
8684	27E	C=C-1 MS
8685	31E	?A<C MS
8686	313	JNC -1E
8687	05E	C=0 MS
8688	33C	RCR 1
8689	0BE	A<>C MS
868A	2FC	RCR 13
868B	0DE	C=B MS
868C	268	WRIT 9(Q)
868D	3D9	?NC XQ
868E	01C	07F6
868F	2F3	JNC -22
8690	278	READ 9(Q)
8691	1E6	C=C+C S&X
8692	1E6	C=C+C S&X
8693	1E6	C=C+C S&X
8694	0AE	A<>C ALL
8695	046	C=0 S&X
8696	0DE	C=B MS
8697	2FC	RCR 13
8698	146	A=A+C S&X
8699	0AE	A<>C ALL
869A	23E	C=C+1 MS
869B	38B	JNC -0F

HEX - OCTal

The HEX-OCT program is an inverse to the OCT-HEX program, allowing you to convert a hexadecimal entry address to the octal form suitable for looking up the entry point in HP's annotated listings.

HEX-OCT starts by placing an H, followed by a space and an underscore in the left of the display (partial key sequencing to the rescue again). The digit keys and the A through F keys are the only ones which are allowed for inputs. Once four digits have been entered, no more may be keyed in. The functions of the backarrow and run/stop keys are the same as for the OCT-HEX program. The output is of the form p-q-aaaa, where p is the page number, q is the quad number in the page, and aaaa is the octal address in the specified quad. A listing for this program starts on the next page.

Address	Hexcode	Mnemonic
869C	149	?NC XQ
869D	024	0952
869E	278	READ 9(Q)
869F	27E	C=C-1 MS
86A0	049	?C GO
86A1	037	0D12
86A2	11E	A=C MS
86A3	05E	C=0 MS
86A4	3CE	RSHFC ALL
86A5	0BE	A<>C MS
86A6	268	WRIT 9(Q)
86A7	1BB	JNC +37
86A8	094	"T"
86A9	003	"C"
86AA	00F	"O"
86AB	02D	"-"
86AC	018	"X"
86AD	005	"E"
86AE	008	"H"
86AF	04E	C=0 ALL
86B0	268	WRIT 9(Q)
86B1	3C1	?NC XQ
86B2	0B0	2CF0
86B3	3BD	?NC XQ
86B4	01C	07EF
86B5	008	"H"
86B6	220	" "
86B7	115	?NC XQ
86B8	038	0E45
86B9	31B	JNC -1D
86BA	04C	?FSET 4
86BB	14B	JNC +29
86BC	35E	A≠0 MS
86BD	3D3	JNC -06
86BE	130	LDI S&X
86BF	007	HEX: 007
86C0	33C	RCR 1
86C1	31E	?A<C MS
86C2	3AB	JNC -0B
86C3	0BE	A<>C MS
86C4	2FC	RCR 13
86C5	3E8	WRIT 15(e)
86C6	106	A=C S&X
86C7	130	LDI S&X
86C8	009	HEX: 009
86C9	146	A=A+C S&X
86CA	149	?NC XQ
86CB	024	0952
86CC	130	LDI S&X
86CD	004	HEX: 004
86CE	33C	RCR 1
86CF	11E	A=C MS
86D0	278	READ 9(Q)
86D1	0BE	A<>C MS
86D2	31E	?A<C MS
86D3	05B	JNC +0B
86D4	05E	C=0 MS
86D5	2FC	RCR 13
86D6	39C	R= 0
86D7	0A2	A<>C @R
86D8	0BE	A<>C MS
86D9	23E	C=C+1 MS
86DA	268	WRIT 9(Q)
86DB	3D9	?NC XQ
86DC	01C	07F6
86DD	2D3	JNC -26

Address	Hexcode	Mnemonic
86DE	3D9	?NC XQ
86DF	01C	07F6
86E0	130	LDI S&X
86E1	020	HEX: 020
86E2	3A8	WRIT 14(d)
86E3	2A3	JNC -2C
86E4	00C	?FSET 3
86E5	043	JNC +08
86E6	130	LDI S&X
86E7	003	HEX: 003
86E8	0BE	A<>C MS
86E9	2FC	RCR 13
86EA	3E8	WRIT 15(e)
86EB	106	A=C S&X
86EC	2F3	JNC -22
86ED	130	LDI S&X
86EE	370	HEX: 370
86EF	106	A=C S&X
86F0	0B0	C=N
86F1	366	A≠C S&X
86F2	22F	JC -3B
86F3	149	?NC XQ
86F4	024	0952
86F5	278	READ 9(Q)
86F6	39C	R= 0
86F7	102	A=C @R
86F8	1EE	C=C+C ALL
86F9	3DC	R=R+1
86FA	054	?R= 4
86FB	3E3	JNC -04
86FC	2FC	RCR 13
86FD	3DC	R=R+1
86FE	102	A=C @R
86FF	042	C=0 @R
8700	1EE	C=C+C ALL
8701	1EE	C=C+C ALL
8702	33C	RCR 1
8703	3D4	R=R-1
8704	102	A=C @R
8705	3D9	?NC XQ
8706	01C	07F6
8707	3BD	?NC XQ
8708	01C	07EF
8709	00F	"O"
870A	003	"C"
870B	014	"T"
870C	220	" "
870D	0AE	A<>C ALL
870E	0BC	RCR 5
870F	31C	R= 1
8710	0D0	LD@R 3
8711	106	A=C S&X
8712	130	LDI S&X
8713	00A	HEX: 00A
8714	302	?A<C @R
8715	027	JC +04
8716	262	C=C-1 @R
8717	242	C=A-C @R
8718	013	JNC +02
8719	0A6	A<>C S&X
871A	3E8	WRIT 15(e)
871B	130	LDI S&X
871C	02D	HEX: 02D
871D	3E8	WRIT 15(e)
871E	2FC	RCR 13
871F	3DC	R=R+1

Address	Hexcode	Mnemonic	Address	Hexcode	Mnemonic
8720	0D0	LD@R 3	872F	3E8	WRIT 15(e)
8721	3E8	WRIT 15(e)	8730	2FC	RCR 13
8722	130	LDI S&X	8731	056	C=0 XS
8723	02D	HEX: 02D	8732	3DC	R=R+1
8724	3E8	WRIT 15(e)	8733	3D8	C<>ST
8725	2FC	RCR 13	8734	054	?R= 4
8726	3D8	C<>ST	8735	3A3	JNC -0C
8727	304	CLRF 1	8736	149	?NC XQ
8728	204	CLRF 2	8737	024	0952
8729	004	CLRF 3	8738	215	?NC XQ
872A	048	SETF 4	8739	00C	0385
872B	088	SETF 5	873A	261	?NC XQ
872C	144	CLRF 6	873B	000	0098
872D	284	CLRF 7	873C	201	?NC GO
872E	3D8	C<>ST	873D	00E	0380

APPENDIX F - Table of Mnemonics

The following table shows the differences between the three types of mnemonics in use. We will only tabulate the mnemonics for the single word instructions. The three types of mnemonics are: HP mnemonics used by HP in all of the annotated listings of their ROMs; Jacobs/De Arras, developed in the early days of the development of MCODE programming by the user community; and ZENROM mnemonics, this version was developed in England and is used in the disassembler of a ROM that is put out by Zengrange Ltd. The Jacobs/De Arras mnemonics were used throughout this book.

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
000	0000	0000000000	NOP	NOP	NOP
00E	0016	0000001110	A=0	A=0 ALL	A=0 ALL
006	0006	0000000110	A=0 X	A=0 S&X	A=0 X
01A	0032	0000011010	A=0 M	A=0 M	A=0 M
00A	0012	0000001010	A=0 WPT	A=0 R<	A=0 WPT
002	0002	0000000010	A=0 PT	A=0 @R	A=0 PT
01E	0036	0000011110	A=0 S	A=0 MS	A=0 S
016	0026	0000010110	A=0 XS	A=0 XS	A=0 XS
012	0022	0000010010	A=0 PQ	A=0 P-Q	A=0 PQ
02E	0056	0000101110	B=0	B=0 ALL	B=0 ALL
026	0046	0000100110	B=0 X	B=0 S&X	B=0 X
03A	0072	0000111010	B=0 M	B=0 M	B=0 M
02A	0052	0000101010	B=0 WPT	B=0 R<	B=0 WPT
022	0042	0000100010	B=0 PT	B=0 @R	B=0 PT
03E	0076	0000111110	B=0 S	B=0 MS	B=0 S
036	0066	0000110110	B=0 XS	B=0 XS	B=0 XS
032	0062	0000110010	B=0 PQ	B=0 P-Q	B=0 PQ
04E	0116	0001001110	C=0	C=0 ALL	C=0 ALL
046	0106	0001000110	C=0 X	C=0 S&X	C=0 X
05A	0132	0001011010	C=0 M	C=0 M	C=0 M
04A	0112	0001001010	C=0 WPT	C=0 R<	C=0 WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
042	0102	0001000010	C=0 PT	C=0 @R	C=0 PT
05E	0136	0001011110	C=0 S	C=0 MS	C=0 S
056	0126	0001010110	C=0 XS	C=0 XS	C=0 XS
052	0122	0001010010	C=0 PQ	C=0 P-Q	C=0 PQ
06E	0156	0001101110	AB EX	A<>B ALL	A<>B ALL
066	0146	0001100110	AB EX X	A<>B S&X	A<>B X
07A	0172	0001111010	AB EX M	A<>B M	A<>B M
06A	0152	0001101010	AB EX WPT	A<>B R<	A<>B WPT
062	0142	0001100010	AB EX PT	A<>B @R	A<>B PT
07E	0176	0001111110	AB EX S	A<>B MS	A<>B S
076	0166	0001110110	AB EX XS	A<>B XS	A<>B XS
072	0162	0001110010	AB EX PQ	A<>B P-Q	A<>B PQ
08E	0216	0010001110	B=A	B=A ALL	B=A ALL
086	0206	0010000110	B=A X	B=A S&X	B=A X
09A	0232	0010011010	B=A M	B=A M	B=A M
08A	0212	0010001010	B=A WPT	B=A R<	B=A WPT
082	0202	0010000010	B=A PT	B=A @R	B=A PT
09E	0236	0010011110	B=A S	B=A MS	B=A S
096	0226	0010010110	B=A XS	B=A XS	B=A XS
092	0222	0010010010	B=A PQ	B=A P-Q	B=A PQ
0AE	0256	0010101110	AC EX	A<>C ALL	A<>C ALL
0A6	0246	0010100110	AC EX X	A<>C S&X	A<>C X
0BA	0272	0010111010	AC EX M	A<>C M	A<>C M
0AA	0252	0010101010	AC EX WPT	A<>C R<	A<>C WPT
0A2	0242	0010100010	AC EX PT	A<>C @R	A<>C PT
0BE	0276	0010111110	AC EX S	A<>C MS	A<>C S
0B6	0266	0010110110	AC EX XS	A<>C XS	A<>C XS
0B2	0262	0010110010	AC EX PQ	A<>C P-Q	A<>C PQ
0CE	0316	0011001110	C=B	C=B ALL	C=B ALL
0C6	0306	0011000110	C=B X	C=B S&X	C=B X
0DA	0332	0011011010	C=B M	C=B M	C=B M
0CA	0312	0011001010	C=B WPT	C=B R<	C=B WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
0C2	0302	0011000010	C=B PT	C=B @R	C=B PT
0DE	0336	0011011110	C=B S	C=B MS	C=B S
0D6	0326	0011010110	C=B XS	C=B XS	C=B XS
0D2	0322	0011010010	C=B PQ	C=B P-Q	C=B PQ
0EE	0356	0011101110	BC EX	C<>B ALL	B<>C ALL
0E6	0346	0011100110	BC EX X	C<>B S&X	B<>C X
0FA	0372	0011111010	BC EX M	C<>B M	B<>C M
0EA	0352	0011101010	BC EX WPT	C<>B R<	B<>C WPT
0E2	0342	0011100010	BC EX PT	C<>B @R	B<>C PT
0FE	0376	0011111110	BC EX S	C<>B MS	B<>C S
0F6	0366	0011110110	BC EX XS	C<>B XS	B<>C XS
0F2	0362	0011110010	BC EX PQ	C<>B P-Q	B<>C PQ
10E	0416	0100001110	A=C	A=C ALL	A=C ALL
106	0406	0100000110	A=C X	A=C S&X	A=C X
11A	0432	0100011010	A=C M	A=C M	A=C M
10A	0412	0100001010	A=C WPT	A=C R<	A=C WPT
102	0402	0100000010	A=C PT	A=C @R	A=C PT
11E	0436	0100011110	A=C S	A=C MS	A=C S
116	0426	0100010110	A=C XS	A=C XS	A=C XS
112	0422	0100010010	A=C PQ	A=C P-Q	A=C PQ
12E	0456	0100101110	A=A+B	A=A+B ALL	A=A+B ALL
126	0446	0100100110	A=A+B X	A=A+B S&X	A=A+B X
13A	0472	0100111010	A=A+B M	A=A+B M	A=A+B M
12A	0452	0100101010	A=A+B WPT	A=A+B R<	A=A+B WPT
122	0442	0100100010	A=A+B PT	A=A+B @R	A=A+B PT
13E	0476	0100111110	A=A+B S	A=A+B MS	A=A+B S
136	0466	0100110110	A=A+B XS	A=A+B XS	A=A+B XS
132	0462	0100110010	A=A+B PQ	A=A+B P-Q	A=A+B PQ
14E	0516	0101001110	A=A+C	A=A+C ALL	A=A+C ALL
146	0506	0101000110	A=A+C X	A=A+C S&X	A=A+C X
15A	0532	0101011010	A=A+C M	A=A+C M	A=A+C M
14A	0512	0101001010	A=A+C WPT	A=A+C R<	A=A+C WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
142	0502	0101000010	A=A+C PT	A=A+C @R	A=A+C PT
15E	0536	0101011110	A=A+C S	A=A+C MS	A=A+C S
156	0526	0101010110	A=A+C XS	A=A+C XS	A=A+C XS
152	0522	0101010010	A=A+C PQ	A=A+C P-Q	A=A+C PQ
16E	0556	0101101110	A=A+1	A=A+1 ALL	A=A+1 ALL
166	0546	0101100110	A=A+1 X	A=A+1 S&X	A=A+1 X
17A	0572	0101111010	A=A+1 M	A=A+1 M	A=A+1 M
16A	0552	0101101010	A=A+1 WPT	A=A+1 R<	A=A+1 WPT
162	0542	0101100010	A=A+1 PT	A=A+1 @R	A=A+1 PT
17E	0576	0101111110	A=A+1 S	A=A+1 MS	A=A+1 S
176	0566	0101110110	A=A+1 XS	A=A+1 XS	A=A+1 XS
172	0562	0101110010	A=A+1 PQ	A=A+1 P-Q	A=A+1 PQ
18E	0616	0110001110	A=A-B	A=A-B ALL	A=A-B ALL
186	0606	0110000110	A=A-B X	A=A-B S&X	A=A-B X
19A	0632	0110011010	A=A-B M	A=A-B M	A=A-B M
18A	0612	0110001010	A=A-B WPT	A=A-B R<	A=A-B WPT
182	0602	0110000010	A=A-B PT	A=A-B @R	A=A-B PT
19E	0636	0110011110	A=A-B S	A=A-B MS	A=A-B S
196	0626	0110010110	A=A-B XS	A=A-B XS	A=A-B XS
192	0622	0110010010	A=A-B PQ	A=A-B P-Q	A=A-B PQ
1AE	0656	0110101110	A=A-1	A=A-1 ALL	A=A-1 ALL
1A6	0646	0110100110	A=A-1 X	A=A-1 S&X	A=A-1 X
1BA	0672	0110111010	A=A-1 M	A=A-1 M	A=A-1 M
1AA	0652	0110101010	A=A-1 WPT	A=A-1 R<	A=A-1 WPT
1A2	0642	0110100010	A=A-1 PT	A=A-1 @R	A=A-1 PT
1BE	0676	0110111110	A=A-1 S	A=A-1 MS	A=A-1 S
1B6	0666	0110110110	A=A-1 XS	A=A-1 XS	A=A-1 XS
1B2	0662	0110110010	A=A-1 PQ	A=A-1 P-Q	A=A-1 PQ
1CE	0716	0111001110	A=A-C	A=A-C ALL	A=A-C ALL
1C6	0706	0111000110	A=A-C X	A=A-C S&X	A=A-C X
1DA	0732	0111011010	A=A-C M	A=A-C M	A=A-C M
1CA	0712	0111001010	A=A-C WPT	A=A-C R<	A=A-C WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
1C2	0702	0111000010	A=A-C PT	A=A-C @R	A=A-C PT
1DE	0736	0111011110	A=A-C S	A=A-C MS	A=A-C S
1D6	0726	0111010110	A=A-C XS	A=A-C XS	A=A-C XS
1D2	0722	0111010010	A=A-C PQ	A=A-C P-Q	A=A-C PQ
1EE	0756	0111101110	C=C+C	C=C+C ALL	C=C+C ALL
1E6	0746	0111100110	C=C+C X	C=C+C S&X	C=C+C X
1FA	0772	0111111010	C=C+C M	C=C+C M	C=C+C M
1EA	0752	0111101010	C=C+C WPT	C=C+C R<	C=C+C WPT
1E2	0742	0111100010	C=C+C PT	C=C+C @R	C=C+C PT
1FE	0776	0111111110	C=C+C S	C=C+C MS	C=C+C S
1F6	0766	0111110110	C=C+C XS	C=C+C XS	C=C+C XS
1F2	0762	0111110010	C=C+C PQ	C=C+C P-Q	C=C+C PQ
20E	1016	1000001110	C=A+C	C=C+A ALL	C=A+C ALL
206	1006	1000000110	C=A+C X	C=C+A S&X	C=A+C X
21A	1032	1000011010	C=A+C M	C=C+A M	C=A+C M
20A	1012	1000001010	C=A+C WPT	C=C+A R<	C=A+C WPT
202	1002	1000000010	C=A+C PT	C=C+A @R	C=A+C PT
21E	1036	1000011110	C=A+C S	C=C+A MS	C=A+C S
216	1026	1000010110	C=A+C XS	C=C+A XS	C=A+C XS
212	1022	1000010010	C=A+C PQ	C=C+A P-Q	C=A+C PQ
22E	1056	1000101110	C=C+1	C=C+1 ALL	C=C+1 ALL
226	1046	1000100110	C=C+1 X	C=C+1 S&X	C=C+1 X
23A	1072	1000111010	C=C+1 M	C=C+1 M	C=C+1 M
22A	1052	1000101010	C=C+1 WPT	C=C+1 R<	C=C+1 WPT
222	1042	1000100010	C=C+1 PT	C=C+1 @R	C=C+1 PT
23E	1076	1000111110	C=C+1 S	C=C+1 MS	C=C+1 S
236	1066	1000110110	C=C+1 XS	C=C+1 XS	C=C+1 XS
232	1062	1000110010	C=C+1 PQ	C=C+1 P-Q	C=C+1 PQ
24E	1116	1001001110	C=A-C	C=A-C ALL	C=A-C ALL
246	1106	1001000110	C=A-C X	C=A-C S&X	C=A-C X
25A	1132	1001011010	C=A-C M	C=A-C M	C=A-C M
24A	1112	1001001010	C=A-C WPT	C=A-C R<	C=A-C WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
242	1102	1001000010	C=A-C PT	C=A-C @R	C=A-C PT
25E	1136	1001011110	C=A-C S	C=A-C MS	C=A-C S
256	1126	1001010110	C=A-C XS	C=A-C XS	C=A-C XS
252	1122	1001010010	C=A-C PQ	C=A-C P-Q	C=A-C PQ
26E	1156	1001101110	C=C-1	C=C-1 ALL	C=C-1 ALL
266	1146	1001100110	C=C-1 X	C=C-1 S&X	C=C-1 X
27A	1172	1001111010	C=C-1 M	C=C-1 M	C=C-1 M
26A	1152	1001101010	C=C-1 WPT	C=C-1 R<	C=C-1 WPT
262	1142	1001100010	C=C-1 PT	C=C-1 @R	C=C-1 PT
27E	1176	1001111110	C=C-1 S	C=C-1 MS	C=C-1 S
276	1166	1001110110	C=C-1 XS	C=C-1 XS	C=C-1 XS
272	1162	1001110010	C=C-1 PQ	C=C-1 P-Q	C=C-1 PQ
28E	1216	1010001110	C=-C	C=0-C ALL	C=-C ALL
286	1206	1010000110	C=-C X	C=0-C S&X	C=-C X
29A	1232	1010011010	C=-C M	C=0-C M	C=-C M
28A	1212	1010001010	C=-C WPT	C=0-C R<	C=-C WPT
282	1202	1010000010	C=-C PT	C=0-C @R	C=-C PT
29E	1236	1010011110	C=-C S	C=0-C MS	C=-C S
296	1226	1010010110	C=-C XS	C=0-C XS	C=-C XS
292	1222	1010010010	C=-C PQ	C=0-C P-Q	C=-C PQ
2AE	1256	1010101110	C=-C-1	C=-C-1 ALL	C=-C-1 ALL
2A6	1246	1010100110	C=-C-1 X	C=-C-1 S&X	C=-C-1 X
2BA	1272	1010111010	C=-C-1 M	C=-C-1 M	C=-C-1 M
2AA	1252	1010101010	C=-C-1 WPT	C=-C-1 R<	C=-C-1 WPT
2A2	1242	1010100010	C=-C-1 PT	C=-C-1 @R	C=-C-1 PT
2BE	1276	1010111110	C=-C-1 S	C=-C-1 MS	C=-C-1 S
2B6	1266	1010110110	C=-C-1 XS	C=-C-1 XS	C=-C-1 XS
2B2	1262	1010110010	C=-C-1 PQ	C=-C-1 P-Q	C=-C-1 PQ
2CE	1316	1011001110	?B≠0	?B≠0 ALL	?B≠0 ALL
2C6	1306	1011000110	?B≠0 X	?B≠0 S&X	?B≠0 X
2DA	1332	1011011010	?B≠0 M	?B≠0 M	?B≠0 M
2CA	1312	1011001010	?B≠0 WPT	?B≠0 R<	?B≠0 WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
2C2	1302	1011000010	?B≠0 PT	?B≠0 @R	?B≠0 PT
2DE	1336	1011011110	?B≠0 S	?B≠0 MS	?B≠0 S
2D6	1326	1011010110	?B≠0 XS	?B≠0 XS	?B≠0 XS
2D2	1322	1011010010	?B≠0 PQ	?B≠0 P-Q	?B≠0 PQ
2EE	1356	1011101110	?C≠0	?C≠0 ALL	?C≠0 ALL
2E6	1346	1011100110	?C≠0 X	?C≠0 S&X	?C≠0 X
2FA	1372	1011111010	?C≠0 M	?C≠0 M	?C≠0 M
2EA	1352	1011101010	?C≠0 WPT	?C≠0 R<	?C≠0 WPT
2E2	1342	1011100010	?C≠0 PT	?C≠0 @R	?C≠0 PT
2FE	1376	1011111110	?C≠0 S	?C≠0 MS	?C≠0 S
2F6	1366	1011110110	?C≠0 XS	?C≠0 XS	?C≠0 XS
2F2	1362	1011110010	?C≠0 PQ	?C≠0 P-Q	?C≠0 PQ
30E	1416	1100001110	?A<C	?A<C ALL	?A<C ALL
306	1406	1100000110	?A<C X	?A<C S&X	?A<C X
31A	1432	1100011010	?A<C M	?A<C M	?A<C M
30A	1412	1100001010	?A<C WPT	?A<C R<	?A<C WPT
302	1402	1100000010	?A<C PT	?A<C @R	?A<C PT
31E	1436	1100011110	?A<C S	?A<C MS	?A<C S
316	1426	1100010110	?A<C XS	?A<C XS	?A<C XS
312	1422	1100010010	?A<C PQ	?A<C P-Q	?A<C PQ
32E	1456	1100101110	?A<B	?A<B ALL	?A<B ALL
326	1446	1100100110	?A<B X	?A<B S&X	?A<B X
33A	1472	1100111010	?A<B M	?A<B M	?A<B M
32A	1452	1100101010	?A<B WPT	?A<B R<	?A<B WPT
322	1442	1100100010	?A<B PT	?A<B @R	?A<B PT
33E	1476	1100111110	?A<B S	?A<B MS	?A<B S
336	1466	1100110110	?A<B XS	?A<B XS	?A<B XS
332	1462	1100110010	?A<B PQ	?A<B P-Q	?A<B PQ
34E	1516	1101001110	?A≠0	?A≠0 ALL	?A≠0 ALL
346	1506	1101000110	?A≠0 X	?A≠0 S&X	?A≠0 X
35A	1532	1101011010	?A≠0 M	?A≠0 M	?A≠0 M
34A	1512	1101001010	?A≠0 WPT	?A≠0 R<	?A≠0 WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
342	1502	1101000010	?A≠0 PT	?A≠0 @R	?A≠0 PT
35E	1536	1101011110	?A≠0 S	?A≠0 MS	?A≠0 S
356	1526	1101010110	?A≠0 XS	?A≠0 XS	?A≠0 XS
352	1522	1101010010	?A≠0 PQ	?A≠0 P-Q	?A≠0 PQ
36E	1556	1101101110	?A≠C	?A≠C ALL	?A≠C ALL
366	1546	1101100110	?A≠C X	?A≠C S&X	?A≠C X
37A	1572	1101111010	?A≠C M	?A≠C M	?A≠C M
36A	1552	1101101010	?A≠C WPT	?A≠C R<	?A≠C WPT
362	1542	1101100010	?A≠C PT	?A≠C @R	?A≠C PT
37E	1576	1101111110	?A≠C S	?A≠C MS	?A≠C S
376	1566	1101110110	?A≠C XS	?A≠C XS	?A≠C XS
372	1562	1101110010	?A≠C PQ	?A≠C P-Q	?A≠C PQ
38E	1616	1110001110	A SR	RSHFA ALL	ASR ALL
386	1606	1110000110	A SR X	RSHFA S&X	ASR X
39A	1632	1110011010	A SR M	RSHFA M	ASR M
38A	1612	1110001010	A SR WPT	RSHFA R<	ASR WPT
382	1602	1110000010	A SR PT	RSHFA @R	ASR PT
39E	1636	1110011110	A SR S	RSHFA MS	ASR S
396	1626	1110010110	A SR XS	RSHFA XS	ASR XS
392	1622	1110010010	A SR PQ	RSHFA P-Q	ASR PQ
3AE	1656	1110101110	B SR	RSHFB ALL	BSR ALL
3A6	1646	1110100110	B SR X	RSHFB S&X	BSR X
3BA	1672	1110111010	B SR M	RSHFB M	BSR M
3AA	1652	1110101010	B SR WPT	RSHFB R<	BSR WPT
3A2	1642	1110100010	B SR PT	RSHFB @R	BSR PT
3BE	1676	1110111110	B SR S	RSHFB MS	BSR S
3B6	1666	1110110110	B SR XS	RSHFB XS	BSR XS
3B2	1662	1110110010	B SR PQ	RSHFB P-Q	BSR PQ
3CE	1716	1111001110	C SR	RSHFC ALL	CSR ALL
3C6	1706	1111000110	C SR X	RSHFC S&X	CSR X
3DA	1732	1111011010	C SR M	RSHFC M	CSR M
3CA	1712	1111001010	C SR WPT	RSHFC R<	CSR WPT

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
3C2	1702	1111000010	C SR PT	RSHFC @R	CSR PT
3DE	1736	1111011110	C SR S	RSHFC MS	CSR S
3D6	1726	1111010110	C SR XS	RSHFC XS	CSR XS
3D2	1722	1111010010	C SR PQ	RSHFC P-Q	CSR PQ
3EE	1756	1111101110	A SL	LSHFA ALL	ASL ALL
3E6	1746	1111100110	A SL X	LSHFA S&X	ASL X
3FA	1772	1111111010	A SL M	LSHFA M	ASL M
3EA	1752	1111101010	A SL WPT	LSHFA R<	ASL WPT
3E2	1742	1111100010	A SL PT	LSHFA @R	ASL PT
3FE	1776	1111111110	A SL S	LSHFA MS	ASL S
3F6	1766	1111110110	A SL XS	LSHFA XS	ASL XS
3F2	1762	1111110010	A SL PQ	LSHFA P-Q	ASL PQ
038	0070	0000111000	C=DATA	READ DATA	RDATA
078	0170	0001111000	C=REGN 1	READ 1(Z)	C=REG 1/Z
0B8	0270	0010111000	C=REGN 2	READ 2(Y)	C=REG 2/Y
0F8	0370	0011111000	C=REGN 3	READ 3(X)	C=REG 3/X
138	0470	0100111000	C=REGN 4	READ 4(L)	C=REG 4/L
178	0570	0101111000	C=REGN 5	READ 5(M)	C=REG 5/M
1B8	0670	0110111000	C=REGN 6	READ 6(N)	C=REG 6/N
1F8	0770	0111111000	C=REGN 7	READ 7(O)	C=REG 7/O
238	1070	1000111000	C=REGN 8	READ 8(P)	C=REG 8/P
278	1170	1001111000	C=REGN 9	READ 9(Q)	C=REG 9/Q
2B8	1270	1010111000	C=REGN 10	READ 10(⊦-)	C=REG 10/⊦-
2F8	1370	1011111000	C=REGN 11	READ 11(a)	C=REG 11/a
338	1470	1100111000	C=REGN 12	READ 12(b)	C=REG 12/b
378	1570	1101111000	C=REGN 13	READ 13(c)	C=REG 13/c
3B8	1670	1110111000	C=REGN 14	READ 14(d)	C=REG 14/d
3F8	1770	1111111000	C=REGN 15	READ 15(e)	C=REG 15/e
028	0050	0000101000	REGN=C 0	WRIT 0(T)	REG=C 0/T
068	0150	0001101000	REGN=C 1	WRIT 1(Z)	REG=C 1/Z
0A8	0250	0010101000	REGN=C 2	WRIT 2(Y)	REG=C 2/Y
0E8	0350	0011101000	REGN=C 3	WRIT 3(X)	REG=C 3/X

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
128	0450	0100101000	REGN=C 4	WRIT 4(L)	REG=C 4/L
168	0550	0101101000	REGN=C 5	WRIT 5(M)	REG=C 5/M
1A8	0650	0110101000	REGN=C 6	WRIT 6(N)	REG=C 6/N
1E8	0750	0111101000	REGN=C 7	WRIT 7(O)	REG=C 7/O
228	1050	1000101000	REGN=C 8	WRIT 8(P)	REG=C 8/P
268	1150	1001101000	REGN=C 9	WRIT 9(Q)	REG=C 9/Q
2A8	1250	1010101000	REGN=C 10	WRIT 10(⊦)	REG=C 10/⊦
2E8	1350	1011101000	REGN=C 11	WRIT 11(a)	REG=C 11/a
328	1450	1100101000	REGN=C 12	WRIT 12(b)	REG=C 12/b
368	1550	1101101000	REGN=C 13	WRIT 13(c)	REG=C 13/c
3A8	1650	1110101000	REGN=C 14	WRIT 14(d)	REG=C 14/d
3E8	1750	1111101000	REGN=C 15	WRIT 15(e)	REG=C 15/e
33C	1474	1100111100	RCR 1	RCR 1	RCR 1
23C	1074	1000111100	RCR 2	RCR 2	RCR 2
03C	0074	0000111100	RCR 3	RCR 3	RCR 3
07C	0174	0001111100	RCR 4	RCR 4	RCR 4
0BC	0274	0010111100	RCR 5	RCR 5	RCR 5
17C	0574	0101111100	RCR 6	RCR 6	RCR 6
2BC	1274	1010111100	RCR 7	RCR 7	RCR 7
13C	0474	0100111100	RCR 8	RCR 8	RCR 8
27C	1174	1001111100	RCR 9	RCR 9	RCR 9
0FC	0374	0011111100	RCR 10	RCR 10	RCR 10
1BC	0674	0110111100	RCR 11	RCR 11	RCR 11
37C	1574	1101111100	RCR 12	RCR 12	RCR 12
2FC	1374	1011111100	RCR 13	RCR 13	RCR 13
388	1610	1110001000	S0=1	SETF 0	SF 0
308	1410	1100001000	S1=1	SETF 1	SF 1
208	1010	1000001000	S2=1	SETF 2	SF 2
008	0010	0000001000	S3=1	SETF 3	SF 3
048	0110	0001001000	S4=1	SETF 4	SF 4
088	0210	0010001000	S5=1	SETF 5	SF 5
148	0510	0101001000	S6=1	SETF 6	SF 6

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
288	1210	1010001000	S7=1	SETF 7	SF 7
108	0410	0100001000	S8=1	SETF 8	SF 8
248	1110	1001001000	S9=1	SETF 9	SF 9
0C8	0310	0011001000	S10=1	SETF 10	SF 10
188	0610	0110001000	S11=1	SETF 11	SF 11
348	1510	1101001000	S12=1	SETF 12	SF 12
2C8	1310	1011001000	S13=1	SETF 13	SF 13
384	1604	1110000100	S0=0	CLRF 0	CF 0
304	1404	1100000100	S1=0	CLRF 1	CF 1
204	1004	1000000100	S2=0	CLRF 2	CF 2
004	0004	0000000100	S3=0	CLRF 3	CF 3
044	0104	0001000100	S4=0	CLRF 4	CF 4
084	0204	0010000100	S5=0	CLRF 5	CF 5
144	0504	0101000100	S6=0	CLRF 6	CF 6
284	1204	1010000100	S7=0	CLRF 7	CF 7
104	0404	0100000100	S8=0	CLRF 8	CF 8
244	1104	1001000100	S9=0	CLRF 9	CF 9
0C4	0304	0011000100	S10=0	CLRF 10	CF 10
184	0604	0110000100	S11=0	CLRF 11	CF 11
344	1504	1101000100	S12=0	CLRF 12	CF 12
2C4	1304	1011000100	S13=0	CLRF 13	CF 13
38C	1614	1110001100	?S0=1	?FSET 0	?FS 0
30C	1414	1100001100	?S1=1	?FSET 1	?FS 1
20C	1014	1000001100	?S2=1	?FSET 2	?FS 2
00C	0014	0000001100	?S3=1	?FSET 3	?FS 3
04C	0114	0001001100	?S4=1	?FSET 4	?FS 4
08C	0214	0010001100	?S5=1	?FSET 5	?FS 5
14C	0514	0101001100	?S6=1	?FSET 6	?FS 6
28C	1214	1010001100	?S7=1	?FSET 7	?FS 7
10C	0414	0100001100	?S8=1	?FSET 8	?FS 8
24C	1114	1001001100	?S9=1	?FSET 9	?FS 9
0CC	0314	0011001100	?S10=1	?FSET 10	?FS 10

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
18C	0614	0110001100	?S11=1	?FSET 11	?FS 11
34C	1514	1101001100	?S12=1	?FSET 12	?FS 12
2CC	1314	1011001100	?S13=1	?FSET 13	?FS 13
39C	1634	1110011100	PT=0	R= 0	PT= 0
31C	1434	1100011100	PT=1	R= 1	PT= 1
21C	1034	1000011100	PT=2	R= 2	PT= 2
01C	0034	0000011100	PT=3	R= 3	PT= 3
05C	0134	0001011100	PT=4	R= 4	PT= 4
09C	0234	0010011100	PT=5	R= 5	PT= 5
15C	0534	0101011100	PT=6	R= 6	PT= 6
29C	1234	1010011100	PT=7	R= 7	PT= 7
11C	0434	0100011100	PT=8	R= 8	PT= 8
25C	1134	1001011100	PT=9	R= 9	PT= 9
0DC	0334	0011011100	PT=10	R= 10	PT= 10
19C	0634	0110011100	PT=11	R= 11	PT= 11
35C	1534	1101011100	PT=12	R= 12	PT= 12
2DC	1334	1011011100	PT=13	R= 13	PT= 13
394	1624	1110010100	?PT=0	?R= 0	?PT= 0
314	1424	1100010100	?PT=1	?R= 1	?PT= 1
214	1024	1000010100	?PT=2	?R= 2	?PT= 2
014	0024	0000010100	?PT=3	?R= 3	?PT= 3
054	0124	0001010100	?PT=4	?R= 4	?PT= 4
094	0224	0010010100	?PT=5	?R= 5	?PT= 5
154	0524	0101010100	?PT=6	?R= 6	?PT= 6
294	1224	1010010100	?PT=7	?R= 7	?PT= 7
114	0424	0100010100	?PT=8	?R= 8	?PT= 8
254	1124	1001010100	?PT=9	?R= 9	?PT= 9
0D4	0324	0011010100	?PT=10	?R= 10	?PT= 10
194	0624	0110010100	?PT=11	?R= 11	?PT= 11
354	1524	1101010100	?PT=12	?R= 12	?PT= 12
2D4	1324	1011010100	?PT=13	?R= 13	?PT= 13
010	0020	0000010000	LC 0	LD@R 0	LC 0

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
050	0120	0001010000	LC 1	LD@R 1	LC 1
090	0220	0010010000	LC 2	LD@R 2	LC 2
0D0	0320	0011010000	LC 3	LD@R 3	LC 3
110	0420	0100010000	LC 4	LD@R 4	LC 4
150	0520	0101010000	LC 5	LD@R 5	LC 5
190	0620	0110010000	LC 6	LD@R 6	LC 6
1D0	0720	0111010000	LC 7	LD@R 7	LC 7
210	1020	1000010000	LC 8	LD@R 8	LC 8
250	1120	1001010000	LC 9	LD@R 9	LC 9
290	1220	1010010000	LC A	LD@R A	LC A
2D0	1320	1011010000	LC B	LD@R B	LC B
310	1420	1100010000	LC C	LD@R C	LC C
350	1520	1101010000	LC D	LD@R D	LC D
390	1620	1110010000	LC E	LD@R E	LC E
3D0	1720	1111010000	LC F	LD@R F	LC F
3AC	1654	1110101100	?F0=1	?FI= 0	?PBSY
32C	1454	1100101100	?F1=1	?FI= 1	?CRDR
22C	1054	1000101100	?F2=1	?FI= 2	?WNDB
02C	0054	0000101100	?F3=1	?FI= 3	?PF= 3
06C	0154	0001101100	?F4=1	?FI= 4	?PF= 4
0AC	0254	0010101100	?F5=1	?FI= 5	?EDAV
16C	0554	0101101100	?F6=1	?FI= 6	?IFCR
2AC	1254	1010101100	?F7=1	?FI= 7	?SRQR
12C	0454	0100101100	?F8=1	?FI= 8	?FRAV
26C	1154	1001101100	?F9=1	?FI= 9	?FRNS
0EC	0354	0011101100	?F10=1	?FI= 10	?ORAV
1AC	0654	0110101100	?F11=1	?FI= 11	?TFAIL
36C	1554	1101101100	?F12=1	?FI= 12	?ALM
2EC	1354	1011101100	?F13=1	?FI= 13	?SERV
024	0044	0000100100	SELPRF 0	SELP 0	PERTCT 0
064	0144	0001100100	SELPRF 1	SELP 1	PERTCT 1
0A4	0244	0010100100	SELPRF 2	SELP 2	PERTCT 2

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
0E4	0344	0011100100	SELPRF 3	SELP 3	PERTCT 3
124	0444	0100100100	SELPRF 4	SELP 4	PERTCT 4
164	0544	0101100100	SELPRF 5	SELP 5	PERTCT 5
1A4	0644	0110100100	SELPRF 6	SELP 6	PERTCT 6
1E4	0744	0111100100	SELPRF 7	SELP 7	PERTCT 7
224	1044	1000100100	SELPRF 8	SELP 8	PERTCT 8
264	1144	1001100100	SELPRF 9	SELP 9	PERTCT 9
2A4	1244	1010100100	SELPRF A	SELP A	PERTCT A
2E4	1344	1011100100	SELPRF B	SELP B	PERTCT B
324	1444	1100100100	SELPRF C	SELP C	PERTCT C
364	1544	1101100100	SELPRF D	SELP D	PERTCT D
3A4	1644	1110100100	SELPRF E	SELP E	PERTCT E
3E4	1744	1111100100	SELPRF F	SELP F	PERTCT F
3C4	1704	1111000100	CLR ST	ST=0	ST=0
3C8	1710	1111001000	RST KB	CLRKEY	CLRKEY
3CC	1714	1111001100	CHK KB	?KEY	?KEY
3D4	1724	1111010100	DEC PT	R=R-1	-PT
3DC	1734	1111011100	INC PT	R=R+1	+PT
058	0130	0001011000	G=C	G=C	G=C
098	0230	0010011000	C=G	C=G	C=G
0D8	0330	0011011000	CG EX	C<>G	C<>G
158	0530	0101011000	M=C	M=C	M=C
198	0630	0110011000	C=M	C=M	C=M
1D8	0730	0111011000	CM EX	C<>M	C<>M
258	1130	1001011000	F=SB	T=ST	F=ST
298	1230	1010011000	SB=F	ST=T	ST=F
2D8	1330	1011011000	FEXSB	ST<>T	ST<>F
358	1530	1001011000	ST=C	ST=C	ST=C
398	1630	1010011000	C=ST	C=ST	C=ST
3D8	1730	1111011000	CST EX	C<>ST	C<>ST
020	0040	0000100000	SPOPND	XQ>GO	CLRRTN
060	0140	0001100000	POWOFF	POWOFF	POWOFF

Hexcode	Octal	Binary	HP mnemonic	Jacobs/ De Arras	ZENROM mnemonic
0A0	0240	0010100000	SEL P	SLCT P	PT=P
0E0	0340	0011100000	SEL Q	SLCT Q	PT=Q
120	0440	0100100000	?P=Q	?P=Q	?P=Q
160	0540	0101100000	LLD	?LOWBAT	?BAT
1A0	0640	0110100000	CLRABC	A=B=C=0	ABC=0
1E0	0740	0111100000	GOTOC	GOTO ADR	GTOC
220	1040	1000100000	C = KEYS	C=KEY	C=KEY
260	1140	1001100000	SETHEX	SETHEX	SETHEX
2A0	1240	1010100000	SETDEC	SETDEC	SETDEC
2E0	1340	1011100000	DISOFF	DSPOFF	DISOFF
320	1440	1100100000	DISTOG	DSPTOG	DISTOG
360	1540	1101100000	RTN C	?C RTN	CRTN
3A0	1640	1110100000	RTN NC	?NC RTN	NCRTN
3E0	1740	1111100000	RTN	RTN	RTN
070	0160	0001110000	N=C	N=C	N=C
0B0	0260	0010110000	C=N	C=N	C=N
0F0	0360	0011110000	CN EX	C<>N	C<>N
130	0460	0100110000	LDI	LDI S&X	LDI
170	0560	0101110000	STK = C	PUSH ADR	STK=C
1B0	0660	0110110000	C = STK	POP ADR	C=STK
230	1060	1000110000	GOKEYS	GTO KEY	GTOKEY
270	1160	1001110000	DADD = C	RAMSLCT	RAMSLCT
2F0	1360	1011110000	DATA = C	WRITE DATA	WDATA
330	1460	1100110000	CXISA	FETCH S&X	RDROM
370	1560	1101110000	C=CORA	C=C OR A	C=CORA
3B0	1660	1110110000	C=C.A	C=C AND A	C=CANDA
3F0	1760	1111110000	PFAD=C	PRPH SLCT	PERSLCT

INDEX

ORDER BLANK

	QUANTITY	AMOUNT
FOR HP-71'S **HP-71 Basic Made Easy** by Joseph Horn, $18.95 per copy	______	$______
FOR HP-41'S **HP-41 M-Code for Beginners** by Ken Emery, $24.95 per copy	______	$______
Inside the HP-41 by Jean-Daniel Dodin, $18.95 per copy	______	$______
HP-41 Extended Functions Made Easy by Keith Jarett, $16.95 per copy	______	$______
HP-41 Synthetic Programming Made Easy (Includes one Quick Reference Card) by Keith Jarett, $16.95 per copy	______	$______
Quick Reference Card for Synthetic Programming $3.00 each	______	$______
FOR HP-10C, 11C, 15C, AND 16C **ENTER** **(Reverse Polish Notation Made Easy)** by Jean-Daniel Dodin, $5.95 per copy	______	$______
Sales tax (California orders only, 6 or 6.5%)		$______

Shipping, per book

within USA, book rate (4th class)	$1.50
USA 48 states, United Parcel Service	$2.50
USA, Canada, air mail	$3.00
elsewhere, air mail	$6.05
elsewhere, book rate (6 to 8 week wait)	$2.00

Shipping for QRC plastic cards (any number)

Free with book order or stamped envelope	
Otherwise	$1.00

Enter shipping total here $______

Total due $______

Checks must be in U.S. funds, and payable through a U.S. bank.

Name______________________________

Address______________________________

____

City______________ State______________ Zip______________

Country______________________________

Mail to:

SYNTHETIX
P.O. Box 1080
Berkeley, CA 94701-1080
(415) 339-0601

ORDER BLANK

	QUANTITY	AMOUNT
FOR HP-71'S **HP-71 Basic Made Easy** by Joseph Horn, $18.95 per copy	________	$________
FOR HP-41'S **HP-41 M-Code for Beginners** by Ken Emery, $24.95 per copy	________	$________
Inside the HP-41 by Jean-Daniel Dodin, $18.95 per copy	________	$________
HP-41 Extended Functions Made Easy by Keith Jarett, $16.95 per copy	________	$________
HP-41 Synthetic Programming Made Easy (Includes one Quick Reference Card) by Keith Jarett, $16.95 per copy	________	$________
Quick Reference Card for Synthetic Programming $3.00 each	________	$________
FOR HP-10C, 11C, 15C, AND 16C **ENTER** **(Reverse Polish Notation Made Easy)** by Jean-Daniel Dodin, $5.95 per copy	________	$________
Sales tax (California orders only, 6 or 6.5%)		$________

Shipping, per book
- within USA, book rate (4th class) $1.50
- USA 48 states, United Parcel Service $2.50
- USA, Canada, air mail $3.00
- elsewhere, air mail $6.05
- elsewhere, book rate (6 to 8 week wait) $2.00

Shipping for QRC plastic cards (any number)
- Free with book order or stamped envelope
- Otherwise $1.00

Enter shipping total here $________

Total due $________

Checks must be in U.S. funds, and payable through a U.S. bank.

Name____
Address______________________________________
____
City____________________State____________Zip__________
Country______________________________________

Mail to:

SYNTHETIX
P.O. Box 1080
Berkeley, CA 94701-1080
(415) 339-0601

HP-41 MCODE FOR BEGINNERS
by Ken Emery

MCODE is the internal machine code used by the HP-41, one level below the set of "user code" instructions that users and programmers are accustomed to dealing with. Some user code instructions like CLX are implemented by the HP-41 in just a few MCODE instructions; other user code instructions like TAN may need hundreds of MCODE operations.

Programs in MCODE are FAST. They run 7 to 120 times faster than user code programs. But the advantage that enthusiasts will appreciate the most is that MCODE gives you total control of the machine. You can make the HP-41 do whatever you want it to do, completely redefining its "personality" and customizing it for your particular applications. MCODE programming requires additional hardware, generally an external box called an MLDL (Machine Language Development Lab). But once you enter the world of MCODE there is nothing you can't do.

This book is your ticket to the world of MCODE.

Simple programming examples lead you step-by-step to an understanding of the principles and practice of MCODE programming. Later examples show you how to use parts of the built-in operating system as subroutines to do input, output, and other useful functions. Even before you finish the examples, you will be able to write your own simple MCODE programs.

For advanced MCODE programmers, there are several features of interest. Complete details of the display instructions are given. This includes the new display that accesses additional LCD characters, and that allows alteration of the contrast. Also explained for the first time is partial key sequencing, which allows you to create functions that prompt for inputs in the same user-friendly way as the built-in functions like STO and LBL.

Two utility programs are included to help in your programming. A debugging program allows you to interrupt an MCODE routine at any point, dumping the contents of the CPU registers for viewing. Also included are base conversion programs to help you use HP's annotated operating system listings.

Move into the FAST lane. Get started programming in MCODE today!

ISBN 0-9612174-7-2