ZENROM

The HP-41 PROGRAMMERS MODULE

- ✻ Synthetic Programming
- ✻ Machine Language Programming

ZENGRANGE

ZENROM 3B

USER'S HANDBOOK

ZENROM 3B

USER'S HANDBOOK

A Programmer's Module
For Use With The HP-41 Handheld Computer

FOREWORD

Many veteran HP-41 programmers will feel slightly cheated by ZENROM. As one who has been hard at it since late '79, through user, synthetic and machine code, I cannot deny at least some sympathy with that view.

It's not the countless hours spent exploring the HP-41 that anyone will mind. If that wasn't fun, why would anyone have done it ? No, it's the painfully developed slick techniques for entering synthetic code, the carefully thought out methods of editorless machine coding, all made suddenly, completely redundant !

One welcomes ZENROM with just a shade of regret. Like one who has survived the rigours of a wagon train and lived on to see the trans-continental railway. A journey of months, through byte jumpers, et al, to an understanding of synthetics, with its sunlit beaches and fertile lands, is reduced to days for any newcomer.

Of course, you would expect me to recommend it. But, since this is a foreword and not an advertisement, you should, if you are reading this, have already bought the module so I'm not here to sell it.

I simply have to tell you that, although I had no part in the writing of either program or manual, I am truly proud to present ZENROM to you.

If you're an old timer and the analogy above strikes a chord, believe me, you'll be amazed how quickly you overcome the regret. If you're a newcomer, eat your heart out. You'll never know what you've missed !

John French
Chairman, Zengrange Limited

June 1984

CONTENTS

LIST OF ILLUSTRATIONS

INSTALLING ZENROM

Before installing or removing the ZENROM module, ensure that the HP-41 Handheld Computer is switched off. If this is not done, damage may result to the module, the computer or its operation may be disrupted.

The ZENROM module may be plugged into any port on the HP-41, although if an HP-82106A Single Memory Module is also plugged in (HP-41C only), then the ZENROM must be in a higher numbered port than the memory module (Port numbers are detailed on the back of the HP-41).

After removing ZENROM, place a port cap into the unused port as protection against dust and dirt.

ZENROM has the same XROM identity, i.e. XROM 05,xx, as that of the HP-00041-15001 STANDARD APPLICATIONS PAC. If both of these modules are plugged in at the same time, then only that module in the lowest numbered port will be seen by the HP-41. To avoid conflict when you wish to use ZENROM, ensure that the STANDARD APPLICATIONS PAC is either removed, or in a higher numbered port to the ZENROM module.

For additional information regarding care and service of ZENROM, refer to Appendix A: OWNER'S INFORMATION.

NOMAS

Synthetic and Machine Code programming fall into a class of activity that HP-User Groups have classified as being 'NOMAS'.

NOMAS is an abbreviation for:

NOt MAnufacturer Supported

(Recipient Agrees Not To Contact Manufacturer) and is a term that you will come across stamped on much of the available documentation relating to the design and implementation of the machine code instruction set on the HP-41.

Most manufacturers put considerable effort into supporting users of their products - and Hewlett Packard has a track record that others could do well to imitate. However, for many reasons, they are not able to support or assist with every activity that users wish to follow. A good example of this is Machine Code Programming. Because the HP-41 was designed many years ago, the Design Team has long since dispersed onto other tasks. Therefore, it is not possible for HP to provide the worldwide effort needed to support such User activities. To overcome this problem, HP have released as much information as commercially viable to the User Groups devoted to their products. However, this has been on the understanding that the material is accepted on an "AS IS" basis and that no further assistance can be given by HP regarding the use of that material.

By the method of NOMAS, HP are able to release more detailed information than would otherwise have been possible. Therefore, please respect the NOMAS status of Synthetic and Machine Code Programming and do not contact Hewlett Packard for assistance — they just will not be able to help. If you need assistance, contact one of the independent User Groups listed in Appendix C. Almost all the published information on the HP-41 has come from members of these groups, and that is where the major expertise lies. For example, even programmers of ZENROM and the authors of this ZENROM User's Handbook are very active in User Groups.

ZENROM QUICK REFERENCE GUIDE

1

ZENROM FUNCTION SUMMARY

Many Users of ZENROM will already have a knowledge of synthetic and other advanced HP-41 programming techniques, or will want to immediately gain hands-on experience of its use. For such people, the following provides a brief, but sufficient introduction to get you started.

However, because ZENROM provides Users with the means to access the HP-41's operating system, either directly via techniques of Synthetic Programming, or indirectly, by machine language programming in conjunction with a machine language device, we would urge all newcomers to read through this handbook before using ZENROM.

Whilst the functions in ZENROM will not harm your HP-41, it is possible for the inexperienced User to cause either:

- loss of user memory contents (MEMORY LOST)
- or a keyboard 'lock-up' (placing the processor into an infinite loop), thereby preventing response to keystrokes. A master reset will be necessary to recover from this, leading to a loss of programs and data in user memory.

ZENROM functions fall into three major categories:

Catalogue Functions -	Language extension functions possessing an XROM identification number;
Operating Modes -	New interactive modes, similar to ED (text editor on the HP-41CX) or SW (stopwatch) and new keyboard layouts;
Direct-Key Synthetics -	Extensions to the HP-41 operating system that provide new facilities and allow the User to circumvent previous system limitations.

1.1 CATALOGUE FUNCTIONS

function	description
CLMM	Clear Main Memory Clears all Main Memory and stack registers. Resets the HP-41 status and flags to a Master Clear condition and displays the message 'MM LOST'.
CLXM	Clear Extended Memory Clears (nulls) all Extended Memory registers — but does not check for the existence of Extended Memory. Displays the message 'XM LOST' if executed from the keyboard.
CODE	Code ALPHA to X Converts the rightmost 14 characters in ALPHA into the 14 digit Non-Normalised Number which they specify, returned to X. No error message is generated if non-hexadecimal digits are contained in the ALPHA string
DECODE	Decode X to ALPHA Replaces ALPHA with the 14-character string representing the 14 digit number in X. If executed from the keyboard, views (and optionally prints) the decoded value. DECODE will perform an AVIEW unless in a running program or when a program is single stepped [SST].
LASTP	Last Program Positions the user program counter to the first line of the last program in program memory (i.e. the program containing the .END.).

NOP

No-Operation
Inserts an empty text line (F0h) as the next program line.
May generate error messages as follows:

PACKING, TRY AGAIN - if there is no room left for the insertion;
ROM - if the insertion is attempted into a ROM program.

Can also be entered as true NOP (XROM 05,07) instruction with a one byte and speed overhead.

NRCLM

Non-normalising Recall by M
Recalls to X the contents of the register whose absolute address is in the least significant 3 digits of status register M, that is, in hex form as the last character and a half of ALPHA.
Generates NONEXISTENT message if the register addressed is not physically connected to the HP-41.

NRCLX

Non-normalising Recall by X
Overwrites X with the contents of the user data register whose register number was specified in X before the function executed.
Saves the address to LASTX.
Generates NONEXISTENT message if the register number is greater than or equal to the current SIZE, or ALPHA DATA error message if the number in X is a string.

NSTOM

Non-normalising Store by M
Stores X into the register whose absolute address is specified in the least-significant three digits of M, that is, the last character and a half of ALPHA.
Generates NONEXISTENT message if the register addressed is not physically connected to the HP-41.

TOGF

Toggle Flag status
Toggles the state (from set to clear, or clear to set) of the flag whose absolute integer value is in X.
Generates NONEXISTENT message if X>55, or ALPHA DATA message if X contains a string.

1.2 OPERATING MODES

MCED

Machine Code Editor (non-programmable)
A full machine language programming and editing environment including facilities for disassembly of M-Code routines and creation of new routines using the M-Code hex-loader (when used with 'Quasi-ROM' (Q-ROM) in a machine language storage device). Use of all functions, except that of DISASSEMBLE, requires the availablility of Q-RAM.
DISASSEMBLE will only direct output to a printer or video interface responding as a printer. When a printer is connected to the system, in TRACE or NORM modes, then all editor commands will be printed.
Executing MCED activates the Editor Keyboard and displays the Editor 'COMMAND ?' prompt. At any time, pressing [SHIFT][CMD] will cancel the current prompt sequence and return you to the main editor prompt. The MCED-keyboard is shown below.

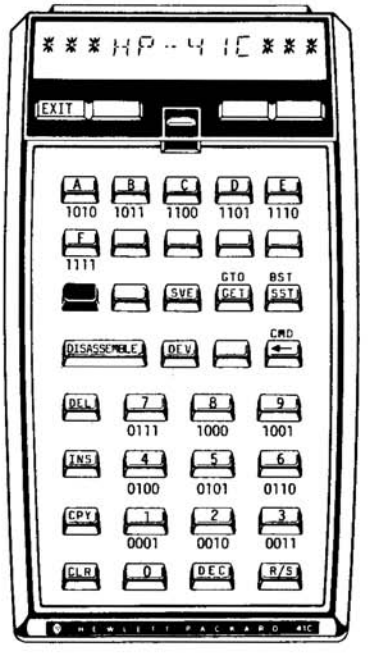


FIGURE 1.2.1 MACHINE CODE EDITOR KEYBOARD

For most of the Editor commands, there is a common input prompt format of:

Command: Start Address , Finish Address

For example, the DISASSEMBLE prompt appears as:

DIS: _ _ _ _ , _ _ _ _

After inputting the hexadecimal start address, the user has an option of specifying a decimal number of words or lines in place of a hexadecimal finish address. This is selected by pressing [DEC] and indicated by a 'd' appearing in the prompt:

DIS: 2 F 0 A , d _ _ _

At this point only the decimal keypad remains active.

Certain MCED functions have an 'A' in the prompt, or have a secondary prompt such as:

LMT: _ _ _ _

(where this allows you to specify the limit address beyond which the action will not take place.) An 'A' in a prompt, indicates that an absolute address input is expected.

Command	Description
[R/S]	Accepts the input and begins the specified action. This provides an escape if an incorrect input was made.
[EXIT]	Pressing this from the main editor prompt will return you to normal HP-41 usage. MCED is set to automatically 'timeout' after approximately two minutes of inactivity in a manner similar to ED in the CX. Executing ON will prevent this.
[SHIFT][CMD]	Cancels the current prompt sequence and returns to the main editor prompt 'COMMAND ?'.
[←]	Deletes the last key input.
[DEV]	Toggles between the ProtoCODER 2 and MLDL type device write formats. Defaults to MLDL type devices. The '0' annunciator is set when a ProtoCODER device is selected.
[DISASSEMBLE]	Requires an address for start and finish. Only the hex-keypad is activated, but after input of the start address, the [DEC] key allows the option of specifying a decimal number of words to be disassembled. [DEC] activates only the decimal keypad.

[GTO] Allows writing of M-Code instructions into your Q-ROM device with the hex loader. Requires input of a start address and responds with a prompt showing:

Address *Current Word* __ __ __
e.g: 1468 154 __ __ __

The hex-keypad allows input of the 3-hex digits representing the word you wish to write to that location. Press [R/S] to accept the new word.
The [SST] and [BST] keys are active within the hex-loader and permit forward and backward movement without changing the word at the current address location. If a printer device, in NORM or TRACE mode, is attached then the entered word is also disassembled to the printer. Single Stepping a word will also cause that word to be disassembled to the printer.

[INS] Allows insertion of a block of NOPs into Q-ROM before the specified address. A specific decimal number of NOPs may be input by pressing the [DEC] key instead of specifying the end hex address. Note that specifying 'd000' will default to 'd001' thus inserting one NOP.
Because INS moves all surrounding code in Q-ROM, to make way for the NOPs, a secondary prompt allows you to specify a 'limit address':

LMT: __ __ __ __

beyond which no code will be changed.
By specifying a LMT address *greater than* that for the end address, the surrounding code is moved up memory (to a higher address). If the LMT address is *before* the start address, then surrounding code will be moved down memory (to a lower address).
DATA ERROR messages will be issued if the end address is less than the start address or the LMT address is between the start and end addresses.

[DEL] Will delete a block of code from Q-ROM at a specified start and end address. A specific decimal number of lines can be deleted by pressing the [DEC] key as before. When deleted, surrounding code is moved in Q-ROM. A secondary LMT prompt specifies a boundary beyond which no code is moved.
By specifying a LMT address *greater than* that for the end address, the surrounding code is moved up memory (to a higher address). If the LMT address is *before* the start address, surrounding code will be moved down memory (to a lower address).
DATA ERROR messages will be issued if the end address is less than the start address or the LMT address is between the start and end addresses.

[CPY] Requires a hex start and end address of a block of code, or a specific decimal number of words to copy. A secondary prompt requests the starting address of the destination.
CPY allows copying to overlapping blocks.
Specifying an end address less than the start will cause a DATA ERROR message.

[CLR] Will clear a block of Q-ROM between the specified start and end addresses. If the end address is less than the start, a DATA ERROR message is issued.

[SVE] Allows M-Code routines to be stored in packed data format in main HP-41 memory. The data can then be transfered to mass storage media.
[SVE] expects a hex start and end address, or decimal number of words, of code to save. A secondary prompt requires an absolute register address, in main or XRAM memory, into which the packed data will be stored.
If a non-existent register address is encountered, SVE will abort with the message NONEXISTENT. Specifying an end address less than the start will cause a DATA ERROR message.

[GET] Recalls main or XRAM memory registers and writes the data contained therein to a Q-ROM device. Expects the first and last register absolute hex-addresses to use, or a decimal number of registers to GET. The ZENROM data format copes with incomplete registers, thereby allowing recall of two consecutive blocks of code without error. A secondary prompt requests the Q-ROM destination address.
A DATA ERROR message results if the end address is less than the start address.

RAMED

RAM Editor (non-programmable)
Provides an editor function, similar to that of the HP-41CX text file editor 'ED', that permits review and replacement of any bytes, or optionally insertion of bytes (program memory only).
Redefines the HP-41 keyboard during execution to allow forward or backwards movement through memory in byte or register increments by pressing the [USER], [PRGM] and [SHIFT] [USER] or [SHIFT] [PRGM] keys.
Pressing the [I] key, toggles between replace and insert mode - signified by the '1' annunciator being lit in the display.
Takes start address from status registers M or b (the program counter), dependent upon mode.
If not in PRGM mode, returns last reviewed address to M upon exit, or if in PRGM mode, exits at line where it entered.
Generates PACKING, TRY AGAIN messages and quits if insertion is attempted when there is no room left to accommodate extra bytes.
During entry of hexcode values, the back arrow key [←] will cancel the first digit input. By pressing and holding the second digit, the whole hexcode entry is nullified - as happens during normal HP-41 key-pressing.
To exit from RAMED, press the [ON] key.

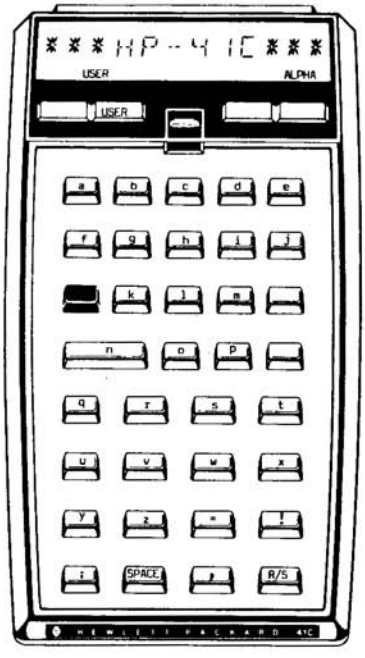
USER ALPHA KEYBOARDS

Entry of alpha characters and text lines, whether as a program line, a postfix to an instruction or directly into the ALPHA register has been greatly enhanced.

ZENROM activates two additional ALPHA-mode keyboards with the [USER] and [SHIFT] keys whilst in ALPHA-mode. These keyboards can be activated whenever ALPHA mode is entered, e.g. even during input of a program label, and provide every displayable HP-41 character plus all lower-case characters defined on keys.
To make text entry easier, a keyboard overlay is included with ZENROM and the two new alpha keyboards are shown below.

The USER ALPHA keyboards do not operate within the HP-41CX Editor ED, nor during a PSE instruction.
Due to HP-41 system restrictions, the normal key rollover does not operate during USER ALPHA entry — take your time typing until you are used to the new keyboards.

UNSHIFTED KEYS



SHIFTED KEYS

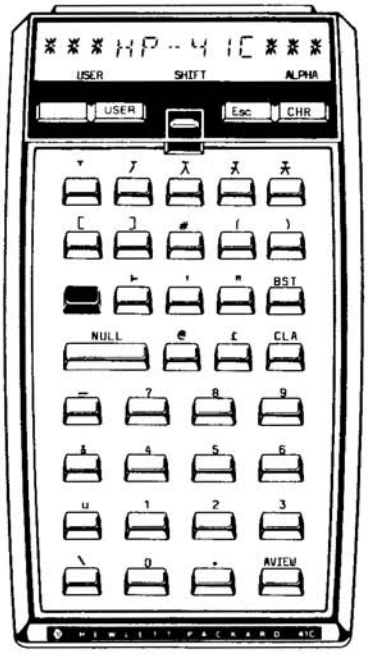


FIGURE 1.2.2 USER ALPHA KEYBOARDS

SYNTEXT ENTRY

Synthetic Text entry allows any of the 256 characters available on the HP-41, whether displayable or not, to be entered in a line of text.

Characters are entered whilst in any ALPHA-mode, by pressing the key sequence [SHIFT][ALPHA]. Two underscore prompts appear at the right-hand edge of the display and the keyboard is redefined so that only the hexadecimal keypad is active, thereby allowing entry of hexcode character values from 00h through FFh.

Like the USER ALPHA keyboards, SYNTEXT entry does not operate during the HP-41CX Text Editor ED, nor during a PSE instruction.

1.3 DIRECT-KEY SYNTHETICS

These non-catalogued functions provide the user with considerable extensions to the HP-41's operating system, thereby allowing access to features that have hitherto only been possible with techniques such as S.P. (Synthetic Programming) developed by members of HP user groups world-wide.

By the use of Direct-Key Synthetics, the User has the possibility of directly accessing the remaining status registers:

M, N, O, P, Q, R (often called ⊢ or append), a, b, c, d, e

as if they were standard user-stack registers.

All HP-41 functions that normally prompt for numeric input, e.g. RCL, STO, VIEW, FIX, etc., have been expanded to allow entry of:

00 through 99
Extended postfix access (100 to 199) by pressing the [EEX] key
Stack register addresses by pressing the [.] key
Indirect arguments by pressing the [SHIFT] key

Note: Due to HP-41 system restrictions, the normal key rollover does not operate during extended prompt entry.

During entry of exponents in PRGM-mode, ZENROM will automatically strip a '1' immediately preceding an [EEX] character if it is the only digit in the mantissa.

CATALOGUE FUNCTIONS

2

CATALOGUE FUNCTIONS

This section of the handbook provides detailed explanations relating to functions appearing in the ZENROM catalogue. MCED (Machine Code Editor) and RAMED (RAM Editor) are more correctly operating modes, and will be detailed in following chapters.

2.1 CLEARING MEMORY

CLMM

CLEARING MAIN MEMORY

CLMM restores HP-41 Main Memory to Master Clear State by storing nulls into every register. In addition, all status registers and flags are restored to default states; all key assignments, timer alarms and input/output buffers are eliminated; and the stack, LASTX and ALPHA are cleared.

HP-41 flags are cleared except for flags 26, 28, 29, 37 and 40 (for a display setting of FIX 4). If a printer is connected, flags 21 and 55 are set.

The size of program memory will be 46 (on the HP-41C and HP-41CV) or 219 on an HP-41CX.

CLMM will operate on all models of HP-41, irrespective of how many memory modules are connected to an HP-41C. All files contained in Extended Memory - both Extended Functions Module and Extended Memory Modules - will be retained completely untouched.

No error message is generated, but the message 'MM LOST' is displayed when CLMM is executed from the keyboard or a running program. Executing CLMM from a running program will cause the program to stop (even if that program is synthetically made to run in Extended Memory, and as such is not erased), because the program counter will be reset to point to the .END., causing the program to halt.

WARNING: Executing CLMM will irrevocably nullify contents of main memory. There is no recovery.

CLXM

CLEARING EXTENDED MEMORY

CLXM overwrites the contents of all existing Extended Memory registers with nulls, whilst still retaining the contents of Main Program and Data Memory in the HP-41.

CLXM will operate on all HP-41 Models and on all valid combinations of Extended Function and Extended Memory Modules, although an error message will not be generated if Extended Memory does not exist.

When executed from the keyboard, CLXM generates the message 'XM LOST'. However, if executed from within a running program this message is suppressed.

After execution, attempting [EMDIR], or [CAT] [4] on the HP-41CX, will produce the message 'DIR EMPTY'.

WARNING: Executing CLXM will irrevocably nullify Extended Memory contents. There is no recovery.

2.2 NON-NORMALISED NUMBERS

In simple terms, a non-normalised number is one that is in a format that the 41 is not used to. The easiest way to describe a non-normalised number is, in fact, to describe a normalised number and then any exceptions to this format can be described as a non-normalised number. For the purposes of this explanation it will be easier to call a non-normalised number a non-normalised register.

Each register comprises 14 digits (each of 4 bits) which will most often contain a real number. This number can be in the range +/- 9.999999999 E99 down to +/- 1 E-99 or 0. Whatever the number is, it is stored in a fixed format in the register - the fourteen digits of the register being:

s m m m m m m m m m m xs x x

where:	's'	represents the sign of the number (0 if +ve, 9 if -ve);
	'mmmmmmmmm'	is the mantissa, ie. the body of the number;
	'xs'	is the sign of the exponent, again 0 if +ve, 9 if -ve; and
	'xx'	are the two exponent digits.

There is an implied decimal point after the first digit of the mantissa.
Example 1: PI = 3.141592654

0 3 1 4 1 5 9 2 6 5 4 0 0 0

Example 2: -1/e = -3.678794412 E-01

9 3 6 7 8 7 9 4 4 1 2 9 9 9

Note that in example 2 the exponent is negative and so the magnitude of the exponent is stored in complement notation i.e.: 1000 - EXP.

In addition, although the second example will display as -0.367... in FIX format, because the decimal point is implied after the first digit of the mantissa, the leading 0 is not stored.

We can call a register non-normalised, if either:

a) the first digit of mantissa is a 0
b) the sign digit is neither 9 nor 0
c) the exponent sign is neither 9 nor 0

There is one other situation that will make a register non-normalised and that is, if any of the digits in that register contain a non-BCD (Binary Coded Decimal) digit - i.e.: a hexadecimal digit A thru F.

For those not familiar with NNNs, their main usages are for data packing, flag control and alpha manipulation.

Using normal HP supported programming techniques, it is almost impossible to create a non-normalised number and most users never even know they exist. This is mostly because the 41 has a nasty habit of 'normalising' numbers. Although this sounds quite painful, it is the process used by the 41 to ensure that a number is in a format that, for example, the maths routines can handle. If a non-normalised number is stored in a register, and that register is recalled using RCL then the register contents will be altered to form a normal number.

There is one special case of NNN that the 41 does support. This is an alpha string which is stored as a series of up to six ASCII coded characters right justified in the register and with the sign digit set to 1 i.e.:

ABC= 1 0 0 0 0 0 0 0 4 1 4 2 4 3

Since it is otherwise impossible to create your own NNNs, a function is required to enable this. When an NNN has been created, it may be displayed in a manner that is not easily decipherable. Therefore, a function is required that will decode an NNN to show its exact contents. Since an NNN can not be recalled from registers without normalisation, functions are also required to allow this. Lastly, since not all of the 41 registers can be directly stored into, a function to allow complete access to all 41 registers, including extended memory, is required.

As a demonstration of the effect of normalisation on a register, for those unfamiliar with synthetic programming, follow this example with the ZENROM plugged in:

```
CF 4                        )
CF 5                        )Ensure flags 4, 5, 6 and 7
CF 6                        )are clear.
CF 7                        )
RCL d | RCL | | . | | D |   ) This a synthetic instruction
```

```
STO 00                          )Store two copies of the contents of
STO 01                          )the mythical register d (an NNN).
RCL 00                          )Recall the NNN from Reg 00
STO d [ STO ] [ . ] [ D ]       ) restore to register d.
```

Notice that the display has now changed to SCI 0 format. This is because register d is the register containing all flags and when the NNN that represented those flags was recalled from Reg 00 it was normalised to the extent that it was changed to 0. Upon restoring to Reg d ALL the flags were cleared. To restore the flags to their original status execute the following instructions:

```
1                               )Recall register 01 without
NRCLX                           )normalisation and restore in
STO d [ STO ] [ . ] [ D ]       ) register d.
```

Notice that the display has now reverted to its original format since when register 1 was recalled it was not normalised.

CODE

CODE ALPHA TO X

Converts the rightmost 14 characters contained in ALPHA into the 14-digit Non-Normalised Number which they specify. This number is returned to the X-register.

CODE does not generate an error message if non-hexadecimal digits are contained in the ALPHA string.

DECODE

DECODE X TO ALPHA

Converts and replaces the contents of ALPHA with the 14-characters representing the 14-digit number in the X-register.

If executed from the keyboard, DECODE instigates an AVIEW instruction to display the returned value. If a printer is attached to the system, DECODE will optionally print the decoded value.

If executed by a running program, DECODE will not perform an AVIEW. To cause the decoded value to be printed insert a PRA instruction into the program following the DECODE.

If single stepped, [SST], DECODE will not instigate an AVIEW.

NRCLM

NON-NORMALISED RECALL BY M

Recalls to register X the contents of the register whose absolute address is in the least significant 3 digits of status register M, i.e. in hex form as the last character and a half of ALPHA. NRCLM generates a NONEXISTENT error message if the register addressed is not physically connected to the HP-41.

The easiest method of entering a register address is via the ZENROM 'SYNTEXT' entry procedure. To use this, go into Alpha mode and press [SHIFT][ALPHA]. At the two digit prompt (righthand side of display) use the hexadecimal keypad (only the keys 0,1,2,....8,9 & A,B,...E,F are now active) to input the characters representing the address of the register. For example, to recall, the Extended Memory register at address 2EFh (see Figure 3.7.1), use SYNTEXT entry in two stages. Remember, however, that NRCLM takes the last ONE AND A HALF characters from register M. Therefore, you must input an extra zero at the front of the address input to the SYNTEXT prompt.

E.g. to input address 2EFh, press the following keys while in Alpha mode:

[SHIFT][ALPHA]	shows two prompts
[0] [2]	input the dummy zero before the first digit of the address. Display shows the starburst character
[SHIFT][ALPHA]	shows new prompts
[E] [F]	displays the second character (also starburst).

To recall the register 2EFh, simply execute NRCLM.

NRCLX

NON-NORMALISED RECALL BY X

Overwrites X register with the contents of the user data register whose register number was specified in X before the function executed. Using NRCLX, only addresses up to the current SIZE can be recalled. The content of register X is saved into LASTX.

Generates NONEXISTENT message if the register number is greater than, or equal to the current SIZE, or ALPHA DATA error if register X contains a string.

NSTOM

NON-NORMALISED STORE BY M

Stores content of register X into the register whose absolute address is specified in the least-significant three digits of M, i.e. the last character and a half of ALPHA.

Generates NONEXISTENT message if the register addressed is not physically connected to the HP-41.

2.3 UTILITY FUNCTIONS

LASTP

GO TO LAST PROGRAM

LASTP positions the user program counter to the first line of the last program in program memory, which is always that program containing the permanent END instruction (i.e. the '.END.' instruction)

In addition to being executed from the keyboard, LASTP can be inserted into a program to produce a very fast GTO during a running program.

NOP

NO OPERATION

When used in PRGM mode, NOP is a 1-byte function that inserts an empty text line, that is an F0h byte, into the program as the next program line.

Alternatively, the function may be entered as an XROM identity, XROM 05,07, by means of the RAM Editor (RAMED), or by assigning NOP to a key and then entering the function into the program with ZENROM removed from the HP-41. If used in this manner, NOP will consume 2 bytes and execute slower.

When entered as a F0h byte, NOP will enable the user stack lift, so it is not really a 'true' NOP. However, when entered as an XROM identity, XROM 05,07, NOP will behave as a true NOP without enabling the user stack lift.

Error messages will be generated as follows:

PACKING, TRY AGAIN	- If there is no room left in program memory for the insertion to take place.
ROM	- If the insertion is attempted into a ROM based program.

TOGF

TOGGLE FLAG

A programmable function to toggle the current status (from set to clear or clear to set) of the HP-41 flag whose number is specified in the X-register as an absolute integer value.

Operates on all 56 user and system flags from 0 to 55. Users should, however, be aware that certain system flags return to a default or conditional status upon halting of the running program. In addition, performing certain operations will also cause the status to reset.

Generates a NONEXISTENT message if the flag number in register X $>$ 55. If register X contains an alpha string, then the ALPHA DATA error message is displayed.

SYNTHETIC PROGRAMMING

3

THE THEORY OF SYNTHETIC PROGRAMMING

To gain the most from the remainder of this User Handbook, we recommend all users to read this chapter. The theory of Synthetic Programming covers many important concepts of the HP-41's operation and a good grasp of this is necessary for any user intending to begin machine language programming.

For this chapter we have assumed the reader is familiar with the HP-41, has an good grasp of User Code (RPN) programming and has read the HP-41 Owners Manual.

It must be stressed that the purpose of this handbook is not to replace those books already written on Synthetic Programming (See Appendix C), but to provide the User with enough information to appreciate the vast benefits that Synthetic Programming and ZENROM provide for the HP-41 programmer. To do this, it has been necessary to pack more than three years work, by User Groups throughout the world, into just a few pages.

If this is your first encounter with Synthetic Programming, we would suggest you take your time reading this chapter and not to worry if you don't fully grasp everything at the first reading. Time spent on this section will be very well rewarded in the future.

Please bear in mind the statement, made at the start of this Handbook, regarding the NOMAS status of Synthetic Programming.

3.1 SP - ORIGIN AND USES

Synthetic Programming (generally called 'SP' for short) is a technique used to enhance the power of the HP-41 programmer. It does this by extending the limits Hewlett-Packard set on the range of instructions executable by the computer.

Whenever a company designs a new product, they set specific limits on what the product is capable of. Such limits may be decided for design, technological or cost reasons, but they also decide implicitly what the product cannot do. So it is with a computer language - and the 'User Code' of the HP-41 is, by any definition of the phrase, a computer language. With a computer, the designers must decide not only what the computer is capable of, but also what the computer User should be capable of.

On the HP-41, this means how the User can manipulate the data entered into the computer, which peripheral devices can easily be communicated with, etc. It also means that there are things which the designers have decided, for one reason or another, that they would rather the User couldn't do. As a simple example; the User cannot (in theory) generate more than ten distinct TONEs, since the TONE instruction will only accept one of ten possible arguments in the range of 0 to 9.

SP is a programming technique widely used among HP-41 users 'in the know' to enhance their programming power by extending the limits set on the computer's language by Hewlett Packard. This is done by taking the individual bytes making up the instructions that can be entered into the HP-41's program memory, and combining them in ways the designers did not anticipate. By this means expanded versions of existing instructions are created or 'synthesised' - which is the origin of the technique's name.

Although there is more to SP than simply rearranging the contents of program memory, other SP techniques (e.g. assigning complete instructions to a key for single-key execution) are best understood once the more basic techniques are mastered. On the HP-41, a complete instruction means an instruction (such as 'STO') and a single argument (such as '00'), combined and available with only one keypress.

As program lines are keyed in, the HP-41 is continually parsing your keystrokes and using them to assemble sensible instructions, according to a pre-programmed dictionary of valid keystroke combinations. Thus, the 41 knows that LBL A is a sensible instruction sequence, but that STO A is not.

Having received a sensible sequence of keystrokes, the HP-41 will store in its memory one or more bytes which were programmed by the designers as representing your keyed-in instructions. For example, LBL A would be stored as two bytes, one of which represents the 'LBL' part of the instruction (also called the prefix byte), and the other representing the 'A' part of the instruction (also called the argument or postfix byte). For LBL 25, the same prefix byte would be used (since this is also a LBL instruction), but with a different postfix byte, since the argument of the instruction is 25, not A. Conversely, a STO 25 instruction would have a different prefix byte (different instruction), but the same postfix byte would be used as for the LBL 25 (same argument).

Fortunately, Hewlett Packard made the instruction set for the HP-41 available to the user community soon after the launch in 1979 - so the coding sequence is well known. Taking the instructions in the last paragraph, we know three possible combinations:

STO 25, LBL 25, LBL A.

The question asked by inquisitive users of the HP-41 was; 'What happens if, somehow, an extra fourth 'illegal' combination, STO A, could be created in program memory?' The answer is that you have a new, 'synthetic' instruction which can, in a single two-byte instruction, store directly into register 102.

To understand how this instruction comes about, and to see the ramifications across all HP-41 programming, let's now analyse the structure of instructions on the HP-41 in more detail. To do this, we need to understand the HP-41 programmers' Rosetta Stone, the HP-41 Byte Table (See Figure 3.1).

This fearsomely complex-looking mass of hieroglyphs contains all the prefix bytes understood by the HP-41, as well as all the postfix bytes and their meanings. In addition, all the display and printer characters represented by the bytes are shown. Although the table is indexed in hexadecimal (to base sixteen) a decimal number index is also shown in each segment. Hexadecimal notation is used because the table contains 256 entries, which nicely fit a square 16 by 16, with base sixteen numbers along the edges.

Probably the most difficult task in coming to grips with synthetic programming is understanding the hexadecimal number system. In everyday life we count in a denary base system (to a base of ten) and numbers increment as follows:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, ...

In order to count in base 16, we need to borrow some new characters from the alphabet to represent the equivalent of 10 through 15. Hexadecimal borrows the first six letters A through F to produce a counting system:

00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, 2F
30, 31,.....

For practice purposes, examine the Byte Table and compare the outer index values against the decimal values in each segment. Note that the table is indexed by row first then column numbers. For example; the value 4Ah (where the 'h' indicates hexadecimal) is: row 4 and column A - which has an indicated decimal value of 74d. For clarification, decimal values will be indicated with a letter 'd' following and hexadecimal values with letter 'h'.

A cross reference chart between the commonest numbering systems is given in the Reference Tables in Appendix E.

HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	CAT	@c (GTO..)	DEL	COPY	CLP	R/S	SIZE	BST	SST	ON	PACK	←(PRGM)	USR/P/A	2 _ _	SHIFT	ASN	
0	NULL 00 ⁻ 0 ◆	LBL 00 01 ⁄ 1 ×	LBL 01 02 ▮ 2 √‾	LBL 02 03 ▮ 3 ←	LBL 03 04 ∫ 4 α	LBL 04 05 ∂ 5 β	LBL 05 06 ⁻¹ 6 Γ	LBL 06 07 ▮ 7 ↓	LBL 07 08 ▮ 8 Δ	LBL 08 09 ▮ 9 σ	LBL 09 10 ▮ 10 ◆	LBL 10 11 ▮ 11 ↘	LBL 11 12 ⁻¹ 12 ↵	LBL 12 13 ⁄ 13 ⁄	LBL 13 14 ▮ 14 ⁻¹	LBL 14 15 ▮ 15 ⁻¹	0
1	0 16 ▮ 16 θ	1 17 ▮ 17 Ω	2 18 ▮ 18 δ	3 19 ▮ 19 ∂	4 20 ▮ 20 ∂	5 21 ▮ 21 ∂	6 22 ▮ 22 ∂	7 23 ▮ 23 Ö	8 24 ▮ 24 ö	9 25 ▮ 25 Ü	. 26 ▮ 26 ü	EEX 27 ▮ 27 Æ	NEG 28 ▮ 28 æ	GTO ⁻¹ 29 ⁄ 29 ≠	XEQ ⁻¹ 30 ▮ 30 £	W ⁻¹ 31 ▮ 31 ∑	1
2	RCL 00 32 32	RCL 01 33 : 33 !	RCL 02 34 " 34 "	RCL 03 35 ∂ 35 #	RCL 04 36 ∫ 36 $	RCL 05 37 % 37 %	RCL 06 38 ∂ 38 &	RCL 07 39 ' 39 '	RCL 08 40 (40 (	RCL 09 41) 41)	RCL 10 42 ∗ 42 *	RCL 11 43 + 43 +	RCL 12 44 , ⁻¹ 44 ,	RCL 13 45 - 45 -	RCL 14 46 . ⁻¹ 46 .	RCL 15 47 ⁄ 47 /	2
3	STO 00 48 ⊡ 48 0	STO 01 49 / 49 1	STO 02 50 ∂ 50 2	STO 03 51 ∂ 51 3	STO 04 52 ∫ 52 4	STO 05 53 ∫ 53 5	STO 06 54 ∂ 54 6	STO 07 55 ∂ 55 7	STO 08 56 ∂ 56 8	STO 09 57 ∂ 57 9	STO 10 58 :▮ 58 :	STO 11 59 ; 59 ;	STO 12 60 ⁄ 60 <	STO 13 61 ⁻ 61 =	STO 14 62 ∫ 62 >	STO 15 63 ? 63 ?	3
4	+ 64 ∂ 64 @	− 65 A 65 A	∗ 66 ∂ 66 B	/ 67 ∂ 67 C	X<Y? 68 ∂ 68 D	X>Y? 69 ∂ 69 E	X≤Y? 70 F 70 F	Σ+ 71 ∂ 71 G	Σ− 72 H 72 H	HMS+ 73 I 73 I	HMS− 74 J 74 J	MOD 75 K 75 K	% 76 L 76 L	%CH 77 M 77 M	P→R 78 N 78 N	R→P 79 ∂ 79 O	4
5	LN 80 P 80 P	X↑2 81 ∂ 81 Q	SQRT 82 R 82 R	Y↑X 83 ∫ 83 S	CHS 84 T 84 T	E↑X 85 ∪ 85 U	LOG 86 V 86 V	10↑X 87 W 87 W	E↑X-1 88 X 88 X	SIN 89 Y 89 Y	COS 90 Z 90 Z	TAN 91 [91 [	ASIN 92 \ 92 \	ACOS 93] 93]	ATAN 94 ^ 94 ↑	→DEC 95 _ 95 _	5
6	1/X 96 ⁻¹ 96 ⁻	ABS 97 ∂ 97 a	FACT 98 ∂ 98 b	X≠0? 99 ∂ 99 c	X>0? 100 d 100 d	LN1+X 101 ∂ 101 e	X<0? A ▮ 102 f	X=0? B ▮ 103 ∂	INT C ▮ 104 h	FRC D ▮ 105 i	D→R E ▮ 106 j	R→D F ▮ 107 k	→HMS G ▮ 108 l	→HR H ▮ 109 m	RND I ▮ 110 n	→OCT J ▮ 111 o	6
7	CLΣ T ▮ 112 p	X<>Y Z ▮ 113 q	PI Y ▮ 114 r	CLST X ▮ 115 s	R↑ L ▮ 116 t	RDN M [▮ 117 u	LASTX N \ ▮ 118 v	CLX O] ▮ 119 w	X=Y? P↑ ▮ 120 x	X≠Y? Q_ ▮ 121 y	SIGN ⊢ ⁻¹ ▮ 122 z	X≤0? a ▮ 123 ∂	MEAN b ▮ 124 l	SDEV c ▮ 125 →	AVIEW d ∫ 126 Σ	CLD e ⊢ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	
	00 01 02 03	04 05 06 07	08 09 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47	48 49 50 51	52 53 54 55	← bit numbers in a 7-byte register		

Figure 3.1a

HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 x̄	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 β	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ	CLRG IND 10 138 ◆	AOFF IND 11 139 ↘	AON IND 12 140 μ	OFF IND 13 141 ∠	PROMPT IND 14 142 r	ADV IND 15 143 ⇞	8
9	RCL IND 16 144 θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 ä	ST ✱ IND 20 148 ä	ST/ IND 21 149 ä	ISG IND 22 150 ä	DSE IND 23 151 ö	VIEW IND 24 152 ö	Σ REG IND 25 153 Ü	ASTO IND 26 154 ü	ARCL IND 27 155 Æ	FIX IND 28 156 æ	SCI IND 29 157 ≠	ENG IND 30 158 £	TONE IND 31 159 ⁂	9
A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 IND 39 167 '	SF IND 40 168 (	CF IND 41 169)	FS?C IND 42 170 *	FC?C IND 43 171 +	FS? IND 44 172 ,	FC? IND 45 173 −	GTO XEQ IND IND 46 174 .	SPARE IND 47 175 /	A
B	SPARE IND 48 176 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9	GTO 09 IND 58 186 :	GTO 10 IND 59 187 ;	GTO 11 IND 60 188 <	GTO 12 IND 61 189 =	GTO 13 IND 62 190 >	GTO 14 IND 63 191 ?	B
C	GLOBAL IND 64 192 @	GLOBAL IND 65 193 A	GLOBAL IND 66 194 B	GLOBAL IND 67 195 C	GLOBAL IND 68 196 D	GLOBAL IND 69 197 E	GLOBAL IND 70 198 F	GLOBAL IND 71 199 G	GLOBAL IND 72 200 H	GLOBAL IND 73 201 I	GLOBAL IND 74 202 J	GLOBAL IND 75 203 K	GLOBAL IND 76 204 L	GLOBAL IND 77 205 M	X<>-- IND 78 206 N	LBL -- IND 79 207 O	C
D	GTO -- IND 80 208 P	GTO -- IND 81 209 Q	GTO -- IND 82 210 R	GTO -- IND 83 211 S	GTO -- IND 84 212 T	GTO -- IND 85 213 U	GTO -- IND 86 214 V	GTO -- IND 87 215 W	GTO -- IND 88 216 X	GTO -- IND 89 217 Y	GTO -- IND 90 218 Z	GTO -- IND 91 219 [	GTO -- IND 92 220 \	GTO -- IND 93 221]	GTO -- IND 94 222 ↑	GTO -- IND 95 223 _	D
E	XEQ -- IND 96 224 ▼	XEQ -- IND 97 225 a	XEQ -- IND 98 226 b	XEQ -- IND 99 227 c	XEQ -- IND100 228 d	XEQ -- IND101 229 e	XEQ -- IND102 230 f	XEQ -- IND103 231 g	XEQ -- IND104 232 h	XEQ -- IND105 233 i	XEQ -- IND106 234 j	XEQ -- IND107 235 k	XEQ -- IND108 236 l	XEQ -- IND109 237 m	XEQ -- IND110 238 n	XEQ -- IND111 239 o	E
F	TEXT 0 IND T 240 p	TEXT 1 IND Z 241 q	TEXT 2 IND Y 242 r	TEXT 3 IND X 243 s	TEXT 4 IND L 244 t	TEXT 5 IND M [245 u	TEXT 6 IND N \ 246 v	TEXT 7 IND O] 247 w	TEXT 8 IND P ↑ 248 x	TEXT 9 IND Q _ 249 y	TEXT10 IND ⊢ 250 z	TEXT11 IND a 251 ⎓	TEXT12 IND b 252 |	TEXT13 IND c 253 →	TEXT14 IND d 254 Σ	TEXT15 IND e 255 ⊢	F
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

For price information and a list of dealers in your area, send a self-addressed stamped envelope to: SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266, USA

Figure 3.1b

3.2 BYTES AND MEMORY

The byte is the basic unit of HP-41 memory and in RAM consists of eight bits. Bits (binary digits) being either 0 or 1; on or off; etc. Each byte can therefore be coded at bit level to a range of 218 entries - which can also be represented by 255 decimal, or FF hexadecimal. It is normal to refer to these bytes as two hexadecimal digits between 00h and FFh.

Every instruction that the 41 is capable of executing comprises one or more bytes from this byte table, which is used by the machine to assemble and later decode these sequences. Whenever the HP-41 attempts to execute an instruction stored in its program memory, it firstly reads the first byte of that instruction. Because many instructions only have one byte, the function can be executed immediately. However, many others have more than one byte, and so the HP-41 must also read these before it can execute the instruction. This is where the idea of prefix and postfix bytes comes from - with some instructions having no postfix; some having one byte to code the postfix; some having two; and even some special instructions having a variable number.

Those instructions that can take one or two postfix bytes have the number of bytes implicitly stored in the prefix byte. E.g., a VIEW instruction always has one postfix byte as it is defined to be a two byte instruction. Once the prefix byte for 'VIEW' has been read and decoded by the HP-41, it knows to fetch the postfix byte before it can understand exactly what is to be executed. Those instructions taking a variable number of bytes actually contain a piece of information telling the 41 how many more bytes to read.

3.3 THE BYTE TABLE

Let's examine the Hexadecimal Byte Table in more detail to determine which bytes code for what and the nature of 'implicit storage' should become clear.

Five rows of the table contain 'single-byte' functions, i.e. having no postfix bytes. These are in rows 4 to 8. There has been some attempt at functional grouping within these rows, but the very nature of the functions makes this difficult to achieve.

Number-entry bytes - the digits 0 to 9, EEX, NEG (which is what you get when you press CHS whilst entering a number, because CHS is also a function in row 5) and 'Point' (' . ' or ' , ' dependent on flag 28) are in row 1, along with two special instructions of variable length (GTO'alpha' and XEQ'alpha', of which more later), and a 'spare' byte (HP didn't use it to code for any particular prefix).

Rows 0, 2 and 3 are 'special' single-byte functions, used as memory savers. HP realised that, in contrast to their previous handheld machines - where the few distinct instructions allowed all possible combinations to be coded with just 256 entries (i.e., all single bytes) - on the 41 there are so many logical combinations of functions with allowable arguments that all functions which took such arguments would consist of more than one byte.

Because, at that time, they were designing the 41C with only 445 bytes of User-programmable memory, they decided to offer some memory saving to the User by allowing the commonest register operations (STO and RCL) to be combined with the most frequently-used register arguments (the lowest ones) and placed in single-byte functions. Thus, STO 00 to STO 15, and RCL 00 to RCL 15 are, single bytes when stored in RAM. (The range of registers, 00 to 15, in these instructions, is no accident — both instructions occupy a single row in the byte table, and use the column index digit to select the register — 0 to F).

HP also realised that their system of only allowing transfer of program control to a label, rather than a line number, meant that a significant proportion of program memory would be taken up by LBLs and GTOs. Normally, a local label is two bytes and a GTO to that label is three (explained later). However, HP set aside part of the Byte Table to give the programmer the ability to trade off memory consumption against a limited local label choice in the range 00 to 14, and maximum branch distance of 16 registers (or 128 bytes if you are in ROM). These LBLs are shown in row 0, and are all single bytes, and the GTOs are in row B. The arguments only have fifteen allowed entries, because one byte from row zero is used as the 'null' byte (one used by the 41 as a place holder in program memory, and normally invisible to the User). Row B was built to mimic this, with the byte B0h being 'spare'. The GTO structure is complex, and will be discussed after the general structure of two-byte functions is clarified.

3.4 MULTI-BYTE INSTRUCTIONS

A two-byte function, with the exception of XROMs and short-form GTOs, always consists of a prefix byte which completely defines its function, and a postfix byte which completely defines its argument. As an example, we'll look at how a TONE instruction can be built up. The prefix byte that defines the function is 9Fh. The argument is found from the argument entry within each table segment. A TONE 4 instruction would need a byte which meant 04 when used as an argument byte. This is byte 04.

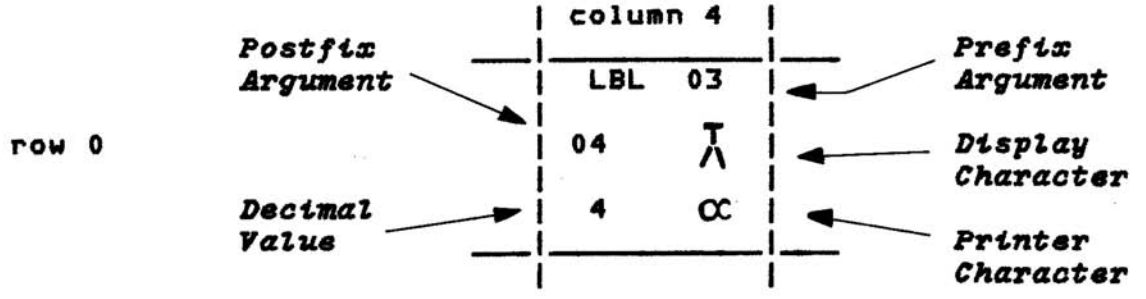


Figure 3.4

In case you're wondering why this doesn't code for a LBL 03, it's because a byte is interpreted according to the context in which it is found. So byte 04 means argument 04 when it is read in by the 41 as an argument byte; LBL 03 if it is read in as a prefix byte; the character ⌶ (a two-legged hangman) when it is placed into the display as part of a text string; and the Greek character α (alpha) when printed out on an HP-82143 or 82162 Thermal Printer. It can even mean the function CLP if it is found in a key assignment register, but we'll come back to that later.

The TONEs 0 to 9 are coded by the byte sequences 9F,00h through 9F,09h. All two-byte instructions are assembled in this format. Two-byte prefixes which operate in this form are row 9; the section of row A from A8h to ADh; and bytes CEh and CFh. Prefix bytes 90h to 9Bh and CEh are all the 41 register operations; 9Ch to 9Eh are display operations; 9Fh is the TONE, A8h to ADh the flag operations; and CFh the local label instruction. According to the HP Owner's Manuals the allowed postfixes for the HP-41 are 00d to 99d, A to J and a to e (for LBL instructions), and X, Y, Z, T, and L.

6	1/X 96 ˣ 96 ˣ	ABS 97 a 97 a	FACT 98 b 98 b	X≠0? 99 c 99 c	X>0? 100 d 100 d	LN1+X 101 e 101 e	X<0? A 102 f	X=0? B 103 g	INT C 104 h	FRC D 105 i	D→R E 106 j	R→D F 107 k	→HMS G 108 l	→HR H 109 m	RND I 110 n	→OCT J 111 o	6
7	CLΣ T 112 p	X<>Y Z 113 q	PI Y 114 r	CLST X 115 s	R↑ L 116 t	RDN M[117 u	LASTX N \ 118 v	CLX O] 119 w	X=Y? P↑ 120 x	X≠Y? Q_ 121 y	SIGN ⊢ ᵀ 122 z	X≤0? a 123 π	MEAN b 124 l	SDEV c 125 →	AVIEW d Σ 126 Σ	CLD e ⊢ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

Row 6 of the byte table contains the postfix bytes that code for postfixes A to J, and row 7 contains those for postfixes X, Y, Z, T and L at the start, and a to e at the end. Earlier, we suggested that STO A could be used to address register 102 directly. If you look at the decimal equivalent of the A postfix, found at segment 66h, you'll see that it is 102. The same applies to postfixes B to J which can access registers 103 to 111. Postfixes T to L, the order in which they appear in row 7, access stack registers T to L in the memory of the 41. The rest of row 7 access registers M to e , but we'll come back to those later.

Adding the two 'unnamed' postfixes 100 and 101, which appear simply as '00' and '01' when used with register or LBL operations, but do access registers 100 and 101, we have 128 postfix bytes, thereby using half the Table. The other half can also access exactly the same registers, but take an INDirect qualifier on the postfix. So, for example, the code for STO IND 29 is 91,9Dh. In order to find the indirect postfix for a known postfix value just add 80h, or 128d to the direct access coding.

Back to the structure of two byte instructions. Byte AEh is the prefix byte for a two-byte function, but does two jobs. If the postfix byte is from the upper half (00h to 7Fh) of the Table, then the instruction is a GTO IND xx - or whatever the register was, while, if the postfix comes from the lower half, then the instruction is an XEQ IND xx. The argument is always INDirect, with the presence or absence of the extra 80h choosing the function.

XROMs represent one of a class of functions which are chosen not by the prefix byte, but by the first digit of that byte - the prefix nybble, if you prefer (a nybble being half a byte). The coding is as follows:

- if the first nybble is A, then the function is an XROM if the second half of the prefix byte is in the range 0 to 7.
- if the first nybble is C, then the function is a global instruction (either an END or an alpha LBL);
- if it is a D, the function is a normal (long-form) GTO;
- if an E, it is a numeric XEQ;
- if it is an F, then the function is a line of text.

The purpose of an XROM instruction is to code for an instruction that is 'not known' to the 41, i.e., a catalogue two instruction. These instructions are coded by giving each possible plug-in device an identifier number (in the range 0 to 31), and allocating each device up to 64 possible functions in its catalogue (numbered 0 to 63). Then, taking the binary for the identifier (five bits), and the binary for the function (6 bits), they can be assembled into the last eleven bits of the two-byte function to give the complete XROM, thus:

1010 0ddd ddff ffff

where:
- ddddd is the device number;
- ffffff the function number;
- 1010 is the A first nybble in binary;
- and 0 at the start of the second nybble is always zero.

XROM numbers can be seen if you place a plug-in device function in program memory, then remove the device that contains it.

In general, function number '00' is used as the header for the ROM or plug-in device, and is a function that normally does nothing except execute a return. The header should be longer than seven characters for the HP-41CX to pick up the ROM header during a CATalogue 2, as it looks for all entries longer than 7 bytes. An exception is the Math Pac 1B header, which is only seven, and can therefore be called as a function. Place it into a program, remove the Math Pac, and see XROM 01,00 in the display. The hex-code for the two bytes in program memory is A0,40h. Similarly, the hex-code for the printer function PRKEYS is A7,4Ch , since its XROM code is 29,12 (decimal). Note that although, in theory, device 0 is possible, in practice it is not and never will be used. The reasons for this aren't important here, but XROM 00,nn can crash the HP-41 by locking up the keyboard, and thereby stop the User from re-gaining control. This should be avoided.

There are two forms of local GTO instruction, called short- and long-form. Short-form GTOs save program memory, but are restricted to GTOs of '00' to '14', and are coded in two bytes. The coding is as follows:

1011 llll drrr rbbb

where:
- llll is the label desired plus one;
- d is the direction in which the destination label lies (0 means forward, the direction of increasing line numbers, 1 backwards);
- rrrr is the number of registers away;
- and bbb the number of bytes.

The postfix byte thus contains the jump to the label. This is added to, or subtracted from (if 'd' is one), the current program counter - the datum that tells the 41 which instruction to execute next. When the GTO is first entered, this byte is a null (zero), that tells the 41 that the instruction has yet to be 'compiled'. Compiling is the 41's process of finding the label and storing the branching information into it - this being accomplished during the first execution of the program line.

If the GTO is in a ROM (a plug-in module), then the last seven bits of the GTO are the number of bytes to the label - because in ROM there are no registers. For ROM programs, this can be up to 127 either way, so a branch in a ROM can actually be slightly further than in RAM. Note also that the direction bit alters in ROM - here a '1' means 'forward' in the program, and a '0' means 'backward'. The reason for this is that as you move forwards in a program, the actual address of the line decreases, while in a ROM program, the address increases. Therefore, a '1' in the direction bit means in the direction of increasing addresses.

The other form of GTO, the long-form GTO, takes up three bytes of memory, and has a structure which is almost identical with that of the numeric XEQ instruction. The coding for these two instructions is:

11tt bbbr rrrr rrrr dlll llll

where:
- rrrrrrrrr is the number of whole registers away;
- bbb the number of bytes in addition to the number of registers;
- d the direction;
- lllllll the label, thus allowing 128 possibilities, (all as for short-form GTOs - detailed above).
- and tt being the bit pair.

The extra piece of information here is the tt bit-pair, which will be either '01' - in the case of GTOs, or '10' in the case of XEQs. Note that there are nine bits used to code for the number of registers to jump - this being the minimum number needed to allow for jumps of 319 registers and the maximum number you are legally supposed to be able to jump within HP-41 program memory.

The HP-41 compiles each line only as it executes it, which is why the line 'speeds up' after one execution. If a short-form GTO cannot be compiled, then the HP-41 conducts a search every time. A program is always decompiled if you:
- delete a line;
- insert a line (overwriting nulls already in a program doesn't cause it to decompile, but the program wasn't packed then anyway if it did contain nulls);
- or delete the END of the program preceding it in memory (this includes using CLP to clear that program.)

3.5 VARIABLE-LENGTH INSTRUCTIONS

Row F is used to code for text strings - these have also been called ASCII-strings by HP . If the first byte of any program line is 'Fn', where 'n' may be any hex digit, then the HP-41 interprets the following 'n' bytes as alpha characters - quite regardless of what they may be, as absolutely any byte, from 00h to FFh, may be found in a text line. The text characters corresponding to each byte are indicated in the Byte Table. In fact, there are only 83 distinct display characters - one of these (the starburst character), which is officially coded for by byte 3Ah, 'stands-in' for all the undisplayable bytes, and as such is the commonest display character in the Byte Table.

Because a text string's length is coded by a single nybble, 0 to F, this means that a string may have up to 15 characters. It may also have none at all - e.g., in the case of the 'empty' string, '', which is coded by F0h in a program. This empty string byte is used by SP'ers as a NOP (No-OPeration = an instruction which does absolutely nothing - other than take up space in a program, and waste a little time when you run it). In actual fact, the F0h byte does enable the stack lift - so is not quite a true NOP. The ZENROM function NOP is a non-programmable function that inserts this line in program memory for you.

Strings also come into the coding of other instructions, namely those of GTO 'alpha', XEQ 'alpha' and LBL 'alpha', where 'alpha' is any string. The coding of the first two of these is quite simple, while the coding of the last is somewhat more complex. Given any string, to turn that into a corresponding alpha GTO, place the byte 1Dh before it in program memory. This byte tells the 41 to expect a string argument instead of a numeric one - as normally occurs on the other two- and three-byte instructions. By placing the byte 1Eh here instead, you have an alpha XEQ line. Placing 1Fh in front of the string creates a curiosity line, W'', but beware as this can cause system crashes.

Alpha labels are one of a category of instructions called 'globals', with the other instruction in this category being the END instruction. These two instructions between them bind the HP-41's program memory together. The HP-41 maintains, somewhere in the dark depths of the Status registers (explained shortly) a piece of information that tells it where in memory the .END. can be found. This is the first of a series of instructions which form what is called a 'linked list'. The .END. contains a pointer (a relative jump, similar to that stored in XEQ/GTO instructions) which allows the 41 to find the next global up memory. Each global in turn points to the next one up, be it a LBL or an END, until the last global in the chain is reached. This contains a pointer value of zero, meaning that there are no more entries in the chain.

If this 'global chain' sounds familiar, do a CAT 1 and watch it go by in reverse order. The simpler of the two instructions, the END, has a structure of:

1100 bbbr rrrr rrrr 0ntx pppp

where:

- bbb and
- r rrrr rrrr are coded as for XEQ/GTO instructions above, but no direction bit is needed, since the next global along the chain is always farther 'up' program memory (nearer the start of CAT 1);
- n is a system status bit, which the HP-41 uses internally, and which shouldn't be altered - normally it is a 0 anyway.
- t marks the type of the END. Where: 0 is a regular END; and 1 is the .END. (the permanent end instruction);
- x is a 'don't care' bit - as it doesn't matter what it is in an END;
- pppp marks the packing status of the program - If it is 1001, then the program is of packed status; if 1011, then the program has been altered and needs to be decompiled. This is the action which takes up time when you quit program mode after an editing session. If 1101,
  then the program has already been decompiled, but still needs packing.

Should the third byte of the global be Fn (where n is any hex digit), then the line is interpreted as a global label. The n bytes which follow the third byte contain two things; the alpha label itself (starting in byte 5), and a byte (byte 4) containing the keycode of the key to which the label is currently assigned. This is why a CAT 2 or CAT 3 function can be assigned anywhere on the keyboard, whereas, a User program can only be assigned on one key location at a time, because the assignment information only allows for one per label. Keycodes will be dealt with shortly. For the moment, however, note that the n in Fn is the length of the label in characters (as it appears in the display) plus one to allow for the keycode byte.

The only other byte in the Table yet to be dealt with is AEh, which serves for two lines - XEQ IND and GTO IND. The suffix byte for this instruction is interpreted as:

trrr rrrr

where:

- rrrrrrr is the register which the 41 will use to supply the indirect branch;
- t is the type of instruction. A 0 here means GTO IND, or a 1 means XEQ IND. This explains why the half of the Table, from which the suffix byte comes, determines the choice of the instruction; either upper-half for the GTO instruction, or the lower-half for the XEQ.

Since the HP-41 has no control over the contents of the register that is chosen to select the branch, the instruction cannot be compiled, so no room need be reserved for this purpose.

The bytes CEh and CFh can be used to code for LBL nn and X<> nn, despite the fact that Cn should be a global instruction. This is because CEh and CFh cannot occur in globals, as there are only 7 bytes per register. This means that the instructions which theoretically code for the references to byte 8 are used for something else instead. Bytes DEh, DFh, EEh, and EFh cannot be similarly used (thus explaining why the XEQ/GTO instructions run right to the end of those rows). The reason is that, in a ROM, all twelve bits comprising the bbb and rrrrrrrrr fields contain instead the number of bytes to the corresponding label - which means that a ROM program can be 4K long. This doesn't affect most normal HP-41 users, but this discrepancy in the Byte Table is worth pointing out.

3.6 REGISTER FORMATS

HP-41 RAM comprises space for 1024 56-bit registers. Of these, 16 are used in the Status Registers, 320 for 41-Main Memory, 128 as base XRAM (Extended Memory inside the EXtended Functions Module) and two blocks each of 239 registers for XRAM1 & XRAM2. The total of 942 registers is unlikely to be extended as many of the currently 'unused' locations are used by system routines.

Each register comprises 14 digits (also called nybbles), which can be treated as 7 bytes. These digits are numbered from the right-hand end of the register as '0' through '13' - these being the bytes '0' through '6'. To make this clearer, let's imagine a typical HP-41 register containing the value for 'minus pi * 100' (-3.141592654 * 10↑2).

Digit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	9	3	1	4	1	5	9	2	6	5	4	0	0	2
Field	s	m	m	m	m	m	m	m	m	m	m	xs	x	x
Byte	6		5		4		3		2		1		0	

Figure 3.6. REGISTER FORMATS

Figure 3.6. REGISTER FORMATS

The notation used indicates that 's' is the sign of the number, while 'xs' is the sign of the exponent. 'mmmmmmmmmm' is the 10 digit number stored with an implicit decimal point after the first digit (between bytes 6 & 5). The actual number is not stored as a binary fraction - as is normal on other larger computers - but in Binary Coded Decimal (BCD) format. This means that the number is stored a digit at a time, with each in the range 0 to 9. The exponent is stored as a BCD integer in the range 00 to 99 for positive numbers and 99 to 01 for negative exponents of -01 to -99. The exact reason for this is not relevant here, but is connected with the most advantageous method of performing mathematical functions. The sign digit is stored as either 9 (negative numbers) or 0 (positive).

Thus the number -1.23456789 * 10↑-3 would be stored as 91234567890997. To find out what the exponent field should be (including sign value), add the exponent to 1000 and take the last three digits.

Of course, registers are also used to store strings or text information. To represent these, the sign value is '1' and the string data are stored in bytes 5 to 0 in right-justified format with zeroes padding to the left. E.g. the string 'STRING' is stored as 10535452494E47, while 'ABC' is stored as 10000000414243. Note that the codes used are those for corresponding characters from the Byte Table.

HP themselves do not use any other digits in the sign field of explicitly stored numbers, although it is possible to create such numbers. These are called Non-Normalised Numbers (NNNs), and the HP-41 has a nasty habit of changing them if it finds out that you are using them. NNN's will be discussed elsewhere in this handbook.

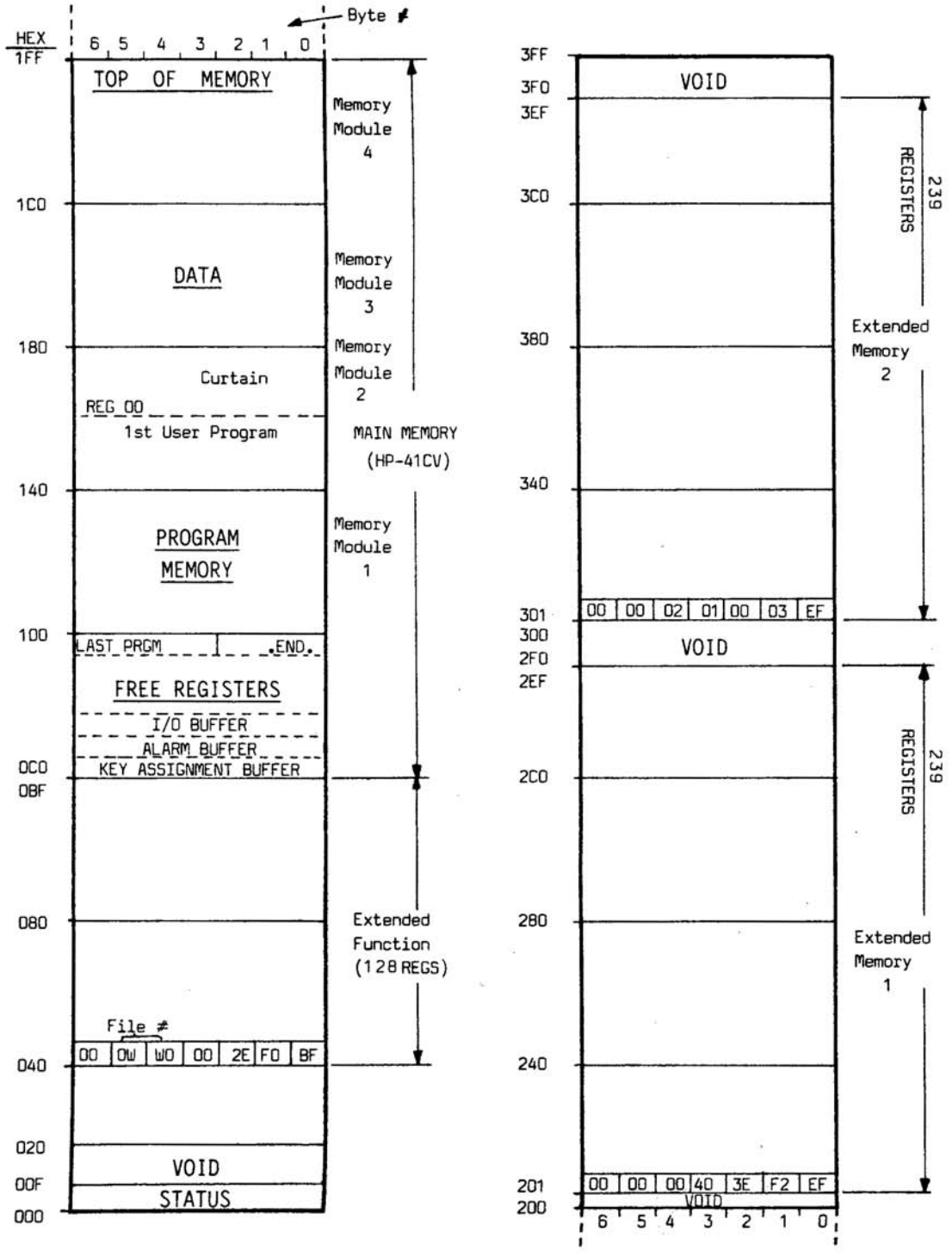


Figure 3.7.1 HP-41 Memory Configuration

3.7 MEMORY STRUCTURE

A detailed memory map of the contents of Main Memory is shown in Figure 3.7.1. This indicates that HP-41 memory can be broken into five areas - these being all 'dynamically allocated', that is, their location in memory will vary according to their presence and size. The only requirement is that the ordering of the five blocks is always maintained. The topmost, or highest-addressed, area of HP-41 memory is the user Data Register block, which is where the numbered registers are to be found. The location of R00 (register 00) is called the 'curtain'. This data block can be zero registers long, i.e., not actually there.

The program area comes next, with the first byte of the first program being in byte 6 of r (curtain - 1). (Note: The manual will adopt the notation of r *xyz* to indicate an absolute register address as marked on the side of the memory map - R*nn* will mean the nn'th register in the User Data space). The last register of the program area always contains the permanent .END. in bytes 2 to 0 of that register - this is the only instruction in HP-41 program memory which always occupies the same place within a given register.

After this comes 'free space'. This means those registers that as yet have not been allocated to any of the other four blocks. Their amount is the number the 41 computes for the display 00 REG nnn. This block can be zero registers long, in which case the 41 has run out of available registers.

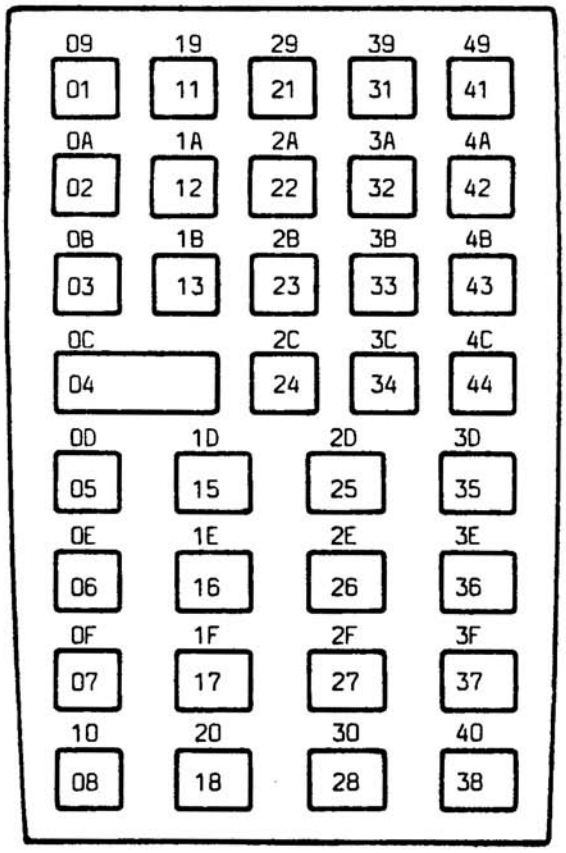
Next comes the buffer block, which contains buffers - a buffer, is a block of contiguous registers currently being used to contain data for a plug-in device, for example, timer alarm information. All buffers have a similar structure, with the lowest-addressed register in the buffer being interpreted as follows:

FIGURE 3.7.2 BUFFER FORMATS

where:
- t is the type of the buffer, and is used by plug-in devices to decide which buffer(s) belong to it (all timer buffers have A here, HP-IL buffers are B (Plotter) or C (IL DEV)).
- ss is the total number of registers in the buffer;
- and x digits are 'don't care' digits.

Digit 13 is usually the same as for 12, but is only ever used when the HP-41 switches on to determine if the buffer is valid (the 41 clears this digit, and then gives any plug-in devices the chance to replace their own type digit here. If, after this, any buffer remains with a 0 here, it is cleared out, and the registers used returned to free space).

If you start playing with buffers, don't create one with an ss field of 00. The HP-41 considers this to be an empty buffer, because it finds an ss field signifying an empty register. Switching the 41 off then on, will cause the keyboard to 'lock-up' - which is a nice way of saying the machine



Hex key codes as used in KA registers and in Global LBL's

Figure 3.7.2 Key Assignment & Global Label Keycodes

will not respond to any keystrokes you make. What is actually happening is that the HP-41 has got stuck in its internal buffer-validating code with no hope of exiting. The only way out is with a Master Reset (MEMORY LOST), or by interrupting the power supply for quite a while.

The highest-addressed register always contains:

F0 XX XX XX XX XX XX

where XX's sometimes are clear, but may contain a string name of the device controlling the buffer. This is not a necessity in buffer construction, but the plug-in devices may require the string to be there, so don't change it.

The buffer area can be empty and it always runs right up against the last, lowest-addressed block, which contains key assignment information. Each Key Assignment register contains two assignments, formatted as:

F0 cc cc kk cc cc kk

where:

- cc cc is the one- or two-byte code representing the function. If from CAT 2, the XROM instruction coding for the desired function is stored here. If from CAT 3, then the first byte is 0n (usually 04h), and the second is the byte from the Byte Table that codes for that particular function. C0h means an END instruction, D0h a GTO and E0h an XEQ. The non-programmable functions are coded from row 0, according to the line above the top row of the Byte Table. For example, 04 codes for CLP when in a key assignment register.
- kk is the field for the keycode of the assignment.

Keycodes for the HP-41 are shown in Figure 3.7.2. Note that these are also the same codes as used for global LBL instructions.

If there are an odd number of assignments, then absolute register 0C0h will contain zeroes in bytes 5 to 3 inclusive. Whenever an assignment is cleared, only the kk field is cleared immediately. However, PACKing memory (or switching the HP-41 off/on) will cause registers with 00 keycodes in both fields to be removed, and adjacent registers each with a single assignment to be merged into one.

There is one restriction regarding the key assignment/buffer area block. It must not contain any completely empty registers - i.e., all zeroes. The HP-41 will consider the first empty register it finds to be the end of the key assignment area (it doesn't actually look inside the buffers, but skips along the first register of each): any further data above this point become 'detached' information that clutters free space, but which the HP-41 isn't intelligent enough to remove. In addition, a buffer must not contain any empty registers because of a bug in some Card Readers. The Card Reader HP-67/97 translation function 'borrows' free space, but searches from the wrong end of memory. While the 41 allocates free space from the .END. downward, the Card Reader looks for it from r 0C0h upwards and stops at the first empty register found. It has never heard of buffers, and so may write rubbish into the buffer area.

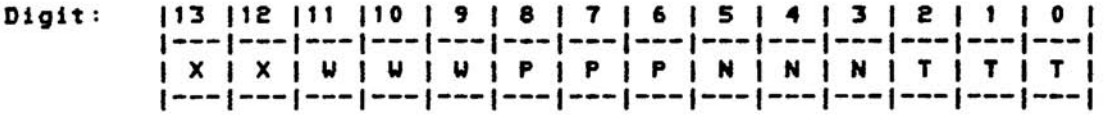
To avoid this problem, the Time Module, and the CX, force non-null bytes into any message string stored with an alarm if that message contains enough contiguous nulls for there to be an empty register in the Timer alarm buffer.

Extended Memory on the HP-41 (XM) is a virtual memory area, which appears to the user to be a contiguous block of registers, either 124, 367 or 600 registers long. In fact, it is composed of three separate blocks of memory, in three uncontiguous locations, linked together by the XFM's internal programming. The lowest block, in the XFM itself, called the 'base' block, is actually 128 registers long. The other two blocks, called XRAM1 (left-hand ports) and XRAM2 (right-hand ports) are each 239 registers long. This gives a total of 606 registers - with 6 registers being for system use.

One of these six, called the 'partition' register, is a register which contains FF:FF:FF:FF:FF:FF:FFh, and marks the end of used XM. It is never available to the User for use - XFM will replace it with its original value at the first opportunity. Two more registers are the overhead for the next file's header registers. Each file uses two registers over and above the size given by EMDIR and the XFM reserves space for the header of the next file still to be created. The three remaining registers are the 'link' registers containing information for the XFM on which XRAM blocks are available, and the order in which the modules containing those blocks were plugged in.

Each of the three blocks has its own link register, located in the last (lowest addressed) register in that block - locations 040h; 201h and 301h respectively. The format of the three link registers is similar:

FIGURE 3.7.4 XRAM LINK REGISTER FORMAT



Each of the three modules has a unique 'module I.D.'; which is simply the address of the first (highest-addressed) register in that block - for the base module 0BFh; for XRAM1 201h; and for XRAM2 301h. These module I.D.s appear in the fields TTT, NNN and PPP respectively, where:

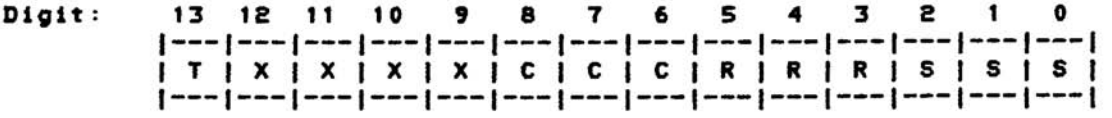
TTT	is the block I.D. (containing the link register in question);
NNN	is the next block I.D. (or 000 if there is no Next block); and
PPP	is the I.D. of the Previous block in sequence - the 41 ignores what it contains in the base block (040h), since there can be no previous blocks. But in the CX, this field is used as a scratch area for the EMDIR function, and may be overwritten, thereby rendering it useless to programmers, especially machine-coders, who need scratch data storage. (By means of the PPP, NNN & TTT fields, each block points to the next block, the last block, and to itself);
WWW	is the number of the working file in hexadecimal.
XX	is a 'don't care' field - put anything you like here, the 41 ignores it completely.

XM files all have similar structures - consisting of a number of registers, specified either by the User at creation (data and text files), or by the system at creation (program files). The size of a file, returned by

EMDIR, refers to the total number of registers available within that file, and is always 2 registers fewer than the total space that file consumes because a two-register 'header' is also stored with the file. This information is seen in the EMDIR listing.

The header is similar for each of the three file types:

- the first register always contains the name of the file - stored in ASCII form (as read from ALPHA) - which is left-justified within the register with the first letter in the name always at the 'left' end of the register). For a name shorter than seven characters, spaces are added to pad out the register. One consequence of this is that, to the 41, the names 'MYFILE' and 'MYFILE ' are the same.
- the second register contains various information on the file, but although these vary between file types, the general layout is always the same:



- 'T' is the file's type, and is a '1' for program files, '2' for data files and '3' for text (ASCII) files. If this digit is '0', then EMDIR will show '@.' where the file's type should be, and '@' for all other cases.
- SSS is the size of the file, coded in hexadecimal and does not include the two header registers. These fields are always the same for any file type.
- RRR is used in program files to store the number of bytes in the program - for data and text files, it stores instead the current record number within the file (hexadecimal).
- CCC is used only by text files, and stores the current character. In fact, only digits 6 and 7 need be used for this, but all three actually are.
- XXXX is a 'don't care' field; in program files this is usually left as zeroes. With data and text files it is used as a scratch area to store the file location during file operations. This field is never read by the 41, only written into.

Data files store data with register 0 following the header. With program files, a byte-for-byte copy is made straight out of memory into the program file. The first seven bytes go into the first register after the header, and so on to the end of the program. The END is also copied, but an extra checksum byte is added - this is found by adding every byte in the file, and performing modulo 256. For some reason, HP didn't use the header CCC field to store the checksum. In text files, data are stored in records of variable length - indicated by a byte preceding the record.

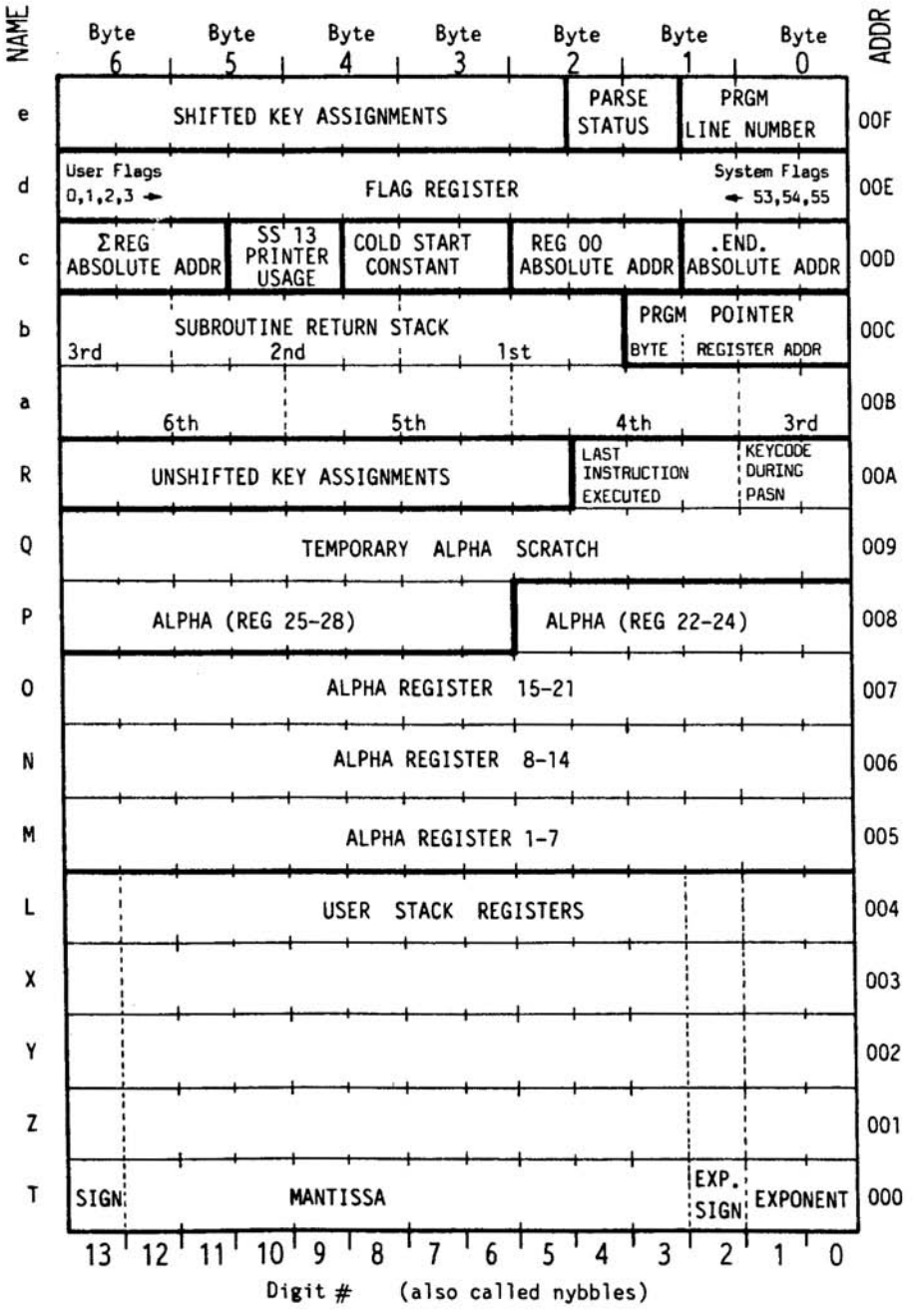


FIGURE 3.8.1 HP-41 STATUS REGISTERS

3.8 STATUS REGISTERS

At the very bottom of HP-41 memory are the System Scratch, or Status, registers. These comprise 16 registers containing a collection of permanently resident information for the 41, including which data can be found where, and in what mode the 41 is currently. Figure 3.8.1 contains a detailed map of the Status Register contents.

The register numbers (up the side) correspond with the postfixes given in the last row in each half of the Table. T,Z,Y,X & L will be familiar to you as the Stack Registers. Registers M,N,O & P (3-bytes only) form the ALPHA display. M contains the 'rightmost' 7 bytes of ALPHA, N the next 7, O the next, and P contains the leftmost 3 bytes. Bytes 6 to 3 of P (also described as P[6:3]) contain scratch fields used by the HP-41 for, amongst other things, running catalogues and controlling digit entry. They can occasionally be used as an extension to ALPHA, as long as there are no number-entry lines or VIEW instructions in the program.

Q is used by every function taking an alpha argument, the printer (whether you use printer functions or not), and a number of other functions, such as digit-entry, P-R, Y↑X, SDEV, SIN, COS, ASIN, ACOS and any time the 41 must place the contents of ALPHA in the display. Provided you can avoid all these obstacles, you can use it.

Register 'R' , normally known as register ⊢ (the append symbol) and register 'e' contain bit-maps for key assignments. Every key position on the keyboard has a corresponding bit in these registers. The bit-map for each key is shown in each of the registers in Figure 3.8.2. If a particular bit is set in register 'R', then that key carries an unshifted assignment. If the bit is set in register 'e' , then it represents a shifted assignment.

Registers 'a' and 'b' contain the subroutine return stack and the User-code program counter. The latter points to the last byte of the line just executed, while the former is simply up to six copies of previous program counters - these point to the last byte of an XEQ that was previously executed (including XROM executes which refer to user code programs in plug-in devices).

The format of the program counter is:

0bbb 000r rrrr rrrr

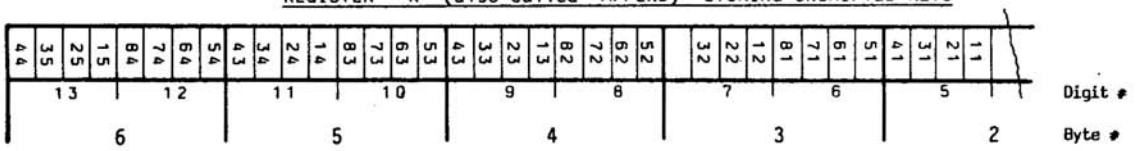
where:
- rrrrrrrrr is the absolute address of the current register;
- bbb is the byte within that register which contains the first byte of the next line, when the program currently executing is in RAM.

When pushed across onto the subroutine return stack, the format alters slightly to that of:

0000 bbbr rrrr rrrr

The reason for this, is to allow the 41 to determine whether or not the return is to a ROM or RAM location.

REGISTER 'R' (also called APPEND) STORING UNSHIFTED KEYS



Note. Keycodes are shown in Row, Column format
ie. 53 = row 5, column 3

REGISTER 'e' STORING SHIFTED KEYS

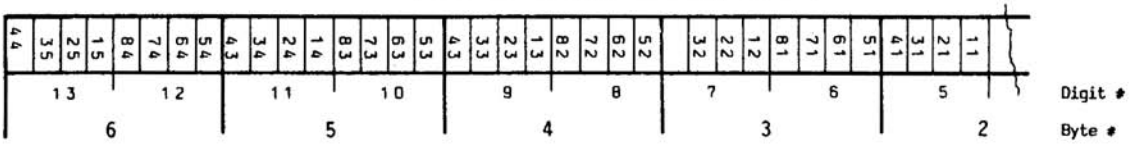
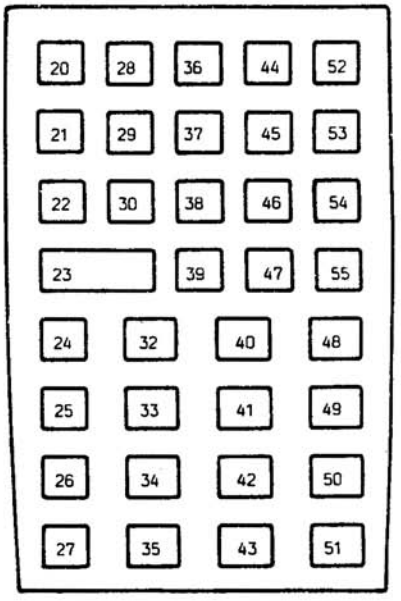


Figure 3.8.2 Key Assignment Bit Maps



Bit locations of assigned keys in registers 'e' & 'R'

Figure 3.8.3 Key Assignment Bit Mapping

If the 41 is currently in ROM, then the program counter simply contains the absolute address of the byte within the ROM. Since no plug-in ROM can have a first digit of 0 in its address (0 being a system ROM address - see later for the 41 ROM page structure), if the first digit is 0, this means a RAM address. If a non-zero digit, it means the XEQ/XROM was in a ROM, and the routine call should return to there. The 41 processor has a flag which determines whether the pointer is interpreted as meaning RAM or ROM. A ROM address pushed onto the return stack is consequently left unaltered.

The stack is pushed from right to left across both registers, with the last two bytes of register b being copied into bytes 1 and 0 of register 'a', and the last two in 'a' being lost. Popping the stack (at a RTN or an END) is the converse of this operation, and its action writes 00 00 into bytes 6 and 5 of 'a'. Whenever bytes 3 and 2 of 'b' contain 00 00, the return stack is said to be empty.

Register 'c' contains memory allocation data. It is the one status register that requires care whilst 'playing around' with SP. The quickest way to destroy everything in memory is with [STO] [.] [c]. Register 'c' contains the absolute address of the register containing the .END. in bytes 2 through 0, the address of the program/data register curtain in bytes 5 to 3, and the absolute address of the current statistics block. This is the register pointed to by the last ΣREG operation) in bytes 13 to 11. The printer uses bytes 10 and 9, and bytes 8 to 6 contain the HP-41's 'cold start constant', which *must* always be 169h. The 41 examines this field at switch-on, whenever a key is pressed (with no program running), and whenever a program returns control to the User (STOP, END, RTN, PSE, etc.). If, when checked, it is not 169h, then the HP-41 assumes that memory has been corrupted and automatically performs a MEMORY LOST.

Register 'd' contains all 56 User and System flags, with flag 00 at the left and flag 55 at the far right. Figure 3.8.5 shows the bit mapping for register d, and the application/purpose of each flag is given in Figure 3.8.4.

Finally, register 'e' also contains the current line number in bytes 2 through 0. When a program starts running, this is set to FFFh, meaning 'I don't know the line number'. Going into program mode, holding down [R/S] (to start a program running) for longer than 100ms, pressing [SST] or generating an error out of program mode (including program errors) causes the current line number to be recomputed.

00 - 10	general purpose	
* 11	auto execute	
* 12	double width print	
* 13	lower-case print	
* 14	overwrite mag. card	
* 15 - 16	HP-IL printer modes	
0 0	MANual	
0 1	NORMal	
1 0	TRACE	
1 1	TRACE WITH STACK	
* 17	(CR) incomplete record	
* 18	enable TINTR	
* 19 - 20	general use	
** 21	printer enabled	
* 22	numeric entry	
* 23	ALPHA entry	
* 24	range ignore	
* 25	error ignore	
26	audio enable	
27	USER mode	
28	decimal point/comma	
29	digit grouping on/off	
* 30	CATalogue	
31	Timer format DMY set	
	MDY clr	
32	Manual HP-IL I/O	
33	HP-IL absolute manual	
34	ADRON clr	
	ADROFF set	
35	disable autostart ROM	
36 - 39	number of digits displayed	
40 - 41	display format	
0 0	SCIentific	
0 1	ENGineering	
1 0	FIXed	
1 1	FIX & ENG	
42 - 43	trig modes	
0 0	Degrees	
0 1	RADians	
1 0	GRADians	
1 1	RADians	
* 44	Continuous ON	
* 45	system data entry	
* 46	partial key sequence	
* 47	SHIFT	
* 48	ALPHA	
* 49	BATtery voltage low	
* 50	message	
* 51	SST	
* 52	PRGM	
* 53	Input/Output	
* 54	PSE	
** 55	printer existence	

Note: * = cleared at turn on
** = cleared only if printer absent

Figure 3.8.4 Description of HP-41 Flags

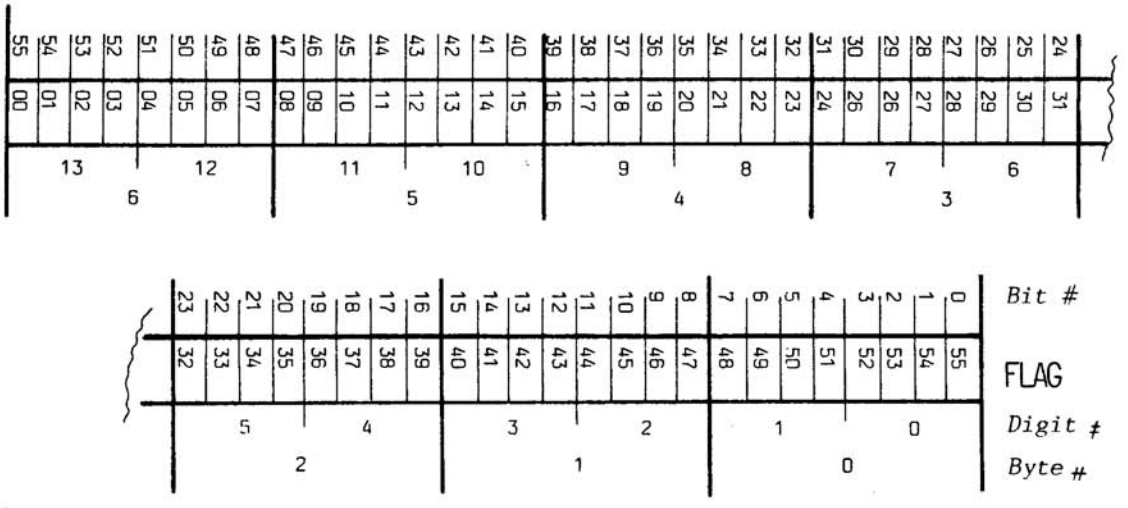


Figure 3.8.5 Flag Bit Locations in Register d

3.9 APPLICATIONS OF SP

There are a number of common, simple applications of SP which are quite straightforward, but which make up the majority of applications for this set of techniques.

3.9.1 Scratch Storage

Quite often, you will find yourself writing a subroutine for a program needing one or two scratch registers outside of the stack. Normally, you are forced to use numbered data registers to satisfy this need. However, this can cause problems when these registers may be used by other routines as part of the main program.

The usually-adopted approach to this problem is to create a small data file. However, for a simple application, this is not really ideal.

The easiest approach, if the function being coded allows it, is to break up the ALPHA register. Remember that ALPHA comprises 4 registers, M through P. It is perfectly permissible to use three of them just like any other (register P must be used with care). In fact, there are two advantages to be gained.

The first is that a RCL, VIEW or X<> from any of the sixteen Status registers is non-normalising. The HP-41 normalises as part of a check to see if the register actually exists. However, the HP-41 knows that the Status registers always exist, and so doesn't need to check and normalise.

The second advantage is speed, for the very same reason - the HP-41's check on the existence of a register takes time (about 6ms), and this is saved for every execution of a register function addressing the Status registers as opposed to anywhere else.

Therefore, with SP, a common place for loop-control variables to be stored (those that are used with ISG and DSE) is in the Status registers somewhere, and similarly for any operations within the loop that access a scratch variable.

3.9.2 Non-Standard Output

Probably the most awkward function in the HP-41 system is BLDSPEC on the system thermal printer - either HP-82143 or 82162. You need to expend about 30 bytes of a program per special printing character, which is hardly a good trade-off. By using SP, you can place a text line in the program that contains exactly what BLDSPEC would have assembled in X anyway, and then simply execute RCL M, ACSPEC. The RCL M is used to recall the last seven characters of ALPHA for sending to the printer.

This number, now in X, is probably neither formatted as a regular number, nor as a string, but rather as a Non-Normalised Number (NNN). Be careful with these. Register access operations that talk to user data space always check the existence of the register first. In the case of a block operation, such as REGSWAP, only the highest-addressed register in each distinct block is checked. During the checking process, normalisation of the contents may occur as follows:

- if the first byte is zero, the register is cleared;
- if the first digit is 0 or 9, the mantissa is forced to contain only digits '0' to '9' and the exponent is forced to be either 000h through 099 (BCD only) or 999 through 901. Digits 'A' to 'F' will be lost;
- if the first digit is other than '0' or '9', then it is forced to be a '1'.

ZENROM contains functions to get around this problem, but avoid using user data register space for storing such numbers, and be careful which numbers you extract out of memory in case you normalise the register. Using the Status Registers gets around these problems, as these registers never get normalised because the 41 need not check their existence.

More than one special character can be entered into ALPHA, of course, and RCL N, RCL O or RCL P used to access it. But remember the limitations on the usage of register P.

Example Program: *Accumulating Special Mathematical Characters into the Printer Buffer*

LBL 'S' *set 'is a subset of' descriptor*

LBL 'IN' *set 'inclusion' descriptor*

LBL 'U' *set 'union' operator*

LBL 'IS' *set 'intersection' operator*

LBL 'M' *set 'is a member of' descriptor*

```
01*LBL "EX2"
02*LBL "S"
  03 "*2*T"
04 GTO 00

05*LBL "IN"
  06 "*J*S*"
07 GTO 00

08*LBL "U"
  09 "*∩∪⊥*"
10 GTO 00

11*LBL "IS"
  12 "*0 _*"
13 GTO 00

14*LBL "M"
15 "*∈R     J"

16*LBL 00
17 ASTO X
18 ACSPEC
19 END
```

The hex-codes for these lines are:

```
03 = F6 01 32 95 2A 54 80
06 = F6 01 4A 95 2A 53 00
09 = F6 00 FA 04 08 0F 80
12 = F6 01 F0 10 20 5F 00
15 = F6 00 71 52 A5 4A 80
```

Use 'SYNTEXT' entry to create these text lines. Remember, however, that you don't need to enter the first F6h byte (= 6 character text line) as the operating system will work this out for you

This method can also provide a way to enter lengthy floating-point constants into a program using fewer bytes and less time than the normal number-entry line does.

Example Program: *Accumulating Floating Point Constants*

LBL 'e' constant e = 2.718281828

LBL 'H' *Planck's constant h = 6.6262*10!34*

LBL 'c' *speed of light = 299 792 459*

LBL 'u0' *permeability of free space = 1.256637*10!-6*

LBL 'G' *gravitational constant = 66.73*10!-12*

LBL 'Z0' *impedence of free space = 376.730 4*

LBL 'F' *double Faraday constant*
*returns 2.892599*10!14 into Reg. X*
and 96 486.7 into Reg. Y

```
01♦LBL "EX1"
02♦LBL "e"
   03 "x̄q♦"
04 GTO 00

05♦LBL "H"
06 "Γbb♦♦σf"
07 GTO 00

08♦LBL "c"
09 "x̄y$Y♦▲"
10 GTO 00

11♦LBL "μ0"
12 "•x¼f7Γ)"
13 GTO 00

14♦LBL "G"
15 "Γ90♦♦σ"
16 GTO 00

17♦LBL "Z0"
18 "÷υsα♦♦x̄"
19 GTO 00

20♦LBL "F"
21 "αdp♦♦αx̄¾♦♦á"
22 RCL \

23♦LBL 00
24 RCL [
25 END
```

The hex-codes for these lines are:

```
03 = F7 02 71 82 81 82 80 00
06 = F7 06 62 62 00 00 09 66
09 = F7 02 99 79 24 59 00 08
12 = F7 01 25 66 37 06 29 94
15 = F7 06 67 30 00 00 09 89
18 = F7 03 76 73 04 00 00 02
21 = FE 09 64 86 70 00 00 04 02 89 25 99 00 00 14
```

SP has also been used to place non-keyable characters into the display, such as [ ], {}, !@≠&, etc., without needing to use a 'character code' and the XTOA function from the Extended Functions Module. As these characters are available on ZENROM USER ALPHA keyboards, this technique is now obsolete.

3.9.3 Register Allocations

This means altering the contents of register 'c'. You can force the curtain, between data and program registers, to be any register in the HP-41. Although, if the program halts you must make sure the register immediately below the one you address actually exists. By forcing the curtain to be register 010h it means that you can directly access any register up to 1FFh. Altering the .END. address or the statistics register pointer isn't so popular, because one can be easily done with a mainframe function, and the other has little point, but both are possible. Again, make sure the register pointed to by the .END. exists if you don't want the annoyance of a MEMORY LOST.

Another use for altering the curtain location is to 'hide' some previously allocated data registers by placing the curtain above them, thereby effectively 'adopting' them into program memory. There is a danger however. When you PACK memory, these registers will be seen as program lines and packed as well, which will more than likely completely mess up your memory contents.

3.9.4 Flag Manipulation

The RCLFLAG/STOFLAG combination, from the Extended Function Module & 41CX, allows you to do some things previously only the territory of SP'ers (such as entering FIX/ENG mode) by directly accessing the flag register. If you normally have a standard or default setting for the 41's flags and wish your programs to enforce it, then simply load a text line containing the chosen register d as a collection of characters. Then follow this with RCL M and STO d (or ASTO d - depending on how you want to affect the first 8 flags). Using ASTO d always leaves flags 00 through 07 clear, except for 03 which will be set.

Clearing the system flag 55 is a common technique used for speed with a printer connected to the 41 - because the machine runs considerably faster if the printer isn't there. The fact that you may not be printing is quite irrelevant, as the 41 polls the printer all the time to see if it needs servicing. By clearing the printer existence flag, f55, the 41 can be made to think there is no printer attached, so doesn't waste time trying to poll it. However, as with most good things, there is a catch: 'a' variety of operations will restore the correct status of the flag.

WARNING: Avoid SSTing programs which modify the flag register - since this directly affects f51, (the SST flag). Also, storing random numbers into register d can cause problems. If, by chance, you set f52 - then executing a number entry line will cause the 41 to begin programming itself. Flag 53 is always automatically cleared by the machine after every operation, so is not really worth bothering with. The Timer Module will automatically clear f30 (the CAT flag) if flags 46 (partial key sequence) and 53 (Input/Output) are set together. DO NOT store 'junk' into the rightmost three digits of register d - if you must temporarily store something there, use nulls for those digits.

Example Program: *Clearing Flag 55*

```
01♦LBL "EX3"
 02 RCL d
 03 STO [
 04 "┣♦♦♦♦"
 05 X<> [
 06 X<> d
 07 CF 23
 08 X<> d
 09 X<> [
 10 "┣♦♦♦"
 11 X<> \
 12 STO d
 13 END
```

This routine clears flag 55 by:
- placing the current flag register contents into M;
- pushing it across by a few bytes until F55 is where flag 23 usually goes and then swapping it back into place;
- clearing f23 (which is really f55);
- swapping it back out and restoring into M;
- pushing the rest of the seven bytes so that what started in M is now in N;
- and then performing RCL N, and STO d to replace it into the flag register.

However, executing this from the keyboard, then testing for FS? 55 will yield 'YES', since halting a running program restores the correct status of F55.

3.9.5 Other Basic Applications

1) A 'NOP' instruction can be very useful in conjunction with an ISG or DSE instruction which you don't want to know the result of. E.g. you may simply wish to increment or decrement a counter, but don't need the 'skip' part of the instruction.

2) A Short-form Exponent can save one byte over the standard instruction. In this case, the '1' in exponent entry, e.g. '1 E3' can be removed, thereby leaving just the 'E3' remaining.

3) Non-Standard Functions can be assigned to the keyboard, e.g. the key sequences VIEW IND X or STO IND Y could be assigned to single keys. Another interesting one is 'eG0BEEP' , a pseudo-function which allows keyboard access to all printer, HP-IL and mass storage functions without the devices being present.

One of the most sophisticated uses of SP (and one of the most difficult to manage) is the use of pre-compiled GTOs and XEQs — this means instructions that already know where to jump to, and which the HP-41 needn't bother searching for. Compiled GTOs needn't jump to LBLs, though. They can jump anywhere you desire, even into the middle of another instruction. A variation of this is using the long-form version of GTOs 00 to 14, thereby allowing the use of the shorter LBL, but still allowing GTOs to reach them from anywhere in the program, not just within the 16 registers either side limitation of local labels.

3.A SUMMARY

As can be seen, with just a little imagination, Synthetic Programming can enhance many aspeccts of the use of the HP-41, and make it even friendlier and more powerful than before, as countless HP-41 programmers have already discovered. There is just one snag! Since the HP-41 has never heard of Synthetic Programming — otherwise it wouldn't be SP — it does its level best to prevent you from exploiting the techniques and facilities available. To overcome this, an incredible number of special keying techniques have been devised over the years, ranging from special magnetic cards (generated on the HP-67) and barcodes, through byte grabbers, jumpers and Q-loaders to prefix maskers and byte-loading programs.

With the advent of ZENROM, all these have been made redundant. If you know what they are, you can quietly forget them. If you've never learned, then you'll never have to. However, we would still recommend that your read some of the many reference books on SP as these cover many interesting application ideas. The next chapter will show you ZENROM can be used to insert all possible byte sequences into your programs, and how to alter the contents of any byte anywhere in HP-41 memory with complete freedom.

4

USING ZENROM TO INPUT SYNTHETIC LINES

This chapter of the handbook assumes that you have read the previous chapter on Synthetic Programming Theory. We would recommend that all ZENROM Users read that section as there are many valuable points that could have been forgotten. For this chapter, we have also assumed you have an good practical understanding of the following concepts:

HP-41 Memory Structure
Register Formats
Hexadecimal Byte Table
User Code Programming
Synthetic Programming

The facilities offered by ZENROM, provide the User with three unique ways to enter synthetic lines:

- using the Direct-Key Synthetics ability which the ROM gives your HP-41 (not available on very early HP-41C models);

- using the new extended alpha keyboards for text entry (available on all HP-41s);

- using the insertion mode of RAMED to insert new bytes anywhere in program memory (available on all 41s).

4.1 DIRECT-KEY SYNTHETICS

Many of the commonest synthetic lines are simply extended forms of regular program lines, such as RCL M, TONE X or a text string '≠2'. If a regular form of the line exists, and you wish to enter a synthetic form, then ZENROM will have a directly-keyable equivalent. If an instruction has a form PREFIX plus POSTFIX (i.e., two bytes or more), where all postfix bytes are normally numeric, then you will find the prompt for that function will have been changed by ZENROM to a two-digit prompt — of course many have a two-digit prompt already, so no immediate difference will be apparent.

For any two-digit prompt, ZENROM allows you to key any of the postfixes which you see in the Byte Table, exactly as it appears there (with one exception — ⊢(append) must be keyed as 'R', this being the letter which alphabetically follows on from Q as register ⊢ follows on from register 'Q') in the HP-41 system. All postfixes from 00 to 99 are keyed as usual: postfixes from 100 to 111 are keyed by pressing the [EEX] key, which adds a '1' ahead of the two-digit prompt, followed by the numbers to make up the postfix you wish (up to 111): pressing [.] places 'ST' in the display — you may now press any of the 16 letter keys (T,Z,Y,X,L,M,N,O,P,Q,R,a,b,c,d,e) to get the corresponding postfix byte appended as part of the function.

At this point, a small note — due to a -41 operating system restriction, ZENROM is unable to validate the digits pressed in response to the '1 __ __' prompt, and thus allows you to enter anything from 100 to 199. E.g. with RCL, responses 128 to 199, if given to the normal (non-indirect) prompt, will generate indirect RCLs from RCL IND 00 to RCL IND 71, while such responses to RCL IND 1 __ __ will generate RCL 00 to RCL 71 (the postfix is value for X MOD 256) — this applies to any function which takes such an argument, not just RCL.

The extended prompting can also be used for creation of synthetic labels (not globals) of status register names X,Y,Z,T,L and M through R , to do this, use the key sequence:

[SHIFT] [LBL] [.] as you would for RCL, STO, etc., and answer the 'LBL ST __' prompt with the status register letter that you wish to use.

This will enter as a local label that behaves exactly like numeric local labels. Note however, that there is no automatic key assignment as there would be for labels A to J and a to e).

In the case of the GTO function, which already had a use for the sequence [GTO] [.], a functional change has taken place:

pressing [GTO] [.] will now give you 'GTO ST __' ,
pressing [GTO] [.] [.] gives you 'GTO. __ __ __',
and pressing [GTO] [.] [.] [.] returns the usual 'GTO..'.

Although this requires a little 're-learning' of the behaviour of the GTO key, it is something which you will get used to very quickly.

There are two other significant synthetic facilities which ZENROM provides on all HP-41s:

- short-form exponent entry,
- NOP instruction entry

If, while in program mode, you press the [EEX] key as the first digit of a number-entry line, then the display shows: '1 E__' . This is a multi-byte instruction - with one byte being wasted on the '1' character. With ZENROM plugged-in, the Direct-Key Synthetics feature takes over and strips out this wasted byte for the '1' character - thereby leaving only the 'E__' prompt in the display. Should it ever be necessary to have the entire '1 E____' sequence in the program, this can be achieved by defeating ZENROM. To do this, press and hold the [EEX] key, then press the first digit to follow the '1 E____'. Finally, release the [EEX] key before the digit entry key. Because the operating system does expect an extra byte to follow, back arrowing the partial exponent entry will very briefly display the next line, before displaying the previous line.

Occasionally, you may need to have a No-OP (No Operation) instruction, e.g. after a conditional test. ZENROM contains a function called NOP that will insert a text string of zero characters (hexcode F0) in your program. When executed by a running program, this F0h byte acts as a dummy instruction. To use, simply execute NOP, and ZENROM will insert the F0h byte as the next program line.

4.2 EXTENDED ALPHA & TEXT ENTRY

Entry of non-standard, or non-keyboard characters has always been difficult on the HP-41. ZENROM has two additional facilities to make keying-in any text string much easier:

- USER ALPHA keyboards, providing all lowercase characters, all displayable characters plus other special characters available from the keyboard;

- 'SYNTEXT' (Synthetic Text) entry, allowing entry of any HP-41 character by input of the hexcode.

4.2.1 USER ALPHA KEYBOARDS

The User Alpha keyboards are illustrated facing (Figure 4.2), and one of the two overlays supplied with ZENROM is for these new alpha keyboards. Normally, when you enter alpha mode, the state of the User flag remains unchanged, as the HP-41 does not have a pre-defined User Alpha keyboard. With ZENROM plugged in, however, entering alpha mode causes the User annunciator to switch off — you now have the normal alpha keyboard available for use.

Whilst in Alpha mode, if you press the [USER] key, the behaviour of the keyboard alters to become the USER ALPHA KEYBOARD: thereby making available the entire lower-case alphabet on the corresponding upper-case keys, and every other displayable character on the shifted locations.

Although this sounds complicated, in fact, by comparing the overlay with the standard Alpha keyboard, you'll see it is quite easy to remember character locations. Wherever possible, we have grouped characters or placed them on locations of similar normal characters. The only one that may need explanation is 'ESC' which is the the escape character (27 decimal, 1Bh) on [SHIFT][PRGM]. This is used quite often in printer control operations. Remember, however, that most lowercase characters are non-displayable - i.e. will only display in the HP-41 LCD as boxed stars - they will, of course, print out as intended. By combining upper- and lowercase characters in a single string considerable byte savings on printing applications can be made.

You may toggle between the USER ALPHA and normal ALPHA keyboards as often as you wish. When you leave alpha mode (whichever it may be), the status of the User flag (and annunciator) is restored to what it was before you entered.

You will notice that some of the USER ALPHA keys are identical to the normal ALPHA keys - e.g. the digit entry keys, AVIEW, CLA, etc. We decided to follow this convention to avoid any possible confusion and repeated switching between the new alpha keyboards whilst keying-in commonly used characters or instructions.

A word of caution regarding APPENDing characters. In alpha mode, if you press [SHIFT][APPEND] this will allow you to attach additional characters to the alpha string. In USER alpha mode, this will insert the ← character into the display as the first character of a NEW string, thereby deleting the existing string. To append to an existing string you must be in ALPHA mode.

UNSHIFTED KEYS SHIFTED KEYS

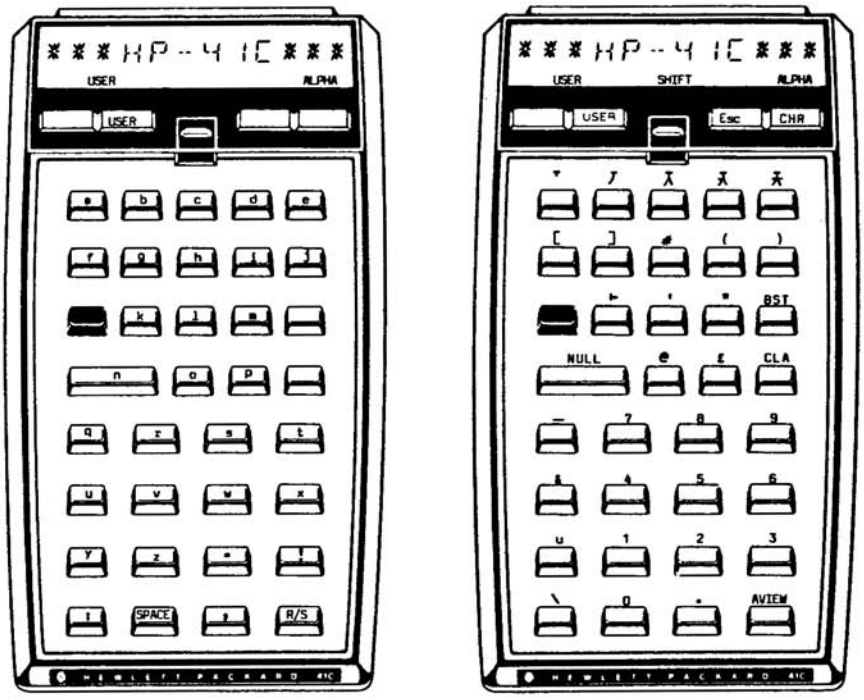


FIGURE 4.2 USER ALPHA KEYBOARDS

4.2.2 SYNTEXT ENTRY

No matter which alpha mode you are in, the key sequence [SHIFT][ALPHA] enables you to input any HP-41 character into text lines or into LBL, XEQ or GTO functions. We have christened this 'syntext' - for synthetic text entry. In this mode, two prompts appear at the far right of the display, and filling these with a two-digit hexadecimal number causes the character with that hexcode to be appended to the current string being entered. This prompt behaves exactly like any other two-digit prompt, but also has the option of being able to null the hexcode entered by holding down the second digit key for about one second — in this case, the original prompts reappear in place of the digits entered, and you may begin again or cancel the prompt with the backarrow key.

The User Alpha keyboards and Syntext entry are available on any HP-41 with a ZENROM plugged in, and operate in all modes where alpha entry is required, including entering text into program memory or any alpha prompt as a function argument, with two exceptions — the feature does not operate inside the CX text editor 'ED', nor does it operate during the execution of a PSE instruction — both of these limitations being due to problems with 'breaking into' the operation of these functions while retaining 41 operating system integrity.

A point of warning to all users — all ZENROM Direct-Key Synthetics facilities, including the special extensions to the alpha keyboard, rely on the ability of ZENROM to temporarily 'take control' of the HP-41 operating system to perform its tasks. This can only occur upon key-release, and will not happen if you press the next key in a key sequence before releasing the previous one. For example, while in alpha mode, press and hold the [SHIFT] key, then press [ALPHA], then release [SHIFT], and you will be switched out of alpha mode, not into syntext entry as you would otherwise expect.

Making sure that each key has been released before hitting the next is only a minor limitation as the HP-41's keyboard is not designed for speed typing. We felt the significant increase in capability afforded by these 'extensions' to the HP-41 operating system will more than offset this problem.

Some HP-41 functions, as well as taking a numeric argument, can also take an alphabetic one, e.g. LBL, GTO and XEQ. Normally, if you need a LBL A, you want a local label (i.e., it doesn't appear in catalogue 1), and ZENROM does not affect this form of label entry. But suppose, instead, you want to have LBL'A', which is a global label. This can be entered by using Syntext entry to specify the character, instead of keying it directly. For example, to get LBL'A', use the key sequence:

[SHIFT] [LBL] [ALPHA] [SHIFT] [ALPHA] [4] [1] [ALPHA]

(where 41 is the hexcode of character 'A'), and this will be entered as a global label instead of a local one. This method of entry also applies to GTO and XEQ as well.

Normally in a User's Handbook like this, there would now be several 'how to do it' type examples covering all the functions introduced. But since the Direct-Key Synthetics are natural and thus, second nature to the user, the best way of learning them is to begin experimenting.

WARNING: Make sure that you have first stored your programs and data onto magnetic cards or other mass-storage media. Experimenting with Synthetics can corrupt programs/data held in memory.

When entering a line of synthetic text, where the characters are given for you as hex-codes, you do not need to enter the first value - indicating the length of the text string - as this is worked out by the operating system. E.g. a seven character string would have the hexcodes beginning: F7 xx xx xx xx xx xx. It is not necessary to enter the F7h byte if you are using SYNTEXT entry. If, however, you are using RAMED to insert the sequence, you must enter the F7h byte as part of the complete character string.

4.3 USING THE RAM-EDITOR (RAMED)

Direct-Key Synthetics will enable most synthetic program sequences to be directly entered from the keyboard. Where you might have some difficulty is with keying significantly non-standard lines, such as functions with alpha arguments longer that 7 characters, XROM codes for devices not connected, or a precompiled GTO.

To allow you to do these (and to provide alternative synthetic entry routines for users of early model HP-41Cs that don't respond to Direct-Key synthetics), ZENROM contains a function called RAMED, which is a simple editor not unlike the HP-41CX text editor ED in its operation. RAMED allows you to alter the contents of all HP-41 memory without restriction and to insert additional bytes anywhere in program memory only. As you can imagine, RAMED is a very powerful tool for programming and memory manipulation, as well as more general exploring. Like all tools, however, it can also be dangerous (although neither RAMED nor Direct-Key Synthetics will harm your HP-41) so keep copies of programs and treat RAMED with respect.

RAMED has two main application areas:

- to modify, replace or insert bytes within any area of HP-41 program memory;

- to modify and replace bytes held in HP-41 main memory (data, program and status registers) or in extended memory.

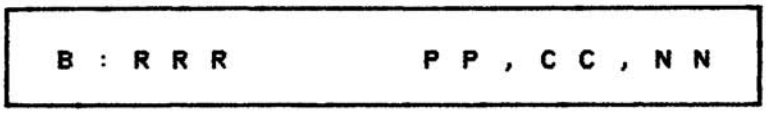
4.3.1 Within Program Memory

The easiest way to modify program memory is to step through your program to the point where modification is required, and then, whilst still remaining in PRGM mode, to execute RAMED.

Let's look at what happens in more detail:

> In PRGM-mode, when you step through a program, a RAM program counter keeps track of the program step as you go. This counter is stored in status register 'b'. When RAMED is executed, whilst in PRGM-mode, it takes the starting address from register 'b' and begins the editing process with the first byte in that program line. The exception to the rule is when the program is not PACKed and contains nulls between the address indicated by register b and the first byte in the line. In such a case, editing will start at the null byte pointed to by the counter and the actual line will be a few bytes further down in memory.

Once in RAMED, you will see a display of the format:



where:

B is the byte position in the register at address RRR,
RRR is the register address currently being edited,
CC is the byte in RAM pointed to by address B:RRR,
PP is the byte immediately 'up' memory from that byte pointed to (which would be the previous byte in program memory),
NN is the next byte 'down' memory from address B:RRR.
Note: If CC, PP or NN display as ' - - ' then that byte either does not exist or cannot be written to.

To facilitate and simplify movement within memory, RAMED re-assigns the HP-41's keyboard as shown below:

keys	Action taken
[PRGM]	to advance the address, such that NN is the current byte (Moves RAMED to the next lowest byte in memory)
[USER]	to retreat back up memory such that byte PP becomes the current byte
[SHIFT] [PRGM]	to advance a whole register down memory
[SHIFT] [USER]	to retreat a whole register up memory
[A] through [F]	
[0] through [9]	hexadecimal input keypad
[I]	insertion mode (active only within Program memory)
[ON]	to exit from RAMED back to program or normal mode dependent upon mode when RAMED was executed
[←]	to delete a partial replacement or insertion of bytes

REPLACING PROGRAM BYTES USING RAMED

At any time, the byte shown at 'CC' may be replaced with any new byte of your choice. To do this, simply key the new byte in, using the hexadecimal input keypad. Upon releasing the second digit key, the substitution will be made in memory, and the address will automatically advance by one byte, making the byte you just keyed the 'PP' byte in the display.

If you have entered one digit, and want to change your mind, simply press the backarrow key to cancel the first digit. If you have pressed, but have not released the second digit key, you can change your mind by holding down the key. After about one second, the attempted substitution will be cancelled, and the display will return to the state it was in before you pressed the first digit key.

INSERTING BYTES INTO PROGRAM MEMORY

Of particular importance to those with very early model HP-41Cs - who cannot use Direct-Key Synthetics - is the RAMED insertion function. Insertion mode is ONLY applicable within the bounds of program memory and operates exactly as the HP-41 does by moving all information down in program memory to free up seven extra null bytes (the size of one register) to accommodate your additional instructions. You can imagine the havoc this would cause were ZENROM to allow you to insert bytes ANYWHERE in HP-41 memory. Null bytes are overwritten when you enter your new instructions, while superfluous nulls are removed upon PACKing the program.

Insert mode is toggled into, and out of, by pressing the [I] key at any time (except while RAMED is waiting for a second digit). During insertion mode, and only if byte 'CC' is not a null byte, will program memory be moved down by one whole register and the register immediately following the current address filled with nulls. If 'CC' is a null, RAMED will simply overwrite the byte just as the HP-41 does normally.

To show the effect your insertion would have on byte 'NN', the display of this byte will be changed. This updating, shown ONLY IN THE DISPLAY, occurs after you press the first digit key. If you decide not to insert the byte, then the display will be restored as it was before you pressed the first digit key.

As with replacement mode, unless and until you release the second digit key, no change is made to HP-41 RAM. Note that if you insert bytes, and this causes register(s) to be 'opened up' to accommodate the additional bytes you enter, then the decompile bits will be set for the program you are editing. The decompile status of a program is never changed by simply replacing bytes (unless you actually change it yourself by replacing the last byte of a program END instruction).

Because RAMED allows editing ANYWHERE in memory, nonexistent register addresses are displayed as two hyphens, thus: ' - - '. Although RAMED does not prevent you from overwriting these in the display, after you release the second digit key ' - - ' is again displayed - thereby indicating that the byte was not accepted into memory.

To exit RAMED, press [ON]. RAMED will position you to the start of the line you were at upon entry.

4.3.2 Outside Program Memory

RAMED can prove very useful for examination of memory and system status register structures plus provide the possibility to directly modify or replace their byte contents. For example, you can directly modify key-assignment information.

To use RAMED out of program mode, the starting address is taken from Alpha — more specifically the rightmost

four hex-digits of register M which are the rightmost two characters as seen in the display. By this you can specify the exact register and byte within that register at which you wish to start editing.

This means that if you know the absolute address of the place in HP-41 memory that you want to edit (See the Memory Map in Figure 3.6.1), then simply use the Syntext entry feature in ZENROM.

To do this:

Enter Alpha-mode,
Press [SHIFT] [ALPHA] (for the Syntext entry prompt),
Enter the byte number, and the first digit of the three-digit register address,

Press [SHIFT] [ALPHA] again (for Syntext entry),

Enter the last two digits of the register address (this results in the address being returned to Reg.M[3:0] in the form expected by the RAMED function).

Come back out of Alpha-mode and execute RAMED. You will be editing memory, starting at the address specified.

As an example, let's take a look at the key assignment registers which have a format as follows:

Byte No	6	5	4	3	2	1	0
Bytes:	F0	A7	20	34	04	61	83

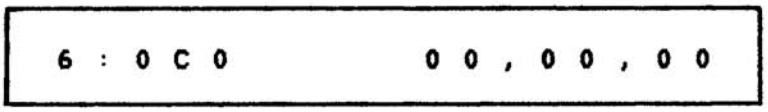
Bytes	Description
0	Keycode of key to which assignment made
1 & 2	Assignment
3	Keycode of key to which next assignment made
4 & 5	Assignment
6	Register ID to specify a key assignment register (F0h

Suppose you wish to edit the lowest key assignment register, which is at address 0C0, and you want to go in at byte 6 (F0h) of that register. In standard RAMED notation this is address '6:0C0' - where the ':' character separates the byte from the register address.

To do this:

Enter alpha mode,
Press [SHIFT][ALPHA] [6][0] [SHIFT][ALPHA] [C][0],
Exit Alpna-mode again,
Execute RAMED.

Assuming there are no key assignments, the display will now show:



You can now begin editing the assignment register. Remember that you will also need to set the key bit-maps in registers R (unshifted keys) and e (shifted keys) depending upon the assignment. If you need further practice with key assignment editing refer to Example Two in the following sub-section.

4.4 EXAMPLES USING RAMED

Although RAMED is very simple to use, some programming examples using RAMED should make its uses much clearer.

4.4.1 EXAMPLE ONE

First, let's take the following dummy program:

RCL M	(recall from status register M)
XROM 05,07	(a peripheral device function)
'(SYNTEXT@╪&)'	(text line with non-keyboard characters)
TONE 45	(a new tone - one of 117 new tones)

which, admittedly, doesn't do much that's sensible, but it does demonstrate a reasonable variety of program lines that you might possibly want to enter. Of these lines, only the XROM instruction cannot be keyed in directly - assuming your HP-41 supports Direct-Key Synthetics - but for the sake of example we'll use a few different methods to get the lines into memory.

Note that in this example, we go into RAMED quite a few times. To save finger work, assign RAMED to a key.

Firstly, we'll use direct insertion for the RCL M and XROM instructions. If you look at the Hexadecimal Byte Table, you'll see that byte codes for RCL M are: 90h, 75h. However, XROM 05,07 is a bit more difficult. The device ID 'XROM 05,xx' is found at hexcode A1h, but what about the function specifier 'XROM xx,07' ? From the chapter on S.P. theory, you'll remember that XROM codes are specified in binary format as follows:

1010 0ddd ddff ffff

where: 10100 is standard throughout XROM codes
ddddd is the device indentifier
ffffff is the function specifier

Subsituting our XROM 05,07 into this format, we get:

format:	1	0	1	0	0	d	d	d	d	d	f	f	f	f	
XROM 05,07	1	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	binary
		A				1				4				7			hexadecimal

There you are. Its easy when you know how !

```
                For practice purposes only.

  What is the hexcode of XROM 25,43 (which is the function
  SEEKPTA from the Extended Functions Module) ?  The answer
  will be found at the end of this example, but no peeking !
  To make it easier, the format is given below:


    format:  1 0 1 0 | 0 d d d | d d f f | f f f f
                     |         |         |
XROM 25,43   1 0 1 0 | 0 _ _ _ | _ _ _ _ | _ _ _ _   binary
                     |         |         |
                 A   |   _     |   _     |   _       hexadecimal


          Therefore:  XROM 25,43  =  A_,__ hexadecimal
```

When using RAMED, remember it inserts ahead of the current byte, so position the program pointer to the line after where you wish to perform the insertion — for this example, just do a GTO... to find some free space, so that we're positioned before the .END. . Now, execute RAMED, press [I] to get into insert mode, and enter 90, 75, A1, 47, then quit RAMED (press [ON]), and see 01 RCL M in the display (RAMED remembers the line you went into memory at in program mode, and tries to restore you at that line when you quit). Step to see NOP (XROM 05,07 is the ZENROM function 'NOP'). An interesting point is that, by entering NOP into a program in this manner, the instruction is now a proper XROM instruction with the name NOP - rather than being just a F0h byte. You can also achieve this by assigning NOP to a key, removing the ZENROM, and then pressing the key in PRGM-mode. However, entering NOP as an XROM instruction not only consumes an extra byte but also takes twice as long to execute.

We'll now edit a text line in situ. In PRGM-mode key-in 'ZSYNTEXTZZZZ', execute RAMED, and see '47,FC,5A' as the three bytes displayed. The 47 is the second byte of the XROM we keyed in above, while 'FC,5A' are the first two bytes of the text line just entered. Each 'Z' in the string is acting as a place-holder for the characters we want to have there eventually, and will appear as a '5A' in the display - this being the hexcode for uppercase Z. We want to alter these hexcodes to be those of the characters shown in the line above, so step to the first '5A' and REPLACE this with '28' to over write the first 'Z' in the string with a '('. Remember that you must be in replace-mode and NOT insert-mode. If you wish to check the change that has taken place, then quit RAMED and take a look. You'll see the line : 03 *(SYNTEXTZZZZ scroll across the display.

Now replace the other 5As with (in sequence) 40,23,26,29 — when you re-enter RAMED, you'll go back in at the first byte of the text line, so do [SHIFT] [PRGM] to skip a register along the string — this will position you to a 58, this is the 'X' in 'SYNTEXT', so skip two bytes along before you start overwriting. When you finish, exit from RAMED to see the replaced line: 03 *(SYNTEXT@╪&) scroll across the display.

Now let's try some prefix and postfix substitution — we'll create a TONE 45 by keying-in a regular TONE 9 and later modify it into a TONE 45. Whilst in PRGM-mode, key in TONE 9, and then execute the RAMED function. The first byte of the TONE will appear as its hexcode of 9F. Now step, using the [PRGM] key, to the 09 postfix byte, and replace this byte with 2D - the hexcode for the 45 postfix being obtained from the Byte Table - then exit RAMED. The replaced program line will now appear as: 04 TONE 5 . However, from the S.P. theory covered earlier, remember that functions normally expecting only one-digit arguments can only display the units digit of their arguments - even when ZENROM allows a two digit value (higher than 9) to be entered. Therefore, the TONE 45 instruction appears as TONE 5 in the display. Press [PRGM] to come out of PRGM-mode and then press [SST] to single step the TONE 45 instruction. You will hear a new longer tone sounded from the beeper.

We could, of course, have just modified the TONE by using prefix subsitution. With this method of producing such a TONE 45, the original program line could have been entered as RCL 45 (or any similar function taking the same postfix argument) and would then have been changed from RCL into a TONE instruction by substituting the hexcodes.

Try this for yourself. As a hint: RCL 45 will appear in RAMED as '90,2D'. The hexcode for the TONE instruction is 9F.

Answer to Practice Problem

format:	1 0 1 0	0 d d d	d d f f	f f f f	
XROM 25,43	1 0 1 0	0 1 1 0	0 1 1 0	1 0 1 1	binary
	A	6	6	B	hexadecimal

Therefore: XROM 25,43 = A6,6B hexadecimal

4.4.2 EXAMPLE TWO

To try something a little more sophisticated, let's create a synthetic global label - e.g. LBL'INVALID KEY' - which is longer than the HP-41 will normally allow, and assign it to the key [SHIFT] [SHIFT]. Following that, let's assign the multi-byte instruction 'VIEW IND X' to the [VIEW] key.

Firstly, we need to establish some kind of global label in memory, so that it will be found by the label-search mechanism that checks for assignments in labels. To be tidy and to make this example much simpler, execute CLKEYS or manually remove any key-assignments.

Then key in LBL'INVALID' (this reduces the number of changes we need to make); then execute RAMED. The display will show 'Cn' in the middle (where the 'n' could be any digit from 0 to D). Advance twice to the 'F8', and change it to 'FC'. The reason for this is that we wish to add four text characters to the label, so we must add four to the label length marker - in hex addition, adding 04h to F8h gives FCh. Next change the '00' in the next place to '0B' which is the hex keycode for the [SHIFT] [SHIFT] key position. Figure 3.7.2 shows the hex keycodes for HP-41 keys.

Now we need to step to the end of the label. Because it is seven characters long, we can skip a single register by using [SHIFT] [PRGM], and this drops us in at the right place to insert the four new bytes of the label. Toggle into insert-mode, by pressing the [I] key, and insert bytes: '20', '4B', '45', '59'. Now exit RAMED to see the LBL 'INVALID KEY' scroll across the display.

Having created the extra-long label, we now need to modify one of the bits in the key assignment maps representing the [SHIFT] [SHIFT] position. Key assignment maps are held in registers 'R' (unshifted) and 'e' (shifted). To do this, we need to edit status register 'e' — in fact, we need to add 40 to byte 2:00F, as this will set the appropriate bit. If you have forgotten the theory behind key-bit mapping refer to Figure 3.8.2.

So come out of program mode, go into alpha mode, and key [SPACE] (the character that has the hexcode of 20h — [SHIFT] [ALPHA] [2] [0] would also have worked, but with more keystrokes), then press [SHIFT] [ALPHA] [0] [F], come back out of alpha mode, and execute RAMED. RAMED will start with byte 2 of register 00F in the display — this byte is probably zero if you have no other key assignments, so just overwrite it with 40h, exit RAMED, and, in User mode, press [SHIFT] [SHIFT], holding down the second time, to see INVALID KEY in the display as the function name being previewed.

We can also use RAMED to alter key assignments stored in the assignment registers at the bottom of Main Memory (from address 0C0 upwards). Assign '+' to the VIEW key; then enter alpha mode and key the address 6:0C0 in using the syntext function [SHIFT] [ALPHA]. Come back out of [ALPHA] mode and execute RAMED — this will place the F0 byte (which is at address 6:0C0) in the middle of the display, indicating the start of the key assignment register. Step along the bytes until you come to 04,40, which are the bytes that code for the '+' function in an assignment register.

Overwrite these two bytes with 98,F3, then exit RAMED. Previewing the key shows XROM 35,51, which is the pseudo-XROM for VIEW IND X. You might like to play with this assignment, and some others too, just to get the feel of what RAMED has allowed you to do: but please remember the warning in the previous chapter about leaving garbage in free space registers.

4.4.3 EXAMPLE THREE

One of the most difficult lines to enter into a User Code program is a precompiled GTO, mainly because the HP-41 insists on decompiling the program upon modification. However, because the HP-41 doesn't treat over-writing of bytes by RAMED as being a modification, the program is not decompiled. This means that you may quite happily alter any bytes in a packed program without penalty. Just to show this working, press [GTO][.][.][.], then key in the following lines:

```
LBL'PRECOMP'
GTO 00
LBL d
GTO 00
RCL IND P
Y↑X
E↑X
SQRT
LN
SQRT
HMS+
Y↑X
X>Y?
GTO 00
VIEW IND 31
.          (Note this is a decimal point)
RTN
LBL 00
'?'
AVIEW
```

Now put an END on the program by: [GTO][.][.][.]
(this will also pack the program, thereby removing nulls)

Come out of PRGM mode and then go to the first line of the program by [GTO][ALPHA] PRECOMP [ALPHA]
Go back into PRGM mode and execute RAMED by [XEQ][ALPHA] RAMED [ALPHA]

Then do the following:

Press [PRGM] to step through the program until the display contains B1,00,CF as the current three bytes,

Key-in 50 to replace the 00, step again until you see B1,00,90, and key-in 61,

Step again until the display contains B1,00,98, and key 92.

Now, exit RAMED and PRGM-mode, and then execute 'PRECOMP', and see what happens. We will leave this as an exercise for the reader.

MACHINE LANGUAGE PROGRAMMING

5

AN INTRODUCTION TO MACHINE CODE PROGRAMMING

In writing a program on your HP-41, you use functions found in catalogues two and three, and maybe routines from other programs you have already written, to build up the desired piece of code. When complete, this will allow the HP-41 to perform a task which none of the individual instructions themselves could perform. User Code programming allows you to exert a considerable degree of control over what you wish your 41 to do, and over how it will be done. Nevertheless, you are limited to those instructions that HP have given you in choosing the tasks that you can do, and the manner in which they are accomplished.

This User Code instruction set represents your 'lexicon' of allowed words or instructions, and the 'sentences' which you can build up are limited by the variety of words you have available. You can, of course, buy extra plug-in modules to expand your vocabulary, but you are still limited by what others have decided are the most useful general-purpose words. Wouldn't it be nice if you could descend to the next level down in the language and devise your own words for your own special needs?

This analogy can be carried to HP-41 programming by operating at the same level as HP's programmers operate when they write the functions in the HP-41 and its plug-in devices, namely, machine code.

Please bear in mind the statement made in Section v regarding the NOMAS nature of Machine Code programming.

5.1 WHAT IS MACHINE CODE ?

In your experience with the 41, you may have believed that all there was to HP-41 programming was User Code, and that each User Code instruction was somehow automatically performed by the hardware lying beneath the case of your machine, in a world to which you were denied access. In fact, there is a completely separate environment operating 'beneath' the one with which you are familiar, and this is far more complex than the User Code environment you have been using up until now.

In the same way that you program your 41 with User Code instructions - to be executed by the underlying operating system - so this extra machine code level is controlled directly by the operating system. This runs its own programs on the 41 processor chip which is programmed using the more primitive instructions that the electronics of the 41 obey directly. These machine code instructions are the 'letters' of our earlier analogy, and represent a method of obtaining as complete a control over the HP-41 as anyone can expect to have.

Although you program in User Code, when the [R/S] key is pressed it is not the User Code that is run, but rather the operating system uses the program pointer held in status register 'b' to fetch the next byte in program memory. The operating system then uses this to determine the machine code routine (representing that individual RPN 'word') that should be run. The CPU itself, can only run machine code routines.

To understand this concept, perhaps it is best to think of every RPN word as being an individual sub-program (written in machine code). In just the same manner as you would build up a program using smaller User Code sub-routines, so the machine code programmer has built up his program using other machine code routines that are in turn built up of the native instruction set of the machine.

5.2 WHY USE MACHINE CODE ?

With such a large instruction set available in User Code (RPN), and with the expansions available using Synthetic Programming techniques, one can be forgiven for wondering why one would need to learn yet another language - for that is all Machine Code is. In much the same manner as a BASIC language user might consider learning the FORTH language to gain benefits of processing speed, I/O (input/output) flexibility and access to the operating system, so the HP-41 User might turn to 41-Machine Code language.

Machine Code programming is normally used for the following reasons:

Speed - up to 100 times faster;
Absolute Control over the operating system - allowing you to perform tasks that would be difficult or impossible with User Code;
Creation of new functions;
Packing of data.

On the debit side, however, when programming in machine code you must perform your own housekeeping. Whilst using RPN functions, the code forming those instructions has been written by HP to take care of housekeeping and associated tasks. This means that greater care must be taken over the coding and that debugging takes substantially longer.

5.3 WHAT YOU NEED TO PROGRAM IN MACHINE CODE

To begin programming in machine code, you will need certain items of equipment and documentation:

Equipment: - A ZENROM module
- A device for storage of the machine coded instructions
- A printer for disassembling the machine code

(After a while, when you build up a library of new functions, you may wish to commit these to blowing in EPROM or even a plug-in module. EPROMS are a much less costly option for small quantities, but do require an interface device for the HP-41 to be able to read them. You can of course, then also exchange your EPROMs with fellow users, and use many of the standard 41-EPROM-sets available through the User Groups.)

Documentation: - The HP-41 VASM Listings released by HP are essential for anybody considering machine code programming.
These are annotated listings (by the original programmers) of the 41's operating system).

The HP-41 has two areas of independently and uniquely addressed memory RAM and ROM. User memory is used to store and run the 8-bit RPN functions, while ROM is capable of storing and running both user code and machine code. Because machine code can only be stored in, and run from ROM (Read Only Memory), we need a way to read AND write our own machine code into some storage. To do this, requires a RAM (Random Access Memory = more correctly Read And Write Memory) interface that is capable of convincing the 41 it really is ROM.

For want of a better term, we have called these Quasi-ROM (Q-ROM) units and will refer to them as such throughout this Handbook. For more information on the many units available, see the addresses given under Appendix D - Bibliography & References (Equipment).

6

PROGRAMMING IN HP-41 MACHINE CODE

To gain the maximum from the machine code sections of the Handbook, all Users are recommended to read this and following chapters.

6.1. WHAT YOU SHOULD KNOW BEFORE YOU START

Before you start to write your own machine code routine it is recommended that you should have a reasonable understanding of synthetic programming - since many of the principles involved require such knowledge.

In particular you should understand:

1) The use of the status registers T through e
2) The structure, organisation and addressing of RAM
3) The format of a register

Moving into the realm of machine code programming requires a rethinking of some of the concepts related to RPN and indeed synthetic programming. We are no longer concerned with registers that contain real numbers, but rather 56-bit integers. We must no longer be reliant on the decimal system because hex and binary calculations are more common.

Other points to note about the following chapters are:

The mnemonics used in this handbook are the same as those employed by the ZENROM disassembler, resident in the MCED function. These mnemonics go under the title of 'ZENCODE' and a full list of the ZENCODE mnemonics, which cover not only 'mainframe' instructions but also peripheral instructions, are listed in Appendix E. In some cases these mnemonics differ from those which you may be used to. The two other sets of mnemonics which are in common use are HPs MASM set and the Jacobs/ De Arras set. The reason for the development of yet another set for ZENROM was that first time users found the old sets both confusing and difficult to learn. We hope ZENCODE will be adopted by the user community as the standard and that users will find them both self explanatory and consistent.

As an example of the confusing and misleading nature of the other mnemonics sets consider the following:

1. The HP-MASM mnemonic for testing status bit (flag) 1 is ?S= 1 1 which stands for: test if status is equal to 1, bit 1.

   We felt that a 41 user was more at home with the word 'flag' rather than 'status bit' and that a flag was either set or cleared rather than being 1 or 0 and so the ZENCODE mnemonic for the same instruction is ?FS 1, i.e: query flag set 1.

2 The Jacobs/De Arras mnemonic for reading the word from a ROM address is CXISA which stands for 'register C eXchange with the ISA line on the 41 I/O bus' - which I think you will agree is fairly meaningless to someone unfamiliar with the 41 bus architecture.

   The ZENCODE mnemonic for the same instruction is 'RDROM' for 'ReaD ROM address'. I hope you need no more convincing!

It should also be noted that some of the Jacobs/De Arras set are incorrect due to the fact that they were established before the complete instruction set was understood. In addition, this set does not include mnemonics for peripheral instructions. ZENCODE also covers all peripheral and HP-IL instructions.

Another convention that is followed throughout this manual is that of specifying fields within a register. Given a register 'r', 14 digits wide, the digits are numbered 13 to 0 from left to right, digit 13 being the most significant.

digit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	s	m	m	m	m	m	m	m	m	m	m	xs	x	x

The mantissa sign digit is referred to as r[S] or r[13].
The mantissa is referred to as r[M] or r[12:3]
The exponent sign is referred to as r[XS] or r[2]
The exponent is referred to as r[X] or r[2:0]
The whole of the register may be referred to as r[ALL], r[13:0] or just r.

If a section of the register, not conforming to one of the above fields is required, then the format is r[h:l], where h is the high order digit and l is the low order digit. For example r[6:3] refers to digits 6, 5, 4 and 3 of register r.

6.2. THE HP-41 C.P.U.

This section describes the HP-41 Central Processing Unit - codenamed 'Nut' by HP's design staff - and its internal register organisation.

The CPU contains:

- three main 56-bit arithmetic registers: A, B and C;
- two 56-bit storage registers M & N;
- one eight-bit register G;
- a four-level subroutine return stack and program counter;
- an eight-bit keyboard 'buffer' register;
- two four-bit pointers P & Q;
- a carry flag or flip-flop and a keyboard flag.
- 14 status flags
- an 8-bit output register

Figure 6.2.1 shows the organisation of these registers and their relationships to each other.

Throughout this chapter and other material relating to machine code, the Reader should be careful not to confuse the CPU-registers, often with similar names, with the Status registers accessible by the User using SP techniques.

6.2.1. The Accumulators (C, A and B)

The HP-41 has two main accumulators, C and A, and one main storage register, B. All three are 56 bit registers resident inside the CPU and should not be confused with status registers a, b and c. The accumulators are the registers upon which almost all of the 41s internal operations are performed.

The 'C' accumulator is the most important register in the whole 41, indeed almost one quarter of all CPU operations are affected or controlled by C. It is the C register that is used to transfer data to and from main RAM and to talk to all peripherals. All arithmetic instructions involve either C or A and you may find it useful to compare C and A to the stack X and Y registers in RPN; they are of similar importance at their respective levels.

The 'B' register is not strictly speaking an accumulator - since although arithmetic operations can be performed on B, the result of such an operation is never stored in B. The only method of changing its contents is by copying or exchanging them with the contents of one of the accumulators. Register B may, however, be referred to as an accumulator for reasons that will become apparent during later discussions (see Class 2 instructions).

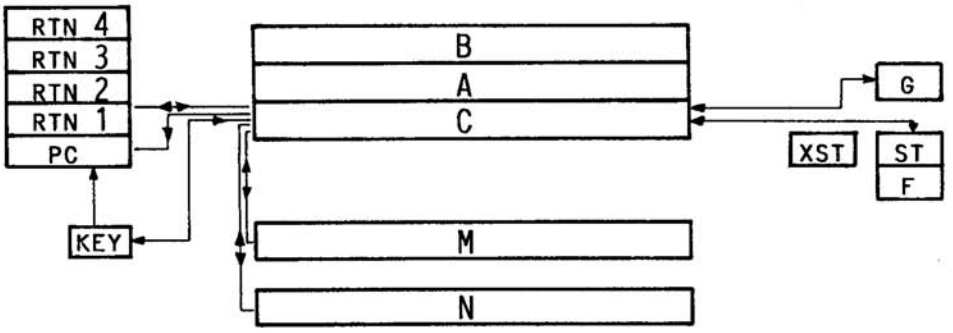


FIGURE 6.2.1. THE HP 41 CPU STRUCTURE

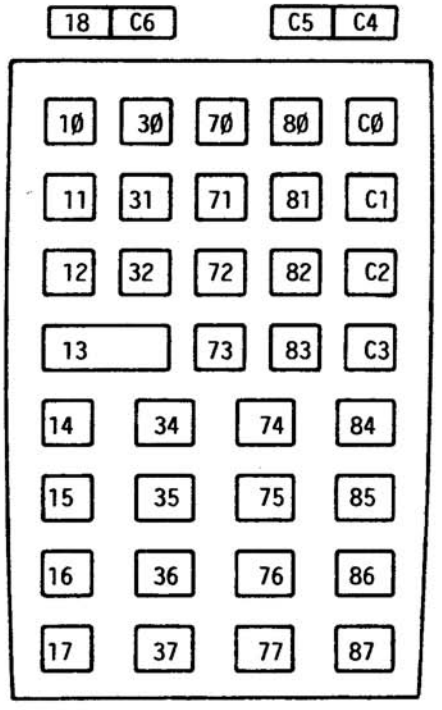


FIGURE 6.2.5. KEYCODE STRUCTURE

6.2.2. The Storage Registers (M, N and G)

There are two CPU temporary storage registers 'M' and 'N'. As with the accumulators they are both 56-bits wide and can thus store a complete 41 register. Again they should not be confused with the status registers Reg M and Reg N which are resident in main RAM and not in the CPU.

These storage registers can only be accessed via primary accumulator C. Drawing another analogy with user code programming; these registers could be compared to user data registers R00 and R01 since most Users employ these as temporary or scratch storage.

In addition to the two 56-bit storage registers, M and N, there is one eight-bit temporary storage register, 'G'. Similar to the other storage registers, the G register can only interchange data with C. Since G is only one byte wide, a coding method is employed to indicate which byte of C you wish G to interact with. The technique for this will be described when the instruction set is discussed.

6.2.3. The Status Bits (ST)

As with user code programming, there are a limited number of status bits, or flags, that are available to the machine code programmer. Once again, these flags are independent of those stored in status register d.

In the CPU there are 14 such flags. All of which may be individually set, cleared or tested. Of the 14 flags, 8 of them can be referred to as the status register ST. This is because the lower eight flags (7 - 0) can be transferred to or from the two least significant digits of C, C[1:0]. This is an important feature, as synthetic programmers will appreciate, because it means that more than one set of flags can be maintained at one time. The HP-41 operating system often uses the other 8 bit register G to store an alternate status set. In user code programming there are user flags (29 - 00) and system flags (55 - 30), the same is true in machine code. Flags 0 through 9 are user, or local, flags which do not have any specific meaning to the system. This does also mean that all of the ST register content is 'local' as well as the two other flags. However, flags 10 through flag 13 are system flags and indicate the following:

Flag 13 set - User code program running
Flag 12 set - Private program
Flag 11 set - Stack lift enabled
Flag 10 set - Program pointer (Reg b [3:0]) in ROM

6.2.4. The Program Counter and Return Stack (PC and STK)

Essential to any CPU is the program counter, 'PC', which keeps track of the next machine code word to be executed. After each machine code cycle, the instruction at the address pointed to (by the PC) is read and the PC incremented.

Of the circumstances when the PC may be altered, the most common is that of jumps. Should the instruction read in be a 'go to' of some form, then the PC is changed to the jump destination. If a 'go sub' (XQ in ZENCODE parlance) is encountered then the PC is copied onto the subroutine return stack and then changed to the jump destination. With the CPU, the return stack is 4 levels deep. The first address on the stack (i.e. the next return) can be transferred to or from the C register with the effect of 'pushing' or 'popping' the stack. Although, it is also possible to write a new address to the PC from C, it is impossible to read the PC directly, hence the single direction arrow shown on the diagram of the CPU. The instructions that deal with the first address on the return stack refer to it as STK.

6.2.5. The Keycode Register and Keydown Flag (KEY)

Whenever a key is pressed, the CPU requires some method of determining which key it was. This is achieved through the 'KEY' register. When a key is hit, providing that another key is not still held down, a keycode for the key is placed in the 8-bit KEY register. The keycodes returned are not the same as those you will be familiar with from synthetic programming but are shown in Figure 6.2.5. The KEY register can be read into C but not the other way around. This register also has a data path to the program counter (PC) which allows the 8 least significant bits of the PC to be overwritten by a keycode, thus enabling branching on a key. This feature is little used in practice due to the range of the keycodes.

In addition to the KEY register, there is also a keydown flag that will be set if the KEY register contains a keycode. This flag can only be tested.

6.2.6. The Flag Out Register (F)

The Flag Out, 'F', register is 8-bits wide and connected to an output pin on the CPU. This output pin is used to switch the beeper on and off. The F register is accessed via the status register ST.

6.2.7. The Pointers (P, Q and PT)

The CPU has two independent pointers 'P' and 'Q' which are used to indicate digit positions in the accumulators. Each pointer can thus have a value from 0 through 13. These pointers are primarily used in arithmetic, shift and comparison instructions to specify different fields of the accumulators to operate on. A description of using the pointers in this fashion is given in the section dealing with Class 2 instructions.

Although there are two pointers, only one of them can be 'active' at any one time. This means that in order to change the value of one of the pointers, that pointer must be selected as the active pointer. Most instructions that use a pointer will use the active one, and refer to it as PT.

6.2.8. The Carry Flag

The 'Carry' (also called 'condition') flag is the bit that will be set if any arithmetic overflow or underflow occurs after a given instruction. This bit will also be set if a test proves true, i.e. testing if a flag is set. This carry flag is also generally used as a basis for branching. For example, after testing a flag, you may wish to branch if the flag was set - i.e. the carry flag is set. Unless an instruction specifically sets the carry flag it will, in almost all circumstances, be cleared after each instruction.

6.3 THE INSTRUCTION SET

The instruction set mnemonics used in this manual are called 'ZENCODE'. The ZENCODE mnemonics are listed in APPENDIX E - Reference Tables.

In a similar manner to that in which user code functions are stored in main memory as a series of 8-bit bytes forming programs, so machine code instructions are stored in ROM or Quasi-ROM as series of 10-bit words to form functions. The format of these 10-bit words, however, is considerably more structured than that of their user code counterparts.

The basic format of the 10-bit CPU instruction (or word) is:

i i i i i i i i c c

The instruction set is split into four distinct and separate classes of instruction. Each of these classes is used to cover a particular type of instruction - such as short jumps or arithmetic operations. The particular class of an instruction is determined by the the least significant two bits of that instruction, indicated by 'c c' in the above example. The remaining eight bits are used to determine the actual instruction within the class.

In some cases, for example: the instruction ?FS 3 (test flag 3), the 8 instruction bits can be further divided into 'Subclass' and 'Modifier' sections, where the Subclass will determine that the instruction is a flag test type, and the Modifier will determine which flag is to be tested.

6.3.1 CLASS 0 Instructions

There are sixteen groups, or subclasses, of instruction in this category and each category has sixteen available modifiers. Diagrammatic representation of this Class is best achieved by means of a table. Although the concept of a byte table will be familiar to users of synthetic programming, to call it such could cause confusion with the standard User Code Byte Table - We will therefore refer to this as a 'Word Table'.

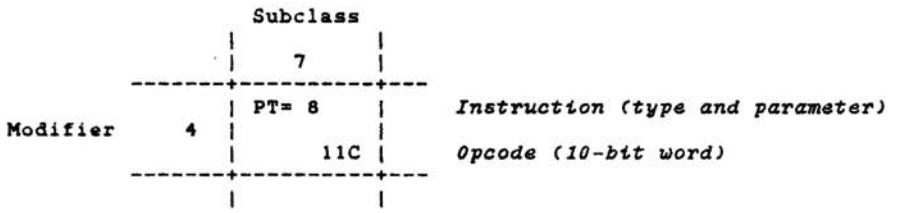
In essence, Class 0 provides instructions for such things as pointers, flags, data storage manipulations, some peripheral handling and other non-branch (or jump), non-arithmetic instructions. Unfortunately, although this is the most complex of all the classes, it is important that it should be covered first - as it includes instructions that must be understood before we introduce the other classes.

A class 0 instruction has the format:

m m m m s s s s 0 0

where: - ssss is the subclass, and
- mmmm is the modifer (the 00 at the end indicate it is from class 0).

The 256 instructions are organised into subclasses and arranged in a Word Table as shown in Figure 6.3.1. Each instruction block has the following format:



In Word Table locations where no instruction is given, only an opcode, this implies that the opcode is unused at the current time.

You will notice from the Word Table that the instruction parameters follow one of two patterns as shown below:

Modifier	Type 'A'	Type 'B'
0	3	0
1	4	1
2	5	2
3	10	3
4	8	4
5	6	5
6	11	6
7	Unused	7
8	2	8
9	9	9
A	7	10 (A)
B	13	11 (B)
C	1	12 (C)
D	12	13 (D)
E	0	14 (E)
F	Special	15 (F)

Flag Instructions (CF, SF, ?FS)

There are three types of flag operation:

Clear Flag (CF Subclass 1);
Set Flag (SF - Subclass 2);
Test Flag (?FS - Subclass 3).

All three instructions take a type 'A' parameter and can operate on any one of the 14 flags. The ?FS instruction will set the carry bit if the flag being tested is set.

FIGURE 6.3.1 MACHINE CODE WORD TABLE - CLASS 0

MODIFIER \ SUBCLASS	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP 000	CF 3 004	SF 3 008	?FS 3 00C	LC Ø 010	?PT= 3 014	018	PT= 3 01C	CLRRTN 020	PERTCT Ø 024	REG=C Ø (T) 028	?PF 3 02C	030	034	RDATA 038	RCR 3 03C
1	WMLDL 040	CF 4 044	SF 4 048	?FS 4 04C	LC 1 050	?PT= 4 054	G=C 056	PT= 4 05C	POWOFF 060	PERTCT 1 064	REG=C 1 (Z) 068	?PF 4 06C	N=C 070	074	C=REG 1 (Z) 078	RCR 4 07C
2	080	CF 5 084	SF 5 088	?FS 5 08C	LC 2 090	?PT= 5 094	C=G 098	PT= 5 09C	PT=P 0A0	PERTCT 2 0A4	REG=C 2 (Y) 0A8	?EDAV 0AC	C=N 0B0	0B4	C=REG 2 (Y) 0B8	RCR 5 0BC
3	0C0	CF 10 0C4	SF 10 0C8	?FS 10 0CC	LC 3 0D0	?PT= 10 0D4	C<>G 0D8	PT= 10 0DC	PT=Q 0E0	PERTCT 3 0E4	REG=C 3 (X) 0E8	?ORAV 0EC	C<>N 0F0	0F4	C=REG 3 (X) 0F8	RCR 10 0FC
4	ENBANK1 100	CF 8 104	SF 8 108	?FS 8 10C	LC 4 110	?PT= 8 114	118	PT= 8 11C	?P=Q 120	PERTCT 4 124	REG=C 4 (L) 128	?FRAV 12C	LDI 130	134	C=REG 4 (L) 138	RCR 8 13C
5	140	CF 6 144	SF 6 148	?FS 6 14C	LC 5 150	?PT= 6 154	M=C 158	PT= 6 15C	?BAT 160	PERTCT 5 164	REG=C 5 (M) 168	?IFCR 16C	STK=C 170	174	C=REG 5 (M) 178	RCR 6 17C
6	ENBANK2 180	CF 11 184	SF 11 188	?FS 11 18C	LC 6 190	?PT= 11 194	C=M 198	PT= 11 19C	ABC=Ø 1A0	PERTCT 6 1A4	REG=C 6 (N) 1A8	?TFAIL 1AC	C=STK 1B0	1B4	C=REG 6 (N) 1B8	RCR 11 1BC
7	1C0	1C4	1C8	1CC	LC 7 1D0	1D4	C<>M 1D8	1DC	GTOC 1E0	PERTCT 7 1E4	REG=C 7 (O) 1E8	1EC	1F0	1F4	C=REG 7 (O) 1F8	1FC
8	HPIL=C Ø 200	CF 2 204	SF 2 208	?FS 2 20C	LC 8 210	?PT= 2 214	218	PT= 2 21C	C=KEY 220	PERTCT 8 224	REG=C 8 (P) 228	?WNDB 22C	GTOKEY 230	234	C=REG 8 (P) 238	RCR 2 23C
9	HPIL=C 1 240	CF 9 244	SF 9 248	?FS 9 24C	LC 9 250	?PT= 9 254	F=ST 258	PT= 9 25C	SETHEX 260	PERTCT 9 264	REG=C 9 (Q) 268	?FRNS 26C	RAMSLCT 270	274	C=REG 9 (Q) 278	RCR 9 27C
A	HPIL=C 2 280	CF 7 284	SF 7 288	?FS 7 28C	LC A 290	?PT= 7 294	ST=F 298	PT= 7 29C	SETDEC 2A0	PERTCT A 2A4	REG=C 10 (R) 2A8	?SRQR 2AC	2B0	2B4	C=REG 10 (R) 2B8	RCR 7 2BC
B	HPIL=C 3 2C0	CF 13 2C4	SF 13 2C8	?FS 13 2CC	LC B 2D0	?PT= 13 2D4	ST<>F 2D8	PT= 13 2DC	DISOFF 2E0	PERTCT B 2E4	REG=C 11 (a) 2E8	?SERV 2EC	WDATA 2F0	2F4	C=REG 11 (a) 2F8	RCR 13 2FC
C	HPIL=C 4 300	CF 1 304	SF 1 308	?FS 1 30C	LC C 310	?PT= 1 314	318	PT= 1 31C	DISTOG 320	PERTCT C 324	REG=C 12 (b) 328	?CRDR 32C	RDROM 330	334	C=REG 12 (b) 338	RCR 1 33C
D	HPIL=C 5 340	CF 12 344	SF 12 348	?FS 12 34C	LC D 350	?PT= 12 354	ST=C 358	PT= 12 35C	CRTN 360	PERTCT D 364	REG=C 13 (c) 368	?ALM 36C	C=CORA 370	374	C=REG 13 (c) 378	RCR 12 37C
E	HPIL=C 6 380	CF Ø 384	SF Ø 388	?FS Ø 38C	LC E 390	?PT= Ø 394	C=ST 398	PT= Ø 39C	NCRTN 3A0	PERTCT E 3A4	REG=C 14 (d) 3A8	?PBSY 3AC	C=CANDA 3B0	3B4	C=REG 14 (d) 3B8	RCR Ø 3BC
F	HPIL=C 7 3C0	ST= Ø 3C4	CLRKEY 3C8	?KEY 3CC	LC F 3D0	-PT 3D4	C<>ST 3D8	+PT 3DC	RTN 3E0	PERTCT F 3E4	REG=C 15 (e) 3E8	3EC	PERSLCT 3F0	3F4	C=REG 15 (e) 3F8	3FC

The special instructions (modifier F) for the three subclasses are:

ST=0	which clears flags 0 to 7, i.e. the status register ST;
CLRKEY	which clears the KEYDOWN flag if no key is down at the time the instruction is executed; and
?KEY	which tests the KEYDOWN flag and, if it is set (i.e. there is a keycode in the KEY register), then the carry flag will be set.

Note that the fact of the KEYDOWN flag being set, does not imply that a key is currently down, just that a key has been pressed since the last keyboard reset. If the KEYDOWN flag has been reset (by CLRKEY) then the ?KEY instruction must be issued before the flag can be set by a new key press.

The Pointer Subclasses (PT= and ?PT=)

There are two main types of pointer instruction:

Set Pointer	(PT= - Subclass 7); and
Test Pointer	(?PT= - Subclass 5).

Both instructions take a type 'A' parameter and can either set the active pointer PT (either P or Q) to a specified digit or test if the active pointer is at a specified digit (13 through 0). Selecting an active pointer is covered under subclass 8. The ?PT= instruction will set the carry flag if the test is true - i.e. the active pointer is at the digit specified.

The special instruction for subclass 5 is 'Decrement Pointer' (-PT). If the pointer is at digit 0 and is decremented, then it will wrap round to point to digit 13. However, this will not set the carry flag. The special instruction for subclass 7 is 'Increment Pointer' (+PT). If the pointer is at digit 13 and is incremented, then it will wrap around to digit 0 without setting the carry flag.

Accumulator Manipulations (RCR and LC)

The two accumulator manipulation operations act on the C register:

Rotate C Right	(RCR - Subclass F) takes a type 'A' parameter and rotates the primary accumulator, C, right by the specified number of digits, 0 to 13.
Load Constant	(LC - Subclass 4) takes a type 'B' parameter, 0 through F, and loads that parameter value into C at the digit indicated by the active pointer. Having loaded the constant, LC then decrements the pointer, following the same rules as for the -PT instruction, thus enabling LC instructions to be strung together for loading more than one digit.

The use of LC and RCR is illustrated by the following example which could be used to load C with the normalised number —2.5 .

Assume all digits of C are 0 and the pointer is at digit 2.

```
Opcode  Mnemonic                    Comments
------------------------------------------------------------------------
250     LC      9       Load the constant 9 at the current pointer position
                        and decrement the pointer.
                        C =  0   0 0 0 0 0 0 0 0 0 0   9 0 0      PT @ 1

090     LC      2       Load the constant 2 at the current pointer position
                        and decrement the pointer.
                        C =  0   0 0 0 0 0 0 0 0 0 0   9 2 0      PT @ 0

150     LC      5       Load the constant 5 at the current pointer position
                        and decrement the pointer.
                        C =  0   0 0 0 0 0 0 0 0 0 0   9 2 5      PT @ 13

03C     RCR     3       Rotate the C register 3 digits to the right
                        C =  9   2 5 0 0 0 0 0 0 0 0   0 0 0      PT @ 13
```

Registers G, M, ST and F

The instructions for accessing registers G, M, ST and F are all in subclass 6. Three instructions are provided for register G:

G=C	load the C register into register G;
C=G	copy the G register into C;
C<>G	which will exchange the contents of the C and G registers.

This would be all very well, except for the fact that C is 56 bits wide, and G is only 8 bits wide. Therefore only two digits of C can be transferred to the G register. The method of indicating which digits of C to use, is by means of the pointer, PT. Data transfers between G and C operate on all of the G register, and two of the digits in C - at the active pointer position, PT, and at PT + 1. Thus if G and C contain the following values and the pointer is at digit 5:

```
C =  0   3 1 4 1 5 9 2 6 5 4    0 0 0        G =  A D
                      ^
                      PT
```

then the instruction C<>G would produce the result:

```
C =  0   3 1 4 1 5 9 A D 5 4    0 0 0        G =  2 6
                      ^
                      PT
```

If the pointer is at digit 13 and a transfer is done between C and G then digit 13 of C will be used, but the other MSD will be indeterminate (see time enable).

Data transfers between C and M (M=C, C=M and C<>M) operate on all 56 bits of both registers.

The next 3 instructions in subclass 6 (F=ST, ST=F and ST<>F) are used to control the CPU output port (the beeper), and their usage is covered in Chapter 8.1. on Special Instructions.

The instructions that transfer data between the Status register (ST) and C operate on all 8 bits of ST (flags 0 through 7) and the two least significant digits of C, i.e. C[1:0].

Subclass C

This subclass includes the three instructions that deal with transfers between registers C and N. As with register M, all the instructions operate on all 56 bits of both registers.

The LDI (LoaD Immediate) instruction comprises two consecutive words, namely, the LDI itself followed by a constant. This means that when the processor encounters an LDI, it does not execute the word following the LDI as if it were a stand alone instruction, but instead treats it as a data byte and uses the word after that as the next instruction. The effect of LDI is to load the value of the next word into C[X]. Hence the sequence:

```
130 LDI
325 CON 805
```

will place 325h into C[X]. The '805d' which will be printed by the ZENROM Disassembler is the decimal equivalent of 325h. Because each machine code instruction is only 10 bits, this means that the maximum value that can be loaded into C[X] using an LDI is 3FF - because the upper 2 bits of C[XS] are cleared during an LDI instruction.

The method of manually pushing an address onto or popping an address off of the return stack is by use of the instructions STK=C and C=STK.

STK=C takes an address from the address field of C, C[6:3], and places it as the first return address on the stack, moving RTN 1 to RTN 2, RTN 2 to RTN 3, ..., until RTN 4 is pushed off the top of the stack and is lost.

C=STK on the other hand, takes the address RTN 1 and puts it into C[6:3] and drops the stack - thus losing RTN 1 from it. The address 0000 replaces RTN 4 at the top of the stack after the latter has been dropped to RTN 3.

GTOKEY has the effect of copying the contents of the keycode register (KEY), into the two least significant digits of the program counter (PC). Basically, the instruction should perform a jump dependent upon the key pressed and, as such, it would have been an extremely useful instruction - had the keycodes been arranged more suitably. Unfortunately, all programmers have their 'off days'. GTOKEY is not used at all by the operating system.

The Read ROM instruction, RDROM, is used for reading a location in ROM. Given an address in C[6:3], RDROM will return to C[X] the word at that address. This instruction is used, for example, when the 41 is running a User code program in a plug in ROM. RDROM is used to fetch the user code byte from the ROM, which can then be processed in much the same way as if the byte was in a main RAM routine.

Two boolean operations are allowed for, 'AND' (C=CANDA) and 'OR' (C=CORA). These both operate on all 56 bits of C and A and leave the result in C.

The instruction 'PERSLCT' is used for peripheral access and will be covered in Chapter 8.1 on Special Instructions.

The two remaining instructions in this subclass, 'RAMSLCT' and 'WDATA' are used to transfer data between the CPU and main memory.

Memory Access Instructions

In order to fully understand this section, you should make sure you are familiar with the structure and addressing of main memory (see Chapter 3.7).

In order to read from, or write to a register in main memory, you must first select that register by using the 'RAM select' (RAMSLCT) instruction. This instruction takes its argument from the least significant 10 bits of C[X]. Thus, if you wish to select the first key assignment register you might use the following sequence of instructions

```
130 LDI
0C0 CON 192     Load the address of the register
270 RAMSLCT     Select the register
```

The selected register remains as the 'active' register until a different address is selected.

To write data to a register the WDATA instruction is used which copies the contents of C to the selected register. To read from a register use the RDATA instruction which will copy the contents of the selected register into C. This method of selecting, reading from and writing to a register can be used for all of the 41's RAM space including the status registers and extended memory. However, there is a much simpler way of accessing the status registers (RAM addresses 000 to 00F) by using the instructions in subclass A (for writing) and subclass E (for reading).

For the instructions in these subclasses to function correctly Chip 0 must be selected. This means that any register between addresses 000 and 00F can be selected. (The sixteen status registers exist physically on the same RAM chip, called Chip 0). Subclass A (REG=C) allows you to write to any of the status registers T through e whilst subclass E (C=REG) only allows reading from register Z through e. It is impossible to read directly from register T since the instruction, which would be C=REG 0/T, is in fact the RDATA instruction. Another point to note is that the instructions C=REG and REG=C have the effect of selecting that register in the same way as RAMSLCT.

This means that if, for example the following instructions are executed

```
046 C=0 X        Clear C[X] to zeroes
270 RAMSLCT      Select Chip 0
0F8 C=REG 3/X    Read the X register
```

then, not only is the X register read, but it is made the selected register and so there are now 2 ways of writing data to X (WDATA and REG=C 3/X) and also 2 ways of reading data from X (RDATA and C=REG 3/X). A further implication of this, is that although register T (address 000) was selected, it is not possible to read or write to T using RDATA and WDATA since the C=REG 3/X effectively deselected that register. This means that normally you will have to select register 000 immediately prior to reading register T.

Other Class 0 instructions

The rest of the class 0 instructions covered in this section do not fall into any particular category but are more general purpose instructions.

Subclass 8

CLRRTN (clear the first return address) is similar to C=STK in that it drops the return stack but it differs in that the return address is not loaded into C, but is simply lost.

PT=P and PT=Q are the instructions used to select either P or Q as the active pointer referred to as PT. The active pointer remains active until the alternate pointer is selected.

?P=Q is a test instruction which will set the carry flag if both the pointers are pointing to the same digit.

?BAT is another test instruction that will set the carry flag if the battery is low. It is this instruction that the operating system uses to determine whether or not to set the low BAT annunciator.

ABC=0 has the effect of clearing all three accumulators and is equivalent to the three class 2 instructions

```
A=0     ALL
B=0     ALL
C=0     ALL
```

GTOC is a branching instruction that takes an address in C[6:3] and loads it into the program counter (PC).

C=KEY fetches the contents of the keycode register (KEY) and copies it into the keycode field of C, C[4:3].

SETHEX and SETDEC are used to select the mode in which arithmetic operations are performed. The HP-41 CPU is able, not only to work in hexadecimal, but also in BCD (Binary Coded Decimal). The only instructions affected by the arithmetic mode are the arithmetic instructions in Class 2. The arithmetic mode will be covered more fully in the section on Class 2 instructions.

POWOFF, DISOFF and DISTOG are used to determine the 'power mode' of the HP-41. There are four possible power modes:

```
    Deep Sleep  - Display off, CPU not running
    Drowsy      - Display off, CPU running
    Light Sleep - Display on, CPU not running
and Run mode    - Display on, CPU running
```

The 41 is in deep sleep when it is switched off; drowsy for a short time after it is switched on, but the display is still off; light sleep whilst it is waiting for a key to be pressed, and in run mode whilst a key is held down or a function is being executed.

POWOFF is a 2 word instruction the second word of which should always be the NOP (000). This instruction will stop the processor running. Note that this does not mean that the 41 will always switch off as that is dependent on the state of the display.

DISOFF and DISTOG will be discussed under Display handling in Chapter 8.2.

The 'return' instructions (CRTN, NCRTN and RTN) are similar to their user code counterpart in that they mark the end of a subroutine and return to either the calling routine or the operating system. RTN is the most common and will normally mark the end of a function - it will always execute a return when encountered. CRTN (If Carry then ReTurN) will only execute a return if the carry flag is set. Note that the carry flag only remains set for one instruction cycle after it is set and so this instruction should be preceded by a test instruction. NCRTN (If No Carry then ReTurN) is the converse of CRTN in that a return will only be executed if the carry flag is not set.

The first instruction in subclass 0 is the machine code NOP (no operation). The main use for this instruction is to ensure that the carry flag is clear.

Subclasses 9 & B are covered in Chapter 8 on Advanced Machine Code Programming along with the other instructions in subclass 0.

All of subclass D is unused.

6.3.2 CLASS 1 Instructions

Class 1 instructions are of the 'branching' type and enable you to either go to, or to execute code anywhere in the 41's 64k address space. All of these instructions in fact comprise of 2 consecutive words. The format for the two words of a class 1 instruction are:

Word 1 - c c c c d d d d 0 1
Word 2 - a a a a b b b b s t

where:
- the two least significant bits of word 1 indicate the class of the instruction,
- the two least significant bits of word 2 (s t) indicate the type of jump.
- if bit 's' is 0 then the instruction is an execute or if it is 1 the instruction is a GOTO.
- bit 't' indicates if the instruction should be executed on Carry (t=1) or on No Carry (t=0).

s t = 0 0 NCXQ - If no carry then execute
= 0 1 CXQ - If carry then execute
= 1 0 NCGO - If no carry then go to
= 1 1 CGO - If carry then go to

- the address to jump to is spread over the two words with the two least significant digits in the 8 most significant bits of word 1 (cccc dddd); and the two most significant digits of the address in the 8 most significant bits of word 2 (aaaa bbbb).

Example: *Work out the two words that represent the instruction* NCGO 0952

First convert the address to binary:

0952h = 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0
(a a a a b b b b c c c c d d d d)

The instruction is an NCGO so bits 's t' = 1 0
Therefore, the words are:

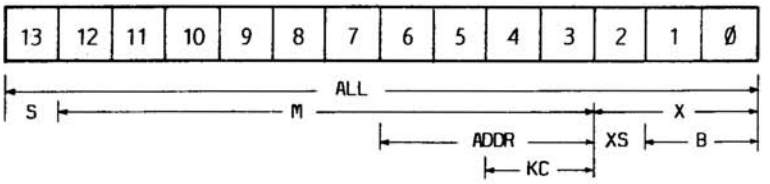
Word 1 = 0 1 0 1 0 0 1 0 0 1 = 149h
Word 2 = 0 0 0 0 1 0 0 1 1 0 = 026h

Try working out the words for these instructions

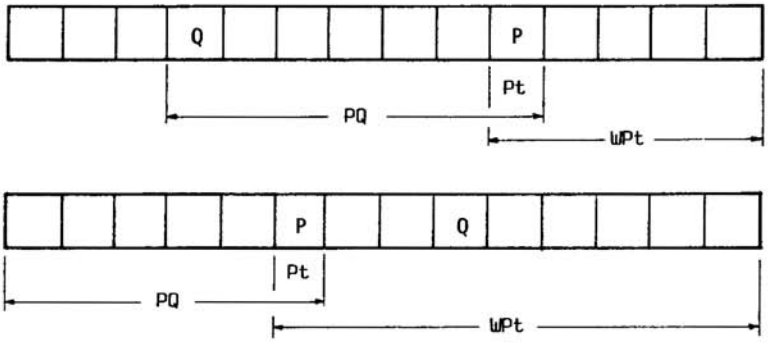
1. NCXQ 2CF0
2. CGO 7D36
3. NCGO 00F0
4. CXQ 3F83

Answers
1. 3C1 0B0
2. 0D9 1F7
3. 3C1 002
4. 20D 0FD

6.3.3 Time Enable Fields



Examples of Using Pointers
(Assuming Pt=P)



ALL	- ALL of the register, digits [13:0]
S	- Sign digit, digit [13]
M	- Mantissa, digits [12:3]
X	- eXponent, digits [2:0], includes the exponent sign
XS	- eXponent Sign, digit [2]
PT	- PoinTer, at the digit indicated by the active pointer [PT]
WPT	- Word through PoinTer, from digit 0 up to the digit indicated by the active pointer [PT:0]
PQ	- from pointer P to pointer Q. If P =< Q then from P up to Q [Q:P] If P > Q then from P up to digit 13 [13:P]
ADDR	- ADDRess field, digits [6:3]
KC	- KeyCode, digits [4:3]
B	- least significant Byte, digits [1:0]

Note: Class 2 instructions use only the first 8 of the above fields.

Figure 6.3.3 Time Enable Fields

From reading the preceding sections of this manual, you will have come across the ideas that a 41 register is 14 digits (56-bits) wide and that it can be subdivided into various fields representing the sections of the register, i.e. exponent, mantissa, etc. So far we have only come across the fields [S], [M], [XS] and [X], but now, with machine code programming, we can extend the number of fields available by introducing the use of the pointers P and Q.

Certain machine code instructions, especially those in Class 2, operate only on specific fields of the registers. The 41 CPU is a bit serial chip which simply means that data is sent on the bus sequentially, one bit at a time. If we consider the process of sending a register, e.g. C, to the I/O bus. The data must be sent one bit at a time, and a total of 56 bits must be sent. We must now start to think in real time. If it takes T seconds to send the whole register, then it will take T/56 seconds to send one bit. Subdivide T into 56 sections (t0 through t55 where t0 represents the time at which the first bit of data (i.e. bit 0 of the register) is sent and t55 is the time at which the last bit of data is sent. We can now specify the fields in terms of time; i.e. [X] can be described as the data sent between times t0 and t11 and [M] as the data sent between t12 and t51. Hence the term 'time enable' fields. In reality, the clock frequency of the 41 is 360kHz - giving a value of t (the time to send 1 bit) of 1/360000s, thus making T = 56/360000 = 156microseconds, which is the time taken by the CPU to execute one machine code instruction.

6.3.4 CLASS 2 Instructions

This class covers the arithmetic, shift and comparison instructions of the CPU. All of the instructions in this class operate on either one or two of the accumulators C, A and B. The format of the words representing class 2 instructions is:

i i i i i m m m 1 0

The two least significant bits of the word are 1 0, indicating that it is a class 2 instruction. The 5 most significant bits (iiiii) are the instruction type - therefore giving 32 different types of instruction (2↑5) - and the remaining bits (mmm) indicate the modifier.

Class 2 instructions use what are called 'Time Enable Modifiers' to determine which digits of the accumulators are used for the operations. Refer to figure 6.3.4. These time enable modifiers, also called 'fields', allow you to specify any particular digit or group of consecutive digits in the accumulators. The modifiers used in class 2 instructions are:

Modifier:	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Field:	PT	X	WPT	ALL	PQ	XS	M	S

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

FIGURE 6.3.4 MACHINE CODE WORD TABLE - CLASS 2

Most of the instructions are self explanatory if you refer to Figure 6.3.4 (Class 2 word table). The first 3 are used to clear the individual accumulators and the next block are used to transfer data between them. Next are the arithmetic instructions that place a result into A, followed by the arithmetic instructions that put a result into C. Of the latter, the only two that need some explanation are C=—C which returns the 16's complement of C if in hex mode or the 10's complement if in decimal mode, and C=—C—1 which returns the 15's complement if in hex and the 9's complement if in decimal. (You are probably more familiar with the terms 1's complement and 2's complement which deal with binary numbers - these instructions are the hexadecimal and decimal equivalent of those terms).

The next set of instructions are the comparisons followed by the shift instructions for shifting any of the accumulators right one digit and finally the A shift left instruction.

The arithmetic instructions in this class, can all set the carry flag if either an overflow or an underflow occurs in the field in which the operation takes place; for example, the following sequences will both set the carry flag:

```
15C PT=    6
210 LC     8
3DC +PT
1E2 C=C+C  PT
```

```
05A C=0    M
27A C=C-1  M
```

Not only does the setting of the carry flag depend on the field on which the operation takes place, but it also depends on the arithmetic mode that has been selected, hex or decimal. If in decimal mode, then the digits are treated as BCD (Binary Coded Decimal) and additions and subtractions are performed accordingly. Digits A - F are not valid BCD digits.

Here is an example of the difference between a hex and a decimal calculation:

```
260 SETHEX
130 LDI
340 CON    832
1F6 C=C+C  XS
1E6 C=C+C  X
```

Result: C[X] = C80 (carry clear)

```
2A0 SETDEC
130 LDI
340 CON    832
1F6 C=C+C  XS
1E6 C=C+C  X
```

C[X] = 280 (carry set)

In brief, the carry flag is set, if either:

a) an addition causes the most significant digit of the time enable field to go from F to 0 in hex mode or from 9 to 0 in decimal mode; or

b) a subtraction causes the most significant digit of the time enable field to go from 0 to F in hex mode or from 0 to 9 in decimal mode.

Note: If a register transfer, shift, rotate, AND or OR instruction is executed in decimal mode, the CPU operates in hex mode long enough to do the operation and then reverts to decimal mode. In this manner, non-BCD digits are not destroyed during the operation. Also, it is still possible to load hex digits A - F into the C register using LDI or LC whilst the CPU is set to decimal mode.

	JNC +	JC +
00	003	007
01	00B	00F
02	013	017
03	01B	01F
04	023	027
05	02B	02F
06	033	037
07	03B	03F
08	043	047
09	04B	04F
0A	053	057
0B	05B	05F
0C	063	067
0D	06B	06F
0E	073	077
0F	07B	07F
10	083	087
11	08B	08F
12	093	097
13	09B	09F
14	0A3	0A7
15	0AB	0AF
16	0B3	0B7
17	0BB	0BF
18	0C3	0C7
19	0CB	0CF
1A	0D3	0D7
1B	0DB	0DF
1C	0E3	0E7
1D	0EB	0EF
1E	0F3	0F7
1F	0FB	0FF

	JNC +	JC +
20	103	107
21	10B	10F
22	113	117
23	11B	11F
24	123	127
25	12B	12F
26	133	137
27	13B	13F
28	143	147
29	14B	14F
2A	153	157
2B	15B	15F
2C	163	167
2D	16B	16F
2E	173	177
2F	17B	17F
30	183	187
31	18B	18F
32	193	197
33	19B	19F
34	1A3	1A7
35	1AB	1AF
36	1B3	1B7
37	1BB	1BF
38	1C3	1C7
39	1CB	1CF
3A	1D3	1D7
3B	1DB	1DF
3C	1E3	1E7
3D	1EB	1EF
3E	1F3	1F7
3F	1FB	1FF
40	-	-

	JNC -	JC -
00	-	-
01	3FB	3FF
02	3F3	3F7
03	3EB	3EF
04	3E3	3E7
05	3DB	3DF
06	3D3	3D7
07	3CB	3CF
08	3C3	3C7
09	3BB	3BF
0A	3B3	3B7
0B	3AB	3AF
0C	3A3	3A7
0D	39B	39F
0E	393	397
0F	38B	38F
10	383	387
11	37B	37F
12	373	377
13	36B	36F
14	363	367
15	35B	35F
16	353	357
17	34B	34F
18	343	347
19	33B	33F
1A	333	337
1B	32B	32F
1C	323	327
1D	31B	31F
1E	313	317
1F	30B	30F

	JNC -	JC -
20	303	307
21	2FB	2FF
22	2F3	2F7
23	2EB	2EF
24	2E3	2E7
25	2DB	2BF
26	2D3	2D7
27	2CB	2CF
28	2C3	2C7
29	2BB	2BF
2A	2B3	2B7
2B	2AB	2AF
2C	2A3	2A7
2D	29B	29F
2E	293	297
2F	28B	28F
30	283	287
31	27B	27F
32	273	277
33	26B	26F
34	263	267
35	25B	25F
36	253	257
37	24B	24F
38	243	247
39	23B	23F
3A	233	237
3B	22B	22F
3C	223	227
3D	21B	21F
3E	213	217
3F	20B	20F
40	203	207

FIGURE 6.3.5. MACHINE CODE WORD TABLE - CLASS 3

6.3.5 CLASS 3 Instructions

Class 3 instructions are all short jumps with a range of +63 or -64 words from the address at which the jump occurs. The format of a class 3 instruction is:

d j j j j j j c 1 1

The two least significant bits indicate that it is a class 3 instruction. Bit 'c' determines whether the jump is On Carry (=1) or On No Carry (=0). The remaining bits d j j j j j are the jump distance - which is in 2's complement form if the jump is -ve (backwards). This means that bit 'd' will be 0 for a forward jump and 1 for a backward jump. The mnemonics for these are:

JNC +	Jump if no carry, forwards
JNC —	Jump if no carry, backwards
JC +	Jump if carry, forwards
JC —	Jump if carry, backwards

Refer to Figure 6.3.5 for a complete list of the jumps and their opcodes.

6.4 THE HP-41 ROM FORMAT

The HP-41 can address 65536 (64k) words of ROM. Each word of ROM space has a 4 digit (16 bit) address at which it is located. The 64k of ROM is split up into 16 pages of 4k, see figure 6.4.1. The lowest 3 of these pages, internal ROMS 0, 1 and 2 are where the 12k operating system resides. Page 3 is used for the internal extended functions module along with operating system extensions in an HP-41CX only. Page 4 is reserved for 'takeover' ROMs that ignore the O/S. Hewlett-Packard's DIAGNOSTIC ROM, used to test 41's returned for servicing, is addressed to this page. Page 5 is where a TIMER ROM resides if installed in your machine. Note that in 41-CXs there are two 4k ROMs both addressed to page 5, the primary one of which is the CX-TIMER ROM, and the secondary one contains much of the code for the extended functions. These two pages are controlled and selected by a method of 'page switching' which is discussed in more detail in section 8.1 . Page 6 is reserved for the Printer ROM, either from the 82143A or the IL module. One point to note about the HP-IL module, is that when you switch to printer disable using the switch on the undercasing of the module is that the address of the printer rom changes to address 4. The coding of the IL Printer ROM is so arranged that it does not try to take over the 41 when it is disabled - as would normally be expected from a ROM at page 4. Page 7 is inhabited by the Mass Storage and Control Functions of the HP-IL ROM. Certain ROMs (TIMER, HP-IL and PRINTER) are hard configured to specific pages - irrespective of the particular port into which they are plugged. Pages 8 to F are dedicated to ROMs plugged into the 41 via the four I/O ports on the rear. Each port is allotted 8k of space to allow for both 4k and 8k ROMS. Most HP ROMs are 4k and as such occupy only the lowest addressed page dedicated to the port into which it is plugged. 8k ROMs such as the PPC-ROM and the HP-IL DEVELOPMENT ROM are addressed to both pages of the relevant port. If no ROMs are present, in any particular page, then that page appears to be full of NOPs (000 words).

xØØØ		XROM number	
xØØ1		Number of catalog entries (n)	
xØØ2		Address of first function	FUNCTION ADDRESS TABLE (FAT)
xØØ3			
:			
x(2n)		Address of last function	
x(2n+1)			
x(2n+2)	ØØØ	End of table	
x(2n+3)	ØØØ		
x(2n+4)		ROM function code	
:			
xFF3			
xFF4		Interrupt vector: Pause loop	
xFF5		" " : Main Running Loop	
xFF6		" " : Deep Sleep Wake up, no key down	
xFF7		" " : Off	
xFF8		" " : IO Service	
xFF9		" " : Deep Sleep Wake up	
xFFA		" " : Cold Start	
xFFB		ROM trailer	
xFFC			
xFFD			
xFFE			
xFFF		Checksum	

ROMs at Pages 3 and 5 through F follow the above format.

FIGURE 6.4.2. ROM FORMAT

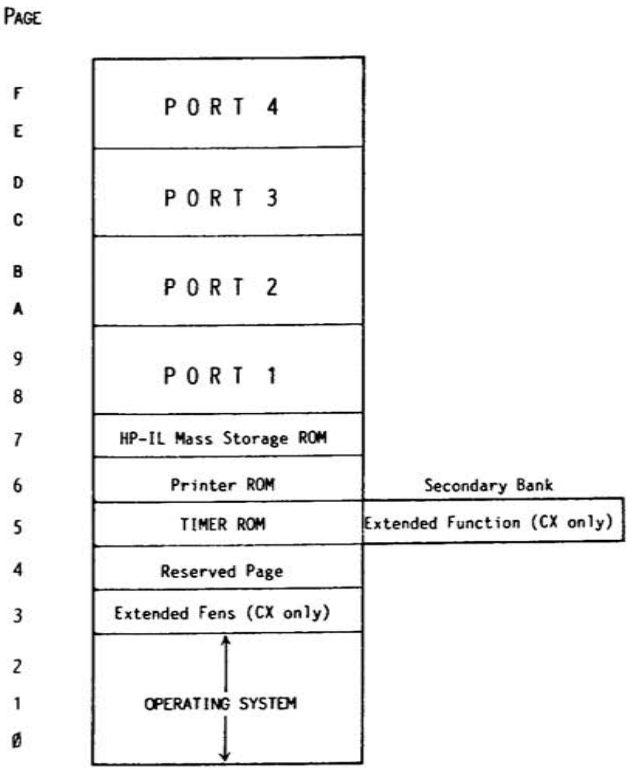


FIGURE 6.4.1. ROM PAGE STRUCTURE

Note that both Q-ROM and EPROM devices do not follow the above strictures, and can be addressed to any ROM page using switches on the devices. Refer to the respective equipment manual for more details.

Another notable exception to the above 'rules' is the ZENROM. When ZENROM is plugged into a port, it occupies the upper page allotted to that port. The reason for this, is to accommodate the growing number of users who are installing modules inside their 41s. Because ZENROM is addressed to the upper 4k of a port, both ZENROM and another standard 4k module can be effectively plugged in to the same port. Contact one of the User Groups (Appendix C) for information on this. Be warned that modifications to either your HP-41 or a plug-in module (including ZENROM) are neither supported by Hewlett-Packard nor Zengrange and will automatically nullify your warranty on the equipment. The modification is mentioned here for information only, and should not - under any circumstances - be taken as a recommendation.

When discussing a ROM it is generally meant as referring to a 4k page of ROM. Any ROM page, except pages 0 through 4, can contain both user code programs and machine code functions, and must conform to a specified format. The format for a ROM is shown in figure 6.4.2.

The first word of a ROM, address 000, is the XROM number of that ROM coded in hex. For example the first word of the TIMER ROM, which has an XROM number of 26, will be 01A. The maximum value of this word is 01F, i.e. XROM 31. The second word of the ROM, address 001, is the number of catalogue entries in that ROM. The header of a ROM, i.e: -ZENROM 3B, counts as an entry in the catalogue. Again this word is coded in hex and can range in value from 000, for no catalogue entries, to 040, for 64 catalogue entries.

The next section of the ROM is called the 'Function Address Table', known as 'FAT'. Each entry in the catalogue requires two words in the FAT to determine the start address of the function (or program) and other status information about that entry. Although a ROM can contain both User code and machine code we are only concerned with machine code functions for the purposes of this manual. The O/S needs to know where each function in a plug-in ROM is addressed in order to be able to execute that function. Therefore, the start addresses are stored in the FAT. Each function has two consecutive words in FAT to indicate its the first executable word of code of that function. The FAT is arranged in catalogue order, i.e: the first pair of words point to the address of the function that will appear first in the catalogue, the second pair point to the function that will appear second, etc. Given a pair of words from the FAT, the least significant digit of the first word indicates the most significant digit of the address of function in the ROM. The two least significant digits of the second word of the FAT entry indicate the two least significant digits of the address of the function. The last entry in the FAT must be followed two NOPs (000) marking the end of the FAT. Take as an example the following start of a ROM:

Address	Word	
000	015	XROM number 21
001	002	ROM has 2 catalogue entries
002	004	
003	02F	First function starts at address 42F
004	001	
005	023	Second function starts at address 123
006	000	
007	000	NOPs to mark the end of the FAT

Were this ROM at page 8, then the two functions would start at 842F and 8123 respectively.

The rest of the ROM, up to the special reserved words starting at address FF4, is available for the functions contained in the ROM. Suppose the second function in the example ROM, described above, is called TEST. Then the format of this function in the ROM would be:

```
Address  Word

  11F     094      T
  120     013      S
  121     005      E
  122     014      T
  123     fff      first word of function
   '       '
   '       '
  zzz     111      last word of function
```

Notice that the first word of the function is at the address pointed to by the FAT and that the preceding words contain the name of the function in reverse order. The function name is stored as 'display coded' characters a table for which is given in figure 8.2. The last character of the label is actually in display coded format, but has the constant 80h added to indicate it is the last character of the name.

There are various alterations to the coding of the function name and the start of the function that can be made to tailor the operation of that function. If the first word of the function is a NOP then that function will be non-programmable (such as MCED). If the first 2 words of the function are NOPs then the function will not only be non-programmable, but will also be 'Immediate execute' (if used whilst in PRGM mode). An immediate execute function is one that cannot be nulled by holding down the key (such as SST).

You are also able to specify the function to be of the prompting variety. Changing values of the two most significant bits in the first two characters of the function name will determine what kind of prompt is required according to the following table:

Value of top two bits

CHR 2	CHR 1	Prompt type	Example
0	0	No prompt	SIN
0	1	Alpha (null input valid)	CLP
0	2	Accept 2 digits, ST, IND, IND ST, +, -, * or /	STO
0	3	Accept 2 digits or non-null Alpha	LBL
1	1	Accept 3 digits	SIZE
1	2	Accept 2 digits, ST, IND or IND ST	RCL
1	3	Accept 2 digits, IND, IND ST or non-null Alpha	XEQ
2	1	Accept non-null Alpha	
2	2	Accept 2 digits, IND or IND ST	SF
2	3	Accept 2 digits or non-null Alpha	
3	1	Accept 1 digit, IND or IND ST	FIX
3	2	Accept 2 digits, IND or IND ST	
3	3	Accept 2 digits, IND, IND ST, non-null Alpha	
		[.] or [.] [.]	GTO

When ZENROM is plugged in however, the type of prompts that are accepted is changed due to the 'Direct-key Synthetics' facility. For a full description of 'Direct-key Synthetics' see section 4.1 .

Thus if we change our TEST label:

```
Address  Word

  11F    094     T
  120    013     S
  121    205     E
  122    214     T
  123    000     NOP
  124    xxx     first word of function
```

the function would be non-programmable and would prompt with 2 underscores and accept either 2 digits, IND and 2 digits or IND ST and a valid stack register.

If a numeric prompt is completed, then when the function starts executing the byte equivalent of the input will be in A[X] or, if an Alpha prompt, then the Alpha string will be left justified in status register Q with the characters in reverse order. For example:

CLP 'ABC' then Q = 0 0 0 0 0 0 0 0 4 3 4 2 4 1

RCL IND 61 then A[X] = 0BD

See the byte table, Figure 3.1, for the byte equivalents.

Normally the first function in the ROM will be the ROM header, i.e. the name of the ROM. The header is coded in the ROM in the same manner as any other function. A header should always be at least 8 characters in length so that it cannot be executed by 'conventional' methods and so that it will show up as a ROM header during the CAT 2 function on a CX. It is customary for the first executable address of the header to be a RTN.

Addresses FF4 to FFA are reserved for interrupt vectors which are polled by the operating system at various times. If a one of these locations contains a non-zero word (normally a Class 3 jump) then control is passed to the interrupt vector when that location is polled. It is recommended that you ensure that these location are kept as NOPs in any device you are using to write your own machine code since their misuse can cause the 41 to lock up.
After the interrupt vectors come 4 words, FFB - FFE, which contain the trailer for the ROM. The trailer should contain 2 display coded letters followed by a display coded revision number. For example the trailer for the plug-in timer module is:

```
Address  Word

 5FFB    003     C
 5FFC    031     1
 5FFD    00D     M
 5FFE    014     T     Trailer is TM1C
```

The last word of a ROM is always the checksum. This is calculated by summing the values of all the other words in the 4k block, i.e. 000 - FFE, in a 10-bit field using end-around-carry, that is when the sum overflows the field an extra 1 is added, and then taking the 2's complement of that sum.

6.5 MACHINE CODE EXAMPLES

By far the best way to explain machine code programming on the 41 is by giving documented examples of machine coded functions so that you can see how things work in reality. The two examples in this section will be useful in not only explaining what has been said in this, and previous chapters, but also the functions might prove useful in their own right.

These examples contain instructions that call subroutines in the HP-41 operating system. It is strongly recommended that you should have a copy of the O/S documented listings which are available from suppliers listed in Appendix C. One of the most important contributors to efficient machine code programming is effective use of the O/S and a good knowledge of the O/S will prove invaluable.

If you are new to machine code programming on the 41, it is recommended that you do not try and key these programs in immediately, but rather wait for the next chapter which will give instructions on how to store these functions into a Q-ROM device. You will, however, find it useful to follow these programs through using the comments provided.

Example 1. Saving the Stack

This short routine is designed to make a copy of the stack registers X, Y, Z, T and L and store them in the first 5 registers of the statistic block. Hence by changing the location of the statistics registers using ΣREG, you can retain numerous different stacks.

The routine works in the following manner:

1. lines 07 - 09 Find the address of the first stats register from status register c.

2. lines 0A - 1E Check that the first and last registers we are going to store into exist. If not then exit via 'ERRNE' which will display the 'NONEXISTENT' message and cope with all the error handling.

3. lines 1F - 2D Take each of the stack registers, L to T, in turn and copy them to the statistics block. Stop when register T has been copied.

```
00      0CE Σ
01      00B K
02      014 T
03      013 S
04      00F O
05      014 T
06      013 S

07      378 C=REG  13/c        Fetch the contents of status register c to C
                               i.e. C[13:11] =  RAM address of ΣREG.
```

```
08    1BC RCR    11        Rotate C so address IREG is in C[X]
09    106 A=C    X         Store the address of IREG in A[X]
0A    384 CF     0         Clear flag to indicate checking validity of
                           start of IREG block.


0B    270 RAMSLCT          Select the register to be checked.
0C    038 RDATA            Read the register into C
0D    11A A=C    M         Store register [M] in A[M] for comparison later
0E    2BA C=-C-1 M         Take the 1's complement of Register [M]
0F    2F0 WDATA            Write complemented form back to the register
10    038 RDATA            Read the register again.  If the register is
                           nonexistent then what is read will not be the
                           same as what was just written.
11    2BA C=-C-1 M         Recomplement the register mantissa
12    2F0 WDATA            Restore the original contents to the register
13    37A ?A#C   M         Set carry if register is nonexistent
14    381 *                "ERRNE"
15    00B CGO    02E0      Register not there so exit via "NONEXISTENT"


16    38C ?FS    0         Has IREG+004 been checked ?
17    047 JC     +8        Yes, then go on to save the stack.
18    130 LDI              No
19    004 CON    04        Load 004
1A    0A6 A<>C   X         C[X] =  Address of IREG,  A[X] =  004
1B    206 C=A+C  X         C[X] =  Address of IREG+004, ie the address of
                           the highest register that will be used
1C    158 M=C              Store the address IREG+004 in M
1D    388 SF     0         Set flag to indicate checking IREG+004
1E    36B JNC    -13       Go back and check the validity of end of IREG


1F    0A6 A<>C   X         Recall to C[X] the 004 which will be used as
                           the start address of the registers to be saved
                           ie register L
20    070 N=C              N =  address of register to be saved
21    270 RAMSLCT          Select the register to be saved
22    038 RDATA            Read the data from the register to be saved
23    10E A=C    ALL       Store the data into A
24    198 C=M              Fetch the address of the stats register into
                           which the data is to be stored
25    270 RAMSLCT          Select the destination register
26    266 C=C-1  X         Decrement the destination address ready for
27    158 M=C              next time around and restore in M
28    0AE A<>C   ALL       C =  Data to be stored
29    2F0 WDATA            Write data to the destination register
2A    0B0 C=N              Fetch address of the stack register just stored
2B    266 C=C-1  X         Decrement the address, carry if decremented 000
                           which indicates all the stack has been saved
2C    3A3 JNC    -0C       Not done yet, save next register
2D    3E0 RTN              Finished.
```

As an exercise you might like to devise a complementary routine which will restore the stack using the first 5 statistics registers.

Example 2. Substituting a character in alpha.

The function ASUB, 'alpha substitute', takes a character from X, either a decimal ASCII character code or, if alpha data, then the first character of the string, and places it at the character position in alpha specified in Y. If the character specified by X is > 256 or no character is specified (null alpha) then a DATA ERROR is generated. The character position in Y should be between 0 (first character) and the length of alpha —1 (last character) otherwise a DATA ERROR will be generated.

The routine works in the following manner:

1. lines 04 - 0B Determine if X is a number or alpha data
2. lines 0C - 12 If X is numeric then convert to hex and check the character code is valid.
3. lines 13 - 1B If X is alpha data then find the 1st character in X
4. lines 1C - 1C Store the character to be substituted in
5. lines 1D - 2C Initialise the search routine
6. lines 2D - 3E Search for start of alpha and then for character position specified by Y.
7. lines 3F - 41 Store the new character into alpha.

```
00    082 B
01    015 U
02    013 S
03    001 A

04    0E0 PT=Q
05    2DC PT=     13          Initialise Q to digit 13
06    0A0 PT=P                Reselect P as active pointer
07    0F8 C=REG   3/X         Fetch character to be substituted into alpha
08    10E A=C     ALL         A =  Character to substitute in

09    27E C=C-1   S
0A    27E C=C-1   S           Carry if in alpha format (ie top digit = 1)
0B    047 JC      +08         In alpha format so find first char in register

0C    0AE A<>C    ALL         Recall character number
0D    38D *                   "BCDBIN"
0E    008 NCXQ    02E3        Convert number to hex in  C[X]
0F    2F6 ?C#0    XS          Is character valid, ie < 256 ?
10    037 JC      +06         No, then DATA ERROR
11    39C PT=     0           Pointer to least significant digit of character
12    053 JNC     +0A         Go on to store char in G

13    35C PT=     12
14    052 C=0     PQ          Clear byte 6 (alpha indicator) of the register
15    2EA ?C#0    WPT         Are there any characters in the register ?
16    0B5 *                   "ERRDE"
17    0A0 NCXQ    282D        No, then DATA ERROR
```

```
18    3D4 -PT
19    3D4 -PT              Decrement pointer to next character position
1A    2F2 ?C#0    PQ       Is there a character here ?
1B    3EB JNC     -03      Yes, then go to save character

1C    058 G=C              Store the character in G

1D    130 LDI
1E    005 CON     05d      Load test value for end of alpha
1F    0E6 B<>C    X        Store test in B(X)
20    0B8 C=REG   2/Y      Fetch the character position to be substituted
21    38D *                "BCDBIN"
22    008 NCXQ    02E3     Convert to hex in C[X]
23    05A C=0     M        Clear out C[M]
24    1BC RCR     11       C[M] =  Character position
25    130 LDI
26    008 CON     008h     Load address of Reg P (start of search address)
27    10E A=C     ALL      Store character position and reg. address in A
28    270 RAMSLCT          Select Register P
29    038 RDATA            Read Register P
2A    15C PT=     6        Pointer to first digit of non-alpha part of P
2B    052 C=0     PQ       Clear the non-alpha part of P
2C    384 CF      0        Clear flag to indicate search for alpha start

2D    394 ?PT=    0        Finished checking this register ?
2E    043 JNC     +08      No, then go to check the next char in this reg
2F    1A6 A=A-1   X        Yes, then decrement register address in A[X]
30    326 ?A<B    X        Finished checking all of alpha ?
31    32F JC      -1B      Yes, then DATA ERROR
32    0A6 A<>C    X
33    106 A=C     X        Duplicate next register address into C[X]
34    270 RAMSLCT          Select the next register
35    038 RDATA            Read the next register
36    3D4 -PT
37    3D4 -PT              Decrement pointer by one byte to check next chr
38    38C ?FS     0        Checking for alpha start or character position?
39    027 JC      +04      Character position
3A    2F2 ?C#0    PQ       Alpha start  -  is this first alpha character ?
3B    393 JNC     -0E      No, then check next character
3C    388 SF      0        Yes, then start search for character position
3D    1BA A=A-1   M        Decrement position counter, carry if pos found
3E    37B JNC     -11h     Position not found yet, carry on search

3F    098 C=G              Position found, pointer is at the relevant byte
                           to recall the substitution character from G
40    2F0 WDATA            Write the updated register back
41    3E0 RTN              Done.
```

To show the advantages that writing such a machine code routine can have when used in your user code programs, let's consider the following search and replace program which will search alpha for all occurrences of the character specified in Y and replace them with the character specified in X. The program uses functions from the extended functions module.

```
01♦LBL "REPLACE"
     02 X=Y?
     03 RTN
     04 X<>Y

     05♦LBL 00
     06 POSA
     07 X<0?
     08 RTN
     09 X<>Y
     10 ASUB
     11 LASTX
     12 GTO 00
     13 END
```

To try this user code routine out, key into alpha the text string:

'ZENCODE MNEMONICS'

Now key as follows:

78 ENTER↑ (character code for 'N')
42 (replacement character '*')
XEQ 'REPLACE'

Now view the Alpha Register, which will show the result: **'ZE*CODE M*EMO*ICS'**

7

USING ZENROM TO INPUT MACHINE CODE

This chapter of the handbook assumes that you have read the previous chapters on Programming in HP-41 Machine Code, and understand the concepts of HP-41 User Code and Synthetic Programming.

7.1 THE MACHINE CODE EDITOR (MCED)

Machine code programming on the HP-41 involves the use of the Machine Code EDitor (MCED) function contained in ZENROM. Executing MCED places you into a new environment with the 'COMMAND ?' prompt in the display. The MCEDitor keyboard is detailed on one of the keyboard overlays provided.

To use most of the MCED functions you will need to connect the HP-41 to one of the available Quasi-ROM devices. The exception is 'DISASSEMBLE' - which allows you to produce a listing of any ROM module plugged into the 41, but only if a printer is connected to the 41 or HP-IL. A printer is not required for other MCED functions, but to program efficiently, and be easily able to debug your routines, a printer will prove invaluable.

Command Level:

This is the main input level and is entered when you first execute MCED. You may also return to this level by pressing the [CMD] key at any time whilst MCED is active. In command level, the display will show 'COMMAND ?' and wait for you to select one of the active key functions detailed on the overlay.

When selected, you will find most of the Editor commands have a common input prompt format of:

Command: Start Address , Finish Address

For example, the DISASSEMBLE prompt appears as:

DIS: _ _ _ _ , _ _ _ _

After inputing the hexadecimal start address, you have an option of specifying a decimal number of words or lines in place of a hexadecimal finish address. This is selected by pressing [DEC] and indicated by a 'd' appearing in the prompt:

DIS: 2 F 0 A , d _ _ _

At this point only the decimal keypad remains active.

Certain MCED functions have a secondary prompt such as:

LMT: __ __ __ __

(where this allows you to specify the limit address beyond which the action will not take place.)

Certain prompts may show an 'A' in the prompt, thus indicating that an absolute address input is expected.

Utility Keys:

[R/S] Pressing [R/S] accepts the input of addresses, etc., and begins the specified action. This allows a check that the correct input was made - which can prove very useful with MCED functions such as CLR.

[EXIT] Pressing [EXIT] from the main editor prompt, will return you to normal HP-41 usage. MCED is set to automatically 'timeout' after approximately two minutes of inactivity in a manner similar to ED in the CX. You can prevent this time out by executing ON.

[SHIFT][CMD] Cancels the current prompt sequence and returns you to the main editor prompt 'COMMAND ?'.

[←] Deletes the last key input.

[DEV] Pressing this key provides a toggle between the ProtoCODER 2 and MLDL type device write formats. The default is for a MLDL type device. The '0' annunciator will be set whenever a ProtoCODER device has been selected.

[DEC] Pressing this key before the input of the second address of a function prompt allows the entry of a decimal value of lines or words.

Function Keys:

[GTO] Allows the writing of M-Code instructions into your Q-ROM device with the hex loader. GTO requires the input of a start address and responds with a prompt showing:

	Address	Current Word	
e.g:	1468	154	__ __ __

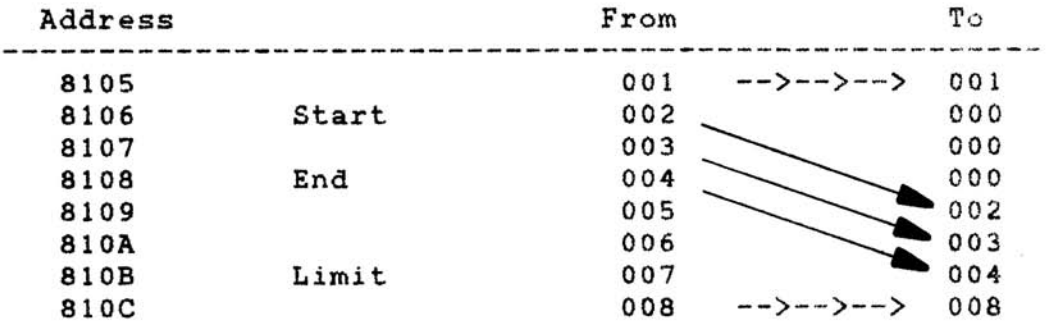
The hex-keypad allows input of the 3-hex digits representing the word you wish to write to that location. Press [R/S] to accept the new word. Both the [SST] and [SHIFT][BST] keys are active within the hex-loader and permit forward and backward movement without changing the word at the current address location. If a printer device (in NORM or TRACE mode) is attached, then the entered word is also disassembled to the printer. Pressing [SST] will also cause that word to be disassembled to the printer.

[INS] Allows insertion of a block of NOPs into Q-ROM before the specified address. A specific decimal number of NOPs may be input by pressing the [DEC] key instead of specifying the end hex-address. Specifying 'd000' will default to 'd001' - thus inserting one NOP instruction. Because [INS] moves all surrounding code in Q-ROM, to make way for the NOPs, a secondary prompt allows you to specify a 'limit address':

LMT: __ __ __ __

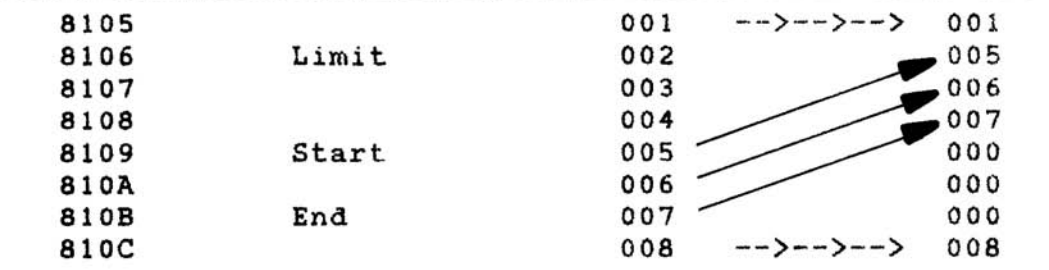
beyond which no code will be changed. By specifying a LMT address *greater than* that for the end address, the surrounding code is moved up memory (to a higher address).

Eg INS: 8106,8108 LMT:810B



If the LMT address is before the start address then surrounding code will be moved down memory (to a lower address)

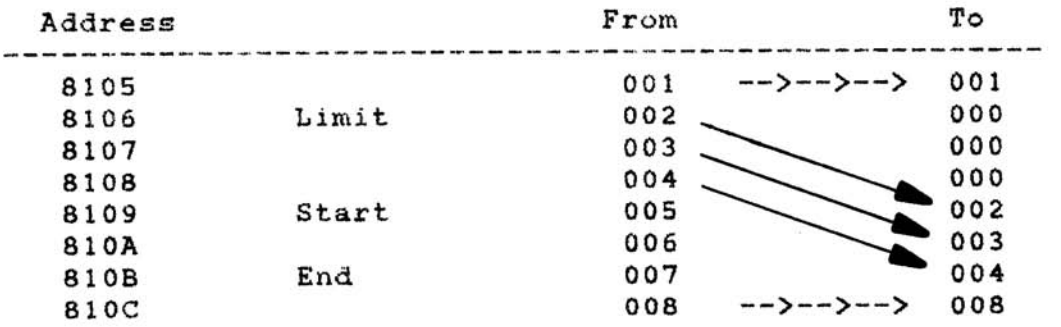
Eg INS: 8109,d003 LMT: 8106



DATA ERROR messages will be issued if the end address is less than the start address or the LMT address is between the start and end addresses.

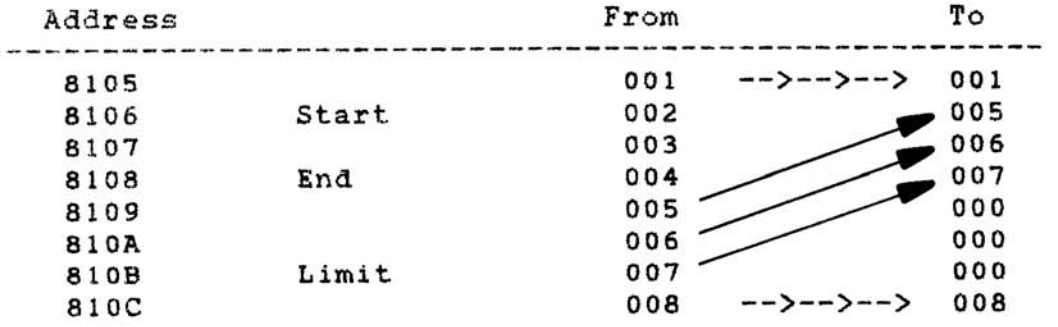
[DEL] Will delete a block of code from Q-ROM at a specified start and end address. A specific decimal number of lines can be deleted by pressing the [DEC] key as before. When deleted, surrounding code is moved in Q-ROM. A secondary LMT prompt specifies a boundary beyond which no code is moved. By specifying a LMT address *greater than* that for the end address, the surrounding code is moved up memory (to a higher address).

e.g. DEL: 8106,8108 LMT: 810B



If the LMT address is *before* the start address, surrounding code will be moved down memory (to a lower address).

Eg. DEL: 8109,d004 LMT: 8106



DATA ERROR messages will be issued if the end address is less than the start address or the LMT address is between the start and end addresses.

[CPY] Requires a hex start and end address of a block of code, or a specific decimal number of words to copy. A secondary prompt requests the starting address of the destination into which the code will be copied. CPY will allow copying to overlapping blocks. Specifying an end address less than the start will cause a DATA ERROR message.

[CLR] Will clear a block of Q-ROM between the specified start and end addresses, or a specific decimal number of words from the start address - by pressing the [DEC] key option. If the end address is less than the start, a DATA ERROR message is issued.

[SVE] Enables Machine Code routines to be stored in packed data-format in main HP-41 memory for subsequent transfer to mass storage media. Although the format is not the same as that for ROM>REG, the two are compatible. [SVE] expects a hex start and end address, or decimal number of words, of code to save. A secondary prompt requires input of absolute register address, in main or XRAM memory, into which the packed data will be stored.

If a non-existent register address is encountered, SVE will abort with the message NONEXISTENT. Specifying an end address less than the start will cause a DATA ERROR message.

[GET] Recalls main or XRAM memory registers and writes the data contained therein to a Q-ROM device. Expects the first and last register absolute hex-addresses to use, or a decimal number of registers to GET. A secondary prompt requests the Q-ROM destination address. The ZENROM data format will cope with incomplete registers, thereby allowing recall of two consecutive blocks of code without error. A DATA ERROR message results if the end address is less than the start address.

7.2 DISASSEMBLING MACHINE CODE

Disassembling machine code is very much like printing or listing a program in main RAM. By this stage you should realise that the term 'ROM' covers both those internal to the HP-41 and those contained in a plug-in module or Quasi-ROM device. Here we are concerned with those containing machine code programs and routines that either make the 41 behave as it does (the operating system), or add extra features or functions (plug-in ROMs). To understand these machine coded instructions, we need some method of listing them - just as you would do a user code routine in main RAM using the printer/IL function 'PRP'. The method of doing this is called 'disassembling' - for which we need a 'disassembler'. A disassembler simply takes the code from the ROM area specified and converts the otherwise meaningless hex-opcodes into more meaningful mnemonics.

ZENROM has a disassembler resident in the Machine Code EDitor (MCED). As a preliminary example of how the disassembler works, lets first have a look at a section of the 41's operating system. To use the disassembler, however, requires a printer - either the HP-82143 or a printer attached via the HP-IL. The printer should be set in TRACE or NORMal mode, by the switches on the front panel or by setting the flags 15 & 16 for an HP-IL printer for the disassembler to work correctly.

Enter the machine code editor environment by: XEQ 'MCED'. The prompt 'COMMAND ?' indicateds that you are now in the 'command level' and that it expects you to select one of the options available on the MCED keyboard. Pressing the [DISASSEMBLE] key (situated on the [ENTER↑] key) selects the option and returns the prompt:

DIS: _ _ _ _ , _ _ _ _

which requests you to input a start and end address for the disassembling. At this point the hexadecimal keypad (0 to 9 and A to F) is active.

Consider disassembling the HP-41 internal routine 'CLA' (CLear Alpha) which is situated at address 10D1h - 1OD6h (You should refer to the VASM listings of the operating system for details of these address locations.) Key in the starting address:

[1] [0] [D] [1]

You may correct mistakes, as normal, by pressing the [←] key.

After keying in the start address, you may either specify an 'end address', or by pressing the [DEC] (on the decimal point key) specify the quantity of lines to be disassembled. For the former, simply input the end address as you did for the start and then press the [R/S] key to accept and begin the disassembly. If you are immediately returned to the command level (the 'COMMAND ?' prompt), then you either have no printer attached, or it is not in the correct mode. Simply repeat the above sequence. If you wish to disassemble a specific number of lines of code, then after entering the start address, press the [DEC] key. At this point, a 'd' will appear in the end address portion of the display prompt:

DIS: 1 0 D 1 , d _ _ _

Only the numeric keys 0 to 9 are now active and you may enter a decimal number of lines to be disassembled.

CLA Routine Disassembled

```
DIS: 10D1,10D6
10D1 04E C=0    ALL
10D2 168 REG=C  5/M
10D3 1A8 REG=C  6/N
10D4 1E8 REG=C  7/O
10D5 228 REG=C  8/P
10D6 3E0 RTN
```

The disassembler produces printed listings in four columns of the format

address word instruction parameter

where the 'address' is given in hexadecimal and the 'opcode' is given as three hex digits. The instruction mnemonics given are those in the ZENCODE set - refer to Appendix E - Reference Tables for a listing. Some of the more esoteric instructions, such as those for the TIMER and DISPLAY codes will not be disassembled correctly. Since the opcodes for these are the same as those used for the more basic instructions, it would be impossible to correctly disassemble these instructions as it would mean having to establish exactly which peripheral is enabled when the instruction is executed.

Certain features do not follow the above standard, but, nevertheless, enable the FAT at the start of an external ROM to be disassembled to provide useful information. To illustrate this, disassemble the FAT for the Printer ROM, that is address 6000h to 6035h for the HP-82143 dedicated printer or to 6039h for the printer ROM in the HP-IL Module.

Partial FAT Listing for IL Printer Rom

```
6000 01D XROM   29
6001 01B FCNS   27
6002 007 -PRINTER 2E
6003 1BB ADDR   67BB
6004 002 ACA
6005 0B4 ADDR   62B4
6006 00C ACCHR
6007 05D ADDR   6C5D
```

Notice that the first line tells you what the XROM number of the ROM is in decimal.

Next are pairs of words that indicate where the functions are located within the 4k block. The first line of each pair indicates the name of the function as it would appear in the catalogue. User code programs have the usual 'super tee' as a prefix. The second line in each pair is the address of the first executable code of the function.

7.3. WRITING MACHINE CODE TO A DEVICE

In this section, Example 1 (STOSTKΣ) from Section 6.5. will be used as the basis of keying a machine code program into a Q-ROM device.

Before starting to actually key in the program, we must first initialise a page of Q-ROM. To do this, firstly plug in the Q-ROM device and set the address switches on the device as described in the owner's manual for the device. The address switches are used to select the ROM-page at which the device will appear to be positioned at. For simplicity, for the remainder of this chapter we will refer to this page as digit 'x'. This is because the exact location will depend upon the individual user.

Having plugged in the device, also attach a printer - if you have one - and set it to NORM or TRACE mode. To make it easier to follow the example key sequence, we must firstly clear the device to get to a known state. To do this use the following sequence :

KEY STROKES	*DISPLAY*	*PRINTER LISTING*
XEQ 'MCED'	COMMAND ?	

MCED allows writing to both MLDL type devices or the ProtoCODER 2. If you are using the ProtoCODER 2 then press the [DEV] key to toggle between the MLDL or ProtoCODER 2 formats. The '0' annunciator will appear in the display whenever the ProtoCODER 2 format is selected.

Now to clear the device press the [CLR] key and then enter the start and end addresses of the Q-ROM block to be cleared, in this case x000 to xFFF.

[CLR]	CLR: _ _ _ _, _ _ _
[x][O][O][O]	CLR: x000, _ _ _
[x][F][F][F]	CLR: x000,xFFF

If you make a mistake whilst keying in the addresses or you select the wrong function, then simply press [SHIFT] [CMD] to abort the function and start again, or use the [←] key to backarrow the input. Once the addresses have been correctly entered press the [R/S] key to accept the input.

[R/S]	CLR:x000,xFFF COMMAND?	CLR:x000,xFFF

When the display reverts to the 'COMMAND ?' prompt the function has been completed.

Next, we need to initialise the page with an XROM number and a FAT. To write code into the device, select the [GTO] function and key in the address at which you wish to start writing code.

[SHIFT][GTO]	ADR: _ _ _ _
[x][0][0][0]	ADR: x000
[R/S]	ADR: x000
	ADR: x000
	x000 000 _____

The display will now show the address pointed to, the word currently at that location and the three underscores which can be filled with the hex digits representing the new word to be written to that address. Let us, for arguments sake, select the XROM number 31 (in decimal) for our Q-ROM.

[0][1][F]	x000 000 01F	
[R/S]	x001 000 __ __ __	x000 01F XROM 31

Notice that if the printer is attached and it is in the correct mode, the word just written will be disassembled. The address pointed to will be automatically incremented and you can then key in the next word to go at the next address.

If you do not have a printer then you can verify that the word has been correctly written by pressing:

[SHIFT][BST]	x000 01F __ __ __

Notice that the word at that address has now been changed to 01F. Pressing [SST] will step on to the next address without altering the current location.

[SST]	x001 000 __ __ __	x000 01F XROM 31

The number of FAT entries in the page is expected at address x001, so key:

[0][0][1]	x001 000 001	
[R/S]	x002 000 __ __ __	x001 001 FCNS 01

Next we must input the FAT entry for the function. If we say that the function will start at address x100 this will leave plenty of space to fill out the FAT with the maximum number of 64 entries and have plenty of space to spare. If the function name will start at x100 then the first word of executable code will be at x107 since 'STOSTKΣ' will consume 7 words. Before keying in the FAT entry it is important to switch the printer to MAN mode for reasons that will be explained later. Key in the FAT entry as follows:

[[0][0][1]	x002 000 001
[R/S]	x003 000 __ __ __
[0][0][7]	x0003 000 007
[R/S]	x004 000 __ __ __

Having input the FAT entry for our function, we can now switch the printer back to NORM or TRACE to resume the printing functions. Since we know that the rest of the ROM is clear there is no need to explicitly key in the end of FAT marker (two NOPs).

We can now start to key in the function. To do this, hit either [←] or [SHIFT] [CMD] to get back to the command level. We now want to start entering code from address x100 :

[SHIFT][GTO]	ADR: __ __ __ __	
[x][1][0][0]	ADR: x100	
[R/S]	x100 000 __ __ __	ADR: x100

Now refer back to the listing of the function and simply enter all the words.

[0][C][E]	x100 000 0CE	
[R/S]	x101 000 __ __ __	x100 0CE
[0][0][B]	x101 000 00B	
[R/S]	x102 000 __ __ __	x101 00B K
...		
...		
[3][E][0]	x12D 000 3OD	
[R/S]	x12E 000 __ __ _	x12D 3OD RTN

When you have finished entering the function you should verify the listing and then exit MCED by returning to the command level and pressing [ON].

Now test the function by setting ΣREG to a known value and then storing some data onto the stack and XEQ 'STOSTKΣ'.

Recall the ΣREGs and check the function has worked correctly. If there is an error, then you should verify the code again.

Now you should enter Example 2 from 6.5 by following the same method except that the Q-ROM is already initialised, so only the FAT needs to be updated.

To update the FAT go into MCED and key :

[SHIFT][GTO]	ADR: _ _ _ _	
[x][0][0][1]	ADR: x001	
[R/S]	x001 001 _ _ _	ADR: x001

Change the number of FAT entries to 2 by keying :

[0][0][2]	x001 001 002	
[R/S]	x002 001 _ _ _	x001 002 FCNS 02

Now single step over the next 2 words which contain the FAT entry for the first function.

[SST]	x003 007 _ _ _	x002 001 STOSTKΣ
[SST]	x004 000 _ _ _	x003 007 ADDR x107

Notice that when address x002 is disassembled the function name is printed. The reason that the printer should be switched to MAN mode whilst keying in a FAT entry is that, because at that time no function name exists, the disassembler does not know this and assumes there is a valid function name at the address pointed to by the FAT entry. Hence an invalid function name will be printed. (In fact, if you are unlucky, it could result in printing a function name containing over 36000 characters.)

Now enter the new FAT entry (first switch printer to MAN mode):

[0][0][1]	x004 000 001
[R/S]	x005 000 _ _ _
[0][3][2]	x005 000 0032
[R/S]	x006 000 _ _ _

This means that the ASUB function name will follow immediately after the RTN at the end of STOSTKΣ

Now key in the function starting at address x12E.

[←]	COMMAND?	
[SHIFT][GTO]	ADR: _ _ _ _	
[x][1][2][E]	ADR: x12E	
[R/S]	x12E 000 _ _ _	ADR: x12E

7.4 EDITING MACHINE CODE

MCED has four editing facilities, CLR (clear), CPY (copy), INS (insert) and DEL (delete).

The operation of CLR has already been demonstrated in the previous section.

[CPY]

The copy function is used to duplicate any section of ROM space into a Q-ROM device. To demonstrate its operation let us duplicate the STOSTKΣ which you have already entered in the Q-ROM. To do this, follow the key sequence:

[CPY]	CPY: _ _ _ _, _ _ _ _	
[x][1][0][0]	CPY: x100, _ _ _ _	
[DEC][0][4][6]	CPY: x100,d046	
[R/S]	ADR: _ _ _ _	CPY: x100,d046
[x][1][7][0]	ADR: x170	
[R/S]	COMMAND?	ADR: x170

What has happened is that we filled the first four underscores with the start address of the block of code to be copied. Then, pressing the [DEC] key followed by [0] [4] [6] specified that we wished to copy 46(decimal) words from the start address. We could, alternatively, have keyed in the end address as: [x] [1] [2] [D]. Having specified the block of code to be copied, we had to specify the start address to which the code would be copied, which in this instance is [x] [1] [7] [0] - which means that the duplicate version of the function will follow immediately after ASUB in the Q-ROM.

Now, having duplicated the function, you must alter the FAT accordingly - no help this time.

If you now exit MCED and run a full CAT 2 you will see two entries of STOSTKΣ

[INS] and [DEL]

Now let's use the insert and delete options to alter the duplicate copy to make it STOSTKE which will save the stack in the first five registers below the permanent end (.END.), checking only that all five registers are available in main memory so that the data will not disturb the first registers of extended memory.

First change the function name accordingly:

[SHIFT][GTO]	ADR: _ _ _ _	
[x][1][7][0]	ADR: x170	
[R/S]	x170 0CE _ _ _ _	ADR: x170
[0][8][5]	x170 0CE 085	
[R/S]	x171 00B _ _ _	x170 08E E
[←]	COMMAND?	

Now referring to the listing of STOSTKΣ in Section 6.5 we can delete lines 08 to 1F and insert in their place:

```
266 C=C-1   X       C(X) = address of .END. -1
158 M=C             Store address of destination block in M
106 A=C     X       Store address in A[X]
130 LDI
0C4 CON     196
246 C=A-C   X       Carry if not enough room in main memory
381 *               "ERRNE"
00B CGO     02E0    No room, so exit via 'NONEXISTENT'
130 LDI
004 CON     04      Load address of first register to be sav
```

To delete lines 08 to 1F key in:

[DEL]	DEL: __ __ __ __, __ __ __ __	
[x][1][7][8]	DEL: x178, __ __ __ __	
[x][1][8][F]	DEL: x178, x18F	
[R/S]	LMT: x19D	
[R/S]	COMMAND?	LMT: x19D

The limit is the address beyond which no code will be changed which, in this case, is the end of the function at x19D.

Now we must make room for the new 10 words to be inserted:

[INS]	INS: __ __ __ __, __ __ __ __	
[x][1][7][8]	INS: x178, __ __ __ __	
[DEC][0][1][0]	INS: x178,d010	
[R/S]	LMT: __ __ __ __	INS: x178,d010
[x][1][8][F]	LMT: x18F	
[R/S]	COMMAND?	LMT: x18F

Here we have specified the start address, the address before which the block of NOPs will be inserted, the number of NOPs to insert (10) and the limit. The limit in this case is 10 words after the end of the function (x185), i.e. at x18F.

Now all that is left to do, is to replace the NOPs starting at x178 with the new words and finally check that STOSTKE works.

Other points to note about INS, DEL, CPY and CLR.

If the specified end address is less that the start address then a DATA ERROR message will be generated.

If Start address <= Limit <= End address then a DATA ERROR message will be generated.

If you specify a decimal number of lines, using [DEC], where: = d000 then it will default to d001 lines.

CPY is able to copy into an overlapping block of code.

7.5 STORING AND RETRIEVING ROM DATA

MCED has a facility for storing ROM data, i.e. your machine code programs, into the 41's RAM and recalling it and placing the code into a Q-ROM device. Remember, however, that this does not mean you can actually run the machine code from the 41's RAM, only that you can download it to RAM in the process of transferring to, or from, mass storage media.

To demonstrate the use of these functions, [SVE] and [GET], let's save the ASUB function which is located at x12E to x16F in the Q-ROM.

Firstly, we must establish at which absolute address Reg 00 is to be found. Exit from MCED and key:

RCL c
XEQ 'DECODE'

The address of register 00 is given by the last 3 but 3 characters displayed. We need to know the absolute address since MCED uses absolute addresses for all its register operations; this means that we are not restricted to which area of RAM can be used for storing ROM data and hence it is possible to store the data directly into extended memory. The address of register 00 will be referred to as 'pqr' since it will vary from user to user.

Having noted the address, now XEQ 'MCED' and do the following:

[SVE]	SVE: _ _ _ _, _ _ _ _	
[x][1][2][E]	SVE: x12E,_ _ _ _	
[x][1][6][F]	SVE: x12E,x16F	
[R/S]	REF: A _ _ _ _	SVE: x12E, 16F

Having specified the block of code to be saved we must now key in the absolute address of the first register into which the code will be saved.

[p][q][r]	REG: Apqr	
[R/S]	COMMAND?	REG: Apqr

If the message 'NONEXISTENT' is returned then there is not enough room to save the code in, so increase the SIZE and try again. You will require a SIZE >= 014.

MCED has now taken the code from the Q-ROM and packed it into a format that can be placed in main memory. For more details on the exact format of the packed data, refer to the end of this section.

Now clear the block of code (from the Q-ROM) that we have just saved. You may like to verify that it has been cleared, before we reinstate the code. To recall the code from the 41's RAM, simply key in:

[GET]	GET: A_ _ _, _ _ _	
[p][q][r]	GET: Apqr,A _ _ _	
[DEC][0][1][4]	GET: Apqr,d014	
[R/S]	ADR: _ _ _ _	GET: Apqr,d014
[x][1][2][E]	ADR: x12E	
[R/S]	COMMAND?	ADR: x12E

Notice that, as with all MCED functions, we can specify a decimal quantity rather than an end address in the initial prompt. (We recalled 14 registers since 66 words were saved and 66 words / 5 words per register = 13.2 registers).

ASUB will now have returned to the Q-ROM.

The 'ROMREG+' format

The ROMREG+ format stores 5 * 10-bit ROM words into one 56-bit register according to the following method:

Bit [55:52] =	0001 this makes the register appear as alpha data to prevent normalisation.
Bit [51] =	0
Bit [50] =	0 if the register contains a full complement of 5 words = 1 if the register is incomplete
Bit [40:41]=	first 10 bit word
Bit [39:30]=	second 10 bit word
Bit [29:20]=	third 10 bit word
Bit [19:10]=	fourth 10 bit word
Bit [9:0] =	fifth 10-bit word

If the register is incomplete, i.e. bit 50 = 1, then the last digit of the register contains the number of words missing from that register.

8

ADVANCED MACHINE CODE PROGRAMMING

This chapter covers some of the more advanced instructions for machine code programming. They are mentioned here for the sake of completeness. This chapter is not intended to be treated as an instruction manual, but rather, presents the basic information needed by an advanced machine code programmer. The beginner should firstly gain experience in writing machine code and contact the User Groups mentioned in Appendix C for further information.

8.1 SPECIAL INSTRUCTIONS

This section covers the Class 0 instructions that were not mentioned in Section 6.3.1, and some extra ones that are used with peripherals.

WMLDL (040h)

This instruction has been adopted by manufactures of some Q-ROM devices as the instruction used to write data to their Q-ROM equipment. When this instruction is executed, the Q-ROM device will take the ROM address to write to from C[6:3] and the 10-bit word to be written to that address from the ten least significant bits of C[X]. Note that this instruction does not write to either the ProtoCODER 1 or ProtoCODER 2 devices. For details on how to write to ProtoCODERs refer to their manuals.

ENBANK1 and ENBANK2 (100h and 180h)

These two instructions are only used by the HP-41CX. The CX has 6 internal 4k ROMs - pages 0, 1 and 2 are similar to those in the CV and form the basis of the O/S; page 3 contains part of the Extended functions ROM along with extensions to the O/S; page 5 is in fact two pages or banks of 4k. The primary page, bank 1, is the TIMER section and the secondary page, bank 2, contains extra code for the extended functions. The two instructions ENBANK1 and ENBANK2 are used for selecting which bank of 4k appears in the page 5 address space. Inside the CX there are only two ROM chips, each containing 12k of the O/S. Chip 1 contains rom pages 0, 1 and 2 and chip 2 contains pages 3, 5 and 5'. In order to switch page 5 banks the instruction has to be located at an address that is physically on the same chip. This means that if you try executing these instructions from a Q-ROM device they will not work. The recommended method for selecting the two banks is:

For bank 1 - NCXQ 5FC7
For bank 2 - NCXQ 5FC9

F=ST, ST=F and ST<>F (258h, 298h and 2D8h)

The F register is the 8-bit flag-out register which is used for controlling the beeper. Whenever there is a non-zero value in register F, the beeper will be making a noise. When the 41 generates tones, both normal and synthetic, it first clears F and then loads FFh into ST. The tone is then actually generated by exchanging F and ST repeatedly thus switching the output port on and off. The number of instructions between swaps is used to control how long the output port is on and off and hence controls the frequency of the beep. The number of times the swaps are performed determines the duration of the tone.

PERTCT (Class 0, Subclass 9)

The HP-41 CPU has the ability to let an external PERipheral Take ConTrol of the instructions being processed by the CPU. If control is passed to a 'smart' peripheral (i.e. one that has a processing ability of its own) by the PERTCT instruction, the following instructions are not observed by the 41s CPU but are rather interpreted in a different manner by the peripheral. Control is returned to the 41 when an instruction with least significant bit of 1 is encountered.

When control has been passed to the peripheral, there are three types of instruction that can be executed; test an external flag (?XF), read a peripheral register (RPREG) and write to a peripheral register (WPREG).

These instructions conform to the following bit patterns:

?XF N (N=0-F)	test an external flag
n n nn00 0011	

This instruction must return control, as the carry flag will be set if the external flag under test is set.

WPREG xy(xy=00-FF)	write a constant xy to the selected peripheral register
xx xxyy yy0r	if r=1 then return control to 41
PREG N (N=0-F)	read peripheral register N to C [1:0]
nn nn11 101r	if r=1 then return control to 41

Class 0, Subclass B

The CPU has a flag input line, for use by peripherals, whose state during each of the digit times represents one of the fourteen input flags. This line is tested during a ?PF (test peripheral flag) instruction and if the test is true, then the carry flag will be set. Most of these instructions are dedicated to certain peripherals such as the TIMER, printer, card reader, etc., and as such have a special mnemonic - to be covered in later sections of this chapter. There is however one 'general purpose' peripheral flag, PF 13, which is the service request flag. If this flag is set, then the 41 will service that peripheral at the next available opportunity. The mnemonic for testing the service request flat is ?SERV

PERSLCT (3F0h)

There are some peripherals that require data to be sent to and from them but do not have the 'intelligent' capabilities of peripherals that can take control (PERTCT). These peripherals, when selected, simply interpret some of the read and write instructions in Class 0 (Subclasses A, C and E) and perform the peripheral operations when these instructions are encountered.

PERSLCT takes its argument, an indentifier for the peripheral, from C[1:0]. Peripherals that use the PERSLCT instruction are:

Peripheral	Indentifier (C[1:0])
Timer	FB
Card Reader	FC
Display	FD
Wand	FE

Since peripherals that are selected using this instruction use the Class 0 instructions that are normally used to read and write to the status register, it is important to deselect any RAM register before selecting such a peripheral. The preferred method for selecting a peripheral is:

```
130 LDI                 }
010 CON    16           }  Select the nonexistent register at 010
270 RAMSLCT             }
130 LDI                 }
0xy CON    xy           }  Select peripheral xy
3F0 PERSLCT             }
```

ROW	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4*	⊢	a	b	c	d	e	¯	⁻	⁄	⁄	⁄	⁄	μ	≠	Σ	∠

FIGURE 8.2. DISPLAY CODED CHARACTERS

8.2 DISPLAY HANDLING

Each character in the display comprises nine bits - where bits 5 to 0 are the character code according to the first four rows of Figure 8.2 (Display codes), i.e. bits 5 and 4 are the row number and bits 3 to 0 are the column number; bits 7 - 6 are used to represent the punctuation following the character and bit 8 is used to indicate a character from row 4 of the display character code table. If bit 8 is set, then bits 5 and 4 should be clear otherwise a space will be displayed. The punctuation field (bits 6 and 7) are worked out in the following manner:

Punctuation	Bit 6	Bit 7
None	0	0
. (period)	0	1
: (colon)	1	0
, (comma)	1	1

The display is in fact organised into 3 main registers A, B and C (not to be confused with the accumulators in the CPU). Display register A contains the lowest 4 bits (bits 3 - 0) for all of the characters, register B contains bits 7 to 4 for all twelve characters and register C contains bit 8 for all twelve characters. There is also a twelve bit annunciator register.

The display responds to 37 instructions:

230 DISOFF	Switch the display off (ie go to drowsy mode)
320 DISTOG	Toggle the display on/off
3FC DISCMP	
3F0 PERSLCT	If c[1:0]=FDh then display is selected else display is diselected
2F0 WRITAN	Copy the bit pattern in C[X] to the annunciators
178 READAN	Read the annunciator bit pattern to CE

When writing to, or reading from the annunciators, the bits in C[X] correspond to the following annunciators:

Bit:	11	10	9	8	7	6	5	4	3	2	1	0
	BAT	USER	G	RAD	SHIFT	0	1	2	3	4	PRGM	ALPHA

The remainder of the display instructions are used to read and write to display registers A, B and C. The format for the ZENCODE mnemonics of these instructions are:

> The first 2 letters indicate whether the instruction is either a read or a write instruction (RD or WR).
> The next 1 to 3 characters indicate which display registers are to be used.
> Then follows a number indicating how many characters are to be included in the operation and the last letter indicates if the characters are to be read from or written to the left or right hand side of the display.

Thus the mnemonic RDAB6L means ReaD from registers A and B six characters from the Left of the display. Note that when writing to the display the characters are pushed on to the specified end of the display thus rotating the display and losing characters from the other end. Reading from one end of the display causes the display to be rotated so that the character(s) just read are taken off the end specified and pushed back on the other end. Also when reading from or writing to the display registers one digit of accumulator C is used for every character being accessed and each display register being used; i.e. WRAB6R uses 12 digits of C (6 characters ,2 display registers).

028 WRA12L	038 RDA12L
068 WRB12L	078 RDB12L
0A8 WRC12L	0B8 RDC12L
0E8 WRAB6L	0F8 RDAB6L
128 WRABC4L	138 RDABC4L
168 WRAB6R	
1A8 WRABC4R	1B8 RDC1L
1E8 WRA1L	1F8 RDA1R
228 WRB1L	238 RDB1R
268 WRC1L	278 RDC1R
2A8 WRA1R	2B8 RDA1L
2E8 WRB1R	2F8 RDB1L
328 WRC1R	338 RDAB1R
368 WRAB1R	378 RDAB1L
3A8 WRABC1L	3B8 RDABC1R
3E8 WRABC1R	3F8 RDABC1L

8.3 THERMAL PRINTER (HP-82143A)

The HP-82143 has the intelligent NPIC (Nut Peripheral Interface Chip) on board and is selected to 'take control' with the instruction:

264 PERTCT 9

When the printer chip is in control it responds to the following instructions:

003 ?XF 0	Is printer busy?
043 ?XF 1	Is printer status valid?
03A RPREG 0	Read printer status to C[13:10]
007 PRINT	Add byte in C[1:0] to print buffer

The printer status bits, when read into C[13:10] are as follows:

digit 13	digit 12	digit 11	digit 10
M1 M0 PRT ADV	OOP LB IDLE BE	LC SCO DW TEOL	LEOL IGN __ __

M0 and M1 indicate the mode

	M1	M0
Manual	0	0
Normal	0	1
Trace	1	0

PRT	— PRINT Key is down
ADV	— PAPER ADVANCE key is down
OOP	— OUT OF PAPER
LB	— Printer battery is low
IDLE	— Printer is idle
BE	— Buffer Empty
LC	— Lower case (ie User Flag 13 set)
SCO	— Special Column Out Mode
DWM	— Double Wide Mode (ie User Flag 13 set)
TEOL	— Type of last End of Line (0=left justified, 1=right justified)
LEOL	— Last byte an End of Line
IGN	— Ignore PAPER ADVANCE key

8.4 THE TIMER MODULE

The Timer chip (Phineas) contains 2 clock registers, timer A and timer B, and various other registers for accuracy factor, status etc. It responds to 26 instructions:

36C ?ALM	Set carry if an alarm is due
1AC ?TFAIL	Set carry if clock register access failed
3F0 PERSLCT	If C [1:0] = FBh then the timer chip is enabled else it is disabled
270 RAMSLCT	Disables the timer chip
060 POWOFF	Increments the time immediately if CPU will stop before the next normal increment

The remaining 21 instructions are only obeyed if the timer chip is enabled:

028 WTIME	Write C to clock register
038 RTIME	Copy clock register to C
068 WTIME-	Write C to clock register and set to decrement
078 RTIMEST	Copy clock register to C and start correction count
0A8 WALM	Write C to alarm register
0B8 RALM	Copy alarm register to C
0E8 WSTS	If timer A then write C to timer status register timer B then write C to accuracy factor
0F8 RSTS	If timer A then copy timer status register to C timer B then copy accuracy factor to C
128 WSCR	Write C to scratch register
138 RSCR	Copy C to scratch register
168 WINTST	Write C to interval timer and start
178 RINT	Copy interval timer to C
1E8 STPINT	Stop interval timer
228 WKUPOFF	If timer A then disable seconds wake up timer B then disable minutes wake up
268 WKUPON	If timer A then enable seconds wake up timer B then enable minutes wake up
2A8 ALMOFF	Disable alarm
2E8 ALMON	Enable alarm
328 STOPC	Stop clock
368 STARTC	Start clock
3A8 TIMER=A	Select timer A
3E8 TIMER=B	Select timer B

8.5 HP-IL

It is recommended that you read 'The HP-IL System' by Kane, Harper and Ushijima before writing machine code to drive the HP-IL.

Data can be copied from C[X] into any one of the HP-IL control registers 0 - 7 using the last 8 instructions in Class 0, subclass 0.

```
200 HPIL=C 0     copy C[1:0] into HP-IL register 0
249 HP1L=C 1     copy C[1:0] into HP-IL register 1

3C0 HP1L=C 7     copy C[1:0] into HP-IL register 7
```

Data can also be read from or written to any of the HP-IL registers by selecting the register using the PERTCT instructions and then using the instructions WPREG and RPREG as described in section 8.1 .

```
24 PERTCT 0      Select HP-IL register 0
064 PERTCT 1     Select HP-IL register 1
       ,
       ,
1E4 PERTCT 7     Select HP-IL register 7
```

Also HP-IL status flags can be testing using instructions in Class 0, subclass B:

```
0EC ?ORAV        Test if Output Register AVailable
12C ?FRAV        Test if FRame AVailable
16C ?IFCR        Test if InterFace Clear Received
26C ?FRNS        Test if Frame Received Not as Sent
2AC ?SRQR        Test if Service ReQuest Received
```

8.6 OTHER PERIPHERALS

The HP 82153 Wand is selected using PERSLCT with C[1:0] = FEh. If the wand is selected then the instruction 038 RDATA will read one byte from the wand buffer into C. Also the instruction 22C ?WNDB can be used at any time to test for data present in the wand buffer.

The card reader is selected using PERSLCT with C[1:0] = FCh. If it is selected then the following instructions are enabled:

```
028 ENDWRIT      End write cycle
068 STWRIT       Start write cycle
0A8 ENDREAD      End read cycle
0E8 STREAD       Start read cycle
168 CRDWPF       Fetch CaRD Write Protect Flag
1E8 CRDOHF       Fetch CaRD Over Head Flag
268 CRDINF       Fetch Card IN Flag
2E8 TSTBUF       TeST card read/write BUFfer
328 SETCTF       SET Card Trip Flag
368 TCLCTF       Test and CLear Card Trip Flag
3E8 CRDEXF       Fetch CaRD reader EXternal Flag
```

1st July 1984 GC FW JP DB IM DP EB

APPENDICES

APPENDIX A

OWNERS INFORMATION

MAINTENANCE

The Zengrange ZENROM programmer's module does not require maintenance, and contains no serviceable parts. During use there are several precautions that you should observe.

CAUTIONS

Do not place fingers, tools or other objects into the plug-in module ports of your HP-41 Computer nor into the ZENROM connector socket. Damage to the ZENROM module or to the HP-41's internal circuitry could result.

Turn off the computer - by pressing [ON] - before installing or removing a ZENROM module.

Modules can only be inserted one way into a port - Do not try to force a module into a port as this could damage the port or module connector.

Although the ZENROM case is strongly constructed, it should be handled with care and kept in the protective dust-proof bag or module holder when not installed in the HP-41 ports.

Protect the HP-41's ports from dust by keeping a port cap installed in the empty ports.

LIMITED ONE - YEAR WARRANTY

ZENROM has been written by Zengrange Ltd and manufactured to the highest possible standards by Hewlett Packard. With the exception of software content, ZENROM is warranted by Zengrange Ltd against defects in materials and workmanship affecting electronic and mechanical performance for a period of one year from the date of original purchase. If given as a gift, the warranty is transferred to a new owner for the remainder of the period. During the warranty period, Zengrange Ltd will replace or, at our option, repair a product that proves to be defective, provided it is returned, shipping prepaid, to Zengrange Ltd.

Zengrange Ltd make no expressed or implied warranty with regard to the program material offered, nor to merchantabilty or fitness of the program material for any particular purpose. Program material is made available to the user on an 'as is' basis, with the entire risk as to quality and performance resting with the user. Whilst every effort has been made to eliminate deficiencies in the program material, the user (and not Zengrange Ltd, nor any other party) shall bear the entire cost of all necessary correction and all incidental or consequential damages.

ZENROM is sold on the basis of specifications as at manufacture. Zengrange Ltd shall be under no obligation to modify or update ZENROM once manufactured.

CAUTION

ZENROM allows access to the HP-41's operating system, both directly via techniques of Synthetic Programming and indirectly, by allowing users to write their own machine language routines. Whilst the functions contained in ZENROM will not harm the HP-41 itself, it is possible for the user to cause either:

- a loss of memory contents (MEMORY LOST)
- or a 'lock-up' of the keyboard (placing the processor into an infinite loop) thereby preventing response to commands or keystrokes. This, and the recovery method, is described in the Owner's Manual for the HP-41CX.

Neither of these problems will cause harm to the HP-41, nor to ZENROM. However, the problem itself, or action taken to recover from a lock-up, may cause loss of user programs stored in RAM. Because these problems are user dependent, Zengrange will accept absolutely no responsibility for any loss or damage - whether consequential or incidential. It is good practice to back-up programs onto mass storage media periodically.

WARRANTY FOR CONSUMER TRANSACTIONS IN THE UNITED KINGDOM

This warranty shall not affect the statutory rights of a consumer whose rights as Buyer and the obligations of Seller are determined by statute.

LIMITATIONS OF WARRANTY

The Zengrange warranty does not, and shall not apply if ZENROM has been damaged by accident, misuse or if attempts have been made to modify the module. No other expressed or implied warranty is given. The repair, or replacement of ZENROM is your exclusive remedy.

SHIPPING FOR SERVICE

In the unlikely event that ZENROM is found defective, return the module, postage prepaid, to:

Zengrange Ltd,
Greenfield Road,
Leeds,
LS9 8DB,
England.

When returning ZENROM, be sure to include the following items:

* A sales receipt or other proof of purchase if the one-year warranty has not expired.

* A description of the problem, detailing when and how the problem occurs.

Whether ZENROM is still under warranty or not, it is your responsibility to ensure that the unit is securely packaged to prevent damage in transit (this is not covered by the Zengrange warranty) and that postage costs to Zengrange are paid.

TECHNICAL ASSISTANCE

The keystroke procedures, program material and operating instructions provided for using ZENROM are supplied with the assumption that the user has a working knowledge of the concepts, terminology, technology and equipment used. Zengrange Ltd's technical assistance is limited to explanations of operating procedures used in the manual and verification of results obtained in the examples.

ZENROM provides users with the means of accessing the operating system of the HP-41, and of producing their own machine language routines. As neither Synthetic Programming, nor Machine Language Programming is supported by Hewlett Packard, users should under NO circumstances contact Hewlett Packard for information.

APPENDIX B

OPERATING LIMITS

You will no doubt have noticed in the earlier sections that 'limitations' have been mentioned against certain ZENROM functions. These limitations are not 'bugs' in the programming of the ZENROM, but rather limitations and restrictions forced upon the design team by the manner in which the HP-41's operating system handles certain requirements.

An example should make this clear:

> If you switch your HP-41 into alpha mode, then press the User key, the processor in the 41 starts running continuously (as it would, for example, in StopWatch mode). This was the only way the HP-41 could be made to respond correctly to your keystrokes while in this mode. However, as in stopwatch mode, peripheral devices; e.g. Wand, Card Reader, Printer or HP-IL cannot 'interrupt' the HP-41's operation, as they otherwise would in normal ALPHA mode.

This is a limitation of the HP-41's internal programming, and is one of several such problems that face a machine language programmer on the HP-41. Whilst it might have been possible to work-around some restrictions, this would have greatly increased the amount of machine code needed for ZENROM routines and also have reduced the number of useful functions we could include. We think you will agree that our choice of powerful ZENROM routines with minor limitations was a good one!

Although there are ways to avoid the limitations in ZENROM, we have listed them below — examine the list carefully and make sure you understand exactly what the limitations are.

DIRECT-KEY SYNTHETICS:

The prompt-expansion functions in ZENROM, that allow you to directly key [RCL] [.] [M], etc., will not work on early HP-41s because of an operating system change by Hewlett Packard. Such 41s will normally have internal ROMs with revision codes before 'GFF'. The particular ROM causing the difficulty is ROM 0 - which must have a revision code after 'F'. Although it is possible for a ZENROM owner to check the ROM-revisions, we have included the relevant serial numbers below:

HP-41 INTERNAL ROM REVISIONS AND SERIAL NUMBERS

Rom Revisions	From Serial No.	To Serial No.
D, D, E	1926xxxxxx	1938xxxxxx
F, D, E	1936xxxxxx	1952xxxxxx
F, E, E	1951xxxxxx	2034xxxxxx
G, F, F	2035xxxxxx	

ROM revisions up to 'FEE' were used only in the model HP-41C, not in the HP-41CV nor HP-41CX. Revisions N, F, L are currently used in the HP-41CX. However, it is possible, if your HP-41C has been returned to Hewlett Packard for servicing, some ROMs may also have been replaced at that time.

Using the MCED function in ZENROM, the User can check ROM revision for each of the internal ROMs. The sequence is as follows:

XEQ 'MCED'	display shows:	COMMAND?
press [SHIFT][GTO]	display shows:	ADR: _ _ _ _
now input the address of internal ROM '0' as '0 F F E' using the hexadecimal keypad		
	display shows:	ADR: 0 F F E
press [R/S]	display shows:	0 F F E 0 0 r _ _ _ _

The 'r' in the second field is the revision digit. Compare this with those in the table below:

byte 'r'	4	5	6	7	8	9	A	B	C	D	E
ROM revision	D	E	F	G	H	I	J	K	L	M	N

You can also check the other internal ROMs by inputting the address as '1 F F E' and '2 F F E'.

If you feel you would like the full benefit of ZENROM, then HP will update your internal 41-ROMs - against a specific request - for the standard service charge. Contact your local HP Service Centre for details.

USER ALPHA KEYBOARDS:

As mentioned in the example above, the USER ALPHA Keyboards require the 41's processor to run continuously - including when the SYNTEXT hex prompt is in the display. This prevents the HP-41 from responding to service requests from the Bar-code Wand, the Card Reader, the Timer, or the Printer [PRINT] key. In addition, pressing the [ADVANCE] key, whilst in USER ALPHA mode during the entry of a program, will cause the printer paper to be advanced rather than allow an ADV instruction to be entered in the program. Simply complete the SYNTEXT prompt or switch off USER to allow these devices to take control. To reduce battery drain during these modes, the HP-41 is set to 'timeout' (switch off) after approximately 2-minutes instead of the usual 10 minutes.

ZENROM uses scratch areas in status registers R and e, to store the status of the USER flag when entering USER ALPHA mode - so that you always enter ALPHA in normal mode and, upon exiting, return to the mode set previously. Executing functions that store information into registers R or e whilst in USER ALPHA mode may overwrite this.

The system routines used by ZENROM to create the 'new' USER ALPHA keyboards will only allow ZENROM to operate upon key release and providing only one key is pressed at any one time.

Try this:

enter alpha mode and clear ALPHA;

now press [USER], then press and hold the [a] key

- this will show 'a' in the display;

now, before releasing [a], press [b], release [a], and then release [b].

- the result is entered as 'aB'.

To correctly use the USER ALPHA keyboards, you must make sure that a key is released before pressing another. This is not really a great problem for the User as the ABCDE layout of the HP-41 keyboard is not designed for speed typing.

Neither the USER ALPHA nor SYNTEXT entry modes will operate during a PSE instruction, inside the HP-41CX Text Editor (ED), nor during GETKEYX.

GENERAL:

Because of the manner in which ZENROM takes over control of the operating system at times, only one ZENROM should be plugged into the HP-41 at any one time.

ZENROM has the same ID as the HP-STANDARD APPLICATIONS PAC. Therefore, a conflict will occur if both modules are present at the same time. Although no harm will come to either module, the HP-41 will only see the module in the lowest numbered port. XROM 05 was chosen because the STANDARD PAC is the only module presently using this ID. Other XROM ID's, particularly those allocated to custom modules, have considerably more conflict of usage.

The HP-IL DEVELOPMENT Module function 'ROMCHKX', if applied to ZENROM, will produce an unusual ROM label of:

05 z)-L:a.

However, the function will then append 'TST' (meaning Testing the ROM) and then should verify correctly by displaying 'OK'.

APPENDIX C

BIBLIOGRAPHY & REFERENCES

USER GROUPS

ZENROM Users interested in finding out more about both Synthetic and Machine Language programming are recommended to contact one of the well established User Groups around the world. These groups are devoted to providing assistance and information for fellow users of Hewlett Packard's portable and hand-held computer ranges.

The most active groups, holding regular meetings and publishing their own regular magazines containing information of benefit to ZENROM Users, are listed below. For information on the many smaller groups in other countries, contact PPC in California, USA.

When writing to one of the User Groups, please enclose a Self Addressed Envelope together with stamps, International Reply Coupons or the common currency of two HP-41 magnetic cards.

PPC

Personal Programming Center,
P.O. Box 9599, Fountain Valley, California, CA 92728-9599, USA
English language journal 'PPC-Calculator Journal' supporting SP and M-Code. Back issues provide valuable information on the development of SP and M-Code programming.

PPC(UK)

Personal Programming Club,
c/o Astage, Rectory Lane, GB - Windlesham, Surrey, GU20 6BW, England. Membership Enquires: c/o Dave Bundy, 9 Kings Court, Kings Avenue, GB - Buckhurst Hill, Essex, IG9 5LP, England
English language journal 'DATAFILE' supporting SP and beginning M-Code.

PPC-DANMARK

Personal Programming Center - Danmark,
Postboks 2, DK - 3500 Vaerloese, Denmark. *Danish language journal 'USER' supporting SP.*

PPC-TOULOUSE

Personal Programming Club - Toulouse,
77 rue du Cagire, F - 31100 Toulouse, France.
French language journal 'PPC-T' supporting SP and active in the development of M-Code.

PPC-PARIS

Personal Programming Center - Paris,
56 rue J.J. Rousseau, F - 75001 Paris, France.
French language journal 'PPC-PC' supporting SP.

CCD

Computerclub Deutschland,
Postfach 2129, D - 6242 Kronberg 2, West Germany.
Largest European group with German language journal 'PRISMA' that has contributed to the development of SP. Active in M-Code development.

CCA

Computerclub Austria,
P.O.Box 50, A - 1111 Wien, Austria.
German language journal.

PPC-MELBOURNE

Personal Programming Centre - Melbourne,
P.O. Box 512, Ringwood, Victoria 3134, Australia. *English language Journal 'PPC-Technical Notes' directed towards more technical development of SP and currently the best coverage of M-Code.*
Note: European Membership & Subscriptions are now handled by: Editions du Cagire, 77 rue du Cagire, F - 31100 Toulouse, France, to whom membership & back-issue enquiries should be directed.

PPC-SYDNEY

Personal Programming Centre - Sydney,
P.O. Box C245, Clarence St, Sydney, NSW 2000, Australia.

PPC-HOLLAND

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

PPC-LAUSANNE

Club PPC-Lausanne,
Case Postale 79, CH - 1000 Lausanne 24, Switzerland.

BOOKS

Since the introduction of the HP-41 in 1979, quite a number of books have been written by members of the above User Groups to aid the new user to become proficient in HP-41 programming techniques. The better of these are listed below:

Synthetic Programming on the HP-41C by Dr W.C. Wickes (1980) Larken Publications, 4517 N.W. Queens Ave, Corvallis, Oregon 97330, USA. *The first book on SP. Techniques for creating SP are dated - but the book contains some valuable information. Treatment is advanced and possibly not suitable a a first SP book.*

German edition: Synthetische Programmierung am HP-41. Heldermann Verlag Berlin, Herderstr. 6-7, D-1000 Berlin 41, West Germany.

HP-41 Synthetic Programming Made Easy by Keith Jarett (1982) Synthetix, P.O.Box 1080, Berkeley, California 94701-1080, USA *Good 'beginners' book on SP.*

German edition: Syntherische Programmierung - leicht gemacht. Heldermann Verlag Berlin, Herderstr. 6-7, D-1000 Berlin 41, West Germany.

PPC-ROM User's Manual by PPC (1981) PPC, P.O.Box 9599, Fountain Valley, California 92728-9599, USA. *Published originally as handbook for the PPC-Custom ROM, but available separately. Valuable source of information and application routines.*

Au Fond de la HP-41 by Jean-Daniel Dodin (1981) Editions du Cagire, 77 rue du Cagire, F - 31100 Toulouse, France. *In French. Good introduction to SP, HP-41 operation and M-Code.*

The HP-41 Synthetic Quick Reference Guide by Jeremy Smith (1983) CodeSmith, 2056 Maple Avenue, Costa Mesa, California 92627, USA. *Very essential pocket sized reference book for SP printed on durable plastic paper.*

Extend your HP-41 by Dr Wlodek Mier-Jedrzejowicz (Summer 1984) PPC(UK), c/o Astage, Rectory Lane, GB - Windlesham, GU20 6BW, England. *Grew out of London User meetings. Answers many questions that trouble the user and dealer. Presents new information covering the whole spectrum of HP-41 use.*

Exploring Extended Functions on the HP-41 by Frank Wales (Autumn 1984) PPC(UK), c/o Astage, Rectory Lane, GB - Windlesham, GU20 6BW, England. *Most compleat treatment of Extended Functions & Memory. Additional comprehensive section about SP - Part of which was adapted, with permission, for Chapter 4 in this handbook.*

Calculator Tips & Routines - Especially for the HP-41C/CV Edited by John Dearing (1981) Corvallis Software Inc, P.O.Box 1412, Corvallis, Oregon 97339-1412, USA. *Compilation of useful utility routines including SP.*

Extended & Translated into German: Tricks, Tips und Routinen fuer Taschenrechner der Serie HP-41. Heldermann Verlag Berlin, Herderstr. 6-7, D-1000 Berlin 41, West Germany.

HP-41C Quick Reference Card by Keith Jarett (1982) Synthetix, P.O.Box 1080, Berkeley, California 94701-1080, USA. *Colour-coded Hexadecimal Byte Table printed on pocket sized laminated plastic card. A must for every serious HP-41 User.*

HP-41 VASM Listings - Zengrange Ltd, Greenfield Road, GB - Leeds, W.Yorks, LS9 8DB, England. PPC, P.O.Box 9599, Fountain Valley, California 92728-1080, USA. Editions de Cagire, 77 rue du Cagire, F - 31100 Toulouse, France. *Annotated listings of the HP-41C/V operating system, as released by H.P. - Note these are:* 'NOMAS' (**NOt MAnufacturer Supported**). *Purchaser agrees not to contact manufacturer.*

Most of these books, and other HP-41 related equipment, can be obtained from one of the following specialist mail order companies:

TH Boekhandel Prins, Binnenwatersloot 30, NL - 2611 BK, Delft, The Netherlands.

EduCALC Mail Store, 27953 Cabot Road, Laguna Niguel, California 92677, USA.

EQUIPMENT

Machine code creation and storage devices are available from the following organisations:

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

ProtoTECH Inc, P.O. Box 12104, Bolder, CO 80303, USA.

PPC-Denmark. Available Autumn 1984. (See User Groups above).

Printed circuit boards, kits and design details for self assembly are available from:

PPC-Calculator Journal, V9N3p27, PPC USA. (See User Groups above).

APPENDIX D

XROM NUMBERS

Function	XROM Number	Byte Code
-ZENROM 3B	05,00	A1,40
CLMM	05,01	A1,41
CLXM	05,02	A1,42
CODE	05,03	A1,43
DECODE	05,04	A1,44
LASTP	05,05	A1,45
MCED	05,06	A1,46
NOP	05,07	A1,47
NRCLM	05,08	A1,48
NRCLX	05,09	A1,49
NSTOM	05,10	A1,4A
RAMED	05,11	A1,4B
TOGF	05,12	A1,4C

APPENDIX E

'ZENCODE'

ZENROM Machine Code Mnemonics

(hexcodes are given in the machine code byte tables)

CLASS 0

NOP		No operation
WMLDL		Write word in C[x] to address in C[6:3] in a MLDL device
ENBANK1		Enable the primary bank of ROM page 5 in CX
ENBANK2		Enable the secondary bank of ROM page 5 in CX
HPIL=C	0-7	Write C[1:0] to HP-IL register n
CF	0-13d	Clear flag n
ST=0		Clear the Status Register (Flags 0-7)
SF	0-13d	Set flag n
CLRKEY		Clear the KEY register and KEY down flag if no key down
?FS	0-13d	Test flag n
?KEY		Test if key is down
LC	0-Fh	Load hex digit n at pointer position in C and decrement pointer
?PT=	0-13d	Test if active pointer is at digit n
-PT		Decrement pointer
G=C		Copy C[PT+1:PT] into G
C=G		Copy G into C[PT+1:PT]
C<>G		Exchange G with C[PT+1:PT]
M=C		Copy C into M
C=M		Copy M into C
C<>M		Exchange C with M
F=ST		Copy ST into F (output port)
ST=F		Copy F into ST
ST<>F		Exchange F with ST
ST=C		Copy C[1:0] into ST
C=ST		Copy ST into C[1:0]
C<>ST		Exchange C[1:0] with ST
PT=	0-13d	Set active pointer to digit n
+PT		Increment pointer
CLRRTN		Clear first return address from stack
POWOFF		Halt the CPU (should be followed by NOP)
PT=P		Select P as active pointer
PT=Q		Select Q as active pointer
?P=Q		Test if P and Q are pointing to same digit
?BAT		Test if Battery is low

ABC=0		Clear all of A B and C
GTOC		GOTO the address in C[6:3]
C=KEY		Copy the KEY register to C[4:3]
SETHEX		Set Arithmetic mode to hex
SETDEC		Set Arithmetic mode to decimal
DISOFF		Switch off the display
DISTOG		Toggle the state of the display
CRTN		If carry then return
NCRTN		If no carry then return
RTN		Return
PERTCT	0-Fh	Transfer control to peripheral n
REG=C	0-15d	Copy C into status register n
?PF	0-13d	Test peripheral flag (flag input)
?EDAV		
?ORAV		Test for output register available (HP-IL)
?FRAV		Test for frame available (HP-IL)
?IFCR		Test for Interface Clear received (HP-IL)
?TFAIL		Test for clock access failure (TIMER)
?WNDB		Test for data in Wand Buffer (WAND)
?FRNS		Test for frame received not as sent (HP-IL)
?SRQR		Test for service request received (HP-IL)
?SERV		Test for service request
?CRDR		Test card reader flag (CARD READER)
?ALM		Test for Alarm due (TIMER)
?PBSY		
N=C		Copy C into N
C=N		Copy N into C
C<>N		Exchange C and N
LDI		Load the next word as a data byte into C[X]
STK=C		Push C[6:3] onto return stack
C=STK		Pop first return address off stack in C[6:3]
GTOKEY		Copy KEY register into least significant byte of PC
RAMSLCT		Select RAM address in C[X]
CLRREGS		
WDATA		Copy C into selected RAM register
RDROM		Read address in C[6:3] to C[X]
C=CORA		Replace C with logical OR of C and A
C=CANDA		Replace C with logical AND of C and A
PERSLCT		Select peripheral identified by C[X]
RDATA		Read contents of selected RAM register into C
C=REG	1-15d	Copy status register n into C
RCR	0-13d	Rotate C right by n digits
DISCMP		
UNUSED		

CLASS 1

NCXQ	addr	If no carry then execute address
CXQ	addr	If carry then execute address
NCGO	addr	If no carry then goto address
CGO	addr	If carry then goto address

CLASS 2

A=0	TE	Clear specified field of A
B=0	TE	Clear specified field of B
C=0	TE	Clear specified field of C
A<>B	TE	Exchange A and B in specified field
B=A	TE	Copy A into B in specified field
A<>C	TE	Exchange A and C in specified field
C=B	TE	Copy B into C in specified field
B<>C	TE	Exchange B and C in specified field
A=C	TE	Copy C into A in specified field
A=A+B	TE	Add B to A in specified field
A=A+C	TE	Add C to A in specified field
A=A+1	TE	Increment A in specified field
A=A-B	TE	Subtract B from A in specified field
A=A-1	TE	Decrement A in specified field
A=A-C	TE	Subtract C from A in specified field
C=C+C	TE	Add C to C in specified field
C=A+C	TE	Add A to C in specified field
C=C+1	TE	Increment C in specified field
C=A-C	TE	Subtract C from A, result to C in specified field
C=C-1	TE	Decrement C in specified field
C=-C	TE	Take the 2's (or 10's) complement of C in specified field
C=-C-1	TE	Take the 1's (or 9's) complement of C in specified field
?B#0	TE	Test if B not equal to 0 in specified field
?C#0	TE	Test if C not equal to 0 in specified field
?A<C	TE	Test if A less than C in specified field
?A<B	TE	Test if A less than B in specified field
?A#0	TE	Test if A not equal to 0 in specified field
?A#C	TE	Test if A not equal to C in specified field
ASR	TE	Shift A right 1 digit in specified field
BSR	TE	Shift B right 1 digit in specified field
CSR	TE	Shift C right 1 digit in specified field
ASL	TE	Shift A left 1 digit in specified field

CLASS 2 TIME ENABLE MODIFIERS

PT	At digit pointed to by active pointer
X	Exponent field (digits 2:0)
WPT	Word through pointer (PT:0)
ALL	All of the register
PQ	If P<Q then Q:P If P>Q then 13:P
XS	Exponent sign, (digit 2)
M	Mantissa (12:3)
S	Mantissa sign (digit 13)

CLASS 3

JNC	+63 /- 64 If no carry jump n words
JC	+63 /- 64 If carry jump n words

DISPLAY

WRA12L	Write C[11:0] to lhs display register A
WRB12L	Write C[11:0] to lhs display register B
WRC12L	Write C[11:0] to lhs display register C
WRAB6L	Write C[11:0] as 6 characters to lhs display registers A and B
WRABC4L	Write C[11:0] as 4 characters to lhs display registers A,B and C
WRAB6R	Write C[11:0] as 6 characters to rhs display registers A and B
WRABC4R	Write C[11:0] as 4 characters to rhs display registers A,B and C
WRA1L	Write C[0] to lhs display register A
WRB1L	Write C[0] to lhs display register B
WRC1L	Write l.s. bit from C[0] to lhs display register C
WRA1R	Write C[0] to rhs display register A
WRB1R	Write C[0] to rhs display register B
WRC1R	Write l.s. bit from C[0] to rhs display register C
WRAB1R	Write C[1:0] to rhs display registers A and B
WRABC1L	Write C[X] to lhs display registers A,B and C
WRABC1R	Write C[X] to rhs display registers A,B and C
WRITAN	Write C[X] into the annunciators
RDA12L	Read 12 digits of lhs display register A into C[11:0]
RDB12L	Read 12 digits of lhs display register B into C[11:0]
RDC12L	Read 12 bits of lhs display register C into C[11:0]
RDAB6L	Read 6 characters of lhs display registers A and B into C[11:0]
RDABC4L	Read 4 characters of lhs display registers A,B and C into C[11:0]
READAN	Copy the annunciator status into C[X]
RDC1L	Read 1 bit of lhs display register C into C[0]
RDA1R	Read 1 digit of rhs display register A into C[0]
RDB1R	Read 1 digit of rhs display register B into C[0]
RDC1R	Read 1 bit of rhs display register C into C[0]
RDA1L	Read 1 digit of lhs display register A into C[0]
RDB1L	Read 1 digit of lhs display register B into C[0]
RDAB1R	Read 1 character of rhs display registers A and B into C[1:0]
RDAB1L	Read 1 character of lhs display registers A and B into C[1:0]
RDABC1R	Read 1 character of rhs display registers A,B and C into C[1:0]
RDABC1L	Read 1 character of lhs display registers A,B and C into C[1:0]

TIMER

WTIME	Write C to clock register
WTIME-	Write C to clock register, set clock to decrement
WALM	Write C to Alarm register
WSTS	If timer A, write C to status reg if timer B, write C to accuracy factor
WSCR	Write C to scratch register
WINTST	Write C to interval timer and start
STPINT	Stop interval timer
WKUPOFF	If timer A, disable 'seconds' wake up if timer B, disable 'minutes' wake up
WKUPON	If timer A, enable 'seconds' wake up' if timer B, enable 'minutes' wake up
ALMOFF	Disable alarm
ALMON	Enable alarm
STOPC	Stop clock
STARTC	Start clock
TIMER=A	Select timer A
TIMER=B	Select timer B
RTIME	Copy clock register to C
RTIMEST	Copy clock register to C and start correction count
RALM	Copy alarm register to C
RSTS	If timer A, copy status reg to C if timer B, copy accuracy factor to C
RSCR	Copy scratch reg to C
RINT	Copy interval timer to C

CARD READER

ENDWRIT	End write cycle
STWRIT	Start write cycle
ENDREAD	End read cycle
STREAD	Start read cycle
CRDWPF	Fetch card write protect flag
CRDOHF	Fetch card over head flag
CRDINF	Fetch card in flag
TSTBUF	Test card read/write buffer
SETCTF	Set card trip flag
TCLCTF	Test and clear card trip flag
CRDEXF	Fetch card reader external flag

PERIPHERAL CONTROL INSTRUCTION

?XF	0-Fh	Test external flag
WPREG	00-FFh	Write constant to selected peripheral register
RPREG	0-Fh	Read peripheral register n into C

Numbering Systems

The commonest numbering systems are shown below in an equivalent table.

Decimal base 10	Hexadecimal base 16	Binary base 2	Octal base 8
0	0	0	0
1	1	1	1
2	2	10	2
3	3	11	3
4	4	100	4
5	5	101	5
6	6	110	6
7	7	111	7
8	8	1000	10
9	9	1001	11
10	A	1010	12
11	B	1011	13
12	C	1100	14
13	D	1101	15
14	E	1110	16
15	F	1111	17
16	10	10000	20
17	11	10001	21
18	12	10010	22
19	13	10011	23
20	14	10100	24
21	15	10101	25
22	16	10110	26
23	17	10111	27
24	18	11000	30
25	19	11001	31
26	1A	11010	32
27	1B	11011	33
28	1C	11100	34
29	1D	11101	35
30	1E	11110	36
31	1F	11111	37
32	20	100000	40
33	21	100001	41
34	22	100010	42
35	23	100011	43

(Note: BCD (Binary Coded Decimal) uses the first ten values of the binary numbering system - to four digits i.e. 0000 to 1001)

CORRECTIONS

After the ZENROM was sent for manufacture, the following difficulties were found:

DECODE The DECODE function name can be entered in a program line as a global execute instruction (i.e. XEQ 'DECODE') with ZENROM not plugged-in.
However, when the program is run with ZENROM plugged-in, the value decoded to Alpha will not be correct.
You should therefore enter the XEQ instruction with ZENROM plugged-in. This sequence gives a saving of 6 bytes and executes faster for all XROM instructions.

MCED During execution of the 'GET' function.
The GET command prompt allows specification of absolute register start and end addresses, or using the [DEC] key, the start address and a decimal number of main memory or XRAM registers from which the data will be recalled.
The [←] (backarrow) key should allow deletion of decimal input, thus changing the 'd' (decimal) prompt back to an 'A' (absolute) prompt.
This sequence does not function correctly and could allow entry of an invalid address.
If you mistakenly select the DEC option, you should cancel the complete command by pressing: [SHIFT] [CMD].

LOW BATTERY ANNUNCIATOR
If the battery voltage falls below the voltage set for switching of the display annunciator 'BAT', this may not be correctly seen by the HP-41.
During RAMED, the annunciator will not be activated. If RAMED is executed in a low BAT displayed state, then the annunciator will be turned off.
During MCED, the low BAT state is sensed by the machine, but when activated, will also display the USER annunciator.

Leeds Booklet Printing Company
LS5 3AE England
0532 744022