ASSEMBLER 3 AND ITS MANUAL

In 1979 Hewlett-Packard released the world's first alphanumeric hand held programmable calculator, the HP-41c. It proved successful and popular far beyond its designers' and its manufacturer's expectations, and hundreds of thousands have been sold throughout the world since that time. The world wide community of users who belong to the PPC Club rapidly developed advanced programming techniques for the new calculator, exploiting unsuspected features of its operating system possibly not anticipated by its designers, then decoded that operating system and taught themselves to write and run their own functions and programs in its internal assembly language (machine code, machine language, microcode). Now there are several devices on the market, containing RAM memory, which allow users to write and immediately run such functions and routines for themselves. Until the development of ASSEMBLER 3 this was a slow and painful process. It first involved hand coding the instruction codes into an EPROM burner, burning an EPROM ROM image, testing it, erasing it and reburning. Programming now, assisted by ASSEMBLER 3, has become simple and straightforward, and the advanced programmer can quickly learn to write and run functions and programs which execute more than 100 times faster than before, and he may retain all of the memory of his 41c exclusively for data storage. To an owner of the already incredibly powerful calculator - or computer - and its HP peripherals, a whole new world of efficient programming is open for conquest.

ASSEMBLER 3, documented in this Manual, is a 4k, HP-41c EPROM ROM image of a set of 41c microcode functions designed to aid and speed the writing of further microcode routines, programs and functions, and to allow easy loading of user RPN programs and routines into the simulated ROM of the most recent of aids to users of that calculator, the plug-in ROM module simulators holding erasable memory. ASSEMBLER 3 may be used in any of the EPROM ROM simulating devices now on the market, though here it is described as employed with the Melbourne manufactured MLI, an adaptation of the MLDL.

With ASSEMBLER 3 functions in a short RPN application program, keying in microcode is as simple as keying in user code RPN instructions. Listings of any ROM or RAM image may be printed as fast as the printer can operate, disassembled into the now standard (in the PPC world) De Arras/Jacobs mnemonics. The user may 'single step' through any ROM or RAM image, either viewing or printing the instructions. ROM images, or any sections of them, may be downloaded to magnetic cards, transferred to cassette tape, or copied from those sources. Blocks of code may be moved to any location in a (RAM) ROM simulator, and user code programs may be built into any RAM ROM image. ROM images may be copied into the RAM of a ROM simulator at the rate of a full 4k image in 5 seconds.

With all these powerful features come also a flexible set of additional microcode functions - some well known, but most totally new. Amongst the new functions are two which, when assigned to keys, allow the keying of any sequence of synthetic or non-synthetic instructions, completely replacing the powerful "LB" (Load Bytes) program of the PPC ROM. The manual contains, besides full and detailed documentation and instructions for using all the 49 functions of ASSEMBLER 3, a complete bibliography of microcode, giving over 90 known references, compact details of XROM structure and of the 41c CPU and its microcode instruction set, and a great deal more, well known to the pioneer microcode programmers, but previously available, if at all, only in a widely scattered form. The beginning microcoder, like the tyro, should find in ASSEMBLER 3, and in its Manual, all he needs to reach the frontiers of the current State of the Art in microcode programming.

* * * * * * * * * * * * * * * * *

ASSEMBLER 3 and its Manual are available from Deep Thinking Software. Write to:
Michael Thompson (8496), 24 Canterbury Road, Camberwell, Victoria 3124, Australia.
The MLI described in this manual is available from Microbaud Developments. Write to:
39 Severn Street, Box Hill North, Victoria 3129, Australia.

Published by PPC Melbourne.

ASSEMBLER 3 Instruction Manual

ASSEMBLER 3 OPERATING INSTRUCTIONS

Prepared by Richard Collett (4523), Michael Thompson (8496),
and John McGechie (3324)

with assistance from Michael Tozer (9807)

(members of the Melbourne Chapter of the PPC Club)

CONTENTS

OPERATING INSTRUCTIONS FOR ASSEMBLER 3

First edition, April 1983

Dedicated to
George Muench,
long a Floridan friend of PPC Melbourne,
for the generousity without which ASSEMBLER 3,
and this, its Manual,
might never have been written.

Material in Appendices A, B and E has been adapted from Jake Schwartz' work in the PPC Southwest Conference Proceedings, January 1983 - there given, in the true PPC spirit, to all to share and profit from his dedication.

* * * * * * * * * * * * * *

Fully annotated listings of the HP-41c/cV microcode routine set, ASSEMBLER 3, are published by PPC Melbourne at A$20 each, plus postage. Write to

PPC Melbourne, PO Box 15, Hampton, Victoria, Australia 3188

Microcode programming, and the devices which make it possible, are in no way supported by Hewlett-Packard, who are unable to supply any information on the subject to HP-41c/cV users, and should not be contacted. Sources of information available to the user community are detailed in Appendix F of this document.

"NOMAS is an Island, Entire Unto Itself . . ."

John Donne A. Bowdler

I INTRODUCTION, ACKNOWLEDGEMENTS - AND A LITTLE HISTORY

ASSEMBLER 3

The 4k EPROM set containing the XROM image known as ASSEMBLER 3, consists of two EPROM's, a 2732 of 32k bits, and a 2716 of 16k bits, only 8k of which are in use at a time.* Together in a suitable device, they provide 4k words of memory. The 'microcode' function set which they contain was primarily designed to be used to provide an operating system for the MLI, a version of the 'Machine Language Development Laboratory' (MLDL) which was designed by Lynn Wilkins (7344), a member of the world wide PPC Club. It may also be used with the ProtoCODER, a unit designed by Nelson Crowle (7019), though with somewhat less ease.

Function access

Once its EPROM's are fitted into a suitable device, the microcode routines of ASSEMBLER 3 may be accessed from the HP-41c keyboard in exactly the same way as those in any application ROM. Apart from the few that are non-programmable, they may also be employed in routines designed by the user. Even without the few specialised functions required for interfacing with the MLI or MLDL, and with the ProtoCODER, the remainder allow a considerable extension of the inbuilt machine functions of the 41c, even of the functions available to the user owning an Extended Functions module.

ROM simulators

Seen only as such, ASSEMBLER 3 may be used with any of three other devices designed to simulate a plug-in ROM module of the 41c by a programmed EPROM set - the HHP-16K (and the HHP-32K) manufactured by F. M. Weaver & Associates, the HP-41 EPROM ROM Simulator manufactured by Dallas Development Systems, and the MC00550A, AMS (Applications Memory System), manufactured by Mountain Computers. (For details of these, see Appendix F.) These devices allow the use of, and access to functions in microcode (or in RPN user language) burnt into EPROM's, but do not allow, as do the MLI, the MLDL and the ProtoCODER, the loading of microcode or RPN routines into any inbuilt RAM memory. The MLI/MLDL can hold 4k of EPROM accessible by the 41c as if ROM, and it is for this unit that the functions of ASSEMBLER 3 have primarily been written. The ProtoCODER, which also holds 4k of RAM memory, has to be used with an interface unit which can at the same time serve to operate a ProtoTECH EPROM unit in which ASSEMBLER 3 may be employed, while the Mountain Computer AMS with an additional RAM board can hold up to 16k of EPROM's and 16k of RAM.

The MLI is marketed in Melbourne, Australia, by Microbaud Developments, under the name Machine Language Interface, since the actual circuitry employed differs from that of the original MLDL. A redesigned MLDL, the MLDL II, which requires no EPROM memory for its operation, and has 8k of RAM memory, is marketed by COMP/STOP In the descriptions which follow, references to the MLI are to be taken as references to the first MLDL also, and conversely, depending on the kind of unit with which ASSEMBLER 3 is to be used. It is very likely that it would be quite usable with the MLDL II, though it would then need installation in an independent EPROM ROM simulator. To date it has not been tried with this unit, or with the Mountain Computer AMS.

Microcode

Microcode programming, the writing of routines for a programmable calculator in an assembly language for the code of its CPU and operating system, has not been possible outside the manufacturers' development laboratories until very recently. Even now it is possible only on the HP-41c. Almost no information about the operating systems of earlier calculators than the 41c was made available by Hewlett-Packard, though their documentation of the architecture of their calculators as

* The top 8k bits of the 2716 duplicate the bottom 8k so that, using an HHP16K, or an MLI, the simulated ROM may be placed at an odd address. See the operating instructions for the HHP-16K for further information on installation.

visible to the well experienced user has generally been superb. Some progress was made towards mastering the operating system of the earlier HP-67/97 by a member of the PPC Club, Tom Napier, late in the '70's, hoping to be able to develop a data collecting interface. This did not come until several years later, and then only for the 41c.

Working from some technical information about the 41c made available to their user's group within a few months of its release in 1979, members of the PPC Club, led by Bill Wickes (3735), discovered many thousands of non-keyable 'synthetic' instructions which considerably extended the intended function set of the machine. One such type of non-standard instruction, the 'byte jumper', assignable by special methods to its keys, was found to read out from program memory to the alpha register, and was immediately used on the address space of plug in application ROM's and of the HP-41c's main operating system. Listings of the 12k words of this system were prepared, but only of the 'lower' 8 bits of the ten.

Using a specially devised interface to a microcomputer, full 10 bit listings were made by several members, including Jim De Arras (4706), studied by him and by Steve Jacobs (5358) with the aid of computer recorded operations of the calculator, and collated with all other PPC member collected and digested understanding of the 41c operation. The function of each instruction code was determined, and a set of mnemonics devised by the two for the possible 1024 individual instruction codes (2^{10}). The complete system was finally deciphered in mid-1981. The mnemonics then introduced were different from those the HP-41c system designers had used, but they are explicit and intelligible. They are used by ASSEMBLER 3, and it is recommended that they become the standard. (The task was not made easier by the fact that some instructions required two words, and often a following data word. There were 262,912 distinct pairs of codes, if one looks at it in that way.)

Late that year the first EPROM interface was designed by Jim De Arras, and marketed by Hand Held Products - the HHP-16K, with the HHP-32K. The first microcode disassemblers, in RPN, were written soon after, by Richard Collett (4523). In January 1982 the earliest user written microcode functions allowed their full automation on the HP-41c itself. The first assemblers, also in RPN, were written later by Michael Thompson (8496), and quickly turned into microcode versions. That in ASSEMBLER 3 is the descendant.

The 'RAM memory accessible as if ROM memory' devices were slower on the scene, heralded by the use by Paul Lind (6157) of an interface to a microcomputer fooling the 41c into supposing that the codes given to it by the microcomputer were codes read from a ROM. The MLDL design was published by Lynn Wilkins (7344) in March 1982, and the ProtoCODER was released shortly after. Until then almost all programming in microcode had been done by burning code into a pair of EPROM's, for running in a ROM simulator. The detection of faulty programming meant reburning the EPROM's, and testing again. With the ROM simulating RAM of the new units, that trouble was eliminated. Now, with ASSEMBLER 3, and the MLI/MLDL or the ProtoCODER, keying in, running, and printing out listings of microcode routines is as simple as doing the same for normal user RPN programming. Microcode has come of age.

The mastery we possess, which all can now acquire, making the astonishing HP-41c/cV even more flexible and powerful, has been due to many: Bill Wickes, Richard Nelson (founder of the PPC Club, and editor of the PPC Journal), Charles Close, Tom Cadwallader, Jim De Arras, Bill Kolb, Steve Jacobs, Paul Lind, Lynn Wilkins, . . . The development of this corner of the computer art has been made possible only through the enormous efforts of these and many, many others, who found out things, tried to understand, and shared unselfishly what they found.

II HP-41c/cV MICROCODE PROGRAMMING

Operating principles

The HP-41c is designed to accept plug-in memory modules of two general kinds, RAM memory, which extends the storage capacity of the calculator, and ROM memory, holding routines or functions. The HP-41c/cV can address up to 64k of words of ROM memory, located at sequential ROM addresses in its ROM memory space, from 0000 up to FFFF, and up to 1024 56 bit (7 byte) program, key assignment, timer alarm, I/O buffer registers and data registers in its RAM memory space. Only up to 942 addresses are actually in use. (16 for status, 128 in the Extended Functions module, 64 in the memory common to the 41c and 41cV, 256 in a Quad RAM module, or the extra memory of the 41cV, and 239 in each of the possible Extended Memory modules. One register in each of the Extended Memory modules and one in the Extended Functions module is used by the operating system for Extended Memory, but counted here. See the maps of the HP-41c ROM and RAM data space in Appendix A.)

Uses of the ROM address space

The operating system of the calculator contains 12k words of ROM, located in three micro chips mounted internally. The addresses of this part of ROM memory run from 0000 up to 2FFF. Nothing is located at addresses 3000 to 3FFF, except when the IL module is in use with its printer ROM disabled, when this ROM appears at the block 3 addresses, and is inaccessible to the operating system, but the plug-in Diagnostic module is set at addresses from 4000 to 4FFF. The addresses from 5000 to 5FFF are reserved for the Timer module, when plugged in, while those from 6000 to 6FFF are for the printer, whether the IL unit, or the earlier dedicated printer. HP-IL commands are located in the plug-in IL module, from 7000 to 7FFF. The remaining 8 blocks of addresses are available at the ports, and are assumed by any plug-in ROM module, depending on the port into which it is plugged. Since applications ROM's can be either 4k or 8k in size, two 4k blocks of addresses are assigned to each port. Port 1 has the addresses 8000 to 9FFF, port 2 those from A000 to BFFF, and so on, but using non-HP devices, every address is available at each port.

RPN user code access to ROM programs and functions

User RPN programs can employ functions written into the 12k operating system, or available in plug-in ROM's. The latter are accessed by what are called XROM instructions (eXternal ROM), which display the accessed function or program name in an application ROM when the ROM is plugged into the calculator, but display an instruction of the form "XROM nn,mm" when the external ROM is removed, where nn is the assigned ID number of the eXternal ROM, and mm is the number of the function in that ROM. 31 XROM ID numbers are available (see the listing in Appendix D), and there may be up to 64 functions in each 4k block of an XROM. The XROM numbers of the XROM images in the 4k (XROM simulating) RAM sections of the MLI (MLDL) and of the ProtoCODER may be set by the user, but it is important that they not conflict with those of any application module or peripheral that may be plugged in. If so, the function in a ROM or simulated ROM in the lower address will be accessed, while that in the other ROM will be ignored. Since the 4k address block at which the MLI or ProtoCODER RAM may be located can also conflict with that of such a ROM or simulated ROM module, causing a crash of the system when powered up, both kinds of address conflict must be avoided.

XROM access

When an ASSEMBLER 3 EPROM set is plugged into any one of the four devices which can simulate a ROM (by reading from an EPROM set), the address range of its 4k words is port dependent (or set by the user), but the address of the RAM in the MLI may be set at any value. (See the operating instructions for the various units.) The XROM number of ASSEMBLER 3 is 21 (a number reserved by HP for user designed application ROM's), and it contains 49 functions, whose XROM numbers thus run from 21,01 to 21,49.

When the HP-41c looks for a called function in an XROM, it searches the

function address table, in each plugged in XROM for the XROM with the number given in the RPN XROM calling instruction. It then reads the address of the start of the called function from the function address table at the start of the XROM, jumps to that location, and starts executing the program or function beginning at that point.

XROM structure

The first word at the lowest address, XØØØ (where X is usually port dependent), of a ROM or simulated ROM, gives the XROM ID number, and is followed by a word giving the number of functions in the XROM. The subsequent pairs of words each give an address, in CAT 2 order, of one of the contained functions. Each function's code sequence is preceded by the name of the function, lettered in reverse order. Each letter of the name, up to a maximum of seven, is coded by a ROM word at the location, intended to be interpreted as a character. Up to 11 letters could be used in a function name, but if functions are to be capable of being keyed into a user, RPN program, no more than seven may be employed. (A maximum of eleven can be read out to the display in a CAT 2. If more than seven are used, access to the function from a user program would be possible only by loading the particular XROM instruction by synthetic means. It is, in fact, feasible to use the XROM name in a program, provided that the ROM words after its name form a non-crashing sequence - e.g. consist of a single RTN. The corresponding XROM number, XROM nn,ØØ would be displayed by a user RPN program when the XROM, or simulated XROM is removed.)

The function address table at the start of an XROM gives the address of the execution entry point of each of the functions, the same address also identifying the start of the function name, running backwards from the same location. User code RPN programs or functions are not named in this reverse order, but start with the first byte of the program. (Data about the size of the program or routine, in this case, is given by the two words at the start of the ROM file, immediately prior to the main global label.) By writing the correct words into the RAM of an MLI or of a ProtoCODER, then, a user may construct his own simulated XROM with microcode, and/or RPN routines or programs, accessible in its 4k RAM. (See the schematic outline of XROM structure in Appendix D.)

The HP-41c/cV CPU structure and instruction set

The CPU of the HP-41c/cV interacts with its RAM and ROM memory through the microcode command set. (Appendix E.) The CPU itself contains five 56 bit, 14 digit, 7 byte registers, C, B, A, M and N, a 16 bit 'program counter' (PC) register on the top of a 4 level subroutine return stack, two further 8 bit registers, ST and FI, for 'status', or 'flag out' and 'peripheral flags', or 'flag in'. There are two further 8 bit registers, T for 'tone' and KY for 'key', two 4 bit registers, P and Q which are used to 'point' to locations in A, B and C. There is a single one bit 'register' holding the 'condition' or 'carry' flag, used for conditional tests.

Register C is the most important of this set to understand, since it is through C that RAM and ROM code is read as data for subsequent processing, and through C that the results of that processing are sent out to the display, card reader, printer, IL Loop, etc., and to RAM memory. (The diagram of these CPU registers in Appendix B should be studied in conjunction with the diagram of the 41c memory space in Appendix A, and the general description of the CPU command set in Appendix E.)

Getting started

The beginning microcode programmer should examine the XROM structure table in Appendix D, write a short function address table, and practise by writing a function name and a simple routine. Annotated listings of the 41c operating system and of HP ROM's and ASSEMBLER 3 are available for study. (The owner of ASSEMBLER 3 may make his own listings. See the descriptions of DISASM in III and of the application programs in IV.)

After mastering simple routine writing, the literature may be consulted for advanced programming information. All known references at the date of publication are given in Appendix F. Beyond that, it should be possible to begin microcode programming using only the information available in this Manual. Every effort has been made to make it complete in itself, though a detailed working knowledge of the HP-41c is assumed from the start. (See Appendix F for suggested background sources.)

III THE FUNCTION SET OF ASSEMBLER 3

Functions available in ASSEMBLER 3

The following descriptions of the functions available in ASSEMBLER 3 should be read in conjunction with the general description of the microcode set of the HP-41c, written by Steve Jacobs (5358), printed in PPC Technical Notes #9, pp.81-90, or the brief summary in Appendix E, and the reference material listed in Appendix F. The mnemonics used for the microcode instructions are those devised by Steve Jacobs and Jim De Arras (4706) in 1981. The disassembler will print or display the full mnemonics, though contracted forms, avoiding the unkeyable characters of the 41c, are used by the assembler. Full details of the contracted mnemonics used are given in the instructions for ASSEM below.

The names of the ASSEMBLER 3 functions will appear in the display, or be printed, when ASSEMBLER 3 is inserted in an MLI, or in one of the other five kinds of ROM simulating units, and CAT 2, or the user language, RPN routine, "XCAT2A" is executed. Their descriptions are given in the order of their appearance in CAT 2. (For applications programs, written using ASSEMBLER 3, see IV, and the MLI Manual.)

The descriptions which follow give the name of each function, its XROM number, immediately below, then below that, its execution entry point in the simulated XROM. The last three digits only are given, since the first digit will depend on the assigned 4k block at which ASSEMBLER 3 is located, and on the device in which ASSEMBLER 3 is used. It is important to remember, when using ASSEMBLER 3 functions for writing microcode routines into an MLI or the equivalent, not to use 21 as the ID number of the MLI XROM RAM image, especially if it is set at a lower address than ASSEMBLER 3, for those new functions will be accessed in preference to those with the same XROM numbers in ASSEMBLER 3. (See Appendix D.)

* * * * * * * * * * * * * * *

ASSEMBLER 3 This is not, strictly speaking, a function in the set at all, though
XROM 21,00 it may be entered as a synthetic two byte NOP in a user program. It
08B will have no effect at all, its 'routine' consisting only of a single
RTN. The name of any ROM may be so used if its code is such. If not, a crash is the almost invariable result.

AND
XROM 21,01
837

Replaces the value of each of the 56 bits in X by the logical product of the values of the 56 pairs of corresponding bits in X and Y. This is carried out for every bit of the 56 bit pairs in the registers. After the operation of this function, each given bit in X is set if both bits of the corresponding pair were set before execution, and is otherwise clear. The original value of X overwrites Last X, and Z and T are unchanged.

Suppose that X contains (right justified, as always, and with the leading zeroes suppressed) the binary number:

	101 1011 0110,	$(=5B6_{16})$
and Y contains:	011 1000 1101.	$(=38D_{16})$
The result in X will be:	001 1000 0100	$(=184_{16})$

OR
XROM 21,02
83F

The operation of this function is very similar to that of AND, forming in X the disjunction of the bit pairs of X and Y, overwriting the contents of X. Each bit in X is set on exit from the routine if either or both of the corresponding bits in X and Y were set on entry. If set in either, it will be set in X on exit. If clear in both, it is clear on exit. The original X overwrites Last X. The stack is not lifted, and Y, Z and T are unchanged.

If X contains:	11 1100 1101 1001	$(=3CD9_{16})$
on entry, and Y contains:	10 0101 1100 1011,	$(=25CB_{16})$
then afterwards X will be:	11 1101 1101 101:	$(=3DDB_{16})$

APPFN
XROM 21,03
3A3

As directed by the rightmost four digits of X, APPFN appends to alpha the address and name of a function in an XROM. X must contain the address, in the Function Address Table, of the address of the function. After execution, X will contain the address of the address of the next function in the XROM. Y, Z, T and Last X are unchanged. The routine below will print out the addresses and names of the functions in an XROM at the 4k block XØØØ if the hex digit X is entered in response to the prompt. (This routine is for illustration only, and will crash the 41c if there is no XROM at the address, or crash sometimes after listing the contained functions. See the application routine, "XCAT2A" in IV.) If the block happens to be empty, the calculator will lock up for at least 45 seconds, frantically searching for a bit 7 of a ten bit word which is set, and finding none. If this occurs, pull out ASSEMBLER 3 from the port. . .

```
-----------------------------------
Ø1 LBL Ø1          Ø7 CODE
Ø2 "XROM ADDR?"    Ø8 LBL Ø2
Ø3 AON             Ø9 CLA
Ø4 PROMPT          1Ø APPFN
Ø5 AOFF            11 PRA
Ø6 "⊢ØØ2"          12 GTO Ø2
-----------------------------------
```

The Function Address Table of an XROM.

The first word of an XROM, at address XØØØØ, gives the XROM ID number. The second word gives the number of functions in the XROM, including the name of the XROM - XROM xx,ØØ. In pairs of words, from the third word on, there follow the addresses of the first words of the functions it contains. If the address of a microcode function is Xabc, the two words will take the form: ØØa, Øbc, where the 244 format for representing the ten bits is used (presupposed by ASSEMBLER 3), and a, b and c are hex digits. When the function is in user (RPN) code, the address takes the form 2Øa, Øbc. The second digit of the first word determines how many prompts. The order in which these addresses of the XROM functions occur is that given by a CAT 2. After the last address of a function, there should be two nulls: ØØØ. APPFN uses this structure to report function names and addresses. The name of a microcode function is given by the words immediately preceding this entry address, at the rate of one letter per word, the letters of the name being given in reverse order. The name of a user routine in an XROM is given by the global label, coded into the words starting with the entry address. The two words which precede that global label give data about the length of the routine for down loading into RAM, whether the routine is private, etc. APPFN will also recognise and appropriately decipher the names of these user routine functions.

If X contains, say, 6ØØ2 (i.e. ØØ ØØ ØØ ØØ ØØ 6Ø Ø2), the FAT address of the address of the printer ROM name, (the first printer 'function', in the above described sense), then the sequence "625B -PRINTER-" will be appended to whatever is in alpha. If X contains AØØC (in its rightmost digits), and the Extended Functions module is in port 2, "A58Ø ARCLREC", will be appended to alpha - the function ARCLREC is the 5th function in the ROM, and its actual executing code starts at ROM address X58Ø, where X is port dependent. (X takes values from 8 to F for the four ports, the odd values for the upper 4k, the even for the lower. The printer and IL functions will give values of 6 and 7 respectively.)

* * * * * * * * * * * * * * *

ASSEM
XROM 21,04
1C3

Along with DISASM, this is one of the most complex functions in ASSEMBLER 3. It allows the user to key microcode instructions (and data words) of all needed types into the RAM memory of the MLDL/MLI, or of the ProtoCODER. Each execution of ASSEM effects the assembly of a whole instruction in microcode, even those consisting of more than one word.

The following is not a description of the full set of microcode instructions, only a description of the manner of their assembly and loading into an external RAM memory designed to simulate external ROM. (For a more complete description of the microcode instruction set, see Appendix E and the starred references in Appendix F.)

The operation of ASSEM

The mnemonic of the instruction which is to be assembled and loaded (into the MLI/MLDL or ProtoCODER) is entered into the alpha register. For some instructions, needed parameters may be keyed either into alpha as mnemonic or as hex digits (depending on the instruction), following the instruction mnemonic, or given, in decimal form, in X. When ASSEM is executed, the required code for the whole instruction (1 to 3 words) will be written into the RAM of an MLI, and also into the stack - for use with the Protocoder. $R_{ØØ}$ then contains the next address in the XROM image RAM space, X contains the current address, right justified, followed by the ten bit word stored there. If the instruction loaded into an MLI RAM was two or three words long, Y and Z, respectively, will contain those preceding words and their storing addresses in RAM. Alpha retains the mnemonic, whose first six characters are in T.

The ASSEM control register.

The hexadecimal address at which ASSEM is to write to the MLI has to be stored in the rightmost four digits of $R_{ØØ}$. After ASSEM writes the code for the instruction to the MLI, it increments $R_{ØØ}$ to point to the MLI RAM location for the next instruction, and returns control to the user, or to the RPN program from which it has been called.

The assembler program, "ASS", below in IV, also reads out and displays the current instruction at the addressed location. The user then may either go ahead and overwrite it by keying in a different instruction, or press R/S, to see the following instruction. "ASS", so seen, behaves in almost the same way as the 41c operating system when keying in user RPN code, except that [R/S] replaces [SST]. A companion "BST" can readily be written. N.b. The entry of an instruction overwrites any existing instruction at the address, it is not inserted. In programming this is like the older HP machines - e.g. the HP-25.

Using ASSEM for instruction entry.

The use of ASSEM, and its handling of the stack contents vary with the class of microcode instructions that are being entered. The following account deals with each class in turn.

There are four classes of microcode instructions, known as classes Ø to 3, where the class is determined by the last two bits of the instruction word. These four classes will be treated separately. Any attempt to enter an instruction mnemonic which ASSEM does not recognise will result in the error message "NOT FOUND" being displayed, if flag 25 is clear, and flag 25 will be cleared, if set.

(1) Keying class Ø instructions.

There are four types of Class Ø instructions, distinguished here by their parameter types, 'f', 'd', 'r' and 'miscellaneous', as so identified in Table Ø of the Class Ø instructions on p.84 of Steve Jacobs' article, and in Appendix E.

(a) Instructions with type 'f' parameters.

Enter the instruction into alpha, followed by a space, and then the parameter. The parameter can be entered in hex in alpha, or in decimal in X, depending on whether or not the calculator is in alpha mode when ASSEM is executed. "RCR C" keyed into alpha has the same effect as keying "RCR " into alpha, and 12 into the X register. Any instructions which require a parameter, must have a space immediately following the instruction, even when the parameter is keyed into the X register. When using "ASS" (see IV), use R/S to enter the instruction, or bypass the existing instruction if no entry is made.

(b) Instructions with type 'd' parameters.

The only instruction with a type 'd' parameter is "LD@R". Parameters for this instruction differ from those of type 'f' in that they must be keyed into the alpha register as hex numbers. Since the actual character sequence "LD@R" cannot be keyed into Alpha ("@" is not keyable), the instruction is keyed in as "LD " (note the space), followed by the parameter.

(c) Instructions with type 'r' parameters.

These instructions are the 'READ' and 'WRIT' instructions for accessing the RAM of the 41c from a microcode routine. Since "⊢" ('append') cannot be keyed into

alpha from the keyboard, the character "+" is used in its place. The register post-fixes may be entered into alpha, or entered into the X register in the way already described, as numbers from Ø to 15. Table I gives the alpha and X register equivalents for keying type 'r' parameters:

TABLE I: Type 'r' parameters

X	Alpha	X	Alpha
0	T	8	P
1	Z	9	Q
2	Y	10	⊢ (+)
3	X	11	a
4	L	12	b
5	M	13	c
6	N	14	d
7	O	15	e

Error messages.

If an illegal register postfix is entered into alpha, "ILLEG PARAM" will be displayed.

(d) Miscellaneous instructions.

Though these instructions do in fact have parameters, their mnemonics need only be keyed in up to the point where they become distinct from those of any other of the miscellaneous instructions. The following table should be used as a guide. The shortened form of the instruction on the left is essential, the portion to the right is optional. Thus the instruction "SETHEX" could be entered by "SETH" alone, or followed by any characters at all - it could even be entered as "SETHARRY". The additional characters will be ignored. (If compulsive, key in "SETHEX" if you wish.)

TABLE II: Class Ø miscellaneous instruction short form mnemonics

Full form	Minimum form	:	Full form	Minimum form
G=C @R;+	G=	:	DSPTOG	DSPT
C=G @R;+	C=G	:	?C RTN	?C
C<>G @R;+	C<>G	:	?NC RTN	?N
M=C ALL	M=	:	RTN	RT
C=M ALL	C=M	:	N=C ALL	N=
C<>M ALL	C<>M	:	C=N ALL	C=N
T=ST	T	:	C<>N ALL	C<>N
ST=T	ST=T	:	LDI S&X	LDI
ST<>T	ST<	:	PUSH ADR	PU
ST=C XP	ST=C	:	POP ADR	POP
C=ST XP	C=S	:	GOTO KY	GOTO K
C<>ST XP	C<>S	:	RAM SLCT	RA
XQ-GO	X	:	WRITE DATA	WRI
POWOFF	POW	:	FETCH S&X	F
SLCT P	SLCT P	:	C=C OR A	C=C O
SLCT Q	SLCT Q	:	C=C AND A	C=C A
?P=Q	?P	:	PRPH SLCT	PR
?LOWBAT	?L	:	ST=Ø	ST=Ø
A=B=C=Ø	A	:	CLRKEY	CL
GOTO ADR	GOTO A	:	?KEY	?K
C=KEY KY	C=K	:	R=R-1	R=R-
SETHEX	SETH	:	R=R+1	R=R+
SETDEC	SETD	:	WROM	WRO
DSPOFF	DSPO	:	NOP	NO

(2) Class 1 Instructions.

These are the 'absolute' XQ and GO instructions. They are keyed into alpha as

follows. (Note the space following the mnemonic):

a) "?NCXQ "
b) "?CXQ "
c) "?NCGO "
d) "?CGO "

The space must be followed by the desired calling address, which has to be given in hexadecimal digits, separated by the space from the instruction mnemonic, e.g. as "?NCGO 2AF9". It is not possible, this time, to use X for the called address in decimal form.

As coded in ROM, all class 1 instructions are two words long. Although ASSEM automatically writes the codes into the MLI, it will also write the first word (in the form aaaawww) into the X register and the second word into the Y register, both right justified, and set user flag Ø6. (These features may be useful if the codes are to be printed while a routine is being assembled, and would be essential when using a ProtoCODER. They may be used in a program, to mimic RPN program entry with a printer set to either TRACE or NORM.)

(3) Port dependent (relocatable) XQ's and GO's.

Since an MLDL device can be set to any 4k page of ROM memory in the 64k ROM memory space of the 41c, the normal Class 1 instructions are useless for calling subroutines in the same 4k block of ROM address space. A call to the address A5E3 may operate perfectly when the block used is that from AØØØ to AFFF, but it could land in empty addresses, or in another ROM altogether, if the same A block is not being used. HP ROM's with fixed address blocks do not suffer from this disadvantage, and it is not necessary, of course, for the mainframe ROM itself. There are routines in the mainframe which have been provided to overcome this problem, though there are a number of drawbacks in using them:

a) Port dependent jump instructions are three words in length.

b) The CPU C register contents are lost, and C may not be used to carry data to the called subroutine. (Use another CPU register, remembering that the subroutine must be written with this in mind.)

c) The CPU must be in HEX mode prior to the execution of the port dependent jump. (There has to be a "SETHEX" instruction prior to the call, with no intervening "SETDEC", though this is not necessary if the CPU already happens to be in HEX mode.)

d) Port dependent jumps cannot be used conditionally in the usual direct manner, though they can be bypassed by a relative jump from higher up in the program. If an attempt is made to use them after conditionals, the second and third words of the port dependent XQ, or GO will be interpreted as ordinary, single word, possibly crash inducing instructions.

Port dependent jumps are entered into the alpha register as "XQ ", or "GO ", followed by the called address in hexadecimal digits. (Note the space following the prefix mnemonic.) The carry flag must always be clear. The address called can be either three or four digits long, as keyed in, since the first digit is, and must be ignored. E.g. "XQ 8496" inserts exactly the same words as "XQ 496". When set by the execution of an instruction, the carry flag will remain set only for the execution of the next word. Whatever that may be, it will be cleared (unless reset by that next instruction.) A test for a condition, then, must immediately follow the command whose outcome is to be tested. Their conditional use requires a subterfuge:

```
?A/C S&X
JC+Ø4
PORT DEP:
XQ
XA3D
...
```

Other addresses recognised by ASSEM.

Port dependent XQ's and GO's call fixed main frame addresses, but there are other such addresses recognised by ASSEM and DISASM. Ø7EP and 77B3 put a message in display. The message should follow the SBR call, and the first digit of the last character of the message should start with a 2. The message runs forward in memory,

not in reverse like the function names. 1C6C is used for putting mainframe error messages into display, using a table in the mainframe ROM 1. Which message is to be displayed is determined by a data word following such a SBR call. Calls to 22F5 will treat the message to be displayed as an error message. This routine tests flag 25, and will cause an error halt. (Other routines in the IL module are similarly recognised.)

(4) Class 2 instructions.

The mnemonics for class 2 instructions also consist of a prefix and a parameter, specifying the field of the CPU register or registers on which the designated operation is to take place They are entered by keying in the prefix mnemonic, a space, and then the field parameter. (Decimal parameter coding in X is not available.) E.g. "A<>C ALL", "LSHFA R", "A=A-B SX". Some field specifications contain unkeyable characters, and these may simply be omitted when using ASSEM. They will, however, be printed, or seen in alpha, when the resulting code is disassembled by DISASM:

TABLE III: Class 2 field parameters

Full mnemonic	Keyed mnemonic
@R	R
S&X	SX
R<	R<
ALL	ALL
P-Q	P-Q
XS	XS
M	M
MS	MS

Error messages.

If an illegal field parameter is keyed in, then the error message "ILLEG PARAM" will be displayed, and behave as normal error messages do. Use the 'back arrow' key to clear, rekey, press R/S.

(5) Class 3 instructions.

These are the relative jump instructions. The jump distance can be given in hex digits in alpha, or in decimal form in X, in the familiar alternative ways. The only difference is that there is no need to enter a space in alpha between the instruction and the jump distance, though doing so will cause no harm.

For example, to jump back 63_{10} words when 'carry' is set, key in either "JC-3F" or "JC", with -63 in the X register. The jump distance, if keyed into alpha, must be given using no more and no less than two digits, even when the distance is less than 16 words. For example, to jump backwards five words when carry is not set, key in "JNC-05" to alpha, or "JNC" to alpha, and -5 to X. The "-" character must always be the third character from the right if the jump is negative, and the distance is in alpha. A '+' sign need not be keyed in for positive jumps - it is assumed by the routine. It is not necessary to key in the "C" in "JNC", since ASSEM tests the second character from the left to see if it is a "C", assuming that it is a "JNC" otherwise. For example, a jump forward of 2E (hex), when carry is not set, could be keyed in as either "J2E", or as "JNC2E". (But why not economise on key strokes?)

Error messages.

If the distance is greater than $+63_{10}$, or less than -64_{10}, the message "TOO FAR" will be displayed. 'Back arrow', and re-enter the instruction.

(6) Keying in data words.

This is necessary for the coding and entry of the function address table (the FAT), and where data is to be read into C by the preceding instruction (usually "LDI S&X"). To write in these words, precede the data string (in alpha) by a colon - e.g. to key in 3E0 (hex), place ":3E0" into alpha. Such data must always be in hexadecimal characters, and is translated as such. Instructions could generally be entered in this way, rather than by the use of mnemonics, and it may be simpler to do so when keying in long routines. If the function HEXKB is used it is easy to check correct entry from the display before pushing R/S. (See "ASSH" in IV.)

(7) Keying in function names.

This feature can only be used with MLI's (MLDL's), and not with the ProtoCODER. Function names are usually tedious to key in, since the code for each character in the name has to be worked out, and then entered in reverse order. ASSEM will take care of all this work. To key in a function name, precede the name, in alpha, by a '$' character. E.g. if the function is to be called "XYZ", then key "$XYZ" into alpha. No more than 6 characters may be keyed in in this way. If a seven character name is required, the last six characters (which come first in the numerical order of the listing), omitting the first character of the name, can be keyed in using the above method, but the first character of the name must be keyed in as data (as described in (6) above).

An example should make this clear. If the function name "COPYROM" is to be keyed in, with its last letter, "M", at address C88D, and its first letter, "C", at C893, the following steps should be taken. We suppose here that the user code routine "ASS" (see IV) is being employed, and that the addresses from and including C88D are so far 'empty, containing only nulls, 000.

(1) See the address prompt: "C88D 000 ?"
(2) Key in "$OPYROM" and R/S.
(3) See the address prompt: "C893 000 ?".
(4) Key in ":003" (for the letter "C" as a 'ROM' character), R/S.
(5) Using UPDFAT (in IV), or manual means, add the new function address to the FAT.

The sequence of instructions entered, when disassembled will be:

C88D 08D "M"
C88E 00F "O"
C88F 012 "R"
C890 019 "Y"
C891 01P "P"
C892 00F "O"
C893 003 "C"

DISASM
XROM 21,05
2CF

This routine will read out, one at a time (one per execution), the words of microcode routines between any two user specified ROM (or simulated ROM) addresses, print those words (in hex digits, in 244 bit format), and disassemble the words, i.e. provide, at user option, the full De Arras/Jacobs mnemonic, if an instruction, or the data item, if data. The result will be printed, if a printer is connected, or viewed if not.

DISASM can be employed in either of two ways:

a) By running the User-code RPN routine "DIS". (See section IV below.)

b) By assigning the function (DISASM) to a key, and pressing it once for each ROM word (or ROM instruction).

The second is much more useful when disassembling without a printer, since there is no scrolling in the alpha register. The address is only momentarily displayed, immediately followed by the instruction. (The address display period depends on the response time of the particular calculator.)

The DISASM control register.

DISASM consults and adjusts the contents of user register R00. The format of its contents is

N FF FF EE EE CC CC

for the 14 digits, as an alpha string with sign digit of 1, where CCCC is the address of the instruction code that is about to be disassembled (it would be the starting address for a run), EEEE is the address at which DISASM is to stop, FFFF is the address at which the last character of the next function name following CCCC, starts, and N is the number of characters, less one, in the next function name.

When using the RPN routine "DIS", FFFF & N are taken care of by NEXTFN (NEXT Function Name) at line 19. (See p.23 below.) The user merely has to fill in two prompts for the start and end addresses.

If NEXTFN is not executed before using DISASM manually (after EEEECCCC has been stored into R00), then function names will not be printed, but will instead be

misleadingly disassembled as normal machine code instructions. To make the most of DISASM, then, place the starting and finishing addresses at the right of R00, and execute NEXTFN. This will determine the values for N and FFFF, and insert them in R00. If operating manually, HEXKB is useful for manufacturing the R00 contents, or CODE could be used. Use X>$ to make the result an alpha string.

The DISASM flag controls.

By employing flags F00 to F05, the user also has control over whether or not routines are disassembled as instructions, or as data. If flag F00 is clear, then the next word is disassembled as an instruction. If it is set, the next word is processed as data. Data can be interpreted in several different ways, according to the settings of flags F01 to F05. The priority given to these flags is as follows:

DISASM control flag priority.

Flag 00: The master control flag. When this is clear, the next word is disassembled as an instruction. When it is set, indicating that the next word is a data item, the settings of the following flags determine the nature of the resulting interpretation. (If 00 is clear, the settings of the flags lower in this list are ignored.)

Flag 03: When set, the next word only is processed as data, and the corresponding character which is displayed or printed is dependent on the settings of flags 01 and 02.

Flag 04: Used for displaying or printing error messages in ROM's. When DISASM encounters any one of the several XQ calls to error message display routines (e.g. to the mainframe 07EF), this flag, and flag 00, are set. The message is then interpreted as a sequence of alpha items. Once the error message has been printed, flags 00 and 04 are cleared.

Flag 05: Used for printing function names. Flag 05 and 00 are set by DISASM, after finding, on consulting the contents of register R00, that the next word is the last character of the next function name (though the first encountered). Once the function name is printed (the name length is given, by the value of N, in R00), flags 00 and 05 are cleared, and NEXTFN is automatically called by DISASM.

Flag 02: When set, everything is processed as normal ASCII data, printing out the code and the appropriate character. (Apart from alpha using instructions in user code programs, the codes for characters in ROM are different from the normal ASCII values.) It may be convenient to leave this flag set, since a listing will then show the ASCII character placed in C after a LDI S&X instruction. (It may be in preparation for loading into the alpha register.) The same thing applies in the case of flag 01.

Flag 01: When this flag is set, everything is printed according to the 41c ROM character codes - where A = 1, B = 2, etc. (According to what has come to be known as the ROM character table. See PPCCJV8N4P10, or PPCTN86, p.57. The table is reproduced in Appendix E.)

Flags 01 to 05 all clear: The word is printed out without any mnemonic or character at all - provided that flag 00 is set.

A>X
XROM 21,06
7D3

Forms a matched pair with the next function. Returns the decimal code (the ASCII code) of the rightmost character in alpha to X, lifting the stack. (This should not be confused with the function ATOX of the X-Functions module, which reads from the leftmost character in alpha, deleting it, while placing its ASCII character number in X and lifting the stack.)

X>A
XROM 21,07
7E1

Appends to alpha the character whose ASCII code is given by the decimal number in X. The stack is undisturbed. (The behaviour is exactly the same as for the X Functions XTOA.)

BCD>BIN
XROM 21,08
13F

This takes a decimal number (ranging in value from 0 to 9999) in X, and replaces it by its hexadecimal counterpart in X, with the digits of the result right justified.

Older versions of BCD>BIN gave the message "NONEXISTENT" for X values in excess of 999. Those usually called up the mainframe routine at 02E3, which, like this function, takes a floating point number, but from the CPU register C. Since this version of the routine takes the modulus of the number in X with respect to 4096, the result cannot be greater than FFF. Values greater than 9999 will give an "OUT OF RANGE" error message. The original value of X overwrites Last X.

BIN>BCD
XROM 21,09
114

This is the inverse of the preceding function. The last three digits (S&X) of X are read as a hexadecimal number, the decimal equivalent overwrites X, whose original value overwrites, in its turn, the contents of Last X.

CF55
XROM 21,10
7C1

Clears flag 55. If a printer is present and is plugged in, flag 55 will normally be set. Under those conditions it will again be set by the operating system as soon as program execution halts - if the old printer is being used. The IL printer will allow flag 55 to remain clear. (Clearing flag 55 speeds up the execution of RPN user code when a printer is connected.)

SF55
XROM 21,11
7CB

Sets flag 55. This will not be cleared when program running halts even though a printer is not plugged in. It allows a running program to reset flag 55 after a stretch over which it has been cleared to speed operation.

CLROM
XROM 21,12
8B3

Clears all 4k RAM memory of the MLDL, if only one is plugged in, and only that of the MLDL/MLI at the lowest address if more than one is plugged in. In order to avoid accidental clearing, it returns the message "DATA ERROR" unless the letters "OK" are found in alpha. COPYROM has no such safeguard. This routine in fact calls up COPYROM, and copies from the empty addresses 4000to 4FFF to the MLDL. Execution time for both is of the order of 5 seconds. If CLROM is downloaded into the RAM of an MLI, and then executed from there, the calculator will crash. (Pull out the MLI to clear.)

CODE
XROM 21,13
0B4

With the following complementary function, this is one of the oldest of special functions written by PPC members. The first versions of CODE and DECODE were written, in synthetics, by Bill Wickes, their inventor, in 1979. The first microcode versions were written by Jim De Arras, and were in the first PPC member written functions in JIMROM 1B.

CODE takes a string of hexadecimal characters from the alpha register, and places the hexadecimal digits corresponding to those characters in the right of X. Thus if alpha contains the character sequence "23FA16D872450B", CODE will replace the contents of X by the 'non-normalised number'

23 FA 16 D8 72 45 0B

Where the number of hex digits in alpha is less than 14, the digits from alpha will be coded into X right-justified, with zeroes to the left. The stack is lifted, if enabled on routine entry. Earlier microcode versions of CODE overwrote the contents of X, and the difference here should be noted.

DECODE
XROM 21,14
0DE

As its name suggests, the inverse of CODE. The contents of the X register are translated into their 14 hexadecimal counterparts in alpha. Though CODE is indifferent whether leading zeroes are present in alpha or not, DECODE will always leave zeroes in alpha corresponding to leading zeroes in X.

COMPILE
XROM 21,15
A62

Packs memory of the 41c, then compiles all GTO's and XEQ's in the program (in 41c RAM) at which the 41c is currently positioned. The display shows the message 'PACKING', followed by the message 'COMPILING'. When placed in an RPN program, it will compile the contents of the file in which it is located.

COPYROM
XROM 21,16
694

Copies any 4k, or 4k section of a ROM into the RAM of an MLDL/MLI. The page address of the ROM that is to be copied has to be in the X register, in decimal form (Ø to 15), on entry to the routine. This allows the contents of a ROM, or of an EPROM set, to be loaded into MLI RAM, provided that the function is available from somewhere. This is a function which should, if only for this reason, be in all EPROM sets. Though 4k ROM words have to be copied, this function runs very quickly, taking only a few seconds to complete its operation. There can be minor avoidable trouble in store for the unwary: if COPYROM is downloaded into the RAM of an MLI, and then executed from there, the calculator will crash when its executing code is overwritten. This can be avoided if it is called from XROM-simulated ROM, or from an address in simulated ROM lower than that to which the MLI RAM is set. (Clear, as with CLROM, by pulling the MLI plug.)

CVIEW
XROM 21,17
4B3

If a printer is present, this routine will print the contents of the alpha register, but if there is no printer, it will instead display those contents. If a printer is present, it will only print the contents of the alpha register, and they will NOT be displayed. (The function consults the setting of the printer existence flag, flag 55.)

VIEWA
XROM 21,18
105

Places the alpha register into the display, and does ONLY that. This function is an alternative to the familiar AVIEW, which prints when a printer is present, and flag 21 is set. VIEWA ignores the settings of both flag 21 and flag 55.

DISS
XROM 21,19
4BF

Given a ROM address in hexadecimal digits, right justified in RØØ, DISS finds the word located at that address, decodes it, and places the result in ALPHA. The incremented address is returned to RØØ. The format in alpha is "AAAA WWW", where AAAA is the address, and WWW is the word at AAAA, in 244 format. (This is very like the old X<>ROM, as used in a short routine to format the word at the address, with the address, in a usable form.)
To illustrate:

(1) Execute HEXKB.
(2) Key in ØØØ1, R/S.
(3) STO ØØ.
(4) Execute DISS.
(5) See in the display "ØØØ1 ØØ6 ". This is the address, and the code, of the second word of ROM Ø.
(6) Execute DISS again, to see in the display "ØØØ2 2B5 ", the third word of ROM Ø.

GETPC
XROM 21,20
B5B

Recalls the current program pointer, and that only (the subroutine returns are left behind) from status register b, placing it in X, but in MM format. (This is needed for use with RCLBYTE, STOBYTE and INSBYTE.) MM format is the same as register b format, except that the byte number (only) is doubled when the program pointer is in RAM. When pointing to ROM, there is no difference. (The resultant format, but of the pointer only, is the same as in register b.) This does not place the subroutine returns into X, only the pointer, and thus allows, with the use of the companion, PUTPC, STO b type jumps, without the disadvantages which those methods have. As currently implemented, the execution of GETPC retrieves more than just the program pointer in MM format: it collects also junk, apparently from the c register. Suppose that the contents of c are 2Ø 1Ø Ø1 69 1F 61 FØ. GETPC, executed at the top of the top file in program memory will give Ø1 69 1F 4Ø Ø1 Ø1 F6. Only the last five digits are needed, and used.

PUTPC
XROM 21,21
C74

The inverse of GETPC. The program counter (PC), in 'MM' format in the right of X, is transformed into register b format, and replaces the program pointer in b. The subroutine stack is not disturbed at all. The pointer in this format may even be stored as an alpha string with the use of X>$, setting the sign digit of X to 1. The beauty of the use of these two is that even if GETPC is executed in one routine, PUTPC may replace the pointer, causing a jump back to that same location, but with a subroutine stack preserved.

HEXKB
XROM 21,22
ADF

Temporarily halts a running program, and redefines the keyboard for hexadecimal digit entry to both alpha and X, lifting the stack. Apart from R/S and the 'back arrow', the only responsive keys are the ten digit keys (Ø to 9), and alpha A to F. Program execution is resumed by R/S, after entry. Pressing 'Shift, R/S' will exit, and halt program execution. On exit, X contains the CODE'd contents of alpha. (The routine actually calls up CODE as a subroutine.) The calculator goes to STANDBY mode between key strokes, and will turn off after 10 minutes in the normal way.

It is probably best to think of this primarily as a function which enables hexadecimal entry to X, the digits flowing in, as keyed, from the right, with alpha containing, on exit, the DECODE'd contents of X. When assigned to a key, and executed in USER mode, an R/S terminates digit entry, both to X and to alpha. The previous contents of alpha are still displayed during entry, the newly keyed digits appearing to their right. As when in alpha mode, or numeric mode, the 'back arrow', key allows deletion of a last entered digit or character, so here that key 'deletes the last entered digit AND character. R/S and shifted R/S act now in identical ways - but with a single exception: if no entries are made after the function is called up, R/S (or shifted R/S) exit, without starting program running at all, lift the stack, placing zero in X, and clear the contents of alpha. The display of the previous contents of alpha is no more than that, a display, though unlike the usual AVIEW display, the (merely) displayed characters there may be deleted, from the right, one by one. The first description of operation is no doubt true of the actual microcode operation of the function, but the second allows HEXKB to be seen as belonging with its real companions: X+Y, Y-X, SXL, SXR, and the rest. They need, but do not have here, a few other arithmetical functions without normalisation. (Who ever thought we would be able, one day, simply to enter non-normalised numbers from the keyboard in such a simple manner!)

INSBYTE
XROM 21,23
B70

This will insert a byte, as specified by its decimal number in X, into a location determined by the contents of Y. Like the mainframe controlled, program instruction entry, it opens a space of seven bytes when the target location is not occupied by a null, and then loads the desired byte in the first byte of the seven cleared. If the target location is null, the byte is loaded without first opening up a space of seven nulls. It thus carries out all of the necessary housekeeping effected by normal program function entry.

Y has to contain the program pointer, in MM format (see above, under GETPC), as positioned at the byte preceding the instruction which would be viewed in program mode.

To use INSBYTE, SST to the instruction BEFORE which the byte is to be loaded, switch to run mode and execute GETPC. Key the decimal byte number into X (lifting the MM format pointer, obtained by GETPC, into Y), and execute INSBYTE. The desired byte will be loaded into memory before the previously viewed program instruction. Since the stack is dropped by INSBYTE, and the MM pointer, previously in Y, but now in X has been incremented to point to the next byte, the operation may be repeated, without continually needing to use GETPC. The previous contents of X are to be found in Last X.

With GETPC and INSBYTE assigned to keys, no Load Bytes program at all is needed to do what that famous program did. Though one is usually at the point in program where the bytes are being inserted, it is entirely possible to be at a different position while carrying out the insertions. Since the bytes loaded into program

memory are placed before a viewed location, counterpart instructions loaded will have a constant line number. If GETPC is executed in an RPN program, just before a jump to a different program file, the latter may write program code back into the original file - exactly in the Load Bytes manner, to follow the GETPC instruction which obtained the (now) addressed location. The safety of this lies in the insertion of bytes, rather than the overwriting which every Load Bytes program with all of its predecessors has done. The called file, into which the code is to be written, could be as simple as this: LBL "ABC", GETPC, END. The written code, in this case, is simply inserted between the GETPC and the END. Load Bytes, once a "lumbering elephant of a program", reduced to a pair of key assignments!

Suppose that the instruction currently viewed in program mode is 25 STO M, and a RCL N is wanted in the preceding line. Switch to RUN, press GETPC, key in 144 (RCL), INSBYTE, 118 (N), INSBYTE. Switch to PRGM, see 25 RCL N, SST, see 26 STO M. This is very useful for writing non-standard text lines. When executed from a running program, it is best to locate the target file lower in CAT 1 order than the writing file - to avoid repetition of loading after every seven bytes. (Due to the opening of a window of seven nulls.)

RCLBYTE
XROM 21,24
B45

Recalls any byte from RAM. Like INSBYTE, this uses the address of the wanted byte in MM format, usually as obtained by GETPC, with the program pointer positioned as described above for INSBYTE. The (recalled) decimal byte number is returned to X, lifting the MM format pointer to Y, while the incremented address pointer overwrites Y. Z, T and Last X are untouched.

STOBYTE
XROM 21,25
B67

Stores a given byte in RAM. Format: decimal byte number in X, with the address at which the byte is to be loaded, in GETPC format, in Y. The behaviour otherwise is the same as for INSBYTE, except that any byte at the target location is overwritten.

These three functions should be used in programs only with great care. It is fatally easy to write a program which fills memory with junk, or which overwrites its earlier lines. The first kind is illustrated by the following: LBL Ø1, GETPC, 100, INSBYTE, GTO Ø1, END. After one run it is LBL Ø1, GETPC, X>Ø7, X>Ø7, 100, INSBYTE, GTO Ø1, END. The insertion of byte 1ØØ, X>Ø7, happens twice, since the first insertion bumped the INSBYTE instruction down past the program pointer, to execute again - as can be seen by SST'ing. Since the bumping down stream is unable to recompile the GTO Ø1, the early part of the growing program file is not rerun. Ten seconds running will place about 60 instructions into the program file. With careful use, some of the under-explored delights of the almost forgotten Bug 2 could be recaptured: writing a program into one file from another, for example. To start, use GETPC in a subroutine positioned at the start of the file into which code is to be written, and use it in the writing program. INSBYTE is strongly recommended for use, rather than STOBYTE, in such a case.

LOADP
XROM 21,26
99E

A non-programmable function, which prompts for a program or routine name, in a manner similar to that of CLP, COPY, etc., and loads a user code, RPN program from the normal program RAM space of a 41c into the RAM of an MLDL. The resulting program in the MLDL/MLI may be run in the same way as any RPN program in a ROM. On entry to LOADP, X must contain the address in the MLDL/MLI RAM at which the program is to be loaded. The routine first calls COMPILE, then loads the resulting program. The display shows "PACKING", "COMPILING", "LOADING". Though there is more to its use than this single operation, for changes may have to be made to the original program before loading, to allow the result to run properly, the eventual XROM image may be transferred to EPROM's, for use in a permanent form, or downloaded onto cassette tape. The use of RPN programs from simulated ROM has the enormous advantage of freeing all of the 41c memory for the storage of data.

The use of LOADP

There has to be at least one global label at the start of the program to be

LOADP'ed. If this condition is not met, the first two bytes of the program, as loaded by LOADP, will be turned into (instant) garbage. There may be more than one global label, and there may be global XEQ's and GTO's, though the latter are not recommended, being pretty slow in execution. If there is no more than one global label (at the start of the program to be LOADP'ed), and no alpha XEQ's required to look for globals in ROM, then loading is quite straightforward.

Changing alpha XEQ's to XROM's.

There are two ways in which alpha XEQ's may be changed into (their corresponding) XROM's. (Consider the change that might be made to turn XEQ "PRPLOT" - which will call up the printer user code routine it names, but by using 8 bytes, and being slow to operate - into XROM "PRPLOT", using only 2 bytes, and executing quickly - XROM 29,14 when the printer is not plugged into the 41c.)

(1) By manually changing each alpha XEQ to an XROM with the aid of the program "XI" (for "XROM INPUT" - see Section IV) and the microcode routine GORAM (also given in IV). The XROM numbers of the desired XROM's which will replace the current global XEQ's must be known.

i. SST through the program to be LOADP'ed, to the first alpha XEQ.

ii. Switch to RUN mode, XEQ GETPC.

iii. Key in the ROM ID number (e.g. 29 for the printer, 23 for the I/O ROM), ENTER↑, function number.

iv. XEQ "XI".

v. At the prompt "PRESS SST", hit the SST key to move the program pointer back to the original program. Switch to program mode to see the desired XROM code. SST to see the now abandoned alpha string which the cleared global XEQ contained. This should now be deleted.

vi. Repeat this procedure for all alpha XEQ's. Note that the plug-in ROM containing the corresponding functions need not be plugged in to carry out this operation.

(2) The second method changes all alpha XEQ's to XROM's, without user intervention, but the corresponding ROM modules must be plugged in. However any alpha XEQ in the program which is being modified, and which calls a global label in the same program will not be revised. If this is the case, the program must be loaded twice. The procedure is simply to XEQ XQ>XR. (See IV.) The name of the program, or any label inside the program, must be in the alpha register in the same manner as is required of many of the X-Functions. If there happens to be an alpha XEQ in the program to be loaded, but there is no corresponding function name in ROM, as may be ascertained by running CAT 2, it will be necessary to revise it, replacing it with the wanted XROM code, in the manner described in (1) above.

Revision of the function address table - the FAT.

Neither method revises the function address table, but this can be effected by following these steps:

i. The hex address at which the program was loaded into the MLI should be placed in the right of X.

ii. XEQ APPLBL. (See IV.)

All of the labels in the program should now be appended to the function address table, the FAT. (N.b. Earlier versions of APPLBL required more complex operations, now taken care of by the ability of this version to operate directly on the XROM image itself, rather than only on the original in the 41c RAM space.) Failing the use of these special functions, the whole operation can be carried out manually.

(The full procedure is described for the masochist in Appendix G.) The manual procedure is little trouble when there are only a few short programs to be loaded, but it is quite time consuming and error prone when programs are long and complex. However it may be of some value to skim through this Appendix, since it not only gives an idea of the work carried out by these convenience routines, but also gives an account of the resulting XROM structure, useful if a user code routine in an XROM image needs revision.

Error messages:
If an MLI is not connected, "NO RAM" is displayed.

MLDL?
XROM 21,27
84A
Reports, as a conditional, whether there is an MLDL in the system. If so, the first, or lowest address in the RAM-simulating ROM is returned to the X register - right justified. If no MLDL is present, the following instruction will be skipped. (When the MLI is disabled, the answer will be "NO", though it may still be cleared by CLROM, with the necessary "OK" in alpha.)

NEXTFN
XROM 21,28
775
This finds and returns the address of the next function in ROM - by searching through the function address table at the start of the ROM (the FAT) for the next function following the current location pointed to by the number in register Rgg. The result is placed in the text string in Rgg. (For details of the format, see under DISASM, on p.11.)

NRCL
XROM 21,29
41F
A non-normalising recall. Returns to X, but without normalisation, the contents of the register whose (relative, user) address, in decimal form, was previously in X. The stack is lifted. When the address in X is negative, it returns to X the contents of the register having that absolute decimal address. (NRCL with 7 in X will have the same effect as RCL Ø7, as RCL IND X, but not changing the contents of Rg7, or the transferred contents in any way. With -7 in X, the effect will be the same as that of RCL N. Note that RCL IND X, with 7 in X and the curtain at register T (RØØ address ØØØ), would normalise the contents of N.)

NSTO
XROM 21,30
C52
The counterpart of NRCL. Loads the contents of Y into the register whose relative address is given by a positive number in X, but into a register whose absolute address would be given by a negative number in X. As the former, this is a luxury, but as the latter, a necessity. For non-normalising storing, the ordinary STO function (STO IND Y, when y is positive) is just as good. Normalising only occurs when the contents of a register have to be used by the mainframe routines, and does not occur when direct (synthetic or stack) status recalls are used.

PCWRT
XROM 21,31
460
Given a four digit ROM address, AAAA, followed by a word WWW in 244 format, in the last seven digits of X (as AAAAWWW), PCWRT will write WWW into AAAA in any protoCODER which is connected (with device select Ø).

ROM>REG and REG>ROM
Of the next two routines, one, ROM>REG, has been revised and rewritten from its original version by Paul Lind and the designer of the MLDL, Lynn Wilkins. The other, REG>ROM, from the same source, has been taken over unchanged. With the use of these two microcode routines, any block of words from a ROM, or from a simulated ROM, may be read into a sequence of data registers of the 41c for storage on data cards, on a cassette tape, or for relocation (or loading) in the RAM of an MLDL (MLI). The block may later be reloaded into MLDL RAM at the same location as that from which it was read, or at the same location in a different 4k ROM address block (a different port, in effect), or into a totally different location.

The invention of these two routines completely solved the problem of microcode routine exchange and storage, and at the same time almost completely eliminated many serious microde editing difficulties. Before the MLDL was designed, and these routines for its use were written, listings and burnt EPROM's had been the only means of exchange of code. In addition, tedious editing of listings, and reburning of EPROM's had been neded for every modification. (For the definitive first account of these routines, see the articles by Lynn Wilkins and Paul Lind in the PPC Journal, V9N3, listed in Appendix F.)

REG>ROM
XROM 21,32
BF6
On entry to REG>ROM, X must contain the address of the header register of a block of data registers holding the code to be transferred to the RAM of an MLI. They may be loaded in three different ways, depending on the contents of Y.

(1) Y=Ø. The stored code is loaded into the same locations as those from which it was originally read by the companion routine, ROM>REG.

(2) Y contains one hex digit (Ø to F), right justified. (The CODE'd form, which may be generated by entering the hex digit into alpha, and executing CODE.) The stored code is loaded into the same addresses as those from which it was read, but now in the 4k block specified in Y.

(3) Y contains a four digit address, right justified. (Again, the CODE'd form.) The stored code is loaded into the MLI, starting from the location given in Y. (This must, of course, be a valid address, in the 4k block to which the MLI is set.)

Apart from the header register, each register in the block of registers from which the code is read contains up to 5 words, stored in the form of an alpha string, usually of a non-standard kind. The header register is at the lowest address of the block of registers, and contains details of the block size, the number of ROM words, and their original addresses in ROM, or in simulated ROM. (See ROM>REG below.)

The header register has the format

1Ø ØØ rr rs ss sn nn

where rrr is the number of registers used, including the header itself, ssss is the starting address from which the sequence came, and nnn is the number of words in the file, less 1. All three are hexadecimal. It will usually have been generated by ROM>REG, when the data block was originally read from ROM, or from simulated ROM.

ROM>REG
XROM 21,33
B92
This will read the contents of a sequence of words from ROM memory, and load them into a specified set of data registers. The contents of X and Y give the addresses from which the sequence of ROM words are to be read, and the start of the block of registers into which they are to be stored, with five words in each register. X contains the starting address of the block of storage registers, while Y contains the hexadecimal number SSSSLLLL, right-justified, where SSSS is the starting address of the block of words, and LLLL is the address of the last word. The simplest method of loading, using older microcode functions, was to key SSSSLLLL into alpha, execute CODE, key the number of the first of the storing registers into X and execute ROM>REG.

The address of the last data register used is left in LAST X. Using the functions in ASSEMBLER 3, it is now much faster to execute HEXKB, key in the pair of addresses (SSSSLLLL), R/S, key in the starting user register number to X, and execute ROM>REG.

Suppose that the function ALENG is wanted in the MLI. From a listing, it is found to run from X145 (X being port dependent) to X176. Key HEXKB, X, 1, 4, 5, X, 1, 7, 6, R/S, 3 (first data register), ROM>REG. Last X contains 13, the address of

the last data register used. The code is now in registers Ø3 to 13, with Ø3 as the header. The stack is unchanged, and the read out routine could now be loaded to cassette, or to cards, or into the MLI, using this data. RØ3 is the header register, which now contains

1Ø ØØ ØØ BX 14 5Ø 31

B, or 11 registers have been used, the starting address from which this came was X145, and the number of words read was 31+1 (hex), or 50 (decimal).

ROM>X
XROM 21,34
C47

This is a revised version of the routine, originally written by Jim De Arras, known as X<>ROM, one of the functions in the first readily available, user written EPROM set, known as JIMROM 1H. (See the bibliography.) It appends to the right of X the value of the word at the address given by the four hex digits at the right of X on entry to the routine. Last X contains not the original value of X, but that value hexadecimally incremented. Thus if X contains (say)

ØØ ØØ ØØ ØØ ØØ ab cd

on entry, on exit it will contain

ØØ ØØ ØØ Øa bc dn nn

where nnn is the word located at (hex) address abcd. Last X now contains

ØØ ØØ ØØ ØØ ab c(d+1)

X>ROM
XROM 21,35
C39

Reads the word from the rightmost three digits of X into the MLDL/MLI address given by the four hex digits in X to the immediate left of that word. Thus if X contains the hexadecimal digits

ØØ ØØ ØØ Øa bc dn nn

the word nnn will be loaded in the RAM of the MLDL/MLI at address abcd. It is the exact inverse of ROM>X, except that X is overwritten by the incremented address, abc(d+1), while Last X collects the previous value of X.

ROM?
XROM 21,36
3F8

Given an XROM number in decimal form in the X register, this returns information regarding that XROM to the stack, provided that the XROM exists. If it does not, the function operates as a conditional, and instead of executing the next user step, skips it. The information returned to the stack gives the number of functions in Y, and the FAT address (the address, in the Function Address Table) of the address of the first function in the XROM to the last four digits of X. (The correct format for the immediate use of NEXTFN.) The stack is lifted, and Z and T contain their original contents unchanged. The XROM number is placed, as it should be for any well regulated function, in Last X. If executed from the keyboard, "YES" and "NO" are displayed - with reasonably obvious meanings.

RXL
XROM 21,37
80E

This forms a pair with the following function, RXR. Rotates the digits of X to the Left, moving the mantissa sign digit to the position of the least significant digit of the exponent.

Before: ab cd ef gh ij kl mn
After: bc de fg hi jk lm na

RXR
XROM 21,38
807

Rotate the digits of X to the Right. The rightmost digit of X is rotated into the leftmost digit position, and the remaining digits are displaced to the right. If the contents of X before execution are

ab cd ef gh ij kl mn,

then afterwards they will be

na bc de fg hi jk lm

SXL
XROM 21,39
7ED

Shift the contents of X Left one bit. The 56 bits of X are shifted to the left of the register, towards the mantissa sign. The leftmost bit is lost, and a null bit is shifted in from the right.

SXR
XROM 21,40
7F5

A near inverse of SXL, but this time the bits of X are shifted to the right, the rightmost bit, the least significant bit of the exponent digit, is lost, and a null bit is moved in from the left, to the previous position of the most significant bit of the sign digit. The rightmost bit of the exponent is lost. (The routines/functions thus are not quite inverses, since executing the pair loses the leftmost, or the rightmost bit, depending on the order of execution.) Used as a pair, bits or digits may be cleared from either end of X, and with the aid of RXL and RXR the cleared portion may be moved to any location. Any digit may be rotated right or left, processed, and replaced.

X>$
XROM 21,41
76A

This is the microcode function which was briefly known as N>ALPHA. (See PPCTN #13, p.54.) The sign digit of X is changed from whatever value it previously had to the value 1, thus converting the contents of X to a (usually non-standard) alpha string, allowing them to be stored and recalled from user registers without normalising taking place. The remaining digits are unchanged.

X+Y
XROM 21,42
815

Effects a hexadecimal addition of the contents of X and Y, placing the result in X. The stack is not lifted, but X is overwritten by the sum, while the original value of X overwrites Last X.

Y-X
XROM 21,43
81F

Subtracts, hexadecimally, the contents of X from those of Y, leaving the result in X. The original value of X replaces Last X. Y is unchanged, and the stack is not dropped.

1CMP
XROM 21,44
828

Takes the one's complement of the contents of X. This inverts every bit of the X register, and paired with AND and OR might well have been called NOT. Suppose X contains FF FF ØØ ØØ FF FF ØØ. After subjection to 1CMP it will instead contain ØØ ØØ FF FF ØØ ØØ FF. The three functions form a complete set for lightning logic operations on 56 variables at the same time, since any truth function may be defined in terms of NOT with either AND or OR.

2CMP
XROM 21,45
830

Takes the two's complement of the contents of X. (The one's complement, plus one.)

1-D	2-D	3-D	4-D
XROM 21,46	XROM 21,47	XROM 21,48	XROM 21,49
426	43F	449	453

These functions decode the last n digits of the fourteen digits of X, where the value of n is as indicated in the individual function names, and appends the decoded result to the contents of alpha. If the number in X happens to be

ab cd ef gh ij 3C D7

1-D will append "7" to alpha, 3-D will append "CD7", and so on.

IV APPLICATION PROGRAMS FOR ASSEMBLER 3

The following microcode and 'user code' (RPN) routines are not included in ASSEMBLER 3, but their use will greatly enhance and simplify the employment of many of the functions in it, especially in writing and loading routines into the MLI.

APPLBL This routine will read the location of every global label in an RPN program loaded into the RAM of an MLI, and append the addresses, in order, to the FAT of the XROM image in the RAM of the MLI.

INPUT The loading address, as for LOADP, in hex digits, right justified in the X register. XEQ APPLBL. (See pp.16-18 above.) BEWARE!!! It is quite essential to give the correct address in X. Any error is likely to scramble the FAT and otherwise to produce quite unpredictable results, with this function looking for, and 'finding' alpha labels, then writing further addresses to the FAT.

XQ>XR This routine will change all alpha XEQ's to XROM's, where this is possible. If, for example, XEQ "GETPC" is entered into a program when ASSEMBLER 3 is not available to the system, and ASSEMBLER 3 is subsequently plugged in, XQ>XR will change the 'XEQ "GETPC"' to 'GETPC'. The same result will be obtained when the addressed ROM routine is in user code. E.g. 'XEQ "PRPLOT"' would be changed into 'XROM "PRPLOT"'.

INPUT The alpha register should contain any global label in the program to be revised. If alpha is clear, the function defaults to the program at which the HP-41c is currently positioned. If the program specified is a user code program in an XROM or XROM image, then "ROM" will be displayed. If there is no program, as specified in alpha, the message "NONEXISTENT" will be displayed.

UPDFAT This routine will append the address of a function to the FAT, given the start address of the function in Rgg (in the same format as is used by ASSEM). After keying in a function name, using ASSEM, execute UPDFAT, and the address of that name will immediately be appended to the FAT.

INPUT The address of the first word after the function name, right justified in Rgg. This address in Rgg will be unchanged, to allow the continued use of ASSEM.

GTOEND Identical in operation to GTO .., but no packing takes place - unless there is insufficient room for the inserted END. (For use with LOADP.)

GORAM This will move the program pointer to a user code program in RAM memory, regardless of the original position of the pointer.

INPUT The address, in RAM, to which the pointer is to be moved, right justified in the X register, but in MM format. Obtain the address by GETPC.

"DIS" At the prompt "START ADR", key in the start address, in hex, R/S, then the end address, R/S. If no specific halting address is wanted, R/S without keying any digits the second time, or place Ø in X. The inversion at lines 11 to 15 places the halting address at FFFF. Set the user flags as for DISASM. (See p.12.)

"DISSUB" This is useful to have in an XROM as a subroutine. Place the wanted end and start addresses in alpha, in the format EEEESSSS, ready for coding.

"ASS" Prompts for the starting address for code entry in the manner of "DIS", then prompts for instructions. Pressing R/S leaves the addressed word unchanged, allowing SST'ing through a routine, checking for necessary revisions. Key in entries as described under ASSEM on pp.6-11 above.

(Continued on p.24.)


```
 01♦LBL "DIS
"
 02 CF 03
 03 CF 04
 04 CF 05
 05 "START A
DR "
 06 TONE ↑
 07 HEXKB
 08 "END ADR
"
 09 TONE ↑
 10 HEXKB
 11 X=0?
 12 1CMP
 13 DECODE
 14 RDN
 15 4-D
 16♦LBL "DIS
SUB"
 17 CODE
 18 STO 00
 19 NEXTFN
 20♦LBL 03
 21 DISASM
 22 GTO 03
 23♦LBL "ASS
"
 24 "START A
DR "
 25 TONE ↑
 26 HEXKB
 27 X>$
 28 STO 00
 29 CF 23
 30♦LBL 01
 31 RCL 00
 32 AON
 33 DISS
 34 STO 00
 35 "⊢?"
 36 TONE ↑
 37 PROMPT
 38 FC? 23
 39 DISS
 40 FS?C 23
 41 ASSEM
 42 GTO 01
 43 END

LBL'DIS
LBL'DISSUB
LBL'ASS
END
121 BYTES
```

```
 01♦LBL "ASS
H"
 02 "START A
DR "
 03 HEXKB
 04 X>$
 05 STO 00
 06 AON
 07♦LBL 00
 08 RCL 00
 09 DISS
 10 "⊢?"
 11 HEXKB
 12 RCL [
 13 X=0?
 14 GTO 00
 15 RDN
 16 ":"
 17 3-D
 18 RDN
 19 STO 00
 20 ASSEM
 21 GTO 00
 22 END
```

```
 01♦LBL "XCA
T2A"
 02 FIX 0
 03 CF 29
 04 "XROM ?"
 05 PROMPT
 06 ROM?
 07 GTO 00
 08 "NO ROM
"
 09 ARCL X
 10 PROMPT
 11♦LBL 00
 12 "XROM "
 13 ARCL L
 14 CVIEW
 15 CLA
 16 ARCL Y
 17 "⊢ FUNCT
IONS"
 18 CVIEW
 19 DSE Y
 20  E3
 21 ST/ Z
 22 RDN
 23♦LBL 01
 24 ADV
 25 CLA
 26 X<>Y
 27 10↑X
 28 2-D
 29 X<> L
 30 X<>Y
 31 "⊢1 "
 32 FC? 55
 33 CLA
 34 APPFN
 35 CVIEW
 36 ISG Y
 37 GTO 01
 38 END
```

```
8800 24C ?FSET 9
8801 02F JC 8806 +05
8802 26C ?FSET 2
8803 3C1
8804 087 ?CCO 21F0
8805 34A ?A≠0 RC
8806 381
8807 08A ?NCCO 02E0
8808 046 C=0 SLX
8809 270 RAM SLCT
880A 3E0 RTN
880B 24C ?FSET 9
880C 3D3 JNC 8866 -06
880D 20C ?FSET 2
880E 1C9
880F 086 ?NCCO 2172
8810 3A8 JNC 8805 -05
8811 092 "R"
8812 018 "X"
8813 03E ")"
8814 011 "Q"
8815 018 "X"
8816 349
8817 08C PORT DEP:
8818 08A XQ 889A
8819 391
881A 08C PORT DEP:
881B 000 XQ 8800
```

```
881C 349
881D 08C PORT DEP:
881E 0D6 XQ 80D6
881F 3C1
8820 082 ?NCCO 00F0
8821 08C "L"
8822 082 "B"
8823 08C "L"
8824 010 "P"
8825 010 "P"
8826 001 "A"
8827 0F8 READ 3(X)
8828 226 C=C+1 SLX
8829 0AE A<>C ALL
882A 341
882B 08C PORT DEP:
882C 1F0 CO 81F0
882D 08D "M"
882E 001 "A"
882F 012 "R"
8830 08F "O"
8831 007 "G"
8832 0F8 READ 3(X)
8833 01C R= 3
8834 0C4 CLRF 10
8835 0AA A<>C RC
8836 06D
8837 03E ?NCCO 232F
```

```
8080 20C R= 13
8088 198 LDER 6
808C 81C R= 3
8080 198 LDER 6
8080 818 LDER 8
8080 818 LDER 8
8090 218 LDER 8
8091 8AE A()C ALL
8092 84E C=0 ALL
8093 158 M=C ALL
8094 81C R= 3
8095 2E5
8096 8A4 ?NCXQ 29B9
8097 86A A()B R(
8098 31C R= 1
8099 2EA ?C≠0 R(
809A 863 JNC 80A6 +0C
809B 18A A=C R(
809C 138 LDI S&X
809D 82C ","
809E 36A ?A≠C R(
809F 873 JNC 80AD +0E
80A0 198 C=M ALL
80A1 23C RCR 2
80A2 8AA A()C R(
80A3 158 M=C ALL
80A4 18E A=A-1 MS
80A5 847 JC 80AD +08
80A6 81C R= 3
80A7 86A A()B R(
80A8 842 C=0 ER
80A9 138 LDI S&X
80AA 085 "0"
80AB 36A ?A≠C R(
80AC 34F JC 8095 -17
80AD 31C R= 1
80AE 846 C=0 S&X
80AF 270 RAM SLCT
80B0 198 C=M ALL
80B1 2EE ?C≠0 ALL
80B2 823 JNC 80B6 +04
80B3 23C RCR 2
80B4 2EA ?C≠0 R(
80B5 3F3 JNC 80B3 -02
80B6 268 WRIT 9(Q)
80B7 158 M=C ALL
80B8 2EE ?C≠0 ALL
80B9 887 JC 80C9 +10
80BA 244 CLRF 9
```

```
80BB 3C4 ST=0
80BC 0CC ?FSET 10
80BD 813 JNC 80BF +02
80BE 208 SETF 2
80BF 141
80C0 0A4 ?NCXQ 2958
80C1 0CC ?FSET 10
80C2 02F JC 80C7 +05
80C3 09D
80C4 0A4 ?NCXQ 2927
80C5 13C RCR 8
80C6 0AA A()C R(
80C7 1ED
80C8 01A ?NCGO 067B
80C9 315
80CA 09S ?NCXQ 26C5
80CB 81C R= 3
80CC 18A A=C R(
80CD 08C ?FSET 5
80CE 368 ?C RTN
80CF 24C ?FSET 9
80D0 368 ?C RTN
80D1 28C ?FSET 2
80D2 181
80D3 81B ?CGO 066C
80D4 1FD
80D5 81A ?NCGO 067F
80D6 3C1
80D7 088 ?NCXQ 2CF0
80D8 38D
80D9 81C ?NCXQ 87EF
80DA 818 "X"
80DB 085 "E"
80DC 811 "Q"
80DD 387 "T"
80DE 38D
80DF 81C ?NCXQ 87EF
80E0 020 " "
80E1 81E "↑"
80E2 020 " "
80E3 818 "X"
80E4 812 "R"
80E5 08F "O"
80E6 08D "M"
80E7 387 "T"
80E8 149
80E9 824 ?NCXQ 0952
80EA 81C R= 3
80EB 08A B=A R(
```

```
80EC 2E5
80ED 0A4 ?NCXQ 29B9
80EE 056 C=0 XS
80EF 2E6 ?C≠0 S&X
80F0 3D8 JNC 80EB -05
80F1 0EA C()B R(
80F2 158 M=C ALL
80F3 0AA A()C R(
80F4 3E5
80F5 0A8 ?NCXQ 2AF9
80F6 14C ?FSET 6
80F7 02B JNC 80FC +05
80F8 09D
80F9 0A4 ?NCXQ 2927
80FA 061
80FB 0A6 ?NCGO 2918
80FC 198 C=M ALL
80FD 0AA A()C R(
80FE 070 M=C ALL
80FF 2E5
8100 0A4 ?NCXQ 29B9
8101 06A A()B R(
8102 18A A=C R(
8103 138 LDI S&X
8104 81E "E"
8105 016 A=0 XS
8106 366 ?A≠C S&X
8107 023 JNC 810B +04
8108 088 C=M ALL
8109 18A A=C R(
810A 38B JNC 80EB -1F
810B 06A A()B R(
810C 359
810D 0A4 ?NCXQ 29D6
810E 02E B=0 ALL
810F 2E5
8110 0A4 ?NCXQ 29B9
8111 3D8 C()ST XP
8112 0EE C()B ALL
8113 3D8 C()ST XP
8114 23C RCR 2
8115 0EE C()B ALL
8116 088 C=M ALL
8117 36A ?A≠C R(
8118 38F JC 810F -09
8119 846 C=0 S&X
811A 270 RAM SLCT
811B 0EE C()B ALL
811C 325
```

```
811D 058 ?NCXQ 14C9
811E 268 WRIT 9(Q)
811F 1D8 C()M ALL
8120 81C R= 3
8121 0AA A()C R(
8122 0F8 C()M ALL
8123 0FC RCR 10
8124 0AA A()C R(
8125 13C RCR 8
8126 0AE A()C ALL
8127 238 READ 8(P)
8128 09C R= 5
8129 0AA A()C R(
812A 0AE A()C ALL
812B 228 WRIT 8(P)
812C 035
812D 098 ?NCXQ 26B0
812E 0EE C()B ALL
812F 238 READ 8(P)
8130 17C RCR 6
8131 10E A=C ALL
8132 07C RCR 4
8133 2CE ?B≠0 ALL
8134 2A8 JNC 8109 -2B
8135 08C ?FSET 5
8136 29F JC 8109 -2D
8137 070 M=C ALL
8138 345
8139 0A4 ?NCXQ 29D1
813A 0EE C()B ALL
813B 07C RCR 4
813C 358 ST=C XP
813D 23C RCR 2
813E 08D
813F 08C ?NCXQ 2323
8140 345
8141 0A4 ?NCXQ 29D1
8142 3D8 C()ST XP
8143 08D
8144 08C ?NCXQ 2323
8145 359
8146 0A4 ?NCXQ 29D6
8147 846 C=0 S&X
8148 08D
8149 08C ?NCXQ 2323
814A 088 C=M ALL
814B 36A ?A≠C R(
814C 3CF JC 8145 -07
814D 338 JNC 8134 -19
```

"XCAT2A" Prompts for the ID number of an XROM, then on R/S prints out a catalogue, with all starting addresses of the contained functions and programs.

"ASSH" A convenience routine for ASSembling in Hex. No colons are needed. Prompting at the start is as for "ASS", and as for that routine, a word may be left unchanged by R/S with no input. For keying long routines from a listing, this is much faster than using "ASS".


```
81CB 094 "T"
81CC 001 "A"
81CD 006 "F"
81CE 004 "D"
81CF 010 "P"
81D0 015 "U"
81D1 244 CLRF 9
81D2 378 READ 13(c)
81D3 03C RCR 3
81D4 270 RAM SLCT
81D5 038 READ 8(T)
81D6 0AE A()C ALL
81D7 0AE A()C ALL
81D8 10E A=C ALL
81D9 046 C=0 S&X
81DA 1BC RCR 11
81DB 23A C=C+1 M
81DC 338 FETCH S&X
81DD 226 C=C+1 S&X
81DE 040 WROM
81DF 1E6 C=C+C S&X
81E0 08A A()C M
81E1 1BC RCR 11
81E2 086 B=A S&X
81E3 386 RSHFA S&X
81E4 386 RSHFA S&X
81E5 0A6 A()C S&X
81E6 24C ?FSET 9
81E7 018 JNC 81EA +03
81E8 236 C=C+1 XS
81E9 236 C=C+1 XS
81EA 040 WROM
81EB 23A C=C+1 M
```

```
81EC 0E6 C()B S&X
81ED 056 C=0 XS
81EE 040 WROM
81EF 3E0 RTN
81F0 81C R= 3
81F1 0AA A()C R(
81F2 3C4 ST=0
81F3 088 SETF 5
81F4 158 M=C ALL
81F5 81C R= 3
81F6 085
81F7 0AC ?NCXQ 2B01
81F8 14C ?FSET 6
81F9 368 ?C RTN
81FA 070 M=C ALL
81FB 198 C=M ALL
81FC 226 C=C+1 S&X
81FD 1BC RCR 11
81FE 138 LDI S&X
81FF 0CD "M"
8200 0A6 A()C S&X
8201 31C R= 1
8202 338 FETCH S&X
8203 362 ?A≠C ER
8204 04F JC 8260 +09
8205 39A ?A(C R(
8206 03F JC 8200 +07
8207 03C RCR 3
8208 0AE A()C ALL
8209 248 SETF 9
820A 349
820B 08C PORT DEP:
```

```
820C 1D7 X9 81D7
820D 088 C=M ALL
820E 333 JNC 81F4 -1A
820F 084 "D"
8210 08E "M"
8211 085 "E"
8212 08F "O"
8213 814 "T"
8214 087 "G"
8215 81C R= 3
8216 378 READ 13(c)
8217 110 LDER 4
8218 81C R= 3
8219 18A A=C R(
821A 070 M=C ALL
821B 139
821C 088 ?NCXQ 224E
821D 2E6 ?C≠0 S&X
821E 027 JC 8222 +04
821F 389
8220 058 ?NCXQ 14C2
8221 03B JNC 8228 +07
8222 0D5
8223 088 ?NCXQ 2235
8224 23E C=C+1 MS
8225 06F JC 8232 +0D
8226 34D
8227 0A4 ?NCXQ 29D3
8228 2E5
8229 0A4 ?NCXQ 29B9
822A 37C RCR 12
822B 2E2 ?C≠0 ER
```

```
822C 81F JC 822F +03
822D 2F6 ?C≠0 XS
822E 3D3 JNC 8228 -06
822F 088 C=M ALL
8230 36A ?A≠C R(
8231 3A8 ?NC RTN
8232 311
8233 0A8 ?NCXQ 28C4
8234 2EE ?C≠0 ALL
8235 089
8236 082 ?NCGO 2882
8237 09C R= 5
8238 318 LDER C
8239 81C R= 3
823A 138 LDI S&X
823B 128 " "
823C 2F8 WRITE DATA
823D 088 C=M ALL
823E 278 RAM SLCT
823F 266 C=C-1 S&X
8240 18A A=C R(
8241 038 READ 8(T)
8242 3D3 C()ST XP
8243 084 CLRF 5
8244 208 SETF 2
8245 3D8 C()ST XP
8246 2F8 WRITE DATA
8247 04E C=0 ALL
8248 278 RAM SLCT
8249 378 READ 13(c)
824A 266 C=C-1 S&X
824B 368 WRIT 13(c)
824C 3E0 RTN
```

```
01♦LBL "MOV
E"
02 "BEG END
"
03 HEXKB
04 0
05 ROM>REG
06 "TO "
07 HEXKB
08 0
09 REG>ROM
10 END

LBL"MOVE
END
34 BYTES
```

```
01♦LBL "XI"
02 X<>Y
03 640
04 +
05 64
06 *
07 +
08 RCL X
09 256
10 ST/ Z
11 MOD
12 X<>Y
13 R↑
14 X<>Y
15 STOBYTE
16 X<>Y
17 INSBYTE
18 R↑
19 "PRESS S
ST"
20 PROMPT
21 GORAM
22 END

LBL"XI
END
48 BYTES
```

"MOVE" Prompts for the beginning and end addresses of a block of code to be moved, using HEXKB, then prompts for the address of a target block to hold the code. The block is copied to that 41c RAM location, and copied back to the XROM image from there.

"XI" See the instructions for the use of LOADP, p.17 above.

V ASSEMBLER 3 FUNCTION ADDRESSES AND ERROR MESSAGES

These may be printed out by using the application routine "XCAT2", given above. There is very little space to spare in ASSEMBLER 3, and the routines, while compact, are subject to revision in later versions that may be released. Accordingly, the addresses given below are those of the first release only, given here to assist those interested in downloading them onto cards, or printing them for study purposes. Only the last three digits of the addresses are given. The first depends on the port into which the ROM simulating device is plugged, or on the setting of addressing switches in the unit, as in the case of the HHP-16K.

It will be noted that some of the routines listed below are starred. If these are downloaded and written to an MLI or Protocoder, even burnt into an EPROM, they will not execute as intended, since their called routines are not at the expected addresses. Routines which do not call up any others, as is indicated in the table, may be freely used in this way, always keeping within the boundaries of copyright protection. The use of the other functions will normally require extensive editing, though when their called functions are at the expected addresses, the remainder of the XROM may be changed. There is assistance in this area - from the full annotated listings of ASSEMBLER 3, available from PPC Melbourne.

WARNING: While the XROM numbers given below are those of the named functions in ASSEMBLER 3, they will not necessarily match those of functions which it contains if accessed from a different EPROM set, or from the *RAM of an MLDL/MLI or ProtoCODER*. When using a routine containing such a function, delete it, and rekey if the function is to be used with a different EPROM or with ROM simulating RAM. It is also possible that the identically named function will not have the same properties as those in ASSEMBLER 3. Test the routine carefully before using to determine whether there are any such differences.

Table IV: Function addresses of ASSEMBLER 3 routines.

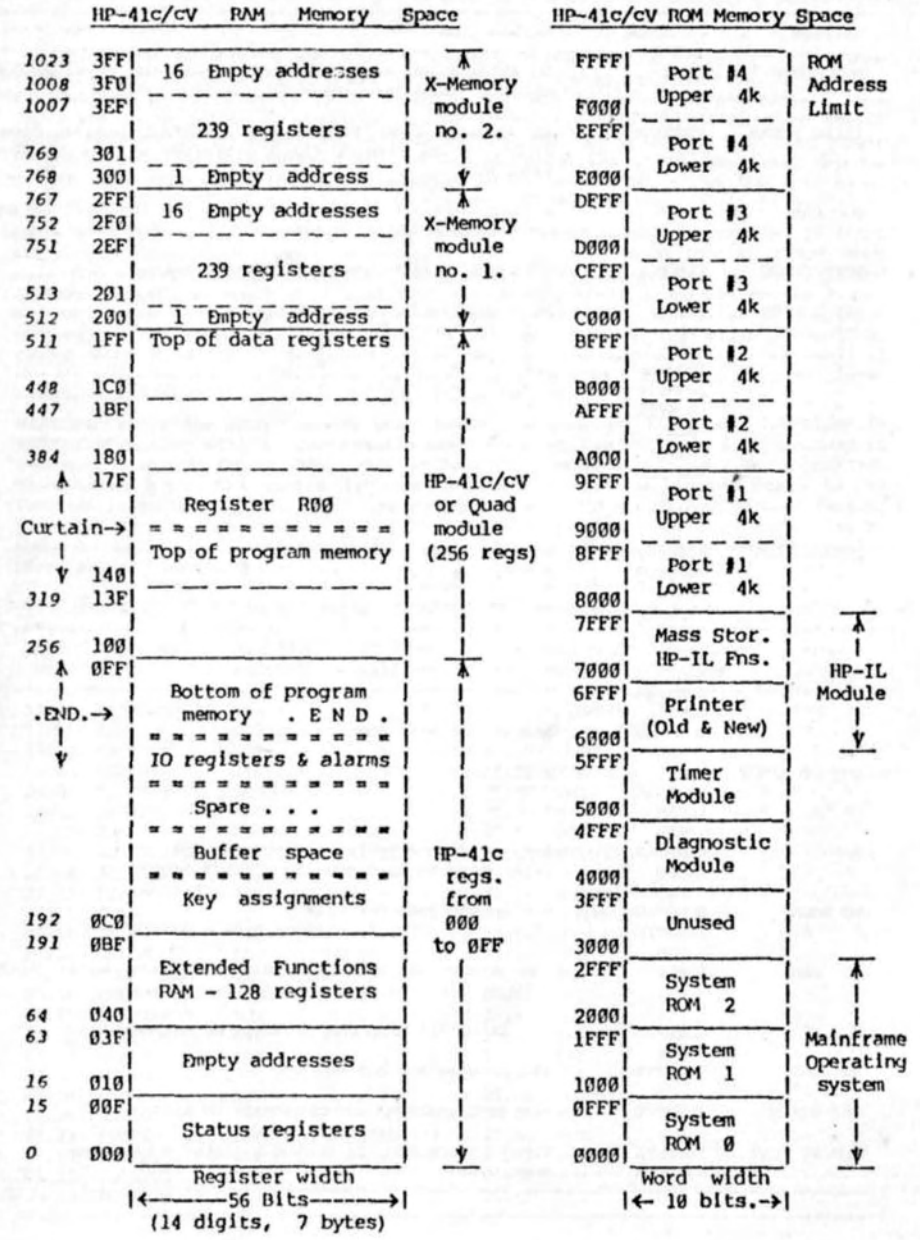
XROM No.	Function name	Starting address	PPC ROM equivalent	XROM No.	Function name	Starting address	PPC ROM equivalent
21,00	ASSEMBLER 3	08B	-	21,25	STOBYTE	B67	"LB"(?)
21,01	AND	837	-	21,26	LOADP *	99E	-
21,02	OR	83F	-	21,27	MLDL?	84A	-
21,03	APPFN	3A8	-	21,28	NEXTFN	775	-
21,04	ASSEM *	1C3	-	21,29	NRCL *	41F	"RX"
21,05	DISASM *	4CF	-	21,30	NSTO	C52	"SX"
21,06	A>X *	7D3	"CD"	21,31	PCWRT	460	-
21,07	X>A *	7E1	"DC"	21,32	REG>ROM	BF6	-
21,08	BCD>BIN *	13F	-	21,33	ROM>REG	B92	-
21,09	BIN>BCD *	114	-	21,34	ROM>X	C47	-
21,10	CF55	7C1	"IF"(?)	21,35	X>ROM	C39	-
21,11	SF55	7CB	"IF"(?)	21,36	ROM?	3F8	-
21,12	CLROM	8B3	-	21,37	RXL	80E	-
21,13	CODE *	0B4	"HN"	21,38	RXR	807	-
21,14	DECODE	0DE	"NH"	21,39	SXL	7ED	-
21,15	COMPILE *	A62	-	21,40	SXR	7F5	-
21,16	COPYROM	894	-	21,41	X>$	76A	-
21,17	CVIEW	4BB	-	21,42	X+Y	815	-
21,18	VIEWA	105	-	21,43	Y-X	81F	-
21,19	DISS	48F	-	21,44	1CMP	828	-
21,20	GETPC	B5B	"Rb"(?)	21,45	2CMP	830	-
21,21	PUTPC	C74	"Sb"(?)	21,46	1-D	426	-
21,22	HEXKB	ADF	-	21,47	2-D *	43F	-
21,23	INSBYTE	B70	-	21,48	3-D *	449	-
21,24	RCLBYTE	B45	-	21,49	4-D *	453	-

General error messages

Message	Function	Explanation
NOT FOUND	ASSEM	An attempt was made to enter a mnemonic which ASSEM did not recognise. (Page 7.)
ILLEG PARAM	ASSEM	An illegal type 'r' register postfix was in X or alpha, or an illegal field parameter was used. (See page 8.)
TOO FAR	ASSEM	A jump distance in excess of 63 words forward, or 64 backward was given in alpha or in X. (See page 10.)
DATA ERROR	CLROM	"OK" not in alpha register (See p.13.)
ALPHA DATA	ASSEM	Alpha string in X register.
" "	X>A	" " " " "
" "	BCD>BIN	" " " " "
" "	COPYROM	" " " " "
" "	INSBYTE	" " " " "
" "	STOBYTE	" " " " "
" "	NRCL	" " " " "
" "	NSTO	" " " " "
" "	REG>ROM	" " " " "
" "	ROM>REG	" " " " "
" "	ROM?	" " " " "
NONEXISTENT	All Fns	ROM simulator not plugged in
"	COPYROM	X register >999
"	INSBYTE	" " "
"	STOBYTE	" " "
"	NRCL	" " "
"	NSTO	" " "
"	REG>ROM	" " "
"	ROM>REG	" " "
"	ROM?	" " "
"	ROM>REG	Header register doesn't exist
OUT OF RANGE	X>A	X register >9999
" " "	BCD>BIN	" " "
" " "	ASSEM	" " "
ROM	COMPILE	Attempted to compile user code in ROM
"	LOADP	Attempting to load program already in ROM
NO ROOM	ROM>REG	Not enough room for file
" "	INSBYTE	" " " " another byte - reSIZE
NO RAM	LOADP	An attempt was made to load a user code program into an MLI, but none is connected. (Page 18.)
" "	COPYROM	Either no MLI/MLDL RAM present, or not enabled
PACKING	INSBYTE	41c program RAM has run out
TRY AGAIN	INSBYTE	Has just packed - encouragement to user
MEMORY LOST	INSBYTE STOBYTE	Vital information in status register c has been destroyed.

APPENDIX A

The address operating space



The HP-41c operates on and with two entirely distinct address spaces. One is normally only occupied by ROM memory data, and the other occupied by RAM. The RAM space is addressed by the microprocessor of the 41c through the last ten bits of the sign and exponent digits of the C register of the CPU. This allows the addressing of 2^{10}, or 1024 locations, at each of which there could be a 56 bit, 14 digit, 7 byte register. The contents of any such addressed register may be read into the C register, or overwritten by the contents of the C register. Individual bytes in a register are handled only by reading the whole of the register to C, treating the individual byte, and writing the result back to the original location. (The address of a byte holding the next RPN program instruction to be executed is kept in register b of status which holds, with register a, the current program pointer and the six level subroutine return stack. Even then, the whole of b is read to the CPU register C for fetching the next program instruction byte, and still has to be isolated using the register b information.)

With the ROM address space, the situation is different, ROM being addressed either through a 16 bit pointer in the CPU, backed by a 4 level subroutine return stack, also in the CPU, or through register C. With 16 bits at its disposal, 64k words of ROM memory may be addressed.

Data access to ROM words is effected through the sign and exponent of register C of the CPU, and individual ROM words may be read into C by a special CPU instruction, called by HP, CXISA. (In Jacobs mnemonics, FETCH S&X.) Each of the words of the ROM memory space is (expected to be) 10 bits in length. They may be treated either as data or as instructions, being handled as data only when read through a pointer in C (in digits 3 to 6, the 'address field'), under control of words from the same data space from which they are read as instructions by the CPU, or when 'loaded' into the sign and exponent digits of C from the following word by a LDI instruction.

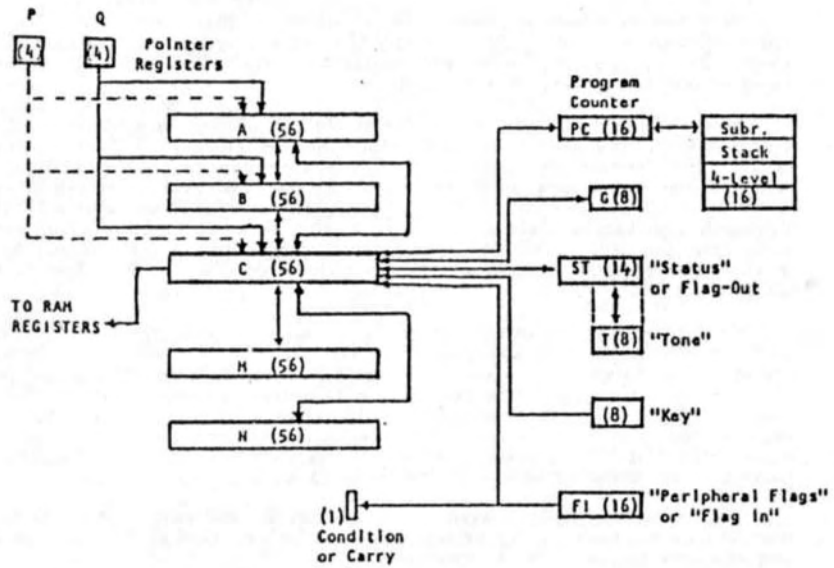
The CPU may be directed to address different 16 register blocks of RAM space (RAMSLCT, followed by the address of the first of the 16), and then directed to any one of the registers within those blocks. A READ instruction will then copy any one of the 16 to register C, while WRIT DATA will overwrite the contents of that address from C. Finally, the instruction POP will read to C the top, and thus the last, address from the subroutine return stack to the address field of C, and drop the stack. PUSH does the reverse. Together they allow computed XQ's and GOTO's, and the reading of sequences of blocks of ROM words as data.

The execution of ROM instructions, whether in RPN user code or in microcode, proceeds from lower to higher numbered addresses, though that of RAM RPN code proceeds from higher to lower addresses.

APPENDIX B

The CPU structure and registers

The following diagram shows the internal registers of the 41c CPU. These are quite distinct from the registers of the RAM memory space on which the CPU operates, and the 41c could still perform in a limited manner even without any of that RAM memory. The diagram also shows, though in an incomplete form, the dynamic interaction of these registers. The arrows indicate either paths of data flow, or paths of control. It should be noted that it is either through register C, or under the control of register C that information is directed through the CPU, and into and out of it, into it from ROM and RAM, and out of it to the display, the printer, the reader, and to the rest of the world..



HP-41c/cV CPU Registers and Information Flow
(After J. Schwartz, January 1983.)

There is some conflict about the specific CPU registers at the disposal of the microcode programmer. Different references give different accounts. (See the HP-Journal for March 1980, the ProtoTECH Manual, Steve Jacobs' PPCTN account and Jake Schwartz in the PPC Southwest Conference Proceedings. Other sources are more authoritative, and have been used as a guide here. It is possible to render all of these consistent.)

Register C 56 bits/14 digits/7 bytes. This is the main register of the CPU, and it is through this that the majority of operations are performed. (See the following Appendix.) ROM words may be read as data, RAM may be read or overwritten, and addressed, peripherals may be read or written to, etc. C interacts directly with most of the other CPU registers, as controlled by individual microcode instructions, or under the direction of the two pointer registers, P and Q. If such a comparison is of any help, the nearest analogue to C is the role played by the X register in the RPN domain. The contents of C are continuously circulated on the main system bus, the line, accessible at the ports, known as DATA.

Registers A and B 56 bits/14 digits/7 bytes. These interact with C and with eachother. Their contents may be exchanged with, compared with, added to or copied from those of C over any range, as determined by individual instructions, or by the pointer registers, P and Q. While C is the main register for arithmetical operations, it is only in conjunction with A and B that most of these arithmetical operations can be carried out.

Registers M and N 56 bits/14 digits/7 bytes. These are used for storage and recall only. They may be copied to C, or from C. Individual parts cannot interact with C. Think of them as useful for scratch or temporary storage only.

Registers P and Q 4 bits, one digit only. These are only used to point to digits in C, or in C together with A or B only. Only one, the selected pointer (which is then known as R) is usually in use at a time, except when the interval between the digits they point to is to be operated on. They may be used as counters, decrementing each time they are employed. Their initial value is not determined through C, but by individual instructions.

The program counter, PC 16 bits, 4 digits, 2 bytes. This holds the address of the microcode instruction currently being addressed, and is incremented after that instruction has been fetched from ROM. Its value may be pushed onto the subroutine return stack, and replaced by the rightmost digits of C, or read to those digits of C, allowing computed XQ's and GOTO's. When an XQ or GOTO instruction is encountered, its value is pushed onto the SBR stack and replaced by the read value, as in the case of an XQ, or simply overwritten in the case of a GOTO.

The SBR stack 4 16 bit registers. The 'subroutine return' stack. This is a set of four registers, arranged in a 'first in, first out' pattern. The value of PC is pushed onto the stack for an XQ, read from the stack for a RTN. The last level is cleared to zero when the stack drops.

Register G 8 bits, 2 digits, 1 byte. Used only for temporary storage of part of the C register, as directed by P or Q or both. Often used to hold part of the flags. While any two digits of C may be stored to, or recalled from G, no arithmetic operations or other operations may be effected between G and C.

Status ST There is conflict over the size of this register, but the HP-Journal (March 1980) and the other sources suggest 14 bits only, rather than the 16 claimed by Jake Schwartz. Since these bits may be individually set, cleared and tested, as 'flags 13 to 0', they might be considered a register, but only 8 of them, the "equal system flags", 7 to 0, may be manipulated as a block - through the exponent of C. In Steve Jacobs' account of the instruction set, ST is this group of 8 flags. The 8 are also used to control the distinct 8 bit 'register', variously called 'T', or 'Tone' (but also called 'FO', or 'Flag Out'), to operate the 'bender'. These are usually copied to T from flags 0 to 7. Flag Out/FO/Tone/T is normally then, a copy of flags 0 to 7. ST is best thought of as this block of 8 flags.

```
Flag #: 13 12 11 10  9  8  7  6  5  4  3  2  1  0
                           -------- ST --------
                           - Equal System Flags -
------------- Flag In - FI -------------
```

Flags 13 to 10 are agreed to have the following significance:

Flag 13 set: Program running.
Flag 12 set: Private program.
Flag 11 set: Stack lift enabled.
Flag 10 set: Program pointer in ROM. (Running RPN code in ROM.)
Flags 9 & 8: Special roles, if any, not known.

Flags 13 to 8 may only be set, cleared, or tested, but flags 7 to 0 may be read or written from the exponent of C. We should, then, add:

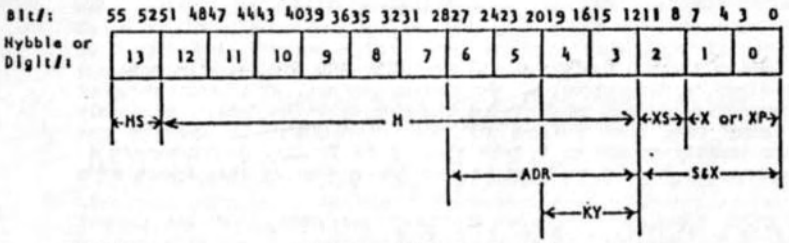
Register T This is an 8 bit register, a copy of flags 7 to Ø of ST, used to send to the beeper, to the 'bender', as HP call it. The contents of this register may be set from C, or cleared. The number of times they are set and cleared per second determines the pitch of the generated TONE. Setting all bits of T to 1 'turns the bender on', clearing them turns it off. The period is determined by the number of microcode instructions between the on and the off, the duration by the number of ON-OFF's.

The key buffer KEY, KY This is an 8 bit register holding the 'key flags', 4 for the row in which a key is located, 4 for the column. This is loaded with a code peculiar to each key when the key is depressed. The key, and the current function assigned to the key is then determined by software control. Changing the software then redefines the key. (See the routine HEXKB of ASSEMBLER 3.) The contents of KY may be read to C by the command C=KEY KY.)

The 'adder carry flag' C This is a single bit register, or flag, which is set when there is a carry digit generated by any arithmetical or comparison operation. When set, it remains so for the next instruction only, and must always be tested immediately by the next instruction. Most tests of C are combined with the instruction whose execution is to be conditional on the state of C. Because of the possibility of confusion with register C, this is perhaps best called 'carry'.

There is, of course, more to the CPU than the collection of its registers, but there is no need here for that further detail, a knowledge of it not being needed for efficient microcode programming. (See the HP-Journal for March, 1980 for the fullest description in the public domain.)

The following diagram from Jake Schwartz' January 1983 summary of microcode information shows the structure and fields of the A, B and C registers of the HP-41c/cV CPU, with the Jacobs' mnemonics for the instructions which determine the field of operation on them of Class Ø.6, Ø.C and class 2 instructions. An account of the counterpart of these for earlier calculators appeared in the HP-Digest for 1977. (See the bibliography in Appendix F.) Note that the bit numbers given in the diagram below are the inverse of the flag numbers of the flag register, register d, of the HP-41c/cV. Flag number ØØ is bit number 55 of register d, flag number 21 is bit number 34, etc.



	Field:	Nybbles:	Bits:		
MS	(Mantissa Sign)	13	52-55	R	Selected pointer (P or Q)
M	(Mantissa)	3-12	12-51	@R	At Nybble pointed to by R
XS	(Exponent Sign)	2	8-11	R←	From Nybble 0 up to nybble R
X or XP	(Exponent)	0-1	0-7	P-Q	From Pth nybble to Qth nybble
ADR	(Address)	3-6	12-27	@R;+	At nybble R and R+1
S&X	(Exp. & Exp. Sign)	0-2	0-11	LD@R,-	Load at nybble R and decrement R
ALL	(Entire Register)	0-13	0-55	SLCT P, SLCT Q	Select which pointer is active

56 - BIT REGISTER FIELDS AND POINTERS IN THE HP41 INSTRUCTION SET. (S. Jacobs mnemonics)

APPENDIX C: Microcoding systems

Requirements

At the start of 1982, just a year before the writing of the present manual began, the situation was simple. There was only one ROM simulator on the market - the HHP-16K. Any wanting to write microcode had to hand burn the last 8 bits of each word into an EPROM, then combine the first two bits of each group of four words into a single byte for burning in another EPROM, plug the result into the HHP-16K, and try it out. There were no other ways. A year later everything from an IL driven EPROM burner to cassette storage of XROM images is available. What is needed? The following summary may help.

A. A ROM simulator. This may be either
- i. An EPROM ROM simulator, or
- ii. A RAM-holding ROM simulator, or
- iii. A combination of i and ii.

(i) This involves using a Dallas Simulator, an HHP-16K or HHP-32K, a ProtoTECH InterFACE and ProtoEPROM, or a Mountain Computer simulator. Some kind of EPROM burner is also needed to write routines, unless a user is content to use EPROM's written by others. PPC Melbourne used an AIM 65 computer with an EPROM burner unit for early experimentation. Later the British made, stand-alone Softy 2 EPROM burner and copier was bought and used extensively. Hand coding in HEX was still needed. Later, routines in BASIC were written (by Richard Collett) for the small Australian designed and manufactured Microbee, and an interface devised to allow routines written using a BASIC assembler on the Microbee to be loaded into the Softy 2 for burning, and for EPROM's read by the Softy 2 to be fed to the Microbee for disassembly. Later, using an IL interface, EPROM images were sent to the Microbee from a 41c for rapid disassembly and revision, and back to the SOFTY 2 through a 232 interface for burning. Much of ASSEMBLER 3 was written this way, though Michael Thompson used a ProtoCODER for much of his early work. Fortunately, EPROM copying is very simple with the Softy 2, which is also convenient for minor revisions.

(ii) Use an MLI, an MLDL I or MLDL II, a ProtoCODER (with InterFACE unit), or a Mountain Computer MCØØ55ØA with their 'RAM Add On Unit', the MCØØ59ØA. The third and fourth do not require any EPROM unit for operation, though the use of such as ASSEMBLER 3 makes the task of loading code enormously easier. The first two and the last accept EPROM's in addition to the RAM they hold. The MLI, the MLDL I, with the Mountain Computer unit, can hold EPROM's in addition to their RAM.

(iii) Use an MLI or MLDL I, both of which accept 4k of EPROM's, or a ProtoCODER with interface unit and ProtoEPROM, the last for holding the EPROM's of ASSEMBLER 3, or a Mountain Computer unit - or use a Dallas unit. The MLI and the Mountain Computer unit are the most convenient, holding as they do, 4k of EPROM's in the former, and up to 16k in the latter. The MLI is probably the most economical purchase for development purposes, though the Mountain Computer unit, with the add-on RAM is very competitive in price. Completed, or part developed microcode or user code ROM images may be stored on cassette and loaded to RAM for use or further revision with any of these combinations, though the use of the ASSEMBLER 3 functions will make this task very much simpler.

B. A computer interfaced to a ROM simulator (or acting as a ROM simulator.

Such a system is described by Paul Lind in PPC Technical Notes # 13, and an earlier unit in PPC Technical Notes #11. (See the Bibliography.) PPCTN#13 describes a 4k RAM unit, interfaced to the computer, which may hold an assembler for writing code to the outboard RAM, from which, in its turn, it may be read as ROM by the 41c.

C. The Richard Collett system, as described above, or similar.

For burning EPROM's from any of these ROM simulating systems, the IL EPROM burner of Mountain Computer could be used. Note also that there are competitors to ASSEMBLER 3 on the market, the first of these being produced by Puget Sound Programming. Details appear in Appendix F.

APPENDIX D: XROM numbering and structure

THE STANDARDISATION OF HP-41c XROM NUMBERS

ROM Name	XROM ID	ROM Name	XROM ID
MATH	Ø1	SECUR	19 *
STAT	Ø2	CLINLAB	19 *
SURVEY	Ø3	AVIATION	19 *
FINANCE	Ø4	MONITOR	19 * †
STANDARD	Ø5	STRTB	19 *
CIR ANL	Ø6	C PPC 1981	2Ø
STRCTA	Ø7	ASSEMBLER 3	21
STRESS	Ø8	IL-DEVEL	22
HOME MN	Ø9	I/O	23 - These two form
GAMES	1Ø *	IL-DEVEL	24 - a single 8k ROM
C PPC 1981	1Ø *	-EXTFCN	25
AS & CD	1Ø *	-TIME-	26
REAL EST	11	- WAND	27
MACHINE	12	-MASS ST	28
THRML	13	(= CTL FNS =	
NAVIG	14	HP-IL MODULE)	
PETROL	15	-PRINTER	29
PETROL	16	CARD READER	3Ø
PLOTTER	17	PPC ROM 2	31
PLOTTER	18		

† Only a small number of this ROM, an early version of the IL-DVT ROM, were made, and they are not stocked or sold by HP.

Those marked with asterisks share their identifying number, and should not be used in the HP-41c/cV at the same time. Of two functions with the same XROM ID, that at the lowest address (i.e. in the lowest numbered port) will be accessed first, and the other ignored. It is recommended that 21 should be used, especially for EPROM sets, unless there is likely to be a conflict with (say) ASSEMBLER 3, and that the number of a function in an EPROM, or in a ROM image in the RAM of an MLI should, so far as is possible, be the same as that of any function with the same name and operation. This will avoid the need to re-key the XROM commands if accessing a function or a program (in user code, RPN) written with a different EPROM set. (See the discussions of these matters in PPCTN from #14 onwards.) It is necessary to be quite clear about the two kinds of conflicts possible: that of address space, avoidable by the user, and of XROM function numbers. To the latter there now seems to be no solution - the time for agreed standards is long past, and conflict is unavoidable.

While selection of an XROM ID number is up to the user of the MLI, though fixed when employing a ROM or an EPROM set, when the number is unable to be changed, short of burning a new EPROM pair, the XROM ID number in a RAM image, as in an MLI or ProtoCODER, may be changed at the discretion of the user. (Change the hexadecimal number at address XØØØ.) The trouble here is that RPN routines using the functions in the unit will need editing by deletion and rekeying of calls so that they may be rerouted to the re-numbered RAM image in the MLI.

XROM STRUCTURE

XROM's are located at whole 4k blocks of addresses. The lowest addresses in an XROM, and a few of the highest have special functions. The remainder may be filled in any way. The locations in the 4k blocks must be filled by ten bit words, giving 2^{10} different codes. They may be read as instructions, or as alphanumeric data. The following summary, adapted from Jake Schwartz' January 1983 PPC Conference paper, should be supplemented by the study of an application ROM, e.g. the Extended Functions module. A listing can easily be prepared by using the ASSEMBLER 3 function, DISASM.

Relative address (hex)	Function of code at that address
XØØØ	The XROM ID number in hexadecimal digits.
XØØ1	The number of functions in the XROM, including the XROM name.
XØØ2-3	Address of XROM name.
XØØ4-5	Address of first routine, program, etc.
XØØ6-7	Address of second routine, etc.
"	" "
"	" "
XØØ2 + 2n) XØØ3 + 2n	Address of nth. routine.
"	" "
"	" "
XØØ2 + 2m) XØØ3 + 2m	Address of last (mth.) routine. m < 64.
XØØ4 + 2m	Null - ØØØ.) Compulsory.
XØØ5 + 2m	Null - ØØØ.
"	" "
"	" "
Add. of ROM name.	Name of ROM (running 'backwards')
"	" "
"	" "
Add. of Fn. #1	Start of Fn. #1 code.
"	" "
"	" "
Add. of Fn. #2	Start of Fn. #2 code.
"	" "
"	" "
"	" "
"	" "
XFF4-A	Special interrupt jump locations.
XFFB-E	ROM name abbreviation and revision #. (E.g. CR1D, WD1E, etc.)
XFFF	ROM checksum for diagnostic module use.

Word pairs containing function addresses

First word of pair:	b	Ø	Ø	Ø	Ø	Ø	a_{11}	$a_{1Ø}$	a_9	a_8
Second word of pair:	Ø	Ø	a_7	a_6	a_5	a_4	a_3	a_2	a_1	$a_Ø$

The address of a function in the same 4k block:

a_{11} $a_{1Ø}$ a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 $a_Ø$

b = Ø when the function is in microcode, 1 when in user RPN code.

When the first word (in 244 format) is ØØ8, and the second ØFE, the address is 8FE, and the routine is in microcode. (X-Functions GETP.) When the first word is 2ØØ, the second ØA1, the address is ØA1, and the routine is in user code. (Printer "PRPLOT".)

ROM words read as data

ROM words may be read as numeric data in a variety of ways, and the results in register C of the CPU could either be treated as decimal or hexadecimal. Not all data is numeric: there are function names (RPN global labels, microcode functions), messages for alpha display of various kinds - error messages and prompts. The last seven bits of the word determine the character which will be displayed or printed. In user code RPN programs the alpha characters in labels and prompt lines, etc., are coded by the normal ASCII values (in hexadecimal). All other alpha coding follows the table below. The displayed form is different from the printed form, both being given here.

b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

Bits 9 and 8: When a ROM word is that for the last character of a name, these two bits will determine the number of prompt lines which will appear when the name is that of a system function. Otherwise these two bits are always clear.

Bit 7: This is normally clear unless the ROM word, as an alpha data item, is the last of a function name (the ROM word at the lowest address).

Bits 6 to 0: These determine the specific character, according to the table below. The value of bits 6 to 4 is down the side, that of bits 3 to 0 along the top. As represented in the hexadecimal 244 form, abc, b ranges from 0 to 4, c from 0 to F. Note that some have a different printed form from their display form. The displayed form below, appears above the printed form. The HP-41c/cV 'ASCII' decimal number of the character is below that, on the third line. This is the byte number for the character, required for its display or printing from text strings and the alpha register.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	P̄	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↗	_
1	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	Sp.	!	"	#	$	%	&	'	<	>	*	+	(	-	→	/
2	Sp.	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	0	1	2	3	4	5	6	7	8	9	⊞	≠	∠	=	≥	?
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	⊢	a	b	c	d	e	¯	⊤	⊤	⊤	⊤	⊤	μ	∠	Σ	∠
4	⊢	a	b	c	d	e	•	⊤	Γ	α	β	°	μ	∠	Σ	∠
	127	97	98	99	100	101	10	96	6	4	5	1	12	29	126	13

User code function names

The address for entry to the execution of a user code routine or program in ROM is that read from the function address table, described above on p.35. In the case of both microcode and RPN routines, the address given is that of the first byte or word of the routine. For user code routines the name is given in the first global label. When the first item in the routine is itself a global label, the alpha string in the label becomes the name of the function. It is preceded by two words, giving data about the length of the routine for downloading purposes, into the 41c RAM. In the printer, for example, the address of "PRPLOT" is 603D. The actual characters are at 6041 to 6046 (the alpha characters in LBL "PRPLOT"). 603D holds 1C8, where C8 is the code for the global prefix. The routine, then, actually begins at 603D after all. The 1 in the start of the XROM word signifies that the byte (8 bits) following in that word, is the first byte of an RPN instruction. The next word at 603E is 000, the first zero indicating that the 00 byte is NOT the only, or first byte of an instruction. (In RAM, this byte, with the last part of the preceding byte, normally contains the global label chaining distance to the next global instruction above in memory. Here there is none, and it is set to zero.) The next is 0F7, the text prefix for the alpha label, then 000, the first zero to show this is NOT the sole, or first byte of an instruction (the other two zeroes filling the key assignment byte for the label), then the words 050, 052, 050, 04C, 04F, 054 - the zeroes at their start having the previous significance, the remaining bytes being the ASCII values for P, R, P, L, O, T. The following words are 18C, 1F6, 04E, 041, 04D, 045, 020, 03F -

to code the text line "NAME 7", 18E, 18B, 19A, 00B to code the following instructions - PROMPT, AOFF, ASTO 11, and so on.

Microcode function names

The names of microcode functions differ. The address of a microcode function, as given by the FAT, is that of the first word of executing code. The name of the function reads backwards from this location, with its first letter coded, in accordance with the above table, by the word before the FAT address. Its last letter is signified by bit seven of the coding word being set. The function FRF of the printer has the address 6A0D. The three preceding words are 090 at 6A0A - the last letter, X, 012 at 6A0B, R, and 110 at 6A0C, the first letter, with bit 8 set to signify a non-programmable function, for the first letter, P.

One of the most famous EPROM sets was JIMROM 1H. Here are parts of its listing:

```
8000 00C                XROM 1D number 12 (hex C).
8001 016                22 functions, including the name (hex 16).
8002 000 }              Address of XROM name, JIMROM 1H, XROM 12,00, at 8039.
8003 039 }
8004 000 }              Address of X<>ROM, XROM 12,01, at 8090.
8005 090 }
8006 000 }              Address of Y<>T, XROM 12,02, at 80A0.
8007 0A0 }
   .
   .
802C 002 }              Address of spoof 'name', JUST KIDDING, XROM 12,21, at 827A.
802D 07A }
802E 000  NOP }         Two NOP's to signify the end of the FAT.
802F 000  NOP }
8030 088  "H"           Last letter of XROM name.
8031 031  "1"
8032 020  " "
8033 00D  "M"
8034 00F  "O"
8035 012  "R"
8036 00D  "M"
8037 009  "I"
8038 00A  "J"           First letter of XROM name.
8039 3E0  RTN           Address of XROM name in the FAT.
   .
   .
808A 08D  "M"           Last letter of name of the function X<>ROM.
808B 00F  "O"
808C 012  "R"
808D 03E  ">"
808E 03C  "<"
808F 018  "X"           First letter of name.
8090 04E  C=0 ALL       First instruction of the function X<>ROM.
8091 270  RAM SLCT
8092 0F8  READ 3(X)
   .
   .
```

APPENDIX E: The microcode instruction set

This short summary of the HP-41c/cV microcode instruction set is intended as a familiarising guide only. A more complete account of their bit structure is given in Steve Jacobs' PPC Technical Notes #9 articles, but full descriptions of their operations have not yet been published, though there is an excellent start (in French) in Jean-Daniel Dodin's excellent Au Fond de la HP-41c. The account here is adapted, with his permission, from Jake Schwartz' concise survey in his HP-41 M-Code Basics summary.

Both RPN user code and microcode routines and instructions are coded in ROM memory by ten bit words. The corresponding microcode instructions separate into four classes, distinguished by the values of the last two (least significant) bits.

Class 1: Two word absolute conditional XQ/GOTO's.

Where a word ends in the binary digits Ø1 it is treated as the first of a two word conditional 'execute' or 'go to' instruction, with the 16 bit target address given by the first 8 digits (AH below) of the word, together with the first eight bits (CD) of the following word. The last two bits of the second word determine which of the following four is intended:

ØØ	?NC XQ ABCD	If the 'carry' bit IS NOT/IS set, execute the subroutine which
Ø1	?C XQ ABCD	starts at address ABCD. ("?NC" = "Is the 'carry' bit not set?")
1Ø	?NC GO ABCD	If the 'carry' bit IS NOT/IS set, jump to address ABCD and
11	?C GO ABCD	continue execution there. ("?C" = "Is the 'carry' bit set?")

Class 3: One word conditional relative jumps.

Where a word ends in the three binary digits a11, it is a single word command, a conditional jump, where the condition depends on the value of a, and the distance backward or forward, as given by the values of the first seven bits, is coded in two's complement form. The maximum distance forward is $3F_{16}$ lines, and backwards 40_{16}. (63_{10} forward, 64_{10} back.)

Ø11	JNC +mn	Jump forward/backward mn words if 'carry' bit not set
Ø11	JNC -mn	
111	JC +mn	Jump forward/backward mn words if 'carry' bit set.
111	JC -mn	

Class Ø: Miscellaneous.

The last two bits of a class Ø instruction word are clear, ØØ. There are four types: Ø, miscellaneous (type 6), 8 and C. The actual type hardly matters when using ASSEMBLER 3, since the coding of the words from their keyed mnemonics is all taken care of by the ASSEM routines.

Class Ø:Ø. The parameter type is indicated by the letters 'f', 'd' and 'r' (see pp.7-8), which are used here to represent the actual hex digits in instances of these mnemonics. The words for these instructions are distinguished by the value of the 5th to 8th bit, where the parameter for each of these is given by the first four (most significant) bits. These instructions take the form: pppp iiii ØØ, where pppp, ranging in value from Ø to F (in most cases), is the parameter/'postfix', and iiii is the instruction/'prefix'. (The values of the corresponding words are not given here. If needed, consult the Jacobs articles. See the end of this Appendix for further comments on this.)

NOP	No operation. (Other Class Ø NOP's are UNUSED. See below.)
CLRF f	Clear flag f (Ø to 13/D).
ST=Ø	Clear all 8 bits of ST to Ø (= flags Ø-7).
SETF f	Set flag f (Ø to 13/D).
CLRKEY	Clear key flag (set when a key is pressed).
?FSET f	Test flag f (Ø to 13/D), and set carry if flag f is set.
?KEY	Test key flag, and set carry if key flag set.
LD@R- d	Load the digit addressed by the currently selected pointer R (one of P or Q) with d (Ø to 15/F), and decrement R.
?R=f	Test whether the currently selected pointer has the value f (Ø to 13/D), and set carry if true.

R=R-1	Decrement the currently selected pointer, R.
R=f	Set the currently selected pointer, R, to value f.
R=R+1	Increment the currently selected pointer, R.
SELP r	Select peripheral r (Ø to 15/F). This disables the CPU, allowing the peripheral to assume control.
WRIT r	Copy the entire contents of C into RAM register r (Ø to 15/F) of the block of 16 selected by a previous RAM SLCT. (See Class Ø.C.)
?FI f	Test specified FI (Peripheral Flag In) register bit f (Ø to 13/D). If the bit/flag is set, the carry bit is set.
READ r	Copy the contents of the designated RAM register in the currently selected block of 16 into C. (The contents of C are overwritten.)
RCR f	Rotate the digits of C to the right by f (Ø to 13/D) digits.

The Miscellaneous Class Ø instructions.
(See Table II on p.8 for the ASSEMBLER 3 short forms of these mnemonics.)

Class Ø.6 A group of 12, taking the form: pppp Ø11Ø ØØ. (In 442 form: X6Ø.)

G=C @R;+	Copy the two digits of C designated by R (the currently selected pointer) and R+1 into Register G.
C=G @R;+	Copy the 8 bits of G over the two digits of C designated by R and R+1.
C<>G @R;+	Exchange the two digits of C designated by R and R+1 with the 8 bits of G.
M=C ALL	Copy C to M, overwriting the previous contents of M.
C=M ALL	Copy M to C, overwriting the previous contents of C.
C<>M ALL	Exchange the contents of C with those of M.
T=ST	Copy the contents of the first 8 bits of the ST register into the T ('TONE') register, overwriting its previous contents.
ST=T	Copy the contents of the T register into the first 8 bits of the ST register, overwriting the previous contents.
ST<>T	Exchange the contents of the first 8 bits of ST with those of T.
ST=C XP	Copy the two exponent digits of C into the first 8 bits of ST, overwriting the previous contents of ST.
C=ST XP	Copy the contents of the first 8 bits of ST into the exponent digits of C, overwriting the previous contents of those digits.
C<>ST XP	Exchange the exponent digits of C with the contents of the first 8 bits of ST (flags Ø to 7 are exchanged with C's exponent digits).

Class Ø.8 A group of 15 miscellaneous Class Ø, type 8 instructions.
These all have the bit pattern: pppp 1ØØØ ØØ. (In 442 form: X8Ø.)

XQ-GO	Pop the subroutine stack, losing the XQ return address. This effectively converts a previous XQ instruction into a GO instruction, since the program counter is unchanged. (RTN also pops the subroutine stack, but places the popped address into the program counter.)
POWOFF	Go into 'light sleep' - i.e. standby mode. In a running program this instruction must be followed by a NOP. (In 'light sleep' the CPU is inactive, but the display is on, with the keyboard monitored by a scan maintained by the display chip.)
SLCT P	Select the four bit register/pointer P as the currently active pointer to the digits of C, etc. (In the Jacobs mnemonics, the currently active pointer is referred to as R.)
SLCT Q	Select Q as the currently active pointer/counter, R.
?P=Q	Set carry if the (pointer) values of P and Q are identical.
?LOWBAT	Set carry if battery voltage is low.
A=B=C=Ø	Clear all digits of A, B and C to zero.
GOTO ADR	Copy the address field (digits 6 to 3) of C onto the program counter. Effectively an 'indirect' GOTO, whose address may then be a computed address, shunted to these digits after computation.
C=KEY KY	Copy to the key field of C (digits 4 and 3, the least significant two digits of the mantissa) from the key buffer, overwriting what was previously there. The last key pressed may then be identified from the new contents of these digits.
SETHEX	Set the arithmetic section of the CPU to operate in HEX mode. (Until cancelled by a counterpart 'SETDEC', all arithmetical commands will be executed in hexadecimal/binary mode.)

SETDEC Set the arithmetic section of the CPU to operated in binary-coded decimal mode, when operations are carried out in base 10.
DSPOFF Turn display off.
DSPTOG Toggle display mode - on to off, off to on.
?C RTN Return from subroutine if carry set. (The stack is popped, its top replacing the current program counter.)
?NC RTN Return from subroutine if carry is clear, if it is NOT set.
RTN Unconditional RTN.

Class Ø.C. A group of 13 miscellaneous Class Ø, type C instructions. These all take the form: pppp 11ØØ ØØ. (In 442 form: XCØ.)

N=C ALL Copy all digits of C to register N, overwriting the previous contents of N. (This is, effectively, a STO N, though the 'N' here is the N of the CPU, not that of alpha.)
C=N ALL Copy the contents of N to C, overwriting the old contents of C. (Analogous to RCL, for this is a RCL to C.)
C<>N ALL Exchange the contents of N with those of C. (Neither contents are lost.)
LDI S&X Load the ten bit word following THIS instruction's location, into the sign and exponent digits of C, overwriting the previous contents of these three digits. That next word, AS instruction, is then skipped. The word following any LDI S&X becomes a data word only - unless there is a jump to that word by a previous GOTO its address.
PUSH ADR Push the 16 bits/4 digits of the address field of C (digits 6 to 3) onto the subroutine return stack. When the next RTN is encountered, this (normally new) address is the one to which there will be a 'RTN', while the previous immediate RTN address is that to be RTN'd to on the next RTN.
POP ADR Pop the subroutine return stack, overwriting the address field of C (digits 6 to 3) by the 16 bits of the (up to then) pending RTN address.
GOTO KY This command writes from the key buffer, holding the 8 bits identifying the last-pressed key, onto only the last 8 (least significant) bits of the program counter. Suppose the program counter had the value abcd. The effect is that of a jump, FROM the address abØØ to the address abkk, where kk is the code for that last pressed key. The jump is to a specified location in the current 256 block of addresses. This may be used to identify (and then execute) commands entered from the keyboard.
RAM SLCT Set the address of the block of 16 RAM registers into which or from which their contents may be written or read by register C. The address of the lowest numbered register of the block is taken from the sign and exponent digits of C.
WRITE DATA Copy over the contents of the lowest addressed RAM register of the block of RAM registers currently selected, the entire current contents of C.
FETCH S&X Copy the ROM word at the address given in the ADR field (digits 6 to 3) of C, into the exponent digits (sign and exponent) of C.
C=C OR A Overwrites each bit, n, of C by the 'logical OR' of that bit and the matching bit of A. After execution, bit n of C will be set if either or both of bit n of C and bit n of A were set before execution. If neither were set previously, bit n remains clear. (See the account of the ASSEMBLER 3 function 'OR' on p.5)
C=C AND A Overwrites each bit n of C by the logical 'AND' of each n bit pair of registers C and A. Each bit of C will be set after execution only if both of the bits n in A and C were set before execution. It will be cleared, or remain clear otherwise.
PRPH SLCT Select the peripheral specified by the contents of the S&X field of C.

WROM This is not one of HP's used codes. As far as the CPU is concerned, it is a NOP, but its detection on the bus by the circuitry of the MLDL causes it to load data into its RAM: the contents of the last ten bits of S&X are copied into MLDL/MLI RAM at the address given in the address field of C.

Class 2 Instructions, field specified.

As mnemonics, all of these instructions have both a prefix and a postfix. They have the bit structure: iiii ifff 1Ø, where the first five bits, iiii i determine, or constitute the 'prefix' or instruction, and fff determine the 'postfix'. This gives

32 (=5^2) prefixes or functions, and 8 (=2^3) postfixes or operands, for a total of 256 possible instructions. (The postfixes are given in Table III on p.10.) The instruction prefixes are given below. Note that an instruction is only carried out on or in the field specified by the postfix. Thus A<>C S&X exchanges only the three exponent digits of A and C, the remainder of these registers being undisturbed. The arithmetic operations are carried out in the mode, hexadecimal (binary) or decimal (BCD - Binary Coded Decimal), to which the CPU was last set. If a carry bit spills over, it has no effect outside the selected field, but the carry flag, 'carry' will instead be set, and may then be tested by the following instruction. The same is true for the tests. (Remember: the carry flag remains set only for the immediately following instruction. When it could not have been set by the preceding instruction or test, a negative conditional instruction becomes an absolute instruction - e.g. ?NC GO will then be an unconditional jump. In the same circumstances, a positive conditional instruction will never be executed - its condition will never be satisfied.)

Class 2 Prefixes. Operations are carried out on the specified fields only.

Carry is set when there is an overflow, or underflow.

A=Ø Clear all digits of A at the field specified.
B=Ø Clear all digits of B at the field specified.
C=Ø Clear all digits of C at the field specified.
A<>B Exchange the contents of A with those of B at the field specified.
B=A Overwrite B at the specified field with the contents of A at that field.
A<>C Exchange the contents of A at the specified field with the contents of B at the specified field.
C=B Overwrite the contents of C at the specified field with the contents of B at the specified field.
C<>B Exchange the contents of C at the specified field with the contents of B at the specified field.
A=C Overwite the contents of A at the specified field with the contents of C at the specified field
A=A+B Add the contents of B at the specified field to the contents of A at the specified field, leaving the result in A - at the specified field.
A=A+C Add C to A - at the specified field.
A=A+1 Increment A by one at the specified field.
A=A-B Subtract the contents of B at the specified field from the contents of A at the specified field.
A=A-1 Subtract one from the contents of A at the specified field.
A=A-C Subtract the contents of C at the specified field from the contents of A at the specified field, leaving the result in A.
C=C+C Double the contents of C at the specified field.
C=C+A Add the contents of A at the specified field to the contents of C at the specified field.
C=C+1 Increment C by one at the specified field.
C=A-C Subtract the contents of C at the specified field from the contents of A at the specified field, leaving the results in C.
C=C-1 Decrement C by one at the specified field.
C=Ø-C Replace the contents of C at the specified field by the arithmetic complement of C at that field.
C=-C-1 Replace the contents of C at the field with the ones-complement of C at the field.
?B≠Ø Set carry if B is non-zero at the specified field.
?C≠Ø Set carry if C is non-zero the the specified field.
?A<C Set carry if A is less than C at the specified field.
?A<B Set carry if A is less than B at the specified field.
?A≠Ø Set carry if A is non-zero at the specified field.
?A≠C Set carry if A is not equal to C at the specified field.
RSHFA Shift the contents of the specified field of A one digit to the right.
RSHFB Shift the contents of the specified field of B one digit to the right.
RSHFC Shift the contents of the specified field of C one digit to the right.
LSHFA Shift the contents of the specified field of A one digit to the left.

Field Specifications
(See Table III, p.10 for the ASSEMBLER 3 keyed forms.)

@R	At the digit pointed to by the value of the currently selected pointer - one of the two 4 bit registers, P and Q.
S&X	The sign and exponent digits, digits 2, 1 and Ø.
R<	The field consisting of all the digits from that designated by the currently selected pointer, up to the right hand - exponent - end of the register.
ALL	The field consisting of all 14 digits.
P-Q	The field consisting of all of the digits from that designated by the value of P up to and including that designated by the value of Q.
XS	The field consisting of the exponent sign, digit 2, only.
M	The field consisting of the ten mantissa digits, digits 12 to 3.
MS	The field consisting of the single digit, #13, the mantissa sign only.

There are a number of words so far not known to have any effect, and not found to be used in any HP routines. Though they might then be used as NOP's, this is not recommended. Class Ø: pppp ØØØØ ØØ, where pppp are all zero is the standard NOP, but where pppp ≠ Ø this also seems to have no effect. It may be that the last 6 of the ten bits only are consulted. No effect has been noted for Class Ø: pppp 11Ø1 ØØ, or for the class Ø: Ø1B, 11B, 21B 31B ØØ3Ø, 1FØ and 2BØ. If any operation is found for these, let the PPC Technical Notes editors know. There are anomalous effects when the values of some of the parameters or pointers are outside their expected ranges - though ASSEM detects most illegal values. There are no flags 14 or 15, for example.

APPENDIX F: Microcoding devices, references, and source materials

The details of non-HP accessories and the other information given below were believed to be correct as of April 1983. No guarantee is given as to its accuracy. Write to the addresses which are given for up to date information.

(1) Devices:

(a) The HP-41 EPROM ROM Simulator. This is a small box, equipped with a short cord terminating in a plug which is inserted into one of the ports of the HP-41c. Into this box a capsule, holding anywhere from 4 to 16k of EPROM's may be plugged. The unit, with a single capsule, costs US$299, with US$49.95 for extra capsules. Dallas provide an EPROM burner service, transferring user programs to EPROM's in the correct format, at US$40 for a 4k EPROM set, and US$80 for a 16k set. They are manufactured by:

Dallas Development Systems, 7410 Stillwater Drive, Garland, Texas, U.S.A. 75042.

(b) The HHP-16K and HHP-32K. These are ROM simulators, capable of holding EPROM sets of up to 16K words, in the case of the HHP-16K, and up to 32K words in the case of the HHP-32K. Like the Dallas ROM simulators, they contain no user RAM. The HHP-16K is US$250 (approximately), the HHP-32K is US$495. These, with other non-HP peripherals for the HP-41c/cV, are manufactured by

F.M. Weaver Associates, Hand Held Products Division,
6201 Fair Valley Drive, Charlotte, North Carolina, U.S.A. 28211.

(c) The ProtoCODER. This is a unit for use with the ProtoSYSTEM INTERFACE. It contains 4k of ten bit RAM words, into which, equipped with suitable software, a 41c user can write microcode or RPN user code (in the appropriate ROM format). The software needed is minimal, but writing in code manually can be tedious. The ProtoSYSTEM INTERFACE will accept other units made by the same firm, the most relevant of which here, is the ProtoEPROM unit. This will accept up to 16K of (ten bit) EPROM's in the same manner as the HHP-16K and the EPROM ROM Simulator, and may be used at the same time as the ProtoCODER, and with the same INTERFACE unit. In order to make use of the powerful functions of ASSEMBLER 3, an INTERFACE and a ProtoCODER, need to be supplemented with a ROM simulator - the ProtoEPROM, or one of the Weaver or Dallas units. The ProtoSYSTEM INTERFACE is US$150, the ProtoCODER US$175, and the ProtoEPROM US$75. These, with other Proto- units, are available from:

ProtoTECH, Inc., 16815 E. Costilla Ave., Aurora, Colorado, U.S.A. 80016.

(d) The MLI, the Machine Language Interface. This is the version of the MLDL to which reference has been made throughout, and also the unit for which the main routines of ASSEMBLER 3 have been written. It contains, in its present realisation, space for 4K of ten bit RAM words, and 4k of EPROM. (It would be possible to extend either section, though not on the present, unmodified board.) The circuit board may be purchased on its own, complete with full assembly and operating instructions, for A$40 (plus packing postage and insurance), or as a fully built up unit, complete with batteries, case and an ASSEMBLER 3 EPROM pair and Manual, for A$199 (again plus packing, insurance and postage). Write to the manufacturers:

Microbaud Developments, 39 Severn St., Box Hill North, Victoria, Australia 3129.

(e) The MLDL II. A fully built unit, a later version of the MLDL I, which was described in the PPC Journal in March 1982, V9N3. This later version does not require the use of any EPROM's at all, and may be loaded with microcode from the unadorned 41c, may have its code read out to registers, loaded on cassette, etc. It holds 8k of ROM simulated RAM (ten bit words), and is available from:

COMP/STOP, Drawer 36600, Tucson, Arizona, U.S.A. 87540.

The same firm stocks other accessories for the 41c, presumably including those listed above.

(f) The Mountain Computer HP-IL EPROM Programmer, MCØØ5ØGA, and HP-41c Application Memory System (ROM emulator), MCØØ55ØA. Using the Programmer, EPROM's may be made directly from the HP-41c, using the HP-IL. This allows permanent EPROM copies to be made of user written XROM images developed on any of the RAM XROM simulators. It is described as able to burn any EPROM's up to the new 27128 (16k x 8 bits.) The projected price for the Programmer is US$450. The MCØØ55ØA, allows up to 16k of EPROM memory to be installed. This is projected to cost $195, and may be used with up to 16k of RAM, for which a RAM board is required, at US $95. Up to ten 6116 CMOS chips (MCØØ595A at $7.20 each) may be installed, to give 16k of RAM. This would appear to be a unit like those made by Hand Held Products and Dallas Developments, but using common circuitry to allow the addition of the RAM storage, in the manner of the MLI and its older brother, the MLDL. A 4k EPROM set containing the operating software is included in the price. The RAM board is designated as the MCØØ59ØA, and plugs into the EPROM unit. These units are manufactured by:

Mountain Computer, 300 El Pueblo, Scotts Valley, California, U.S.A. 95066.

Mountain Computer are also proposing to provide custom written microcode, and a burner service for RPN, similar to those offered by Weaver and Dallas.

(g) Puget Sound Programming, operates as designers of hardware, and writers of software for the HP-41c/cV system. Their products are expected to include EPROM sets holding microcode routines, but no details were available at press time. Write to:

Puget Sound Programming, 3912 171st. Place, N.E., Redmond, WA, U.S.A. 98052.

(h) More software, and perhaps hardware, will be forthcoming from Deep Thinking Software, the producers of ASSEMBLER 3. Announcements will appear in PPC Technical Notes. Otherwise write to:

Michael Thompson (8496), 24 Canterbury Road, Camberwell, Victoria 3124, Australia.

(2) Periodicals

The PPC Journal

The available literature on the subject of microcode, and on the devices allowing its use by 41c owners, is quite extensive, but almost inaccessible outside the major programmable calculator user group in the world, PPC. This group was first formed in 1974 when the first handheld programmable calculator, the HP-65, was released by Hewlett-Packard. It was then known as the 65 Users' Club, changed its name to the PPC Club in 1978, and was Incorporated in 1982. Amongst a wide range of other activities, it publishes what was originally called 65 Notes, but became the PPC Journal in 1978. With only one or two exceptions, there was nothing in print on microcode (the name originally given by Hewlett-Packard to the various assembly language, or machine codes of their generations of hand held calculators, both programmable and non-programmable) outside of the pages of the PPC Journal (and the 65 Notes), until 1980.

The PPC Journal is available only to members of the PPC Club, at an annual subscription which varies round the world. For a sample copy, and membership application forms, write to

PPC, 2545 West Camden Place, Santa Ana, California, U.S.A. 92704,

enclosing an A4 sized self-addressed stamped envelope (two ouncen, airmail, or International stamp vouchers for the appropriate amount). The same material is available from PPC Melbourne.

PPC Technical Notes

The Melbourne Chapter of the PPC Club, PPC Melbourne, started its own users' publication, PPC Technical Notes, in 1979, and has published information on the subject of microcode from 1980 on. Subscriptions are A$20 annually, and back issues are available at A$10 per set, plus postage. (Airmail rates on back issues run to well over A$10, depending on the destination.) Send Bank Cheques or money orders, made payable to PPC Melbourne, to:

R.M. Eades, Box 15, Hampton, Victoria, Australia 3188.

PPC-T

This is a growing 'newsletter', edited by Jean-Daniel Dodin, now an excellent periodical, published by members of the Toulouse Chapter of the PPC Club. It has had original material on microcode from the early issues. Subscription, 150 Francs, by air outside France, 100F in France. Write to

PPC, 77 Rue Cagire, 31100, Toulouse, France.

(3) Books

Any wanting to program in the HP-41c/cV microcode should have a good grasp of the operating system of that machine. Suitable texts are

Wickes, W.C. Synthetic Programming on the HP-41c.
(Larkin Publications, 4517 NW Queens Avenue, Corvallis, Oregon, U.S.A. 97330.)

Jarett, K. HP-41 Synthetic Programming Made Easy.
(SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, California, U.S.A., 1982.)

PPC Inc. The PPC ROM Manual.
(Published by PPC. The manual for the PPC written, 8k Custom ROM. See above)

Dodin, J-D. Au Fond de la HP-41c. (In French.)
(J-D. Dodin, 77 Rue du Cagire, 31100 Toulouse, France.)

The latter is (to date) the only publication to give any kind of introduction to the subject of microcode programming. In this it is quite superb, with well thought out diagrams of the CPU, of program flow and detailed descriptions of the instruction set. It also functions as an introduction to the operating system of the HP-41c, and to the subject of synthetic programming, a knowledge of which is essential to serious microcode work. Strongly recommended, and still very useful even to those knowing no French at all. The Operating Manual of the ProtoCODER has a fairly complete description of microcoding, but uses the NOMAS (HP) mnemonics, though the standard PPC Jacobs/De Arras mnemonics are also covered.

It must be stressed that the ability to write microcode presupposes a full understanding of the principles of operation of the HP-41c/cV. Any of the four texts above will impart this knowledge. The most efficient writing of such routines will also require an understanding of synthetic programming. Both of these may be obtained from study of the PPC ROM Manual, which contains excellent articles on the operation of the 41c and on synthetic programming. In addition the documentation of the PPC ROM routines and programs approaches completeness, and the its routines and programs fully represent the 'state of the art' at the time of its publication. Since it was released before the later HP accessories were released, some of its routines can interfere with the use of (e.g.) the timer module, and study of the PPC Journal from 1981 on is recommended.

(4) Articles and papers.

All of the known references are given below, but a beginner is advised to use only those marked with an asterisk. Many of the following were pioneering efforts, are commonly plagued with errors, and should be used with caution. This compilation started from the only other bibliography which has appeared in print, that in the Lind-Wilkins article 'M-Code, Hardware and Software'. The authors have been innovators in more than one way . . .

1) Anonymous Advantages of Using Microcode.
(PPC Journal, V9N7P18, 1982.)

2) Anonymous Microcode: Electronic Building Blocks For Calculators.
(Hewlett-Packard Digest, No.3, 1977, pp.4-6.)

3)* Anonymous The NOMAS Listings.
(Available From PPC Melbourne, PPC California.)

4) Bailey, B. Time Module Alarms and I/O Buffer.
(PPC Journal, V9N7P11, 1982.)

5) Bouldin, C. Report from Seattle Base to Moon Relay.
(PPC Technical Notes #13, pp.79-80.)

6)* Bouldin, C. & Trin, P.
Two Microcodings: SAVESTA and RESSTA.
(PPC Technical Notes #13, pp.81-83.)

7) Cadwallader, T.
CATalogue 3 Function Address Table in ROM 1.
(PPC Technical Notes, #6, Supplement, pp.1-4.)

8) Cadwallader, T.
RAM and ROM Pointers.
(PPC Technical Notes, #4, pp.33-4.)

9) Cadwallader, T.
Pointer and XROM Numbers.
(PPC Technical Notes, #4, pp.41-2.)

10) Cadwallader, T.
ROM Labels, Microcode Addresses, XROM Structure, Etc.
(Various articles and notes in PPC Technical Notes #4, pp.12-13, 33-34, 37-46.)

11) Cadwallader, T.
Printing HP-41c Microcode Byte Tables.
(PPC Technical Notes #6, pp.48-9.)

12)* Cadwallader, T.
Byte Jumping: A Window Into ROM.
(PPC Northwest Conference Proceedings, August 1981, pp.37-47.)

13) Collett, R. Getting Into a Non-Normalised CAT.
(PPC Technical Notes, #8, pp.4-6

14) Collett, R. The MALMAC December Disassembler.
(PPC Technical Notes, #10, pp.51-5.)

15)* Collett, R. Machine Code Formatting For EPROM Burning.
(PPC Technical Notes, #12, pp.55-7.)

16)* Collett, R. Microcode Instruction Word Recoding.
(PPC Technical Notes #8, p.83.)

17) Collett, R. HP-41c Rosetta Stones Microcode - Disassembler.
(PPC Technical Notes #9, pp.54-60.)

18) Collett, R. HP-85 Assembler for HP-41c/cV Microcode.
(PPC Technical Notes #12, pp.74-80.)

19) Collett, R. & McGechie, J.E.
Some Bits of Microcode
(PPC Technical Notes #7, p.55.)

20)* Collett, R. & Thompson, M.
Hard Melbourne Microprogramming on Softy 2.
(PPC Technical Notes #13, pp.53-5.)

21) Collett, R., Groom, R.Q., & McGechie, J.E.
Microcode Mantras.
(PPC Melbourne, 1981)

22) Cook, M.J., Fichter, G.M. & Whicker, R.E.
Inside the New Pocket Calculators.
(Hewlett-Packard Journal, November 1975, pp.8-12.)

23)* Crowle, N. "C = KEY KY" Keycodes.
(PPC Journal, V9N7P16, 1982. Also PPC Technical Notes, #13, p.3.)

24)* Crowle, N. Operating Manual For the ProtoCODER.
(ProtoTECH, Inc., 1982)

25) Crowley.W.L. & Rode, F.
A Pocket Sized Answer Machine For Business and Finance.
(Hewlett-Packard Journal, May, 1973, pp.2-8.)

26) De Arras, J. HP-41c Bus Interfacing.
(PPC Journal, V7N3P20, 1980.)

27) De Arras, J. HP-41c/cV Assembly Language Programming.
(PPC Northwest Conference Proceedings, August 22nd., 1982, p.53ff.)

28) Dickinson, P.D. & Egbert, W.E.
A Pair of Program-Compatible Personal Programmable Calculators.
(Hewlett-Packard Journal, November 1976, pp.2-8.)

29)* Dodin, J-D. ECRAN-QUAD, HDUMP.
(PPC-T, #4, pp.24-5. 1983)

30) Dodin, J-D, Gengoux, E.
Utiliser un Port-X-tender.
(PPC-T, #4, pp.13-14. 1983.)

31) Egbert, W.E. Personal Calculator Algorithms. (I-IV)
(Hewlett-Packard Journal. I: May 1977, pp.22-24.
II: June 1977, pp.17-20. III: November 1977, pp.22-23.
IV: April 1978, pp.29-32.)

- Gengoux, E., Dodin, J-D.
Utiliser un Port-X-tender.
(PPC-T, #4, pp.13-14. 1983.)

32) Groom, R.Q. Access To ANY 41c ROM Address?
(PPC Technical Notes, #4, p.59.)

33) Groom, R.Q. ROM Addresses, Printer, Reader, Wand and Mainframe Functions.
(PPC Technical Notes, #4, pp43-4.)

34) Groom, R.Q. HP-41c ROM Readout Character Table.
(PPC Technical Notes #6, p.57, and PPC Journal, V8N4P10, 1981.)

35) Groom, R.Q. The MEMORY LOST Microcode Routine.
(PPC Technical Notes #12, p.81.)

75) Rath, C. Debug Descovered in Labradacadian Wilds.
(PPC Technical Notes, #12, pp.28-30.)

76) Rath, C. The Labradacadian Sees Seattle From Europe.
(PPC Technical Notes, #11, pp.65-6.)

77)* Rath, C. Debugging On the MLDL.
(PPC Journal, V9N4P25, 1982.)

78)* Rath, C. HP-41 Assembly Hints.
(PPC Technical Notes, #13, p45, PPC Journal, V9N7P10, 1982.)

79)* Rath, C. ROM XRUM's and RUM XROM's.
(PPC Technical Notes #13, p.20.)

80)* Rowell, G. Annotated HP ROM Listings.
(PPC Sydney, forthcoming.)

81)* Schwartz, J. HP-41 M-Code Basics.
(PPC Southwest Conference, Proceedings, January 1983, pp.31-43.)

82)* Tozer, M. & Winkler, C.
Machine Language Interface.
Owners' Notes and Construction Guide.
(Microbaud Developments, Melbourne, 1983.)

83)* Thompson, M. ASSEMBLER 3 Annotated Listing.
(PPC Melbourne, forthcoming.)

- Thompson, M. & Collett, R.
Hard Melbourne Microprogramming on Softy 2.
(PPC Technical Notes #13, pp.53-5.)

84)* Trinh, P. M-Code SST and BST.
(PPC Journal, V9N7P49, 1982.)

- Trinh, P. & Bouldin, C.
Two Microcodings: SAVESTA and RESSTA.
(PPC Technical Notes #13, pp.81-83.)

85) Tung, C.C. The 'Personal Computer': A Fully Programmable Pocket Calculator.
(Hewlett-Packard Journal, May 1974, pp.2-7.)

86) Whitney, T.M., Rodé, F. & Tung, C.C.
The 'Powerful Pocketful': An Electronic
Calculator Challenges the Slide Rule.
(Hewlett-Packard Journal, June 1972, p.2ff.)

87)* Wilkins, L. HP-41 Machine Development Lab.
(PPC Journal, V9N3P27, 1982.)

- Wilkins, L. & Lind, P. (44)
A Proposed M-Code Standard.
(PPC Journal, V9N3P40, 1982.)

- Wilkins, L. & Lind, P. (45)
M-Code, Hardware and Software.
(PPC Southwest Conference, Proceedings, January 1983, pp.22-27.)

Various other articles in the HP-Journal are worth consulting:
On the Wand - January 1981, and on the original HP programmable, the HP-9100A, September 1968. Forthcoming issues of PPC Technical notes are planned to contain many articles on microcode, on microcode writing and already written routines.

APPENDIX G: Manual Use of LOADP

The use of LOADP

Convenient methods are given in Section III for the use of LOADP, but they require the use of functions already loaded into the MLI. The following procedures, though tedious except for the shortest of routines, allows that loading without the need for the use of auxiliaries. This is not a full SDS ROM compiling system, but rather a compromise. There are restrictions on the structure of programs that may be LOADP'ed, reflected in the double loading procedures that must be followed.

(1) The program must start with a global label, otherwise the first two bytes of MLI RAM where the program is loaded will be turned to rubbish, and the program will not be able to be COPY'ed.

(2) The address in the MLI at which loading is to begin has first to be placed in X. The area in the RAM memory of the MLI must, of course, be free for loading. Take care not to overwrite existing programs. Execute LOADP. On the prompt "LOADP _", press ALPHA, key in the name of any global label in the program to be LOADP'ed, and terminate entry by pressing ALPHA again. If the program pointer is in the program file to be loaded, simply press ALPHA twice. (Compare COPY, CLP, etc.) LOADP then completes execution. The first, unrevised copy is placed in the MLI RAM.

(3) The Function Address Table must now be revised manually. To do so:

i. The FAT address of the first label is two more than the address in X used when LOADP is called. This address must be added to the FAT, and the # of functions, in the second word of the MLI RAM, increased by one. Note that the first word of the pair of address words in the FAT must begin with 2, it must be of the form 2XX (the first bit of the ten must be set). If there are any more global labels in the program, they must be added as follows:

ii. Delete the first global label of the program in 41c RAM.

iii. GTO the global label in the MLI, SST to the next global label (GTO .nnn may be used, but not GTO "NAME". The Function Address Table has not yet been revised.) Switch to RUN mode, execute GETPC, DECODE the result. This address, less one, must be placed in the FAT for that label. Repeat the process for any remaining labels, noting them as you go, and revise the FAT to add the addresses, not forgetting to increase the number of functions in the second word. This number has to include the name of the XROM image, as well as those of its contained functions. Remember also to make the first word of an address start with a 2. (See Appendix D.)

(4) If there are any XEQ's in the program that is being loaded, calling global labels in the same program, they should now be changed to XROM's. (The same has to be done for any which call routines or programs in other as yet unloaded user code/RPN programs. Note, however, that calls to global labels are much slower than calls to local labels. If the calls are made in the same program file, it is better to have a local label immediately after the global label, and call the program or routine using the local.) To do so:

i. If it has not already been done, LOADP the program, with its XEQ's, into the MLI.

ii. Put the XROM addresses of all the global labels in the copy of the program, as just loaded into the RAM of the MLI, in the Function Address Table.

iii. In the RAM program, which should still be in the 41c, change all the XEQ's to XROM's, where necessary. Operating on each global label in the program, called from within the same program, delete it. Stepping through the program, delete and rekey each of its calling XEQ's (changing them, in so doing, into XROM's). Restore the label, and repeat the operation for each of the remaining (called) global labels.

The first global label will have been deleted at step ii, and will now need to be replaced, if this has not already been done.

iv. Re-LOADP the program to the same address in the MLI. (LOADP will PACK, and compile all XEQ's and GTO's before writing to the MLI - and say so.)

v. Revise the Function Address Table in the same way as before. (The location of some of the global labels may have been changed. XROM's take up only two bytes, and may have replaced XEQ's of two or three times the length.)

(5) There must be enough room for the program to fit in between the loading address in the external RAM of the MLI, and its end (at XFFF). Failure to observe this will result in a futile attempt to write to the next 4k page, and overwriting of the interrupts. Running any program so loaded is absolutely guaranteed to produce crashes.

(6) There must be an END, rather than the .END. to the file from which the program is to be loaded. Loading from the last file in memory will leave wasted nulls before its ROM END.

Error messages:
If an MLI is not connected, "NO RAM" is displayed.

FUNCTION INDEX

References in arabic numerals are to page numbers, in small Roman to the four pages of the cover, in capital Roman to Sections, and by capital letters to the Appendices. The index is intended to be useful, rather than exhaustive . . .

ASSEMBLER 3 functions

Application routines:

GENERAL INDEX