ON ANNOUNCEMENTS AND OTHER THINGS ABOUT ICODE

Below is one of our occasional announcements/advertisements - this time of a superb set of functions. If Dear Reader is wanting to take an intelligent interest (why DO interests have to be intelligent? Who would, or could take an UNintelligent interest? What on earth is the implied contrast involved???) in machine language programming, has a fair command of synthetics, but is not yet financially equipped to plunge into the full thing, but may think that EPROM's with his own customised favourite programs would be a good thing, and thinks a Jim Box of some kind would be worth it, then: during those long Winter Nights, instead of playing the Gramophone(Phonograph/Victrola/ . .) he may Invest in an ICODE EPROM set and approach the mastery of Malmacronian with a Linguaphone manual and 41c. (I am reliably told that such stuff as this is is eventually intelligible. One of .ED.'s 5,000 indulgences, I guess.) But the real purpose with which I started to type all this was to remark that after the following bit of blatency there is the opening section of Paul's Manual (not yet automated) for the ICODE EPROM set - the 'Brief Overview'. After that puff comes The Real Thing. *Said .ED.*

icicicicicicicicicicicicicicicicicicicic

THE I-CODE EPROM SET IS NOW AVAILABLE FROM MELBOURNE

The EPROM set comprises 101 functions which enable you to perform operations on 2 scratch registers (maintained and managed by the EPROM as mini programs) including:

1. Bit or nybble rotations and shifts of data
2. Decimal or hexadecimal operations on selected fields within the scratch registers
3. Storing or recalling of bytes to or from RAM or ROM (including emulated ROM-- ROME or now called XRAM)
4. Virtually all microcode tasks can now be performed with I-CODE routines, and you have the advantage of single-stepping, editing inserting etc. Furthermore, only an EPROM box is required- XRAM or ROME is not necessary to enable you to perform tasks normally needing microcode.
5. Most sytem data is instantly retrievable into the scratch registers eg current size, number of key assignment registers used, number of free IO registers, location of curtain and .END. etc. etc.
6. Commands are supplied which permit exchanges with and bit level manipulation of system and user flags.
7. Many program pointer control functions are provided (eg for creation of indefinite numbers of subroutine calls)
8. Many, many more utility functions eg clear alarms, clear selected user registers, CAT 2 beginning at a selected XROM number) etc. etc.
9. Multiple scratch registers can exist, and they can be exchanged with the XYZT or Alpha registers etc.

A manual describing all of the commands (18 pages) is supplied with the EPROM set, and 17 pages of application routines are included to start you off.

The EPROM set costs $35 Australian including post and packing. Please only send cheques in Australian currency to: Paul Cooper.* Overseas personal cheques unfortunately cannot be cashed here.

* See back cover for address.

BRIEF OVERVIEW OF I-CODE AND THE ICODE EPROM SET

If one was given the task of permitting the user to operate machine code like instructions without having to write machine code routines (and therefore not requiring an MLDL or MLI), one might proceed to emulate the machine code functions by a variety of functions which could be called up in a sequence to perform the required task from within a user code program. Thus one would only have to have a ROM or EPROM containing the required functions and the emulation would work (although it would be slower than actually working in machine code). The above strategy has the difficulty that there are many many machine code instructions and a 4K EPROM can only have 64 directly callable functions. This means that only 64 functions could be coded for, and even so, parameters would have to be passed to many of these functions (eg RCR 1, RCR 7, RSHFA S&X, RSHFA M etc.). This leads to the second complexity: 4K is not a lot of room to code in such functions. I am sure, however, that it could be done, and in fact I am currently working on this approach for a second set of EPROMs (ICODE 2).

Instead of taking a direct emulation approach, I decided to perform a partial emulation of the CPU by providing scratch registers for bit, byte and whole register manipulation of data. I then provided a pivotal function (ICODE) which can be used to access a further 33 functions via parameters passed to it (from the keyboard or from program memory). These functions would not be direct CPU functions, but would instead perform mini tasks which can be included into larger programs. For example, functions to return the number of key assignment registers used, the current size, the curtain address etc. are provided. Although such functions can be performed by synthetics, and some have been already available in machine code in other ROMs/ EPROMs, this set of functions is CONSISTENT with the use of the scratch registers provided, and do not impede the operation of ANY normal or synthetic instructions in a user program. Thus, this combination of CPU emulation instructions and utility functions is known as I-CODE because they are intermediate in speed and utility between machine code and user code.

It is true that many of the tasks which could be written by an I-CODE program (see appendix) could be coded in machine code, BUT that would require :

1. An MLDL or MLI containing emulated ROM so that machine code routines can be written to RAM in the device.

2. An intimate knowledge of how machine oode works (eg how to set up an F.A.T., how to use the CPU registers, flags etc.). Until a machine code book comes out, this knowledge is not gained overnight by a beginner (I know that from bitter experience).

3. An assembler/dissassembler is required if one is not to go mad while trying to write machine code routines.

Even after all the above, machine code can NOT be single stepped with execution (not quite true- I have written a machine code routine to do this, but it is difficult), and can not be easily edited or altered. Furthermore, debugging is often difficult. Experience overcomes many of the foregoing points, but I-CODE overcomes most of them very simply:

With I-CODE:

1. You need only have an EPROM box, as no simulated ROM is required.

2. You can use as many scratch registers as you like for data operations, or just for additional storage (non-normalising of course).

3. You can debug, single-step and edit any program containing I-CODE functions, and unlike sythetics, this will never cause improper operation of the calculator.

While I-CODE may fulfil some of the requirements to be called a new language for the calculator, it may be helpful to regard it as an expanded set of mutually supporting functions. Thus I-CODE does not replace normal user code or sythetics, but it does add speed and power to many programs which would otherwise use cumbersome techniques in their operation. If you are firmly entrenched into writing machine code routines, then I-CODE may not be for you. However, consider this: would you purchase a 4K EPROM set containing only certain programs (eg as per a machine code version of PPC ROM), or would you prefer to have a set of tools which you can combine to perform most tasks you would wish a machine code routine to do?

The rest of this manual will describe the functions of the I-CODE set, and will discuss the creation and operation of the scratch registers. The central ICODE function itself has been described in detail in PPCTN # 16. Finally, some application routines are provided merely to show the scope of uses to which I-CODE could be put. I am sure that people reading this information will have many more imaginative ideas than the ones I have presented. Bar code is also included for these programs.

icicicicicicicicicicicicicicicicicicicicicicicicicicicicicicicic

MICROBEE DRIVE TELLS HP-41c ITS INNERMOST SECRETS

Time now to rush out for a quickie before the main feature starts (I Would Marry You If ICODE). There we find Michael Thompson blazing, waving a strip of thermal paper. All agog at the latest marvels we are about to have revealed to us, we hold our breath as he speaks of 1.2 Mb drives crashing about him, the RS 232 interface speaking from the Microbee to the 41c, the 41c valiantly trying to make sense of the flood of bytes flowing in, but sympathetically assisted by the kindly Michael it eventually produces, on the very same loop as that from which the RS232 told its all, this veritable miracle of facsimiles of the screen of the Microbee, all prettified and intelligible to all. "They said it could not be done," said Michael, "but between us this is now demonstrated to be False." Having so said, he placed an armlock on .ED., forcing him to swear to present it to all TN readers, waiting for such a SIGN for years. At the Right, then, Dear Reader (voyour?), let thine eyes feast. But don't ask me how, ask Michael. If you don't have a printer with your personal computer, why - a 41c might prove just the thing.   And now the bells are ringing, the main feature is about to commence.

drop dead .ED.

beecatbeecatbeecatbeecatbeecatbeecat

```
Drive A: ---------+
----41cROMS 001----+
---- * => R/O -----+
---302k-Bytes Free
ASSEM1  .BIN___6k |
EXT-I/O .BIN___6k |
ILPRT1E .BIN___6k |
SDS-1A  .BIN___6k
ASSEM3  .BIN___6k |
HPILDEV1.BIN___6k |
MLEPRMIH.BIN___6k |
SYSROM0G.BIN___6k
EPROM   .COM___2k |
HPILDEV2.BIN___6k |
PLOT1A  .BIN___6k |
SYSROM1F.BIN___6k
ERAMCO  .BIN___6k |
ICODE   .BIN___6k |
PLOT2A  .BIN___6k |
SYSROM2F.BIN___6k
EXT     .ASM__74k |
ILBURHIB.BIN___6k |
PPC/MELB.BIN___6k |
WM      .HLP___4k
EXT     .BAK__74k |
ILMODIH .BIN___6k |
SDS     .BIN___6k |
XFCHIB  .BIN___6k
----------24 Files +-
--------32 Extents +-
--------------------+-
--274k-Bytes Used
```

INTERMEDIATE CODE HAS LIFTOFF!!! *Paul Cooper (9910)*

Good news needs quick delivery - for this pathetic reason (just thought up on the spur of the moment) this very latest material from Paul's hands has been inserted (using intermediate code - what else?) in the near start of this issue of TN. It is later than other material of his which is here, and will be in TN#17, but deserves to be first. Paul and the Melbourne Chapter will be making EPROM sets of ICODE available fo a minimal charge, and they will be available in France - and through most of Europe, from Jean-Daniel Dodin. See the announcement in this issue. For the rest, start thinking how to exploit the power of Paul's excellent software. We will be publishing many of the routines of ICODE in TN for Serious Students of The Art. Watch these spaces. *.ED.*

icicicicicicicicicicicicicicicicicicicicicicic

The conception of Intermediate Code was announced in TN#14 (pp.72-78) and a note mentioning minor revisions in the implementation was on the foot of p.78. The earlier version used status registers a and b for scratch purposes, though keeping the program pointer undefiled. Since then, as a result of fasting and profound meditation (staring at the HP-41c navel), it was decided to use registers in an ICODE created program file at the end of program memory. The deliberations behind this will surface elsewhere.

THE USE OF SCRATCH REGISTERS BY INTERMEDIATE CODE ROUTINES

Nearly all of the I-code routines require consultation of a special pair of registers designated Sa (scratch reg a) and Sb (scratch reg b) respectively. These scratch registers are created just below the permanent END, but are configured with a header label and a new permanent END with a newly constructed CAT 1 linkage. Thus each of the routines can simply examine the registers above the permanent END to see if they hold the correct header LBL (consisting of two unkeyable chars - hex 01). If not, then either the scratch registers do not exist, or the user has added in a new program after the last END (thus cutting off access to the scratches). The I-code routines will create new scratch registers if appropriate, or will indicate 'NONEXISTENT' if they require the contents of the scratch registers for their operation. In the latter case, the user can still recover the previous scratch contents by making use of two utility routines. The first does the equivalent of a manual GTO^{T}✱✱ and places the program pointer at the first set of scratch registers found by climbing the global chain. A second utility routine may then be used to copy the contents of this set of scratches to a new set synthesized just below the new permanent END. Of course the registers may be duplicated, erased, swapped and of course decoded and examined at any time. Many of the I-code routines allow direct entry of data to or from the status registers and/or user registers, as well as allowing many operations to be performed on the data within the registers.

In more detail, the scratch registers are configured as shown below:

```
                 ← Text byte
LBL^T ✱✱ F14
Sb
Sa
.END. ← Right justified
```

Within the scratch registers, various fields have been specified to allow more convenient use of the contents:

```
Byte #          6              5            4         3     2       1   0
Nybble #      13       12       11   10     9      8    7    6 5 4    3 2 1 0
Sb field    Pntr 1   Pntr 2  Flags Temp byte  Loadable Address   Temp store
                               :     store       byte
                               :
                 Bit #  3     2'     1       0

                 Use: +/-   Pntr Carry Dec/Hex
```

The Sa register is very simple and is basically used to hold the operand upon which the operation is to be performed.

I will describe now the various fields within Sb:

<u>Nybble 13</u>: This nybble stores the current value of pointer 1. For example if pointer 1 was currently pointing to nybble 8 of Sa, then nybble 13 of Sb would hold 8. Although this nybble can hold values between 0 and 15, values 14 and 15 have no relevence for operations on Sa, as Sa holds only 14 nybbles. (#13 to #Ø)

<u>Nybble 12</u>: This nybble is as for nybble 13 except that it refers to pointer 2.

<u>Nybble 11</u>: This nybble is further subdivided into the four component bits. Bit 3 is set if the operations are required to be SUBTRACTIVE. It is clear if they are to be ADDITIVE. Bit 2 is set if pointer 2 is selected and is clear if pointer 1 is selected. The selection of active pointers is important for the I-code operations which increment or decrement the pointers. Bit 1 is set if a preceding operation resulted in an arithmetic overflow, or if a test was true. It is otherwise clear. Note that it is ONLY cleared by arithmetic or test operations, or by direct manipulation of Sb nybble 11. Bit Ø is set if decimal mode is selected, clear if hex.

<u>Nybbles 10 & 9</u>: These two nybbles provide a convenient storage for two nybbles. Various commands have been provided which enable these nybbles to be exchanged with other Sb register nybbles.

<u>Nybbles 8 & 7</u>: These two nybbles constitute a major operating field within the Sb register. These nybbles (byte 3-4) can be loaded from a running program, can be arithmetically operated on (in HEX or Decimal) and can be exchanged or added to other bytes within the Sa register. Nybble 7 is also used as part of the address field for specifying 4 nybble addreses (eg ROM addresses). See also below.

<u>Nybbles 6, 5 & 4</u>: These three nybbles are the address field of Sb reg. They may be operated on, exchanged with register Sa fields or may be grouped with nybble 7 to specify 4 nybble addresses. The use of this field allows RAM addresses to be specified and ROM words to be written to an MLI or retrieved and placed into Sa register.

Nybbles 3, 2, 1 & 0: These nybbles are basically used for temporary storage of various bytes and nybbles, and the contents of this field may be exchanged with many other Sb fields. These nybbles are also used to permit tests based on the bytes present in nybbles 4 to 7 to be performed (see below).

I-CODE commands

Name	Function	No. Bytes
Register Operations		
XY-S	Places the contents of the X reg into Sa, and the Y reg into Sb. Creates Sa and Sb if not present.	22
S-XY	Places the Sb reg to the Y reg, and Sa to X reg. Gives 'NONEXISTENT' if Sa and Sb do not exist.	24
XY<>S	Swaps Y reg with Sb and X reg with Sa. Gives 'NONEXISTENT' if Sa and Sb do not exist.	33
ARGIN	Places the X reg into Sa. Creates Sa and Sb if they do not exist.	15
ARGOUT	Places the Sa contents into the X reg and lifts the stack if enabled. Gives 'NONEXISTENT' if Sa and Sb do not exist.	15
ARGEX	Swaps the X reg contents with Sa. Gives 'NONEXISTENT' if Sa and Sb do not exist.	20
GET	Gets the contents of the register specified by nybbles 4,5 and 6 of Sb, and loads into Sa. Gives 'NONEXISTENT' if that register does not exist, or if Sa and Sb were not found.	40
PUT	Places the contents of the Sa register into the register specified by nybbles 4,5 and 6 of Sb. Gives 'NONEXISTENT' if Sa and Sb do not exist, or if the specified register does not exist.	30
EXA B	Swap Sa and Sb register contents. Give 'NONEXISTENT' if Sa and Sb do not exist.	25
Field exchanges		
SWPADR	Swap Sb nybbles 4-7 with Sb 0-3. Gives 'NONEXISTENT' if Sa and Sb do not exist.	23
SWPTMP	Swap Sb nybbles 7-10 with Sb 0-3. Gives 'NONEXISTENT' if Sa and Sb do not exist.	23
SWPTR	Swap Sb nybbles 7 and 8 with Sb nybbles 12 and 13. This enables the pointers to be loaded via the loadable byte (nybbles 7 and 8) in one or two steps. 'NONEXISTENT' if Sa and Sb are not found.	19

Name	Function	No. Bytes
SWPBYT	Swap Sb nybbles 7 and 8 with 9 and 10. Provides convenient storage for a byte which has been loaded into the Sb register. 'NONEXISTENT' if Sa and Sb are not present.	20
SWPFLG	Swap Sb nybble 9 with the Sb flag nybble 11. Allows the flag statuses to be modified or examined via byte operations.	24
SWPSPC	Swap nybbles 9, 10, 12, 13 of Sb with nybbles 4 - 7. Provides a means of storage for intermediate calculations based on the address fields (4-7, and Ø-3). Gives 'NONEXISTENT' if Sa and Sb do not exist.	29

Field exchanges between registers

EXFLG	Exchange user flags 0-7 with the bits in nybbles 9 and 10 of Sb. Enables bit level manipulation on nybbles 7 and 8. Also provides convenient storage and retrieval of user flags without using user registers. Gives 'NONEXISTENT' if Sa and Sb do not exist.	26
CPUFLG	Recalls or stores the status of the bits in Sb nybbles 0-3 into or from the internal CPU flags 0-13. If Sb is in minus mode (bit 3 of Sb nybble 11 set), then CPUFLG sets bits 0-13 of Sb nybbles 0-3 as per the CPU flags. These flags control PRIVATE status, stack enable, program running in ROM etc. If the Sb register is in PLUS mode (bit 3 of nybble 11 clear), then CPUFLG takes the bits 0-13 of Sb nybbles 0-3 as the new CPU flag settings. After setting the flags, it ALSO does the following task: the CPU A register is loaded with the contents found in the Sa register, the CPU C register is loaded from the X register, the CPU B register is loaded from the Y register, and CPU M and N registers are loaded from the Z and T registers respectively. Furthermore, the nybbles loaded into the Sb 4-7 are treated as a ROM address, and a machine code jump is made to that address. Thus CPUFLG (in PLUS mode) can be used to branch to any desired mainframe or plug-in ROM address, but you can jump there after having set-up the required CPU registers and flags.	

If you just wanted to alter the status of the CPU flags, and you don't want to perform a mainframe entry, then the best plan is to place a harmless mainframe address into nybbles 4-7 of Sb (see D-H below for one way to do this). An example might be the mainframe standby polling routine. Obviously CPUFLG is best used with care after consulting the NOMAS listings. Using it, you could (for example) branch to the XASN entry point to enable you to perform synthetic key assignments with machine-code speed. Obviously the keycode bytes etc. would have to be set up into the required stack and Sa registers prior to the call to CPUFLG. When used with care, CPUFLG is one of the most powerfull I-CODE routines.

Name	Function	No. Bytes
EXNYB	Exchange Sb nybbles 4-6 with Sa 0-2. This is useful to load or unload the address field of the Sb reg. Gives 'NONEXISTENT' if Sa nad Sb not found.	25
EXBYTS	Exchanges Sb nybbles 4-7 with Sa 0-3. As for EXNYB but acts on a further nybble. Useful for loading addresses for operations of ROM addresses.	28
EXADR	Exchanges Sb nybbles 0-3 with Sa 0-3. Allows the temporary storage field of the Sb reg to be loaded or unloaded with a 4 nybble address (eg for a compare or arithmetic operation). Gives 'NONEXISTENT' if Sa and Sb were not found.	25
PTA<>B	Exchanges Sb nybbles 7 and 8 with the nybbles pointed to in the Sa register by the currently selected pointer. Exchanges at the pointer and pointer + 1 position. Gives 'NONEXISTENT' if Sa and Sb do not exist.	41
<u>Pointer operations</u>		
SLCT 1	Clears bit 2 in nybble 11 of Sb register, so that pointer 1 is selected. Gives 'NONEXISTENT' if Sa and Sb not found.	23
SLCT 2	Sets bit 2 in nybble 11 of Sb reg so that pointer 2 is selected. Otherwise performs as SLCT 1.	9
SETP	Sets the selected pointer to the value of the nybble 7 in the Sb register. It does not check the input range of the nybble 7, and does not alter the carry flag. Gives 'NONEXISTENT' if Sa and Sb do not exist.	31
SWPTR	See description given under field exchanges above. This function also does not validate the pointer values and does not disturb the carry flag.	19
PNTROP	Checks the status of bit 3 in nybble 11 of Sb. If it is clear, then it increments the value of the currently selected pointer. If it is set, then it decrements the pointer value. In both cases, the pointer argument is validated and cannot exceed 13. The carry flag will be set if a pointer operation tries to exceed this value. Gives 'NONEXISTENT' if Sa or Sb do not exist.	56
CLRBYP	Clears the Sa register between selected pointers. Gives 'NONEXISTENT' if Sa and Sb do not exist.	35
DECODA	Decodes the Sa register between selected pointers and appends the characters to the alpha register. This function enables you to decode only the central bytes of a register without having to rotate or truncate	

Name	Function	No. Bytes
DECODA (continued)	the data. It also does not clear the alpha register, thus allowing more flexible use than the normal DECODES. Gives 'NONEXISTENT' if Sa and Sb not found.	70
Shift and rotate instructions		
SHFA	Shifts the Sa register left or right one nybble. The direction of shifting will be right if bit 3 of nybble 11 of Sb is clear, and will be left if it is set. This is easy to remember: PLUS mode means RIGHT, MINUS mode means LEFT. Gives 'NONEXISTENT' if Sa and Sb not found.	21
SHFOP	As above, except shifts a variable number of bytes determined by the value found in nybbles 7 and 8 of Sb.	36
SHFBIT	As above, except shifts left or right a variable number of bits as determined by the value in nybbles 7 and 8 of Sb.	40
RBYB	Rotates the Sa register left or right the number of bits specified by nybbles 7 and 8 of Sb register. Direction of rotation is right if in '+' mode, and left if in '-' mode. Gives 'NONEXISTENT' if Sa and Sb are not found.	36
Jumps		
NCSKIP	Skips the next instruction in the user program if the carry flag is clear. Clears the carry flag. Gives 'NONEXISTENT' if Sa and Sb not found.	23
CSKIP	As above, except skips if the carry flag is set (see bit 1 of nybble 11 in Sb).	20
****** ROTA	This belongs with the shifts and rotates above. It rotates left or right the Sa register by one nybble only. See above for how to set the direction of rotation.	39
Load and store commands		
LDB	Loads the following two characters in program memory as two nybbles to be placed into Sb nybbles 7 and 8. The characters must follow the LDB instruction as alpha characters, and must be in HEX. LDB does the equivalent of a CODE on the two chars, but this operation does not disturb the alpha register. If characters other than 0-9 and A-F follow LDB, then 'DATA ERROR' will arise. An example should make it clear: LDB ⊤ØB will load hex 0B into nybbles 7 and 8 of Sb reg. If you single step the program with LDB in it, then the LDB instruction will automatically jump the following alpha characters, thus preventing the argument from	147

Name	Function	No. Bytes
	disturbing the alpha register. LDB in combination with SWPTR is a fast way of setting both pointers (eg for a DECODA instruction).LDB must always be followed by <u>two</u> chars.	
XBYTE	Takes an argument found in the X register and converts the absolute value into two hex nybbles which are then loaded into the Sb nybbles 7 and 8. Gives 'ALPHA DATA" if an alpha string is in the X reg. Gives 'OUT OF RANGE' if a decimal value greater than 255 is in the X register. Creates the Sa and Sb registers if they are not found.	30
FETCH	If the Sb register is in HEX mode, then FETCH consults the nybbles 4-7 in the Sb reg.,and recalls the word at the ROM address specified by those nybbles. The word is placed into the nybbles 0-2 of the Sa register after first rotating the Sa register 3 nybbles left..Nybbles 13 and 12 of the Sa register are then replaced with the HEX digits 1 and 0, so that when the Sa register is placed into a user register, it will appear as an alpha string and will therefore not be normalized. Up to four ROM words may be recalled into the Sa register before it becomes full. This operation is made easier because FETCH automatically increments the address in Sb nybbles 4-7. If the Sb was in DEC mode (bit 0 of nybble 11 set) then FETCH treats the nybbles 4-7 of Sb as a RAM program pointer. It retrieves the byte at that RAM location then resets the program pointer to point to the next RAM byte. The retrieved byte is placed into Sa nybbles 0 and 1 after rotating the Sa register two nybbles left. As before, the Sa register is then made into an alpha string. Thus up to 6 RAM bytes may be recalled with FETCH before the Sa reg is full. The program pointer may be placed into the Sb register nybbles 0-3 by a utility program described later.	70
STOW	This is a near inverse of FETCH. If the Sb register is in HEX mode, then the word in nybbles 0-2 of the Sa register is written to an MLDL or MLI at the ROM address in Sb nybbles 4-7. The ROM address is incremented, and the Sa register is rotated 3 nybbles right. If the Sb register was in DEC mode, then STOW places the byte in the Sa register nybbles 0 and 1 into RAM at the location indicated by the program pointer at Sb nybbles 4-7. The program pointer is updated to point to the next byte, and the Sa register is rotated two nybbles right.	88
CODA	This function performs a CODE on the alpha register, but places the coded nybbles into the Sa register rather than the X register. It creates the Sa and Sb registers if they do not exist. This function is different to the CODE in ASSEMBLER 3 in the following way: only the number of coded nybbles derived from coding the ALPHA register are REPLACED with the nybbles in the	

Name	Function	No. Bytes
CODA	Continued Sa register. For example, if the Sa register contained: ABCDEF01234567 and the ALPHA register contained the chars: 9876, then after the CODA operation, the Sa register would be: ABCDEF01239876. The ASSEMBLER 3 CODE function would have cleared the Sa register, and then entered the 9876 nybbles. Thus CODA may be used to selectively load the Sa register without losing previous contents (especially when used in conjunction with the rotate and shift commands). Note that leading nulls will not be loaded, and that a cleared ALPHA register will load nybble 0 of Sa with HEX 0. For example, if the ALPHA register held: 009876, then the contents of the Sa register would still be as in the example above. That is, the leading nulls are not loaded. To enter leading nulls, you can use one of the other load commands, or you can preclear part of the Sa register (see CLBYP). ICODE 03 (HEXKB) works as per CODA, but it functions from the keyboard.	
XADR	This function takes a decimal number from the X register, converts it to HEX, then loads it into the address nybbles of Sb (4-6). Thus a register location or part of a ROM address could be loaded in one step with this command. Maximum input number is 4095.	
Arithmetic functions		
SET-	Sets bit 3 of nybble 11 in the Sb register so that subsequent operations will be subtractive, and shift operations will be to the left.	18
SET+	This is the default condition. Bit 3 on nybble 11 of Sb is cleared. Operations will be additive, and shifts will be to the right.	18
SETDC	Set bit 0 of Sb nybble 11 so that operations will occur in decimal mode. Also FETCH and STOW will assume RAM program pointer operations.	19
SETHX	Clear bit 0 of nybble 11 in Sb. This is the default so that all arithmetic operations will occur in HEX. Also, FETCH and STOW will assume ROM addresses.	19
CURTOP	Adds or subtracts the current curtain address to or from Sb nybbles 4-6 depending on the setting of bit 3 of Sb nybble 11. Always performs in HEX mode regardless of the setting of bit 0 of Sb nybble 11.	42
ADDROP	Adds or subtracts Sb nybbles 0-2 to or from Sb nybbles 4-6 in HEX or DEC mode, depending on the setting of bits 0 and 3 in Sb nybble 11. Clears the carry flag at the start and sets it if an arithmetic overflow occurred.	42
BOPA	Adds or subtracts the Sb nybbles 7 and 8 to or from the Sa register between the current pointers in HEX or DEC mode. Initially clears the carry flag, but sets it if an arithmetic overflow occurred. The current pointers are determined by the values found in Sb nybbles 13 and 12, and bits 0 and 3 in nybble 11 of Sb determine the mode of operation (see above).	83

Name	Function	No. Bytes
D-H	Takes a number in the X register as a decimal, and converts it it to hexadecimal, and places the result back into the Sa register right justified. The absolute value of the input number is used. 'ALPHA DATA' error will occur if an alpha input is given. Numbers greater than 65535 will result in 'OUT OF RANGE'. The Sa register is left shifted apart from the nybbles 0-3 which contain the answer. This function may therefore be used to load the Sb nybbles 4-7 (when followed by EXBYTE) without disturbing ALL of the Sa register.	
H-D	This function takes the nybbles of Sa numbers 0-2 and treats them as a HEX argument. This argument is then converted to a decimal number in proper BCD format which is placed back into the Sa register. This may then be read out with an ARGOUT for example, or stored in a user register etc. Maximum value of the number will be 4095.	135
H-D4	Nearly the same as above, but expects 4 nybbles of argument. That is, it takes nybbles 0-3 of Sa and converts it to a decimal number. The maximum value of this number will be 65535 for an input of FFFF.	70
ADR+-	This function increments or decrements nybbles 4-6 of Sb in HEX or DEC mode. It clears the carry flag, and sets it upon arithmetic overflow. Consults nybble 11 of the Sb register for its mode of operation.	40
B+-	As for ADR+-, but acts on nybbles 7 and 8 of the S b register.	39
BOP	Adds or subtracts nybbles 9 and 10 of Sb to or from nybbles 7 and 8 of Sb in HEX or DEC mode. Initially clears the carry flag, and sets it if an arithmetic overflow occurs.	48
Tests		
BYTE?	Sets the carry flag (bit 1 of nybble 11 in the Sb reg) if nybbles 7 and 8 are equal to nybbles 9 and 10. If not then the carry flag is cleared.	32
ADR?	Sets the carry flag if nybbles 0-2 of the Sb regsiter are equal to nybbles 4-6. Clears the flag if not.	24
EQADR?	As above, but tests nybbles 0-3 with nybbles 4-7 of the Sb register.	17
<ADR?	Sets the carry flag if nybbles 4-6 are less than nybbles 0-2 of the Sb register. Clears flag if not.	19
LTADR?	As above, but tests nybbles 4-7 for being less than 0-3.	32
PTR=B?	Sets the carry flag if the currently selected pointer is equal in value to the nybble 7 of the Sb register. Clears the flag if not.	39
A=R?	Sets the carry flag if the contents of the Sa register between the current pointers is equal to the contents of the RAM register specified by the RAM address in nybbles 4-6 between the pointers. Clears the carry flag if not.	61

Name	Function	No. Bytes
A<R?	As above, but tests if the Sa contents is less than the selected register between the pointers. Note that this is a direct less than test, and makes no assumptions as to the format of the Sa register. For example, if a BCD number is in the Sa register, and the pointers select for the entire Sa reg, then it will NOT be the same as X<Y?	7
A=0?	Sets the carry flag if the Sa register is zero between the current pointers. Clears the flag if not.	48
XROM?	Takes the value in nybbles 7 and 8 of the Sb register as an XROM number (in HEX), and looks for that ROM in the ports. If it finds it, then it sets the carry flag, and the start address is placed into nybbles 4-7 of the Sb reg. If the carry flag is cleared, then that ROM is not present. (ROM? now starts looking at address 3000, and is able to cope with the X Functions located there in the new 41cX.)	49

<u>Other</u>

ICODE	This is the function called by all the above routines. It can be used as a function in its own right however. When called from the keyboard it prompts for the entry of a two digit decimal number which corresponds to a further function you want executed. This can be one of the 23 utility functions listed below. If placed into a program, it examines the bytes following it for instructions on what it is supposed to do. If the following bytes are anything other than an F1 or F2 text byte, then it checks for the scratch registers. If not found, then they are created. If found, then they are cleared. If the byte following ICODE is an F1 text byte, then it checks the next byte. If it codes for the character 'D', then the current scratch registers are dissolved, and the .END. is replaced onto the previous program. If the character is an 'E', then it dissolves or erases as many scratch registers as it can find, then it returns. If the character was a 'D' by the way, and no scratch registers were found, then 'NONEXISTENT' results. 'E' never gives an error message. If the character was an '7', then the current set of scratch registers will be duplicated, and you may now operate on a new set of scratch registers without disturning the old set's contents. The old contents may be recovered following a dissolve command or by using the swap scratches utility function described below. If the letter 'S' follows the ICODE function, then a new set of scratch registers are ALWAYS created regardless of whether old ones existed. This permits ICODE routines to call up other ICODE routines as subroutines without disturbing the first lot of registers. The subroutine should start with ICODE 'S', and should end with ICODE 'D' which will dissolve only the scratches used by the subroutine. The calling routine will then be returned to with its scratch registers (if any) intact. See also the utility functions (below) for further manipulations of the scratch registers. If a single character other than the above appears, then 'DATA ERROR' will occur. If an F2 text byte is found, then ICODE expects two alpha chars which will code for an argument representing the number of the utility function to	

be performed (see below). For example, ICODE ØØ will run the zero'th utility function in the same way as entering two zeroes in response to the keyboard prompt when ICODE is executed manually. ICODE validates the characters, and if any other than Ø - 9 are found the message 'DATA ERROR' results. 574

Utility functions

These are a set of functions which are executed by providing a two digit decimal argument to the ICODE routine. They may be run from the keyboard (by filling in the two prompts after executing ICODE) or they may be run from a program by placing ICODE into the program and following it with 2 decimal alpha characters corresponding to the argument required. See ICODE above for further details. Also see LDB for the behaviour of ICODE and LDB during single stepping of programs containing them. Note also that to place ICODE into a program, you will have to enter 2 digits before ICODE will place itself into program memory. These digits disappear as soon as they are entered, however, and you must follow with the 2 alpha character arguments as explained above. This strange behaviour has to do with the main-frame prompting routine not expecting external ROM functions which are prompting to be programmable.

Function No.	Description
00	Does a goto scratch register operation. It looks for the first scratch register up from the .END., and places the program pointer at the synthetic label that each of these scratch registers has. It is most useful in combination with the following routine if you have lost access to you scratch registers (eg by editing a program or adding new programs etc.). It is best to actuate manually from the keyboard or else it will execute the scratch register as a program. This does no harm except that it will load the contents of the scratch registers into the alpha register.
01	Copies the 14 bytes following the NEXT byte into a new set of scratch registers. Let us say that you have lost access to the scratch registers which you previously had. Execute ICODE 00, then SST. You will now be placed at the text byte F14 which precedes the follwing 14 bytes (which are the Sb and Sa register contents in that order). Execute ICODE 01 and you will have created a new copy of the scratch registers with the old register's contents as a currently usable set of scratches. The old set are not disturbed in any way. You might use ICODE 01 as a more general way of placing program bytes into scratch registers for later decoding etc., although the FETCH function may be a more controllable way of doing this.

FOUNDING FATHER IS RETURNING TO HIS HOMELAND

Graeme Dennes was a member of PPC while most of us were still innocent, and was one of the 16 who met in July 1979 to establish PPC Celbourne. His help then and until he left us in 1980 to work in the USA was quite invaluable - but now he is to be back here for a few days in Melbourne, before going to Queensland. He arrives on April 13th. Address not yet known. While in Georgia he fell victim to an IBM PC, and founded a user group in Macon. If possible, we will organise some kind of occasion to meet him before he again leaves. John .ED.

Function No.	Description
02	This returns the correct number of free registers as a HEX number and places it in the nybbles 4-6 of the Sb register. It is the equivalent of doing a GTO.. then going into PRGM mode. Unlike the F? function of the PPC ROM, it always accurately reflects how many registers are available.
03	This calls up the HEXKB routine as found in ASSEMBLER 3, but it places the CODEd argument directly into the Sa register. It will create Sa and Sb if they were not found.
04	This is a routine euphemistically called 'MUSIC'. It is basically as published in PPCTN 15 P73. It treats the first byte in the Sa register as an argument for the TONE function, then shifts a copy of the Sa contents two bytes right, and treats the next byte as a variable length pause (F will give a pause of about 0.5 seconds). It then shifts the contents two bytes right, and checks to see if the register is now empty. If it is, then the routine exits, if not, it cycles back to the TONE/pause routine. Thus a maximum of five TONE's and four pauses may be executed by one call to ICODE 04.
05	This suspends the key assignments by clearing the bits in the key assignment flag registers. It was written by Neil Hunter-Blair.
06	This restores the key assignments (including programs) that were suspended by SKEY. It requires the presence of the HP-IL or the extended functions module. It was written by Neil Hunter-Blair. See PPC TN 15 P31 for ICODE 05 and 06.
07	Returns the address of the last used key assignment register to nybbles 4-6 of Sb in HEX format. Creates the Sa and Sb regs if they were not found.
08	Returns the address of the last free IO register (counting down from the .END.) into nybbles 4-6 of the Sb register. Creates Sa and Sb if not found.
09	Swaps the last two sets of scratch registers if two or more were created (see the ICODE write-up on how to have more than one set of scratch registers). Gives 'NONEXISTENT' if no scratches are found, but does not check for the existence of BOTH sets before swapping, so careful when you use it.
10	This function takes a decimal number in the X register, and regards it as the XROM number from which to start an CAT 2 catalogue. If the ROM does not exist then: 'NONEXISTENT'. This is as per Jean-Daniel Dodin (see PPCTN 15 P58).
11	Clears all timer alarms. Written by Neil Hunter-Blair. See PPCTN 15 P31.
12	Returns the address of the summation registers to Sa nybbles 3 to 5, and the current SIZE to nybbles 0 to 2.

Function No.	Description
13	Suppresses trailing zeroes on a number. It places the display into SCI format, then slects the minimum number of digits to be displayed so that all trailing zeroes are blanked. It is meant to save on storage space if you are placing alpha numbers into ascii files. Its a quick and dirty routine at present though, because it also places the calculator into DEG mode.
14	Clears registers specified by a number placed into the X register in the format: BBB.EEE. If the BBB register does not exist, then 'NONEXISTENT'. If the EEE register is greater than the last existent register, then this function will clear up to and inculding the last existent register, but will not go further (eg will not clear extended memory). 'DATA ERROR' results if EEE is less than BBB.
15	Performs a shift of the alpha registers. The M register is lost, N goes to M, O goes to N, and the first 3 characters in P are moved to O, and the rest of O is cleared. The P register is cleared. This function is meant to be used after the next two ICODE functions in much the same way as ASTO and ASHF are used.
16	Stores the contents of the alpha M register into the Sa register. It creates the scratch registers if they are not found.
17	Swaps the alpha M register contents with the Sa register. Gives 'NONEXISTENT' if Sa and Sb are not found.
18	Goes to the .END. It moves the program pointer from ROM into RAM, and lands at the .END. All subroutine calls are cleared.
19	Tiny little routine to return the mantissa of a number. So easy to do in M-code that I just had to put it in.
20	Moves the program pointer to the global label found in the X register (eg after an ASTO X) then stops program execution. This is the programmable equivalent of a GTO label command performed from the keyboard, and does not require STOPs to be placed immediately after global labels if you want the user keys to become active (eg for local labels).
21	Recalls the program pointer to Sb nybbles 4-7. The nybbles could then be moved to nybbles 0-3 or otherwise manipulated prior to a FETCH or STOW command. Creates the Sa and Sb regsiters if they were not found. Sets the carry flag if PP was in ROM, else clears.
22	Places the address of the .END. into the Sb register nybbles 4-6. Creates the Sa and Sb registers if they were not found.
23	Stores the Sb nybbles 4-7 into the status reg b program pointer. If the carry flag was set, then the PP will be set to ROM.
24	Performs HEX addition of the Sa and Sb registers, with the sum ending up in Sa. This operation functions over the entire registers and does not set the carry flag if the digits overflow.
25	Subtracts in HEX mode, the Sa register from the Sb register, with the difference going to Sa. Occurs over the entire registers. Carry is not set on underflow.

26 Recalls status registers a and b and places the contents into Sa and Sb registers respectively. Creates Sa and Sb if not already present. When combined with ICODE 27, can be used to extend the subroutine call stack to an indefinite length.

27 Recalls the Sa register and places into status register a. Recalls the Sb register and places into status register b, but does not overwrite the program pointer in register b. Thus this can be used with ICODE 26 to extend the ability of the HP41 to make subroutine calls.

28 Clears all subroutine calls, but does not alter the program pointer.

29 Transfers the alpha registers to the stack in the following fashion: M to X, N to Y, O to Z, P to T.

30 Transfers the stack contents to the alpha register in the following fashion: X to M, Y to N, Z to O, T to P.

31 Exchanges the alpha M and N register contents with the stack X and Y register contents respectively.

32 Sets the message flag then places the contents of the Sa register into the display. In other words, it is like a VIEW Sa command, except that it does not print the display and it does not normalize the Sa register contents. Thus if you want to display a message (up to 6 chars) to the user during a program, but you don't want to disturb the ALPHA reg (or the stack or numeric registers) then you can set up the Sa register (eg with ASTO X then ARGIN) so that ICODE 32 will display the information. If a BCD number is in the Sa register then it will be displayed as a number. That is, the normal rules for displaying ALPHA or numeric nybbles in the X reg. are followed. This includes respecting the display format (eg number of decimal places, radix etc.).

<u>FURTHER DATA ON SOME I-CODE ROUTINES</u>:

<u>SWPFLG</u>

This function swaps Sb nybble 9 with nybble 11. Internal bits 0-3 of the Sb nybble 11 may thus be set with one operation. For example, if HEX 7 was in nybble 9, then after SWPFLG, the carry flag would be set, the Sb reg would be in + mode, and would be set at pointer 2, and it would be in DECIMAL mode.

<u>EXFLG</u>

This function permits the bit manipulation of data stored in the scratch registers. It exchanges user flags 0-7 with the Sb 9-10 nybbles. The most signif. bit of Sb 10 will correspond to flag 0, and the least signif. bit $_{\wedge}^{of 9}$ will correspond to flag 7. This command could be used to save user flags 0-7 prior to some program run, for later restoration, or it can be used to manipulate the bits in the Sb 9-10 nybbles. For example, say we want the second bit in nybble 9 to be set. EXFLG SF 06 EXFLG would perform that task. Of course the nybbles in Sb 9-10 may have been placed there from the Sa register or other places etc. etc.

COMMANDS WHICH CONSULT NYBBLE 11 IN THE Sb REGISTER:

CPUFLG	ADR?
SLCT1	EQADR?
SLCT2	<ADR?
SETP	LTADR?
PTA<>B	PTR=B?
PNTROP	A=R?
SHFA	A<R?
SHFOP	A=0?
SHFBIT	XROM?
RBYB	ICODE 21
ROTA	ICODE 23
NCSKIP	
CSKIP	
FETCH	
STOW	
SET-	
SET+	
SETDC	
SETHX	
CURTOP	
ADDROP	
BOPA	
ADR+-	
B+-	
BOP	
BYTE?	

EXAMPLES OF THE USE OF I-CODE FUNCTIONS:

GET

An example of the use of this function might be to retrieve a non-normalized number from a user register. Alternatively, it can be used as a replacement for synthetic instructions. For example, if Sb 4-6 holds HEX 00E, then the GET function would recall the 14th register from the bottom of memory. This is equivalent to RCL d, except that the X register needn't be disturbed, and flag operations may be performed on the Sa registers without fear of SST destroying the flag register contents.

PUT

In similar fashion, STO c is a nasty function to have in a program if you are single stepping without carefully watching the stack contents. HEX 00D placed into Sb 4-6 followed by PUT will store the Sa contents into register c, and the stack may be analyzed as required during SST.

SWPTR

This function is most useful for setting up both pointer positions with the one command. Have a look at the example ICODE programs. You will often see it used with LDB which first loads Sb 7-8. SWPTR then exchanges nybble 7 with the pointer 1, and nybble 8 with pointer 2. Thus the pointers can both be set, and tested with this command.

EXAMPLES OF ROUTINES USING ICODE FUNCTIONS

```
01+LBL "CUP
"
02 ICODE
03 X<0?
04 SET-
05 ABS
06 ARGIN
07 D-H
08 EXNYB
09 CURTOP
10 LDB
11 "03"
12 RCL c
13 ARGIN
14 SET+
15 RBYB
16 EXNYB
17 SET-
18 RBYB
19 ARGOUT
20 STO c
21 END
```

This routine performs as per the CU routine in the PPC ROM ie it takes a decimal value from the X register (+ or -) and adjusts the curtain address in the C status register accordingly. The first ICODE command is followed by a test. Since this is not a text byte, the ICODE command will create and/or clear the scratch registers. If the number was a negative, then SET- will set the Sb register for subtraction. ARGIN moves the absolute X reg value (see step 05) into the Sa reg. D-H converts it to hex, and EXNYB places it into Sb 4-6. Curtop then adds or subtracts the curtain address to or from this value. LDB 03 places hex 03 into Sb 7-8. RCL c retrieves the current status register contents, and ARGIN places this into Sa. SET+ ensures that Sb is now set for additions regardless of the previous status. RBYB rotates the Sa register clockwise (because we are in + plus mode) the number specified in Sb 7-8 (hex 3). EXNYB places Sa 0-2 (the curtain address) into Sb 4-6, and the new curtain is exchanged into Sa 0-2. The Sa register is rotated anti-clockwise 3 digits, and the new register c contents are placed into the X register for transfer to status reg c via STO c.

```
01+LBL "IOF
REE?"
02 ICODE
03 ICODE
04 "08"
05 SWPADR
06 ICODE
07 "22"
08 SET-
09 ADDROP
10 EXNYB
11 LDB
12 "20"
13 SWPTR
14 "IO FREE
="
15 DECODA
16 AVIEW
17 H-D
18 ARGOUT
19 END
```

The following routine demonstrates the use of the utility functions provided with the I-CODE routines to permit useful system information to be obtained. In this case it is desired to find out how much free IO space is present ie the number of registers between the last KAR or ALARM and the .END. We require the output to be in HEX appended to the alpha register, and we want the output in X as a decimal number. First we clear the Sa and Sb registers by ICODE. The second ICODE has 08 as its argument, so function 08 is run. This returns the lowest free IO register to Sb 4-6. This address is placed into Sb 0-3 by SWPADR. ICODE 22 returns the .END. address to Sb 4-6. We set the Sb register for subtraction by SET-, then ADDROP performs .END. minus last free IO. This gives us the HEX number of free IO registers in Sb 4-6. EXNYB places this to Sa 0-2. LDB 20 places hex 20 to Sb 7-8, then SWPTR places this byte into nybbles 13 and 12 of Sb (the pointers). We now load up the alpha register with a message, then DECODA decodes the Sa register between the pointers (ie Sa 0-2) and appends the characters to the alpha reg. AVIEW displays the result, H-D converts Sa 0-2 to a normalized BCD number in Sa, and ARGOUT places this number into the X register.

```
01+LBL "OH"
02 "OCT?"
03 PROMPT
04 DEC
05 GTO 01
06+LBL "DH"
07 "DEC?"
03 PROMPT
09+LBL 01
10 CF 29
11 FIX 0
12 SF 21
13 ARGIN
14 D-H
15 LDB
16 "30"
17 SWPTR
18 "HEX="
19 DECODA
20 AVIEW
21 "DEC="
22 ARCL X
23 AVIEW
24 RTN
25+LBL "HD"
26 SF 21
27 "HEX?"
28 ICODE
29 "03"
30 H-D4
31 ARGOUT
32 CF 29
33 FIX 0
34 "DEC="
35 ARCL X
36 AVIEW
37 OCT
38 "OCT="
39 ARCL X
40 AVIEW
41 END
```

The following I-CODE converts Octal to Dec and HEX, HEX to DEC and OCT, and DEC to HEX. It starts by setting up the alpha register for the input of an octal number. This is converted to decimal by the mainframe. A jump is then made to LBL 01. If you entered at the global label DH, then it is assumed you wish to convert from decimal to hex. Thus the appropriate alpha prompt is set up for that.

Both routines come to LBL 01 for common processing. Flag 29 is cleared for alpha formatting of numbers, FIX 0 is set, and flag 21 is set to allow the printout of alpha views.

The decimal number in the X register is placed into Sa via ARGIN, and is converted to HEX by D-H. LDB 30 places hex nybbles 30 into Sb 7-8. SWPTR places these nybbles to Sb 31 and 12, thus setting up the pointers. The alpha register is then set up, and DECODA places the Sa register decoded between the pointers after the alpha string in the alpha register. This is then AVIEWed. The Decimal number is placed into the alpha register via a further R/S if desired.

For conversion from HEX to DEC (and to OCT), flag 21 is again set to allow AVIEW to print. The alpha reg is set up for an input, and ICODE 03 functions as HEXKB, with the input going directly to the Sa register. More than 14 characters may be entered, but only the rightmost four will be converted to a BCD normalised number in the Sa register by H-D4. This number is transferred to the X reg by ARGOUT. It is then displayed and converted to OCT and further displayed as required.

```
01+LBL "H-"
02 "ARG 1?"
03 ICODE
04 "03"
05 EXAB
06 "ARG 2?"
07 ICODE
08 "03"
09 ICODE
10 "25"
11 LDB
12 "30"
13 SWPTR
14 "HEX SUM
="
15 DECODA
16 AVIEW
17 H-D4
18 ARGOUT
19 END
```

Perfomms HEX subraction without disturbing the user STACK. The first step is to set up the alpha reg ready for the first argument to be input via ICODE 03 (ie HEXKB) with the coded result being placed into the Sa register. EXAB swaps the Sa and Sb regs. ARG 2 is now input, again via ICODE 03, then HEX subtraction is performed on the Sb - Sa registers via ICODE 25. For convenience, I have then used LDB to place hex 30 into the Sb 7-8 nybbles, then SWAPTR to set the pointers to these nybbles, so that DECODA will then append the nybbles Sa 0-3 into the alpha register. This pair of nybbles may be changed to any values including D0 if the entire Sa register is to be decoded. The HEX subtraction is perfomred over the entire Sa and Sb registers. Finally, H-D4 converts the Sa 0-3 nybbles to a BCD number and ARGOUT places it into the X reg. This step may be removed if the required res-lt is wanted in HEX only,

```
 01*LBL "XRA
Y"
 02 CF 01
 03 CF 21
 04 FS? 55
 05 SF 21
 06 CF 23
 07 "HEX?"
 08 AON
 09 PROMPT
 10 AOFF
 11 FC?C 23
 12 SF 01
 13 SWPADR
 14 CF 29
 15 FIX 0
 16 NCSKIP
 17 GTO 01
 18 SETDC
 19 LDB
 20 "10"
 21 SWPTR
 22*LBL 04
 23 SWPADR
 24 CLX
 25 ARGIN
 26 .9
 27*LBL 02
 28 "BYTE "
 29 ARCL X
 30 "├="
 31 ISG X
 32 X<> X
 33 FETCH
 34 FC? 01
 35 DECODA
 36 FC? 01
 37 AVIEW
 38 FC? 01
 39 GTO 02
 40 H-D
 41 ARGOUT
 42 ARCL X
 43 AVIEW
 44*LBL 03
 45 CLX
 46 ARGIN
 47 RDN
 48 GTO 02
 49*LBL 01
 50 LDB
 51 "20"
 52 SWPTR
 53 SETHX
 54 GTO 04
 55 END
```

This program permits the byte by byte analysis of a program resident in RAM or ROM. The dissection can be in decimal (flag 1 set) or HEX (flag 1 clear). The program pointer must first be placed into scratch register Sb 4-7. This can be done by ICODE 21 when you have placed yourself at the program you wish to dissect. The carry flag will be set if the PP was in ROM. Flag 21 is set to enable printer operation, and flag 23 is clear for alpha number formatting. Thus when you XEQ XRAY, you should press an alpha character if you wish HEX decoding, or press R/S if decimal.

Flag 1 will be set accordingly.
The PP in Sb is placed into Sb 0-3 by SWPADR, then the carry flag status is checked by the NCSKIP step. If carry was set, then GTO 01 will be executed.

If not, then we now SETDC to set the FETCH command to decode a RAM byte rather than a ROM byte (see the FETCH command). LDB 10 loads hex 10 into Sb 7-8, and SWPTR sets these nybbles as the pointer positions. SWPADR replaces the PP back to Sb 4-7, and X is cleared to allow the ARGIN command to clear the Sa register. 0.9 is entered into the X register as the byte counter.

The alpha register is then loaded with the alpha string for outputting, and the byte number is appended with ARCL X.

FETCH is then executed to bring the byte at the current PP into the Sa register 0-1. The PP is moved to the next location. If flag 1 is clear, then DECODA is executed to append the HEX byte Sa 0-1 to the alpha register. If flag 1 is clear, then the alpha register is AVIEWed, and a jump is made to LBL 02. If flag 1 was set, then a H-D conversion is done on the Sa contents, and a BCD number is now in Sa to be output to X with ARGOUT. This is then appended to the alpha register with ARCL X and is then AVIEWED. The Sa register is then cleared by the CLX ARGIN sequence, and a jump is made to LBL 02.

If the program has jumped to LBL 01, then the PP was in ROM, and the pointers must be set up to decode a 3 nybble ROM word. This is done by LDB 20 then SWPTR. Hex mode is set to make sure that FETCH recalls from ROM rather than RAM (see the FETCH writeup). A jump is then made to LBL 04 which can now take care of the formatting of the output series of ROM words. FETCH autoincrements after each operation, so that it steps through ROM one word at a time.

```
BYTE  1=260
BYTE  2=1C8
BYTE  3=000
BYTE  4=0F7
BYTE  5=000
BYTE  6=050
BYTE  7=052
BYTE  8=050
BYTE  9=04C
BYTE  10=04F
BYTE  11=054
BYTE  12=18C
BYTE  13=1F6
BYTE  14=04E
BYTE  15=041
BYTE  16=04D
BYTE  17=045
BYTE  18=020
BYTE  19=03F
```

This is an XRAY vision of the printer PRPLOT user code program resident in the printer ROM. It is decoded here in HEX form.

```
BYTE  1=C6
BYTE  2=00
BYTE  3=F5
BYTE  4=00
BYTE  5=58
BYTE  6=52
BYTE  7=41
BYTE  8=59
BYTE  9=A9
BYTE  10=01
BYTE  11=00
BYTE  12=00
BYTE  13=00
BYTE  14=00
BYTE  15=00
BYTE  16=A9
BYTE  17=15
BYTE  18=AC
BYTE  19=37
```

This is an XRAY vision of the XRAY program itself, starting with the global label.

The nulls here mean the program wasn't packed. The decoding was done in HEX here.

```
BYTE  1=198
BYTE  2=0
BYTE  3=245
BYTE  4=0
BYTE  5=88
BYTE  6=82
BYTE  7=65
BYTE  8=89
BYTE  9=169
BYTE  10=1
```

This is the same as above (ie a view of the XRAY program itself), but the decoding has been done in decimal.

```
 01+LBL "NRC
L"
 02 ICODE
 03 D-H
 04 EXNYB
 05 CURTOP
 06 GET
 07 ARGOUT
 08 END
```

Performs a non-normalising recall from the register specified in the X register. The first ICODE command is followed by a byte other than an F1 or F2, so it clears the contents of the current scratch registers (or creates them if they are not present). D-H converts the decimal number in the X reg to HEX, and places the nybbles into Sa 0-3. EXNYB places nybbles 0-2 (the reg address into Sb 4-6. CURTOP adds the curtain address to this register address, and GET recalls the contents.
    ARGOUT places the value into the X register (and lifts the stack.

Notice that there is another way to load the Sb 4-6 nybbles: replace steps 3 and 4 with XADR which directly loads the HEX conversion of the X reg number into Sb nybbles 4-6.

This I-CODE routine neatly combines the SIZE? function with a routine to compute how many free registers are available. Firstly ICODE 02 is executed which calls up the mainframe routine to return (in HEX) the number of free registers to Sb 4-6. EXNYB brings these nybbles to the Sa0-2 nybbles, and H-D converts these nybbles to a decimal number. ARGOUT moves this number to the stack, and flag 29 and FIX 0 are used for the alpha recalling of this value. The SIZE? function is then executed to return the number of free registers which is then brought into the alpha register. Thus the X reg holds the SIZE and the Y reg holds the number of free registers after execution.

```
 01+LBL "MEM
"
 02 ICODE
 03 "02"
 04 "REG "
 05 EXNYB
 06 H-D
 07 ARGOUT
 08 CF 29
 09 FIX 0
 10 ARCL X
 11 SIZE?
 12 "├ SZ "
 13 ARCL X
 14 ARGIN
 15 AVIEW
 16 END
```

Note: ICODE 12 now returns the SIZE to nybbles Ø-2 of Sa, hence it could be used in the above routine in place of the function SIZE?, should that not be available. (See page 37 above.)

```
01+LBL "MUSIC"
02 "40064057048"
03 ICODE
04 CODA
05 ICODE
06 "04"
07 STOP
08 "43050043050043"
09 ICODE
10 CODA
11 ICODE
12 "04"
13 STOP
14 "40039048039048"
15 CODA
16 ICODE
17 "04"
18 END
```

This demonstration program shows how ~~three~~ different sets of tone sequences may be simply generated using the ICODE 04 function. The Alpha register is set up with the required bytes as shown in step 2. The ICODE 04 function will read these nybbles from right to left: first 2 nybbles as a tone, next nybble as a pause. Thus we will have TONE hex 48 then no delay (value of nybble is 0), then tone 57, no delay etc. Step 3 makes sure that the Sa register is empty before the CODA step (see write-up for CODA).

In the course of an investigation of the True Algorithm for the computation of ROM/XROM checksums, Paul wrote the following, desperately slow program which will do the job accurately, but in rather unreal time. (So try doing it without benefit of ICODE? Well, with the aid of ASSEMBLER 3, etc., etc.) As Paul is not around at this time of compilation of this issue, evil .ED. stuffed it in. It is, by the way, an example of a program/routine which is best written in microcode (remember that?), mainly for its speed. The checksum computation effected by the Diagnostic ROM takes only about twenty seconds to deliver its report. Routines such as this are very much longer running. .ED.

+=+

```
 01♦LBL "CHS
UM"
 02 CF 01
 03 ICODE
 04 "3FF"
 05 CODA
 06 EXAB
 07 "XROM NO
?"
 08 PROMPT
 09 XBYTE
 10 ROM?
 11 CSKIP
 12 RTN
 13 FETCH
 14 .003
 15 ENTER↑
 16 .999
 17 GTO 06
 18♦LBL 01
 19 ISG X
 20 GTO 06
 21 ISG Y
 22 GTO 07
 23 FS? 01
 24 GTO 10
 25 SF 01
 26 RDN
27 .096
28 GTO 06
29♦LBL 10
30 EXAB
31 "400"
32 CODA
33 EXAB
34 ICODE
35 "25"
36 LDB
37 "20"
38 SWPTR
39 "CHECK="
40 DECODA
41 AVIEW
42 RTN
43♦LBL 07
44 RDN
45 .999
46 GTO 01
47♦LBL 06
48 FETCH
49 SWPSPC
50 EXNYB
51 ROTA
52 ROTA
53 ROTA
54 EXADR
 55 ADDROP
 56 EXADR
 57 <ADR?
 58 CSKIP
 59 GTO 02
 60♦LBL 03
 61 EXNYB
 62 SWPSPC
 63 GTO 01
 64♦LBL 02
 65 ADR?
 66 NCSKIP
 67 GTO 03
 68 SET-
 69♦LBL 05
 70 ADDROP
 71 <ADR?
 72 CSKIP
 73 GTO 05
 74 SET+
 75 GTO 03
 76 END

LBL'CHSUM
END
172 BYTES
```

STOP__PRESS

At the left is a sample of the output from the teletype printer interfaced to Paul Cooper's HP-41c through his MLI. See page 64.

```
LISTING OF THE EPROM FUNCTIONS REQUIRED TO TALK TO A TELETYPE PRINTER.

XROM 29
25 FUNCTIONS                                          REG 000 = 3.141593
                                                      REG 001 = 23.140693
                                                      REG 002 = 1.382588E23
00: E864 PROM                                         REG 003 = 3.718316E11
01: E3E8 ACA                                          REG 004 = 609779.9775
02: E562 ACCHR            14: E79F COPYROM            REG 005 = 780.884100
03: E5A3 ALV              15: E69B OUTROM             REG 006 = 27.944304
04: E5B2 PAPER            16: E1A6 PRREG              REG 007 = 0.035785
05: E503 ACX              17: E705 INROM              REG 008 = 0.189170
06: E445 CHARIN           18: E6A0  -                 REG 009 = 2.299182
07: E486 CHROUT           19: E5CA PRSTK
08: E093 PRA              20: E10D PRX                21.29E3
09: E8C6 TEST             21: E755 KLBL?              EXAMPLE OF PRA
10: E386 PRBUF            22: E4BE SKPCHR
11: E6A0  -               23: E8E0 HEXKB
12: E6A0  -               24: E927 ♦OUTNLI            ACCHR
13: E6A1  -                                           21.02.1984
                                                       6:50 PM
```

```
 01+LBL "FLG
X"
 02 ICODE
 03 ENTER↑
 04 ABS
 05 55
 06 X<>Y
 07 -
 08 XBYTE
 09 14
 10 XADR
 11 GET
 12 RBYB
 13 PTA<>B
 14 SWPBYT
 15 EXFLG
 16 RDN
 17 RDN
 18 X<0?
 19 CF 07
 20 X>0?
 21 SF 07
 22 EXFLG
 23 SWPBYT
 24 PTA<>B
 25 SET-
 26 RBYB
 27 PUT
 28 LASTX
 29 END
```

This program functions nearly as per IF in the PPC ROM. That is, it will set or clear the flag number found in the X reg. If the number is positive, then the flag will be set. If neg., then the flag will be cleared. Firstly, the number is subtracted from 55 to give the bit position from the right hand end of reg d.

The function XBYTE then places the decimal number as the converted HEX nybbles into Sb7-8. XADR places the X reg number (14 here) into Sb 4-6, so that the following GET will recall the 14th register (reg d). RBYB rotates the Sa register right the number of bits loaded into Sa 7-8, and PTA<>B swaps the Sa0-1 nybbles with Sb nybbles 7-8 when the pointer is at 0 (it is because we started with cleared scratch regs). SWPBYT then swpas nybbles Sb7-8 with Sb9-10, and EXFLG exchanges Sb9-10 with user flags 0-7. Now we can set or clear the rightmost bit by setting or clearing user flag 7 (the leftmost bit would be user flag 0 etc.). The setting is dependent on the sign of the flag number. EXFLG and SWPBYT now bring the bits back into the Sb 7-8 nybbles, and PTA<>B swaps them with nybbles Sa 0-1 again. We now do a SET- so that the next RBYB will rotate the Sa left by the number of bits we previously rotated it right, and then PUT places the contents into the reg d, because the Sb 4-6 nybbles are still undisturbed.

<u>*ONE HUNDRED AND ONE THINGS TO NOTE ABOUT ICODE*</u>

On the opposite page an XCAT2A of the ICODE XRAM image in my MLI is followed by a listing of the rest of the functions which it contains, in much the same format. (I shall take to the grave the details of the routine which produced this - and no peeking, either.) It is, I think, fair to list these as separate functions. Though the double (DOUBLE!!!) duty ICODE itself (themselves?) functions both as name of the XRAM (as it still is) image and as function which will accept 33 postfixes from the keyboard as well as from text lines when in a program, it will also accept four postfixes from text lines, alpha postfixes, in program. The resulting operations are so varied that one should not think of ICODE, except nominally, as a prefix like (say) STO, or SF IND, which always do the same to the gizmo signified by the postfix. But Paul's postfixes here just act as signposts, directing the flow of microcode execution initiated by the operation of his function, to differing stretches of code, to carry out differing tasks. Since the number of such operations is limited only by the XRAM/XROM space, there is no particular reason, except lack of perspicuousness, why a useful image should not have a name only - in the way that HP count these things, have no functions at all. But the name, with Cooperian postfixes, could access as many functions as could be crammed into a 4 or 8K image. The cost is in the user code program bytes - the ICODE is a normal (well, almost normal, for it is XROM 19,00) two byte XROM function, and its F1 postfixes also take two bytes, while its two digit, F2 post fixes take three.

Anyway it is something to ponder on: there are one hundred and one reasons why YOU, Dear Reader, should soon have an ICODE EPROM set to plug into your Jim Box/MLDL/MLI, etc., etc.

Said .ED.

+=+

```
XROM 19                    PAUL COOPER (9910)
101 ROUTINES

00: E00A ICODE
01: EEA1 BOP          35: E98C SWPFLG
02: E8A8 XADR         36: E9D5 EXADR          68: ICODE "0"
03: E315 ARGIN        37: E9F0 EXBYTS         69: ICODE "1"
04: E326 ARGOUT       38: EA0D CPUFLG         70: ICODE "2"
05: E336 XY-S         39: EA59 CLBYP          71: ICODE "3"
06: E34D S-XY         40: EA80 PTA<>B         72: ICODE "4"
07: E367 XY<>S        41: EAAB BYTE?          73: ICODE "5"
08: E30F EX           42: EACD CSKIP          74: ICODE "6"
09: E3ED EXFLG        43: EAE4 NCSKIP         75: ICODE "7"
10: E408 SLCT1        44: EAFA ADR?           76: ICODE "8"
11: E420 SLCT2        45: EB15 EQADR?         77: ICODE "9"
12: E429 SETP         46: EB24 GET            78: ICODE "10"
13: E44B PNTROP       47: EB56 PUT            79: ICODE "11"
14: E482 SET-         48: EB7A ADDROP         80: ICODE "12"
15: E495 SET+         49: EBE8 SWPSPC         81: ICODE "13"
16: E4B3 SETHX        50: EC00 LTADR?         82: ICODE "14"
17: E4C9 SETDC        51: EC22 <ADR?          83: ICODE "15"
18: E4DD ROTA         52: EC38 PTR=B?         84: ICODE "16"
19: E505 LDB          53: EC61 ARGEX          85: ICODE "17"
20: E586 CURTOP       54: EC77 SHFOP          86: ICODE "18"
21: E5B3 EXNYB        55: EC9C SHFA           87: ICODE "19"
22: E5CD H-D          56: ECB5 ADR+-          88: ICODE "20"
23: E5F6 D-H          57: ECDD B+-            89: ICODE "21"
24: E63A RBYB         58: ED74 BOPA           90: ICODE "22"
25: E661 H-D4         59: EDCA A=R?           91: ICODE "23"
26: E6F8 DECODA       60: EE09 A<R?           92: ICODE "24"
27: E745 SWPTR        61: EE14 SHFBIT         93: ICODE "25"
28: E854 CODA         62: EE3D A=0?           94: ICODE "26"
29: E868 XBYTE        63: EE70 XROM?          95: ICODE "27"
30: E925 FETCH                                96: ICODE "28"
31: E951 STOW         64: ICODE "D"           97: ICODE "29"
32: E976 SWPADR       65: ICODE "E"           98: ICODE "30"
33: E98F SWPTMP       66: ICODE "t"           99: ICODE "31"
34: E9A7 SWPBYT       67: ICODE "S"          100: ICODE "32"
```

THE SUICIDAL REPLY

Nick Reid (8703)

The above, or a reasonable facsimile, was sent by Paul to Nick, who replied in his usual style, with critical and practical comments. On some things he is right, on others there is disagreement. One thing that is clear: ICODE is a programmer's EPROM, and a frustrated mcode programmer at that. But its routines are very powerful indeed, and several allow flexible entry to the operating system, even without benefit of any XRAM, much in the manner of Jim-the-ROM's GOTOROM function which so puzzled me when I first tried that pioneering ROM. But here is Kami, as he was on that momentous day, November 15th., 1983: Be warned, however, .ED. put in the headings.

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

Dear Paul,

Thanks for the I-CODE stuff, which arrived today: will peruse it at the weekend.

Location of the F14 ICODE Buffer

When developing the idea of the F14 buffer (actually F5, F14/F5, F14/ . . . /(LBL "IF ANY")*/END/" (concerning that LBL "IF ANY": the user decides if he wants a label, and if so makes it to his own specs for reading, writing, etc. No need for "unkeyable": just delete), but Gordon [Rowell] at first forgot how to get the first F14 spaced correctly), the notion of putting it at the TOP of program memory was to guarantee the "microcode/user code interface" scratch registers the ABSOLUTE MAXIMUM POSSIBLE creatability, accessibility, manipulability and deletability - from all angles: keyboard and user code, as well as microcode, and with OR WITHOUT an EPROM box on.

The argument for the top of memory

This latter point is important when, for portability, or whatever reason, one is separated from an EPROM box (or doesn't possess one, and has to play the microcode game in snatches on borrowed gear, as several students in PPC have to do) as it enables a certain amount of microcode (or ICODE) analysis and preparation, allowing all memory-allocation and user-code preparation to be set up/tested with the 41c alone. The curtain's "stabilising" of the "placing" of the registers enables all of this, and ensures that (say) a STO into the regs won't be foiled by a displacing of the FK byte by any passing PACK.

Time of creation and access of buffers and buffer stacks

There are many other passing et ceteras. To figure out how I might precreate a buffer (properly spaced) at top of memory, say, with the 41c only, is easy. Putting one after the present .END. needs a lot more thought. From the I-code point of view, all this is immaterial: ICODE is a plug-in page of 10 bit code (well, almost immaterial: I could think of useful situations where RPN access to I-code droppings, or vice versa, would be valuable), BUT the top-of-program-memory buffer was proposed as a general-purpose microcode-rpn interaction buffer (no doubt one is needed), before news of I-code, the first type of such "interaction" is seriously implemented. Its major disadvantage, from the microcode side of such interaction, is the time taken to create new pairs of buffers (or single buffers with F7), having to push down any program, as key assignments have to push up any I/O, but, given its conception, the greatly increased ease of all other sorts of control/access/etc. seemed to override that. The entire field of microcode/user code interaction ("RPN-LIB"???), of which I-code is the first "codified" part, is (I think) communally interesting enough (useful enough) to propose a bit of damned standardisation about basic requirements, like common storage areas, etc., so at least a gnat's worth of portability between individual 41c-enhanced systems remains.

The alternative - top to bottom of memory swapping

Although I personally wouldn't worry about the time of creation at the top of program memory, seeing as it is more-or-less a one time thing: once they've been put there, so all else is equal, timewise: nevertheless, if you're stuck on doing it at the .END.for I-code, then I'd suggest that the notion of a general-purpose interaction-buffer requirement, and the maximisation of its interactibility, BE LOOKED AT. Given this very fast requirement, I think it goes (in a stack) at the top of program memory. Maybe it doesn't, but if it does, it would be worth having as basic. I-code routines, two of which (at least) swap the current I-code registers at the .END. for the second and third down under the curtain, so as to make the major "interaction" development to date easily loadable from the registers I set up in the train on the way here tonight . . . or whatever. It would mean being committed to a bit of "damned standardisation" in advance of its existence, but, as HP keep on demonstrating to us, that's how to advance its existence.

Gordon (The Count) Rowell repents, and the ProtoCODER gets ABS minded

The other news from Sydney is that G. Rowell is about to return to the 10 bit fold, it seems. The sniggers got to him. Apparently a new development from N. Crowle has interested him, given his almost-finished ProtoCODER adaption of ASSEMBLER 3. The said Nelson C. is even now, rummaging around with his Exacto, cutting traces and jumping here and there, so that all future ProtoCODERs write on "ABS" (1077, to be exact), which has a direct return to microcode, and, apart from the formatting of the instruction to LJ and add. C, is as direct to use from microcode as 040 is for MLDL's. Details of changes to update existing Proto Interfaces to be published soon.

FAT addressing is wider than was supposed [the SUMO table, then?]

Also new from Sydney is our learning from the navigation ROM that the function address table will locate an address with ANY amount of offset. So if the entry to ABC in a ROM is put in the table as

```
                    00A           (or 20A)
                    0BC
then
                    01A           (or 20A)
                    0BC
```

will send the pointer to the next ROM page up (ROM 0, if the table is in Page F), and

```
                    0FA           (or 2FA)
                    0BC
```

to the next ROM page down.

Dissembling disavower dissents from dissent

Apart from my interest in a general buffer, and the importance of talking to I-code through such a thing, I'll leave most of the correspondence to Gordon: I generally hate writing (& loathe typing), and in fact believe that all 'self-expression', including programming, is a disgustingly infantile activity for which we shall deservedly be put up against the wall when the Universe revolves. (The trouble with Swift was his incurably tolerant and kindly view of humanity.) The less of it I've got documented against me the better, I reckon.

Regards, Nick 8703.

+-+

Without any more ado about something, here is Paul in full cry, on November 21st.:

Dear Nick,

Thanks for your letter. John and I have read it with interest. Whilst your points about scratch register access are accepted, I still think that below the .END. is the best place for them. For one thing, if you call up subroutines which create scratch registers, you do not have to do all the housekeeping necessary to update the program pointer when the registers are shuffled down. However, I take your point that access may not be as easy when the EPROM box is not plugged in. There is no need to have a machine code routine to perform the scratch register exchange function you mentioned in your letter: such a task is perfect for I-CODE. Thus I have enclosed the following routine which takes the two registers at curtain -2 and curtain -3 (could be changed to curtain -1 and curtain -2 etc. etc.) and exchanges them with my scratch registers Sb and Sa respectively:

```
01♦LBL "SWA
P"
02 S-XY
03 ICODE
04 CURTOP
05 SET-
06 ADR+-
07 ADR+-
08 GET
09 X<>Y
10 ARGEX
11 PUT
12 X<>Y
13 ADR+-
14 GET
15 ARGEX
16 PUT
17 XY-S
18 END
```

I will leave you to figure out how it works! By the way, I think you received a somewhat preliminary version of my I-CODE documentation. Apart from several minor mistakes, the KASN program has been deleted, and in its place are routines to allow the user to enter a mainframe ROM address with complete control over how to load up the CPU registers and CPU flags. Plus there is a routine to allow Sb 4-6 to be loaded directly from the X register etc. etc. I will send the new documentation along with application routines to any who wish to purchase the EPROM set ($35). I also noted your comments regarding your horror of typing. We all have our worries about what we say on paper which may incriminate us when 'The Revolution Comes'. HOWEVER, some poor sod has to type the stuff up somewhere along the line, and that poor fellow has been John. As secondary editor and partial business editor for PPCTN, I have prepared the enclosed notice which will get sent to ALL who transgress the 'neat or typed BUT not scrawled' rule.* So having said my say, I hasten to add that all info will be read with interest, and if it is really topical etc. etc. then I or John will type it up. But PLEASE if you can, then type (we will even burn it after reading it if it will prevent your soul being jeopardised--John even offered to eat your material, but that was after a VERY long session of editing).

On other topics, glad to hear that Gordon has not deserted the HP41 scene. We are eagerly awaiting more EPROM madness from you all up there. Things are winding down here at the moment, as many of the club members are sitting exams. John and I have just finished a set of CAE courses which should enable the club to purchase the IL EPROM burner. I am looking forward to committing my MLI image to the safety of an EPROM set. Anyway, if I ramble on any more I'll get even more boring, so I'll stop right now,

Cheers,

Paul Cooper

Paul Cooper 9910
39 Brisbane St,
Ascot Vale 3032

* See below, p.57 for more on this matter.

TWO KEY ASSIGNMENT ROUTINES FROM ASSEMBLER 2 Michael Thompson (8496)

Before there was ASSEMBLER 3, there were first ASSEMBLER 1 and ASSEMBLER 2. A listing of the functions in ASSEMBLER 1 was published in PPCTN, #13, p.55, and a description of those of ASSEMBLER's 2 and 3 in PPCTN #14, pp.38-40, as well as in the PPCCJ, V10N3Pp9-10.

Most of the functions of ASSEMBLER 2 were the same as those of ASSEMBLER 3, but it did include two key assignment functions, entirely in microcode, called ASG and KA, for which there was no room in the later EPROM image. In addition, the early versions of ASSEMBLER 2 had sundry bugs which were purged from the counterpart routines of ASSEMBLER 3 before its release. The old ASSEMBLER 2 has now been debugged by Michael, and is available from him at the same price as ASSEMBLER 3, equally usable with the MLDL, but with room made available by omitting some of the functions, not needed for writing microcode/mcode. (For a function listing of ASSEMBLER 2, see PPCTN #17, p.70. The XROM number of the version you will get from Michael will not necessarily be 31 - Paul burned me a bug free version at the February meeting, and at my request, changed its XROM ID number to avoid conflict with that of ASSEMBLER 3. The routine listing below, however, was from an earlier version of ASSEMBLER 2, with the same XROM ID as ASSEMBLER 3, yet had to be effected using ASSEMBLER 3 routines. Quite a hassle! Because of this, the adresses of the following listing slightly differ from those of the debugged version.) The routines omitted are: LOADP, COMPILE, INSBYTE, STOBYTE, PUTPC, though it does retain RCLBYTE and GETPC, VIEWA and COPYROM. CLROM is there, under the name CLROME. Besides the ASG and KA, it has ROM>AS and AS>ROM, taken care of in ASSEMBLER 3 in a different manner.

ASG functions rather like the sophisticated routines Tapani Tarvainen and Gerard Westen have been writing, accepting alpha mnemonics for full synthetic assignments. In operation, execute it just as one does the mainframe ASN. It prompts 'ASG _', and now may be used exactly as if it were ASN, but on pressing ALPHA, it allows, and accepts, such mnemonics as 'STO M', 'RCL a', 'ISG Q', etc. On terminating alpha entry, it prompts for a key, just as does ASN, and once a key is pressed (shifted or unshifted) the assignment is completed. All valid mainframe prefixes are accepted (STO, RCL, ST+, XEQ, etc.), together with the now common status postfixes, except that, as in ASSEMBLER 3, '+' is used for the 'append' sign. (See the ASSEMBLER 3 Manual, p.8.)

KA is very like the PPC ROM MK in that it accepts the bytes and keycodes from the stack - from X. It is programmable, unlike ASG. The format is: aaa.bbbcc, in X, then execute KA, where aaa is the decimal code for the prefix byte, bbb that for the postfix, and cc is the keycode. Leading zeroes in bbb must be entered (KA 144/87, say, to the LN key, must be entered as 144.08751, rather than 144.8751). An invalid keycode gives the message 'KEYCODE ERR'. The stack is lifted, and X is transferred to Last X. *Use negative values for shifted keys.*

It was rather odd to compile notes for these functions, and to test them out while sitting in a camper van at Apollo Bay in Victoria, about a mile from the place where the first keycode to key assignment register byte routine was written in 1980 (KA 3 or 4, I think). Alas for those happy, innocent synthetic days!

John McGechie (3324)

kakakakakakakakakakakakakakakakak