PPC TECHNICAL NOTES #15

P P C   M I C R O C O D E   N O T E S   # 1 5

C O N T E N T S

REGISTERED BY AUSTRALIA POST PUBLICATION NO. VBG4080 (CATEGORY B)

M-CODE PROGRAMMING

Chris Rath (6287)

This article is stolen, with Chris' blessing, from the Proceedings of the Rhode Island New England PPC Conference, held on May 7th. and 8th. 1983. The novel nature of the new information given here will be apparent to anyone knowing anything of recent work in microcode. (This is the REAL name, in Malmacronian, of that which Chris perversely refers to as m-code.) Please notice that this material has been lightly .ED.ited by .ED. in consultation with Michael Thompson, who spent several hours at .ED.'s home, caressing the keys of .ED.'s Osborne to get this stuff into it. Had it not been necessary to so rekey - since the photocopy Chris sent to us had the effects of his using an emphasis pen noticed by the copier he used for us - neither of the criminals just mentioned would have tampered at all. But word processing is the standard invitation to such unsolicited sin. Beware, then. Send us stuff so beautifully typed, or printed, in exactly the format we can use, and **nothing** will happen to it apart from its appearing in print. (.ED. is very likely to respell 'EXISTANT's as 'EXISTENT's, and such things.) The moral is obvious: send important stuff, scrawled, or otherwise needing this kind of work, and .ED. will be his normal evil .ED. (He does it to his own stuff. Word processors are supposed to cut down on writing and revising time. In Malmacronia they seem not to do so.) Anyway: may the Chris forgive us . . .   Yr. Obt. Pedagogue-ing .ED., who has collected at the .END. all of the longer comments which would otherwise interrupt the text at the end. The notes are noted in the text by <1>, <2>, etc.

mcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmc

1(a) What is m-code?

The 41 is usually programmed in what I will define here as 'user-code'. This is the language of the familiar STO's, ENTER's, etc. These commands represent keystrokes, and we place sequences of these in memory as the programs we write. These programs perform specific tasks, and sets of related programs are frequently used, and grouped together. (E.g. the Complex Arithmetic programs in the PPC ROM.)

In actuality our familiar user-code is itself just a very large set of programs, and the language that **they** are written in is **m-code.** I.e. when we press a key, we are just indicating to the 41 which m-code program we wish to run.

Because m-code is a step down from user-code (down at the level of the 41 itself), we can do much more with m-code. But also inherent in the step down is the fact that every task must be carried out in smaller steps than we are used to in user-code. For example, STO Ø1: in m-code we must place the contents of the X-register in an internal working register, then select what is presently Register Ø1 (found by looking at the address of RØØ in status register c), and then finally put the data in register Ø1. But don't think that smaller steps always cause a complication of our task - some programming problems are simplified by the low level of m-code. Amongst the best known of these are the problems set by, for example, CODE and DECODE.

1(b) The m-code programmer's attitude.

One must be in a rather different frame of mind when programming in m-code, a frame of mind different from that appropriate to non-synthetic user-code programming. This 'new' outlook is very similar to that which serious synthetic programmers must have: one involving a respect for the integrity of the 41 operating system. We may not like the 'user-code' illusion that exists, but when we write an m-code program, we are writing a tool for use in that user-code dimension, and that being so, we are obliged to help maintain that illusion. <1>

As most of us are painfully aware, the operating system works to preserve this illusion at almost any costs: the normalization of registers, the cold start constant, etc. I am not suggesting that everyone who intends to program in m-code must examine and memorize the 41 and its peripherals and programs, only that when your m-code makes a change to something, you should be sure that the 41 operating system can cope properly with the change.

If you destroy the user-code program counter, then be sure to invalidate the line counter, and reset the user program counter to something which the operating system can accept, such as the address of the .END.

Digit shifts and rotations are provided, but not bit manipulations, hence:

Left shift one bit	C=C+C
Right shift one bit	C=C+C and RCR used in appropriate combinations

Rotate left one bit	C=C+C
	JNC +02
	C=C+1

Rotate right bits	use rotate left and RCR

The above can be effected in more ways than are shown here, but those above use only the C register.

If you write a non-programmable routine, you have status registers a and b to use for scratch. In the light of 1(b) above, you should either attempt to save the user program counter from b, or invalidate the line number, resetting the program counter to something recognizable.

If you run short of scratch registers, and the status register (of the CPU) is full of valuable data, but you still must test the state of a bit, then place the bit in the C register, rotate it to either XS, or MS fields, and then use C=C+C XS, or MS an arbitrary number of times to shift the desired bit into the carry flag. Then you may make a conditional branch, and allow the state of the bit to affect your program.

Does the result of a subroutine have to be noted by the calling program? Then don't forget that in m-code the return address can easily be altered. I.e. if our subroutine is to have a true/false result, then if true we will execute

```
POP ADR
C=C+1 M
PUSH ADR
```

When the return is encountered, the program must take the form:

```
?NC XQ 'Subroutine'
JNC to an 'if false routine'
  .
  .
  . <- continue here if true
```

This idea of incremented return addresses can easily be extended to simulate a computed GOTO.

Any who are programming in m-code, and have a 143A printer and the IL module, should keep in mind that when the IL module printer ROM is disabled it has just changed addresses: it has moved from page 6 in memory to page 4 (where the diagnostic ROM usually resides). Take care, then, if you are experimenting with page 4, and have the IL module in place.

2(b) Task size.

Of the many problems we try to solve on our 41's, there are really two types: those which can easily be broken down into a few small repetitive pieces, and those which can only be broken down into large pieces, comparable to user-code instructions. I will take the time here to define two terms describing these types:

i) Low level tasks: Those readily broken down into small sub-tasks. Though these component tasks are not always repetitive, the pieces are generally smaller than those behind user-code instructions.

ii) High level tasks: Those which may only be divided into large sub-tasks - of the size carried out by user-code instructions, or sequences of such instructions.

As should be readily evident now, low level tasks are those which are best, and most easily programmed in m-code, rather than in user-code.

CODE and DECODE provide familiar examples of low level tasks. In CODE we are changing ASCII bytes into hex nybbles, and in DECODE we are changing hex nybbles into ASCII bytes. As we are all aware, there are no user-code instructions for dealing with 'halves' of characters easily. It can be done, but only with finagling and coaxing, and usually only in the flag register. Its task is also very repetitive: nybble to byte over and over, or the converse. It is a task which is ideally suited for m-code.

An example of a routine one might like to see in m-code, but shouldn't, is a quadratic root solver. Why not? Because it is a high level problem. The optimized user-code version, which most are familiar with, is only 35 bytes (with a two character label and end). But an m-code version (which I was foolish enough to write) is over 80 bytes, and there is no significant increase in execution time. Why? Because the m-code routine calls the same math subroutines as the user-code version, together with a few extra routines to put things in and out of scratch. With this type of routine it would be advantageous to place it in an EPROM, but in user-code, since this provides an inherent speed increase.

A major task can often be broken down into smaller parts, some of which are low level tasks. This is the real place of m-code in our use of the 41. When we use it here, it makes our programs run faster, and allows the code to flow more easily when writing the program.

I realise that this doesn't give hard and fast rules for determining the suitability of a problem to m-code solution, but at present no rules exist. However, if you make the low level/high level concept an extension of your program design, you should have no difficulty with the decision.

3 Peripheral controls.

'Smart' peripherals are those devices to which the 41 can pass control, using the SELP r instruction. When SELP r is executed, the 41 CPU stops executing instructions, and the peripheral begins executing them. This continues until an opcode is encountered that has bit Ø set. Control is then passed back to the 41 CPU after execution of the instruction by the peripheral.

The main source of information on the 41c m-code instruction set has been Steve Jacobs' article in PPC Technical Notes #9, but there are errors in his account of the peripheral instructions (see pp.84-5) which I would like to note here. SELP is listed as having a type 'f' parameter, but its parameter is in fact of type 'r'. [See the parameter definitions at the foot of p.84. .ED.] The importance of this becomes evident when we start defining the peripheral, IL module m-code instruction set. <2>

Peripheral instructions of the SELP type:

1) **SELP r** Select and pass control to peripheral 'r'.
r ranges in value from 0 to 15 (F). Bit structure: ???? 1ØØ1 ØØ
This is a normal class Ø instruction. It passes control to the peripheral numbered 'r'. [See Jacobs' TABLE Ø.]

2) **?PFSET r** Is peripheral Flag r Set?
r ranges in value from Ø to 15 (F). Bit structure: ???? ØØØØ 11
A peripheral may have up to 16 flags. If flag r is set, then the 41 CPU carry flag is set, as with any compare or test instruction. Note: this instruction immediately returns control to the 41c CPU.

3) **PLDI $ab** Peripheral Load Immediate.
ab may range in value from hex ØØ to FF. Bit structure: aaaa bbbb ØX
This instruction sends the byte of data (ab, in hex digits - bits: aaaabbbb) to the selected peripheral. If bit Ø (bit X) of the instruction code is set, control is immediately returned to the 41c CPU.

4) **C=PREG r** Load exponent of C from the peripheral register or data line r.
r ranges in value from Ø to 15 (F). Bit structure: ???? 111Ø 1X
The contents of the register, or dataline r of the selected peripheral are placed in digits Ø and 1 of the C register in the 41 CPU. If bit Ø of the instruction (bit X) is set, control is passed back to the 41c CPU.

Since the IL module is such a smart peripheral, we we will set out defining an instruction set for specific use with that peripheral. Control of the module, and hence of the IL, is effected through use of the SELP type of instruction, together with another of the class Ø instructions.

Instructions for use with the IL module:

A) **WREG r** Write register C exponent digits to peripheral register r.
r ranges from Ø to 7. Bit structure: 1??? ØØØØ ØØ.
This is a quite normal, **class Ø, type Ø** instruction. (The only other [used?] type Ø is the NOP, ØØØ.) <3> It writes the contents of the 41 CPU register C (digits Ø and 1) to register 'r'. Note that this is **not** part of the SELP set as defined above. <4>

B) **RREG r** Read register r of peripheral r' to exponent of C.
r (r') ranges from Ø to 7.
This, as you will see, is made up of **three** SELP type instructions. A formal definition is being given, in order to assist in IL module applications, and also because of an idiosyncracy. RREG r is to be understood as the sequence:

```
           ( SELP r
  RREG r = ( C=PREG r'  <5>
           ( ?PFSET r  (with bit Ø set)
```

The reason for the ?PFSET instruction is not presently known. It is serving to pass control back to the 41 CPU, but that could have been done with

C=PREG r. Paul Lind reports that the flag test never appears to be used as anything but a NOP. Perhaps the IL module requires the extra instruction cycle to carry out all of its internal processing?

It should be emphasized here, that after the SELP r instruction has been issued, any of the SELP type commands may be executed, and indeed if one is to operate upon an IL register, then SELP r must be given to select that register [peripheral??] before any other commands are executed.

C) Paul Lind reports that it is advantageous to define a couple of special cases: IL WRITE & IL READ.

IL WRITE is just writing a message out over the IL.
IL READ is the act of reading a message from the IL, or more precisely, the last received message.
I am leaving a formal definition of these for Paul, as he is compiling other IL related m-code material for publication, and I am not familiar enough with this to be sure of such a definition.

Also used with the IL module is the **?FI f** command. This is the external flag line test, and it allows the IL module to inform the 41 of IL status, without any direct contact with it. ?FI 8, ?FI 9 & ?FI A are all used in the IL module, but just what each indicates I do not so far know. Paul feels that they indicate

i) Frame not returned as sent,
ii) Output register available and,
iii) Frame received.

One must keep in mind that for these flags to possibly be true [set], the FLGEN flag (bit Ø) in IL register #1 must be set, for this flag enables the FI line from the module to the 41 CPU.

We are just beginning to learn about the IL module, and if we are to learn more, then others with greater insight than I must examine its code. Those who wish to program in m-code, using the IL module, will find that it is necessary to purchase the various manuals that are available (see PPCCJ, V10N1P34), and the development module, and a second 41c will prove to be of great assistance when debugging m-code/IL programs, so Paul informs me. [One to transmit data over the IL, the other to receive it? .ED.]

4 Assemblers and disassemblers.

I present this here as the result of nine months' work last year. I will work from the inside of the Assembler/disassembler, to the outside, presenting two external representations that are really very similar internally.

We require two data files for the routines to work with:

i) containing a table of opcodes
ii) containing a table of mnemonics

The assembler looks through the mnemonic table, and when it finds a mnemonic matching the one you have entered, it jumps to the corresponding entry in the opcode table and stores it in ROME (Emulated ROM). The disassembler is just the opposite in concept, but it is not that easy!

The assembler must know whether or not there is a parameter/field to go with the command. It must then either ask you for it, or parse from the already entered string. (The second method is a bit more complicated for the programmer, but much easier for the eventual user.) The assembler should also know whether or not the XQ/GO's entered require translation to relocatable form. Furthermore, since parameters can take several forms, they must all be checked. It all sounds very easy, but try programming it!

The disassembler is much trickier. When the disassembler examines a word it must know the answers to several questions to properly decode it:

- is this the second word of an XQ/GO, or the data word of a relocatable type, or the data word for a message or error routine?
- is this the second word of an LDI?
- are we in SELP mode?
- is this ASCII or ROM display data?

The answers to all of these affect the way the word is decoded, often with disastrous results when a wrong assumption is made. The disassembler can determine most of this information for itself, but only if we start decoding 5 to 10 words before the point we wish to start disassembling. (This gives the disassembler a chance to sort itself out.)

This internal structure can be utilized with two external systems. The difference between the two consists in the method of entering data to the assembler. This can be done in two ways:

i) By reading ASCII data from ALPHA [or numerical data from X, as in ASSEMBLER 3].

ii) By allowing each of the pair of routines to take over, entering the commands just as we normally do, by single keystrokes for each function, i.e. by redefining the keyboard.

At present there are no programs of type ii), but I know of three using type i). One of these is my own, the other two being by Michael Thompson & Richard Collett, and by Nelson Crowle. <6>

I realize that I have not presented much information here, but I am only trying to lay a foundation for anyone who may be attempting to write such a beast.

* * * * * * * * * * *

Addendum. An important mainframe entry point.

This is added here to present an important entry point uncovered by Michael Thompson (8496) & Richard Collett (4523) of **PPC Melbourne**. It is a remarkable piace of disassembly on their part, and this is only meant to convey its (basic) use quickly to all.

Entry Point ØE52

If you XQ ØE52, a few display odds and ends of tasks are performed, after which the 41c goes into light sleep. [The display and the display chip alone are active, the display chip busily monitoring the keyboard for a sly push.] This leaves the return address of your XROM routine on the CPU return stack. The entry point also sets the partial key sequence flag (which is essential when coming out of sleep) and the message flag. When the next key is pressed, the 41 notes that the partial key sequence flag is set, and by way of a RTN, hands control over to your XROM routine. Note that because the message flag is set, the display must contain something on entry to ØE52 or a crash will result [since the display is left justified. And if the display is full of spaces, then guess what? .MT.). When control is passed to your XROM routine, the logical keycode of the key that was pressed is left in register N (digits 1 & 2). The G register must be cleared before entry, or the keycode returned will be incorrect.

Now we just need to find out why HP doesn't use this entry point in their own routines. . .

A tribute: I would like to thank Paul Lind and Lynn Wilkins for their guidance in my m-code work, and the group I will name as PPC Melbourne for their support.

mcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmcmc

Notes by .MT. and .ED.

<1> See the quite different thoughts on this matter expressed by Kamikaze in TN#14, preprinted in the Journal, PPCCJV10N3Pp.11-12. But all is not on Kami's angelic side. If we are to use those three applications ROM's of the 41c, provided free by HPwith each one purchased, we still do well to heed Chris' words here. Look out for HP's corporate conception of what the HP-41c is, while using the buttony black box in accordance with it, but also think outside the same constraints when no numbers are waiting in the wings to be crunched. The HP-41c is black box + operating system defining software. It is not itself with different software which we are at liberty to define. .ED.

<2> Note that Chris is HERE proposing mnemonics **for** an **existing** instruction set **used** in the IL module ROM. Standard listings of the instructions of that ROM should be revised in accordance with Chris' proposals below. Dumb .ED. found what was going on quite unclear until this particular penny dropped. As there could possibly be others with the same central nervous system disorder . . . It also seems to .him. that the instructions below, where they **seem** to be of class 1, 2 or 3, are actually those **following** a SELP prefix. We trust these little comments help? 1) below, it seems to deadhead .ED., should be regarded as the **prefix** to 2), 3) and 4). The other way to regard these is as instructions which are executed by their corresponding

codes only **when** a peripheral has been selected, and while it **continues** to be selected. The same seems to be true of instruction A of the IL set below, in relation to 1), 2) and 3) again.

<3> According to Jacobs' TABLE Ø, **all** these are NOP's. This seems to be the (presumably) solitary exception.

<4> Of the IL device - or of the IL **module**? Previously labelled 'UNUSED' in the Jacobs TABLE Ø. .ED.

<5> Since .ED. has added the 'dash' to the r of the second mnemonic, be warned: normally one would be working on one peripheral at a time in a routine. Whis would be selected by the value of r in SELP r. Once this has been executed, peripheral r is active - now a specific one byte register of peripheral r is to be read to the exponent of C - which register is now determined by the value of the r parameter in the second instruction of this sequence, C=PREG r - but the value of **this** r need not be the same as that of the first. Thus the r'. Deadhead .ED.

<6> If you are contemplating writing one, then give very careful consideration to the mnemonics you use. There are already about 10 assemblers around, most of them in user-code, and all of them use slightly different variats of the Jacobs/De Arras mnemonics. Please try to keep **exactly** to the Jacobs mnemonics so that others can use your assembler with ease. If you are writing a disassembler as well, then try to make the two consistent with eachother, i.e. what is keyed into the assembler should be the same as the disassembler displays or prints out. Since there are so many dis/assemblers around, there is little point in writing yet another, but I suppose this is against the PPC religion! NOW **Michael Thompson** is being .ED.! A jolly forlorn hope. Look in this issue of TN, and the last. When are PPC'ers going to stop writing key assignment routines and programs? I spent four months on them, and the thought of any return is positively nauseous. We will have assemblers/disassemblers with us for several years, for sure. Yr. Sic .ED.

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

(Continued from p.66.)
the fourth wheel I bought, a nice 15 pitch wheel. (Wordstar does not support proportional spacing, but the F10-40 does - can be switched to that mode, and Spellbinder, in its HP Word 125 incarnation, my first love in word processing software, does.) There are, apparently, no proportional spacing Diablo wheels usable on the ITOH F10-40, known, I think in the USA as the ITOH Starwriter, and anyway they are probably WPS, or in some other awful non-standard character sequence. There is only ONE proportional spaced Qume wheel available with the normal character sequence - and I will get it (the distributer has no stock, but I have located a supplier), but it seems to me, in my ignorance, quite absurd that this should be so. When a user pays like hell to buy a 'precision' printer, as these are called (line height adjustable by 1/48th of an inch, horizontal motion by 1/120th inch), why do they not want to purchase proportional spaced wheels? If there were a demand, there would very likely be a supply.

Now this ITOH printer (which I would commend to anyone wanting an excellent printer - my wife prefers it to the Diablo 630, partly because it is quieter, as well as being about half the price), can be programmed to accept any printwheel sequence - but there is simply no inducement to do so on my part. The Jinglish ITOH manual is one obstacle, the rotten chore of calculating the right sequence of bytes for a given wheel is the other. Anyway, once programmed, resetting the printer clears the programmed sequence. Does any PPC member know of a way around this? (Yes, I COULD scrap the ITOH and buy a Qume, which is switch selectable, switching to a second ROM. The distributers of the ITOH here were utterly unhelpful . . .) Programming the ITOH would not be much good, unless I print from a disc file, since the two word processing programs I use reset the printer before printing. Is there any good word processing CP/M software on the market that will solve this problem - or a patch to Wordstar or Spellbinder???

Desolate, disproportionate .ED.