An alternative to Bank-Switching in the 41CL

Monte Dalrymple

Background

Bank-switching was not part of the original 41C hardware or software. But with up to 64 functions possible in the 4096-word address space of a page, it is quite easy to run out of address space in a plug-in module. With bank-switching, and up to four banks per page, the equivalent of 16,384 instruction words will be available in each page. This paper will review how bank-switching is used in the 41C, and introduce an alternative allowed by the Memory Management Unit (MMU) in the 41CL.

Bank-switching

Bank-switching is controlled by four instructions (with mnemonics `ENROM1`, `ENROM2`, `ENROM3` and `ENROM4`) that are ignored by the CPU in the 41C. These instructions control the state of two extra bits of program address. These extra two bits of address are not global, but are specific to the address of the page(s) where the `ENROM` instructions are executed, because the hardware that holds the bits usually exists in the module where the code resides. Thus each page (or pair of pages) has an independent bank select facility.

Because of the way the Ports are related to page addresses in the 41C family, only one set of bank-select bits exists for each port. Thus pages 8 and 9 share two bank-select bits, as do pages A-B, C-D and E-F. Pages 4, 6 and 7 can have individual bank-select bits. HP embraced bank-switching in the 41CX, and pages 3 and 5 share a set of bank-select bits, used for the X-Functions in the 41CX. The operating system pages (0-2) do not support bank-switching. Had the 41C series continued beyond the CX, I suspect that might have changed.

The 41C family actually powers down during light sleep (between keypresses), and the state of the bank-select bits are not preserved during light sleep. This is both a blessing and a curse. It is a blessing because it means that nothing special needs to be done at the end of a function to restore the state of the bank-select bits to known values. The light-sleep state does that automatically. The drawback of this operation is that any bank switching must be done every time a function is started or restarted (with a partial key sequence, for example).

Switching the current bank can be tricky. Think about it: the CPU is happily fetching instructions, when it encounters an `ENROM` instruction that it treats as a `NOP` (No Operation). The CPU blithely continues on and fetches the instruction at the subsequent program address. Meanwhile the page-select bits have been modified, meaning that the memory address is actually pointing at a completely different section of memory. This is why many people who write software don't care for bank switching, and in-line bank switching is often referred to as "pulling the rug out from under the running program."

Actually, the problem can be even worse, if the timing of the actual bank switch is not clearly specified relative to the execution of the bank-switch instruction. In the case of the 41C, at least for the HP-designed ROM used in the 41CX, the word immediately following the `ENROM` instruction is fetched from the current bank, while all subsequent instructions are fetched from the new bank.

The usual way to deal the problem of switching banks in-line in a program is to use a single software interface routine to switch banks. This saves the software writer from having to think about always having to make the code match up at certain locations in different banks. With a common interface routine only one piece of code needs to line up between the various banks.

Bank switching was only implemented in a few pieces of 41C software, and I'll illustrate three different techniques that I have found that were used for switching banks.

An example from the 41CX X-Functions is shown in ***Listing 1***. The bank-switching done in this case is unique, because of the way the code is arranged. The main code for the X-Functions is hardwired in Page 3 of the CPU program address space, but only Page 5 has a second bank. This means that the `ENROM` instruction is actually changing the bank-select bits for a page other than the page where the instruction resides, completely avoiding the inline-code problem. So the code merely switches the bank in Page 5 and then jumps to a Page 5 address.

```
3089 180                     ENROM2                ; select bank 2 in page 5
308A 145162                  GOLNC   5851          ; call a subroutine in page 5
```

Listing 1

The second example, shown in ***Listing 2***, is from the HP-41 Advantage module. This module occupies two consecutive pages, with a second bank only in the upper page, so it is also somewhat unique. Bank switching is only done from the lower page, and automatically jumps to a specified location in the upper page, again completely avoiding the inline switching problem.

Code in the second bank is accessed through a subroutine call, with ten bits of the destination address specified by the word following the actual subroutine call. Much like the port-dependent subroutine call used to start the bank-switch (the `GSB41C` mnemonic), there are four different subroutine entry points, depending on which 1K section (called a quad) of the page the destination resides in. The assembler program will automatically pick the correct subroutine entry point. The software in Bank 2 (of the upper page) returns using a normal `RTN` instruction, without needing to restore the Bank 1 selection. The Bank 1 selection is done after the return.

```
8249 36D08C1DF               GSB41C  (GB2Q2) 85DF   ; bank 2 call
824C 08C                     #08C                   ; at address 988C
...
85D6 1B0      (GB2Q0)        C=STK                  ; entry for quadrant 0-3: get target addr
85D7 330                     CXISA                  ; 0 1 2 3
85D8 0CB                     GONC    +25 85F1
85D9 1B0      (GB2Q1)        C=STK                  ; entry for quadrant 4-7: get target addr
85DA 330                     CXISA                  ; . . . . 0 1 2 3
85DB 1F6                     C=C+C   XS             ; . . . . 0 2 4 6
85DC 1F6                     C=C+C   XS             ; . . . . 0 4 8 C
85DD 236                     C=C+1   XS             ; . . . . 1 5 9 D
85DE 06B                     GONC    +13 85EB
85DF 1B0      (GB2Q2)        C=STK                  ; entry for quadrant 8-B: get target addr
85E0 330                     CXISA                  ; . . . . . . . . 0 1 2 3
85E1 1F6                     C=C+C   XS             ; . . . . . . . . 0 2 4 6
85E2 236                     C=C+1   XS             ; . . . . . . . . 1 3 5 7
85E3 1F6                     C=C+C   XS             ; . . . . . . . . 2 6 A E
85E4 03B                     GONC    +7 85EB
85E5 1B0      (GB2Q3)        C=STK                  ; entry for quadrant C-F: get target addr
85E6 330                     CXISA                  ; . . . . . . . . . . . . 0 1 2 3
85E7 1F6                     C=C+C   XS             ; . . . . . . . . . . . . 0 2 4 6
85E8 236                     C=C+1   XS             ; . . . . . . . . . . . . 1 3 5 7
85E9 1F6                     C=C+C   XS             ; . . . . . . . . . . . . 2 6 A E
85EA 236                     C=C+1   XS             ; . . . . . . . . . . . . 3 7 B C
85EB 1F6                     C=C+C   XS             ; . . . . 2 A 2 A 4 C 4 C 6 E 6 E
85EC 013                     GONC    +2 85EE
85ED 236                     C=C+1   XS             ; . . . . . . 3 B . . 5 D . . 7 F
85EE 1F6                     C=C+C   XS             ; . . . . 4 4 6 6 8 8 A A C C E E
85EF 013                     GONC    +2 85F1
85F0 236                     C=C+1   XS             ; . . . . . 5 . 7 . 9 . B . D . F
85F1 23A                     C=C+1   M              ; increment return address
85F2 170                     STK=C                  ; and put on stack
85F3 3DA                     CSR     M              ; shift current page to C<3>
85F4 3DA                     CSR     M
85F5 3DA                     CSR     M
85F6 23A                     C=C+1   M              ; increment to next page
85F7 1BC                     RCR     11             ; rotate computed address to C<6:3>
85F8 180                     ENROM2                 ; enable bank 2
85F9 1E0                     GOTOC                  ; and jump into bank at target addr
```

Listing 2

The final example, shown in ***Listing 3***, is from the HEPAX module. This is the most complex bank-switch code, because it allows the use of all four banks. This bank-switch code at addresses `FC3-FCA` must be replicated in all four banks for the common interface. In this case the two words following the call to the bank-switch subroutine specify the bank and destination address within the bank. But unlike the HP implementations, this case is actually a jump rather than a subroutine call, so the code only returns to Bank 1 with an explicit jump back. Of course it would be easy to modify this code to implement subroutine calls into another bank.

```
814F 3B508C39A              GSB41C  (GOBNK)  8F9A         ; go to address in  different bank!
8152 3FA                    #3FA                          ; BANK 3, address 7FA
8153 007                    #007
...
8F9A 1B0      (GOBNK)       C=STK                         ; get address of first argument
8F9B 330                    CXISA                         ; and then the argument
8F9C 23A                    C=C+1   M                     ; point at second argument
8F9D 0B6                    ACEX    XS                    ; put bank digit in A.XS
8F9E 1BC                    RCR     11
8F9F 0B6                    ACEX    XS
8FA0 03C                    RCR     3
8FA1 0B6                    ACEX    XS                    ; restore A.XS, bank digit is in C.S
8FA2 27C                    RCR     9                     ; align argument in C<7:4>
8FA3 170                    STK=C                         ; and save it on stack for now
8FA4 0BC                    RCR     5                     ; align address of second arg
8FA5 330                    CXISA                         ; and fetch the arg
8FA6 2BC                    RCR     7                     ; get it out of the way
8FA7 1B0                    C=STK                         ; and retrieve first arg from stack
8FA8 0BC                    RCR     5                     ; target address to C<2:0>
8FA9 3DA                    CSR     M                     ; five shifts to get page into C<4>
8FAA 3DA                    CSR     M
8FAB 3DA                    CSR     M
8FAC 3DA                    CSR     M
8FAD 3DA                    CSR     M
8FAE 1BC                    RCR     11                    ; align target addr to C<6:3>
8FAF 170                    STK=C                         ; and put this address on the stack
8FB0 276                    C=C-1   XS                    ; bank digit is in C.XS, decrement it
8FB1 276                    C=C-1   XS                    ; dec again, if carry go to bank 2
8FB2 0BF                    GOC     (GOB2)  +23 8FC9
8FB3 276                    C=C-1   XS                    ; dec again, if carry go to bank 3
8FB4 07F                    GOC     (GOB3)  +15 8FC3
8FB5 083                    GONC    (GOB4)  +16 8FC5      ; else go to bank 4
...
8FC3 140             (GOB3) ENROM3                        ; select bank 3
8FC4 3E0                    RTN                           ; go to address on stack
8FC5 1C0             (GOB4) ENROM4
8FC6 3E0                    RTN
8FC7 100             (GOB1) ENROM1
8FC8 3E0                    RTN
8FC9 180             (GOB2) ENROM2
8FCA 3E0                    RTN
```

Listing 3

Bank-switching is a powerful way to expand program address space, but it is not without complications. The examples above show how these complications were managed, both by HP and the authors of the HEPAX module. The last two cases are complicated mainly because of the necessity to interpret the one or two words required to encode the destination address into the program lines.

The 41CL fully supports the bank-switching hardware and software of the 41C, with the bank-select bits contained in the CPU logic itself. The 41CL contains a large physical memory, and it was easiest to load the contents of four banks of code in successive pages of this memory. This physical memory organization is also the most efficient for implementing the PLUG functions that insert an image into a Port. But the downside of this memory organization is that unless four pages of space are reserved up front for an image, it becomes a compatibility nightmare to upgrade an image (like the 41CL Extra Functions) from a non-bank-switched version to a bank-switched version. This led me to look for alternatives to bank switching.

Dynamic page-switching

The MMU in the 41CL allows pages to be dynamically mapped and unmapped under program control, and I deliberately spaced out the MMU entries in memory to provide room for up to different four MMU entries for each page and bank. My original thought when I spaced out the MMU entries in memory was that this would allow multiple "personalities" that could be selected under user control, but the software to implement this feature has not yet been written. (I'm actually working on it right now.)

The way that HP used bank-switching was the inspiration for the page-switching that I will describe here. Instead of remapping the current page, this software will program the MMU so that Page 4 accesses a specified page in the physical memory of the 41CL. Then the software can access routines in Page 4 directly, with known logical memory addresses. At the end of the routine that dynamically loaded Page 4, the previous MMU programming for Page 4 is restored and the machine continues on as if nothing happened. I chose Page 4 because this page will almost always be empty. Only 'Angel Martin's Library-4 software currently uses Page 4.

I remap Page 4 to provide fast access to subroutines at fixed addresses. If you have done much coding for the 41C in machine language, you know that calling a subroutine with a fixed address is much more efficient than dealing with what is called a port-dependent address. Port-dependent subroutine calls (using the `GSB41C` mnemonic) and port-dependent long jumps (using the `GOL41C` mnemonic) actually call helper subroutines in the 41C mainframe code to calculate the destination address, because code in a plug-in module usually has no *a priori* knowledge of which page it is executing from. These helper subroutines are great for the programmer, as long as you remember the restrictions that come with them.

The first restriction is that the routines use the C register in the CPU, so you cannot use this register to pass information to your target subroutine. The second restriction is that the helper routines use an additional subroutine level, beyond the one required to return to the calling routine. Given that the subroutine stack is only 4 levels deep in the CPU, and the Operating System return address is already using one level, this can lead to unexpected behavior if you are not paying attention. I have accidentally violated both of these restrictions during development, and once in released code! In addition, as you saw in ***Listing 2*** and ***Listing 3***, these helper routines also require three words of program space, where a normal subroutine call requires just two program words.

The code for the combined load/unload routine is shown in ***Listing 4***. The entry point to map a page into Page 4 is `LLIB` and the entry point to restore the previous Page 4 mapping is `ULIB`. Most of the operations are the same for the two cases, so a common routine made sense. Calling the `LLIB` routine requires that the physical address for the "library" code be loaded to the X field in the B register in the CPU prior to calling the subroutine. The physical address for the MMU is twelve bits, so it won't fit in a program word. I could have used the technique of ***Listing 2***, namely ten bits in the next program line and four different entry points to signal the final two bits, or the technique of ***Listing 3***, with two following program lines, but it seemed simpler to use a register to hold the destination address.

Even though it is not necessary at this point, given the dearth of software that uses Page 4, the code here implements a virtual four-level stack for the MMU entry for Bank 1 of Page 4. Only the first bank is remapped because I don't anticipate ever using bank-switching in a Page 4 library (and only the latest 41CL FPGA programming supports bank-switching in Page 4 anyway). If access to more code space is required, just load a new page to Page 4.

The MMU entry for Bank 1 of Page 4 is stored at physical address 0x804040, and the three succeeding addresses are used for the virtual stack. A call of `LLIB` moves the contents of address 0x804042 to address 0x804043, 0x804041 to 0x804042, and 0x804040 to 0x804041. This makes room for the new programming for the MMU. A call of `ULIB` reverses this sequence, discarding the "bottom" of the stack and placing a null entry on the top of the stack.

```
0EAE 03E      (LLIB)          B=0     S       ; load library tagged by B.S = 0
0EAF 023                      GONC    (LIBSU) +4 0EB3
0EB0 05E      (ULIB)          C=0     S
0EB1 23E                      C=C+1   S
0EB2 0FE                      BCEX    S       ; unload library tagged by B.S = 1
0EB3 1303F0   (LIBSU)         LDI     3F0
0EB5 270                      DADD=C
0EB6 3F0                      PFAD=C          ; select 41cl peripheral
0EB7 04E                      C=0     W
0EB8 2DC                      PT=     13
0EB9 090                      LC      2       ; C = 20000000000000
0EBA 010                      LC      0       ; C = 20000000000000
0EBB 210                      LC      8       ; C = 20800000000000
0EBC 010                      LC      0       ; C = 20800000000000
0EBD 110                      LC      4       ; C = 20804000000000
0EBE 010                      LC      0       ; C = 20804000000000
0EBF 110                      LC      4       ; C = 20804040000000
0EC0 090                      LC      2       ; C = 20804042000000
0EC1 010                      LC      0       ; C = 20804042000000
0EC2 150                      LC      5       ; C = 20804042050000
0EC3 2DE                      ?B#0    S       ; load/unload case?
0EC4 023                      GONC    (LLIB1) +4 0EC8       ; branch if load case
0EC5 15C                      PT=     6
0EC6 050                      LC      1       ; C = 2x804041050000 (unload case)
0EC7 013                      GONC    (LIB0) +2 0EC9
0EC8 210      (LLIB1)         LC      8       ; C = 20804042058000
0EC9 01C      (LIB0)          PT=     3
0ECA 10E                      A=C     W       ; A = 2x80404205*000
0ECB 082                      B=A     PT      ; load lib: 8, unload lib: 0
0ECC 1FC      (LIB1)          #1FC            ; read mmu stack
0ECD 0B8                      C=REGN  2       ; get the data
0ECE 0AA                      ACEX    WPT     ; combine address & data
0ECF 0AE10E                   C=A     W       ; and get it in C
0ED1 15C                      PT=     6
0ED2 2DE                      ?B#0    S       ; load/unload case?
0ED3 02F                      GOC     (LIB2) +5 0ED8 ; branch if unload case
0ED4 1A2                      A=A-1   PT      ; load library: next lower (src)
0ED5 222                      C=C+1   PT      ;               next higher (dst)
0ED6 05C                      PT=     4       ; can't do jump here because of carry!
0ED7 023                      GONC    (LIB3) +4 0EDB
0ED8 162      (LIB2)          A=A+1   PT      ; unload library: next higher (src)
0ED9 262                      C=C-1   PT      ;                 next lower (dst)
0EDA 05C                      PT=     4
0EDB 110      (LIB3)          LC      4       ; write command digit
0EDC 1FC                      #1FC            ; load: 2>3, 1>2, 0>1, unload 1>0, 2>1, 3>2
0EDD 0AE10E                   C=A     W       ; set up C for next write
0EDF 1BE                      A=A-1   S       ; decrement loop counter in A.S
0EE0 363                      GONC    (LIB1) -20 0ECC       ; repeat loop if not done
0EE1 0EA                      BCEX    WPT     ; almost done, get final mmu data
0EE2 15C                      PT=     6
0EE3 2DE                      ?B#0    S       ; load/unload case?
0EE4 01F                      GOC     (LIB4) +3 0EE7 ; branch if unload case
0EE5 042                      C=0     PT      ; load library: bottom of stack
0EE6 013                      GONC    (LIB5) +2 0EE8
0EE7 262      (LIB4)          C=C-1   PT      ; unload library: top of stack
0EE8 05C      (LIB5)          PT=     4
0EE9 110                      LC      4
0EEA 1FC                      #1FC            ; write final data
0EEB 149026                   GOLNC   [ENCP00] 0952  ; deselect 41cl peripheral and return
```

Listing 4

The code in ***Listing 4*** is obviously specific to Bank 1 of Page 4, but the exact same technique can be used for any or all pages that are mapped by the MMU, because all MMU entries are spaced apart in memory.

Conclusion

I am currently in the process of upgrading the 41CL Extra Functions to use dynamic page-switching, expanding the feature set to make several of the functions more useful to users. I make no claim that this technique is better than bank-switching, and in fact some of the downsides should be immediately obvious. In particular, it requires both more code and a longer execution time than even the HEPAX-style bank switching.

For example, loading a new Page 4 mapping requires 93 instruction times, and restoring the previous Page 4 mapping requires another 92 instruction times. But it is important to remember that the code in Page 4 has a fixed address, which allows for shorter, and faster, jumps and calls, so much of this overhead may be reclaimed during function execution. If the software stack of MMU entries were not required loading a library would be much faster, and unloading would not be required at all.

A second issue has to do with error handling. Normally, exiting a function in the case of an error means sending an error message to the display and jumping to a mainframe error cleanup routine. And if the "Error Ignore" status bit is set the error message step is skipped and the code returns directly to the operating system. In either case there is no opportunity to unload the dynamic code from Page 4! The way around this problem is to use short "stubs" of code in Page 4 to return to error message code in the calling page. This special error message code unloads Page 4 before proceeding with the normal error message exit. Somewhat annoying, but still manageable.

If nothing else, I expect that the dynamic page-switching presented here demonstrates the true power of the MMU in the 41CL.