Appendix XI
The Alternate Register Set.

The Z80 microprocessor has two sets of registers - the normal set (AF, BC, DE and HL) and the alternate set (AF', BC', DE' and HL'). Unless the techniques outlined in this appendix are implemented the user is prohibited from using the alternate register set. This is because the alternate register set is used by the firmware (the Kernel in particular) for storing certain system values and flags. Providing that the user never enters the firmware then the alternate register set may be used without restriction. Of course this would mean that the user would be unable to use any facilities provided by the firmware. Furthermore, the user would also have to disable interrupts as interrupts cause firmware routines to be executed.

In the sections below a number of different methods are described that allow the user to overcome these restrictions. The method chosen will depend on what use is to be made of the alternate register set.

a. The firmware's use of the alternate register set.

The Kernel stores a couple of system variables in the alternate register set. This allows the Kernel to access these variables easily and thus speeds up a number of operations (particularly entry to and exit from firmware routines). Only BC' and the alternate carry flag (carry') are used to store values, however, routines do make use of the other alternate registers and so firmware routines may corrupt them.

B' is used to store the I/O address of the gate array (#7F). C' is used to store the value required to set the current ROM state and screen mode:

Bits 0..1:	Set the screen mode.
Bits 2:	Disables the lower ROM.
Bits 3:	Disables the upper ROM.
Bits 4..7:	System value to select gate array function.

By changing the ROM state bits and performing an OUT (C),C instruction the user can enable or disable ROMs. (N.B. The Z80 OUT (C),r and IN (C),r instructions use B as the top 8 bits of the I/0 address. The hardware uses these top bits for decoding the I/O address, it ignores the bottom 8 bits!) OUT (C),C may be used to change the ROM state during the interrupt path when the normal Kernel entries (e.g. KL U ROM ENABLE) may not be called because they enable interrupts.

Carry' is normally false. When carry' is true this indicates that the firmware is in the interrupt path. The firmware occasionally uses this flag to allow it to take a different action when it is in the interrupt path to the action it takes when it is not in the interrupt path (usually avoiding enabling interrupts).

b. Simple use of the alternate register set.

The technique described in this section allows use of the alternate register set providing that no firmware routines are called and that interrupts are disabled.

After disabling interrupts registers A', DE' and HL' may be used as required. If registers BC' or F' (in particular carry') are used then their original contents must be restored before interrupts are re-enabled. The user may alter bits in C' (as described in (a) above) and need not restore the original value provided that an OUT (C),C is performed to keep the hardware abreast of the current state. The machine will not function correctly if the hardware and the value in C' are out of step when interrupts are enabled.

This technique requires interrupts to be disabled for the duration of the operation being performed. This is acceptable if the operation is short but not if the operation is lengthy. Disabling interrupts for a lengthy period will stop many firmware functions such as timers (and hence ink flashing, sound generation and keyboard scanning). If the operation to be performed is lengthy then it might be better to consider the use of one of the techniques described in sections (c) or (d) instead.

Example.

The user might want to provide a routine that performs an LD A,(BC) from RAM (similar to the RAM LAM pseudo-instruction provided by the firmware).

The code for this routine could be written as follows:

```
A_FROM_BC:
    PUSH    BC
    DI              ; ** About to use alternate registers
    EXX
    POP     HL      ;Transfer BC to HL'
;
    LD      A, C
    SET     2, A    ;Set the disable lower ROM bit
    SET     3, A    ;Set the disable upper ROM bit
    OUT     (C), A  ;Tell the hardware
;
    LD      A, (HL) ;Read the value from RAM
;
    OUT     (C), c  ;Restore the old ROM state
;
    EXX
    EI              ; ** End of use of alternate registers
    RET
```

N.B. This routine must be RAM resident or disabling the ROMs will have an unfortunate effect!

c. Use of the alternate register set with interrupts enabled.

The technique described in this section allows the alternate register set to be used and interrupts to be enabled. It does not allow firmware routines to be called.

The simplistic use of the alternate register set by disabling interrupts as described above is unsatisfactory if this results in interrupts being disabled for an extended period of time. By patching INTERRUPT ENTRY in the low Kernel jumpblock interrupts can be trapped and appropriate action to restore the firmware registers can be taken. The actions that must be performed are as follows:

Before starting to use the alternate register set the firmware's BC' is saved and INTERRUPT ENTRY is patched so that the user's interrupt routine is used.

When the user has finished with the alternate register set the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine.

When an interrupt occurs the user's alternate registers are saved, the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine. The latter is done in case a second interrupt occurs whilst processing the events kicked from the interrupt path of the first interrupt (remember that the event processing is performed with interrupts enabled).

After interrupt processing has finished the firmware's BC' is saved, the user's alternate registers are restored and INTERRUPT ENTRY is patched back to the user's interrupt routine again.

Note that when INTERRUPT ENTRY is patched it is vital to ensure that the lower ROM is disabled and remains disabled. It is impossible to patch the ROM version of INTERRUPT ENTRY! If an interrupt occurred whilst the lower ROM was enabled then the firmware would jump straight into its interrupt routine without restoring its alternate registers first.

Example.

The following routines implement the scheme described above:

```
;
; The following storage locations are used
;
FIRM_BC:   DEFS 2          ; Two bytes to store firmware's BC'
FIRM_INT:  DEFS 2          ; Two bytes to store the address of the
                           ; firmware's interrupt routine
USER_AF:   DEFS 2          ; Two bytes to store the user's AF'
USER_BC:   DEFS 2          ; Two bytes to store the user's BC'
USER_DE:   DEFS 2          ; Two bytes to store the user's DE'
USER_HL:   DEFS 2          ; Two bytes to store the user's HL'
```

```
; This routine sets up the environment so that the
; user may make use of the alternate register set.
; N.B. Interrupts are enabled.
;
USER_ALTERNATE:
   DI                                ;An interrupt would be disastrous
   EX     AF, AF'
   EXX                               ;Swap to the alternate register set
;
   LD     (FIRM_BC), BC              ;Save the firmware's BC'
;
   LD     HL,(INTERUPT_ENTRY+1)
   LD     (FIRM_INT), HL             ;Save the firmware's interrupt routine
;
   LD     HL, USER_INTERRUPT         ;Use the replacement interrupt routine
   LD     (INTERRUPT_ENTRY + 1), HL
;
   LD     HL, (USER_AF)
   PUSH HL
   POP    AF                         ;Restore user's AF'
   LD     BC, (USER_BC)              ;Restore user's BC'
   LD     DE, (USER_DE)              ;Restore user's DE'
   LD     HL, (USER_HL)              ;Restore user's HL'
;
   EXX                               ;Swap back to the standard register set
   EX     AF, AF'
   EI                                ;We have finished with the alternate regs
   RET
;
; This routine restores the environment for the
; firmware to use the alternate register set.
; N.B. Interrupts are disabled and not re-enabled.
;
FIRM_ALTERNATE:
   DI                                ;An interrupt would be disastrous
   EX     AF, AF'
   EXX                               ;Swap to the alternate register set
;
   LD     (USER_HL), HL              ;Save user's HL'
   LD     (USER_DE), DE              ;Save user's DE'
   LD     (USER_BC), BC              ;Save user's BC'
   PUSH AF
   POP    HL
   LD     (USER_AF), HL              ;Save user's AF'
```

```
LD   HL, (FIRM_INT)              ;Restore the firmware's interrupt routine
LD   (INTERRUPT_ENTRY + 1), HL

LD   BC, (FIRM_BC)               ;Restore the firmware's BC'
OR   A, A                        ;Set the firmware's carry' to be false

EXX                              ;Swap back to the standard register set
EX   AF, AF'
RET                              ;N.B. May be about to enter the interrupt
                                 ;path so no EI.
;
;
; This routine replaces the firmware's interrupt routine
; when the user is using the alternate register set
;
;
USER_INTERRUPT:
CALL FIRM_ALTERNATE      ;Switch the environment to the firmware
CALL INTERRUPT_ENTRY     ;Run the normal interrupt routine
JP   USER_ALTERNATE      ;Switch the environment back to the user
```

To start using the alternate register set the user obeys the instruction:

```
CALL USER_ALTERNATE
```

To finish using the alternate register set the user obeys the instructions:

```
CALL FIRM_ALTERNATE
EI
```

d. Calling firmware routines whilst using the alternate register set.

The technique described in this section extends the technique described in section (c) to allow the user to call firmware routines whilst using the alternate register set.

To call a firmware routine requires exactly the same action as is required for the interrupt routine:

Before calling a firmware routine the user's alternate registers are saved, the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine. The latter is done in case an interrupt occurs whilst executing the firmware routine.

After running the firmware routine the firmware's BC' is saved, the user's alternate registers are restored and INTERRUPT ENTRY is patched back to the user's interrupt routine again.

As indicated in section (c) it is vital to ensure that the lower ROM remains disabled while the alternate register set is in use since INTERRUPT ENTRY in the ROM is not patchable and jumps straight to the firmware's interrupt routine.

Using the routines defined in section (c) a firmware routine may be called by using the following sequence:

```
..
CALL FIRM_ALTERNATE      ;Switch the environment to the firmware
EI                       ;FIRM ALTERNATE disables interrupts
CALL firmware            ;Run the firmware routine
CALL USER_ALTERNATE      ;Switch the environment back to the user
```

The above code is rather long if a lot of firmware calls are to be made (10 bytes per call). The following routine takes the address of a firmware routine to call to as an inline parameter (and only uses 5 bytes per call).

```
;
; This routine saves the user's alternate registers, calls a
; firmware routine whose address is passed inline and then
; restores the user's alternate register set afterwards.
;
FIRM_ROUTINE:
        CALL    FIRM_ALTERNATE      ;Switch to the firmware environment
        EXX                         ;N.B. Interrupts are disabled
        POP     HL                  ;Recover address of routine to call, uses
        LD      E,(HL)              ;firmware's DE' and HL' which may be
        INC     HL                  ;corrupted
        LD      D,(HL)              ;Get routine to call into DE'
        INC     HL
        PUSH    HL                  ;Put the real return address back
        LD      HL, USER_ALTERNATE  ;Restore the user environment when
                                    ;the firmware returns by putting a
        PUSH    HL                  ;fake return address on the stack
        PUSH    DE                  ;Save the routine to call
        EXX
        EI
        RET                         ;Jump to the routine to call
```

To call a firmware routine using the above routine the following sequence should be used:

```
…
CALL FIRM_ROUTINE
firmware                 ;Address of routine to call
…                        ;FIRM ROUTINE returns here
```